Generic N-Dim R-Tree Explorations

or how I learned to calm the hyperspatial index

Dave Rostron — @yastero

June 13, 2015

All our wisdom is stored in the trees.

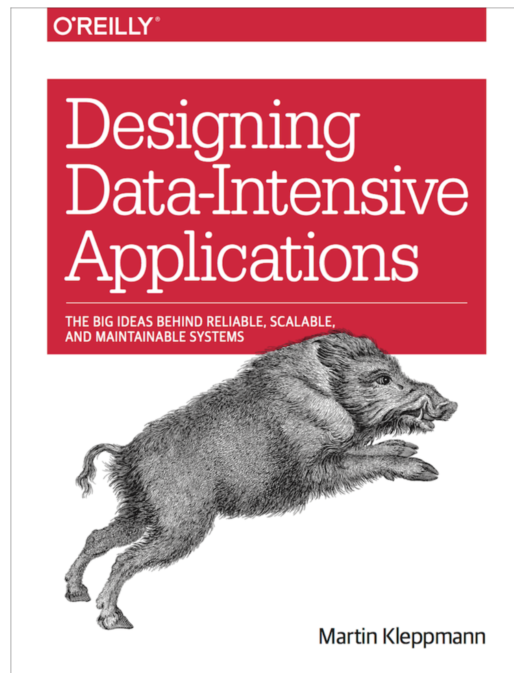
— Santosh Kalwar

opening notes

- this is not complete
- more of a chance to dig into shapeless than look in depth at R-Trees (*at least initially*)
- just having some fun, please join me

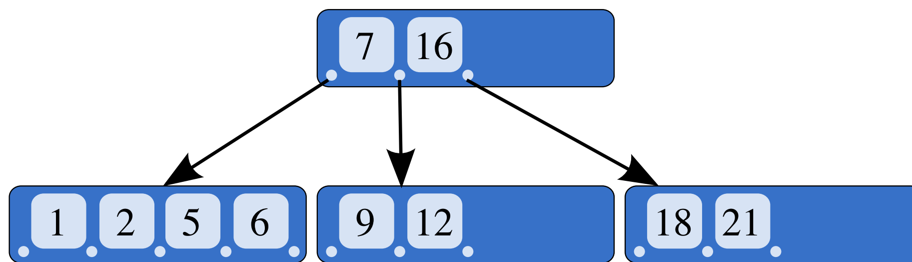
exploration inspiration

"Designing Data-Intensive Applications" by Martin Kleppmann (*recommended*)



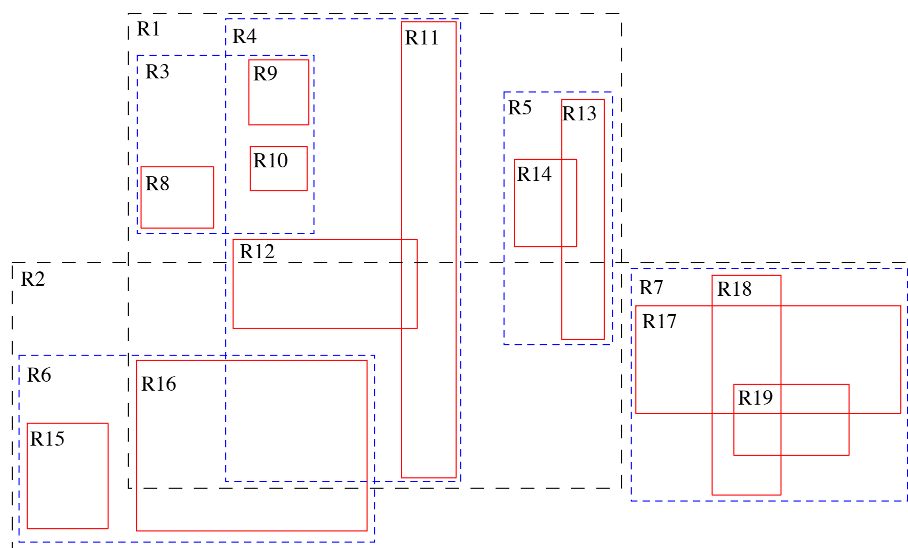
how do we find the needle in the haystack?

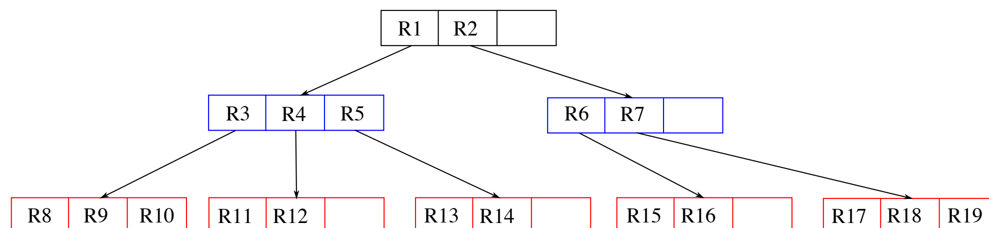
- could scan through the whole stack
- maybe there’s something faster
- B-Tree to the rescue



wait, we want to index our data over multiple dimensions

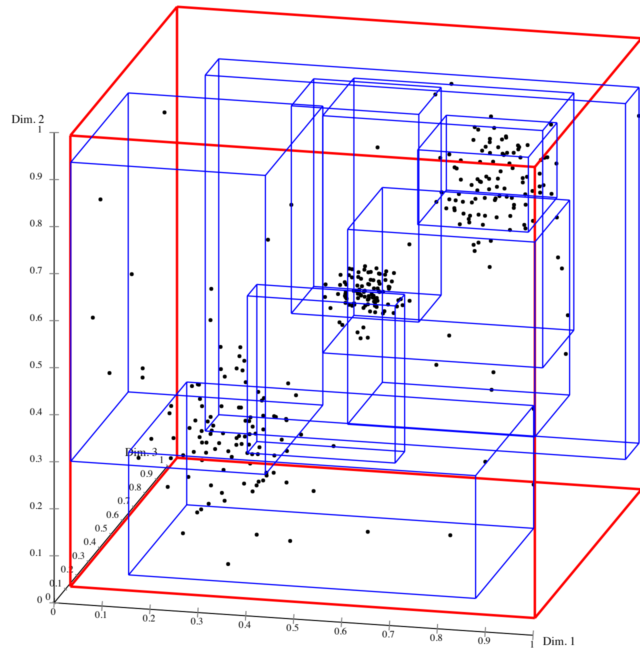
- could get part of the way there with a B-Tree but what about a range query over both dimensions?
- R-Tree to the rescue





R-Tree Overview

- spatial n-dimensional index



- variants: M-Tree, X-Tree, Hilbert R-tree
- sounds useful, tell us more…

how bout some code

```scala
trait Data2D {
  case class Point[T](x: T, y: T)
  case class Entry[V, T](value: V, point: Point[T])
  case class Box[T](xLower: T, xUpper: T, yLower: T, yUpper: T) // inclusive
  sealed trait RTree[V, T]
  case class Empty[V, T]() extends RTree[V, T]
  case class Leaf[V, T](entry: Entry[V, T]) extends RTree[V, T]
  case class Node[V, T](
    box: Box[T], left: RTree[V, T], right: RTree[V, T])
    extends RTree[V, T]
}

object data2D extends Data2D
```

```
import spire._, algebra._

trait Ops2D {
  import data2D._
  def initBox[T : Order](point1: Point[T], point2: Point[T]): Box[T]
  def expandBox[T : Order](box: Box[T], point: Point[T]): Box[T]
  def expandBox[T : Order](box1: Box[T], box2: Box[T]): Box[T]
  def withinBox[T : Order](box: Box[T], point: Point[T]): Boolean
  def overlaps[T : Order](box1: Box[T], box2: Box[T]): Boolean
  def add[V, T](rtree: RTree[V, T], entry: Entry[V, T]): RTree[V, T]
  def remove[V, T](rtree: RTree[V, T], entry: Entry[V, T]): RTree[V, T]
  def find[V, T](rtree: RTree[V, T], point: Point[T]): Option[Entry[V, T]]
  def contains[V, T](rtree: RTree[V, T], entry: Entry[V, T]): Boolean
  def search[V, T](space: Box[T]): List[Entry[V, T]]
}
```

what if we have more than 2 dimensions?

```
trait DataDynamicNDim {
  case class Point[T](terms: List[T])
  case class Interval[T](l: T, u: T)
  case class Entry[V, T](value: V, point: Point[T])
  case class Box[T](intervals: List[Interval[T]])
  sealed trait RTree[V, T]
  case class Empty[V, T]() extends RTree[V, T]
  case class Leaf[V, T](entry: Entry[V, T]) extends RTree[V, T]
  case class Node[V, T](
    box: Box[T], left: RTree[V, T], right: RTree[V, T])
    extends RTree[V, T]
}
```

potential runtime pitfalls

```
import spire.implicits._, spire.math.{ min, max }

object dynNDim extends DataDynamicNDim {
  def initBox[T : Order](p1: Point[T], p2: Point[T]) = Box(
    p1.terms.zip(p2.terms).map { case (i, j) => Interval(min(i, j), max(i, j)) })
  val lossy = initBox(Point(List(0, 0)), Point(List(2, 3, 5, 7)))
}
```

```
scala> dynNDim.lossy
res0: dynNDim.Box[Int] = Box(List(Interval(0,2), Interval(0,3)))
```

Traveling through hyperspace ain't like dusting crops, farm boy. Without precise calculations we could fly right through a star or bounce too close to a supernova, and that'd end your trip real quick, wouldn't it?

— Han Solo, Star Wars Episode IV: A New Hope

I hear type systems perform calculations

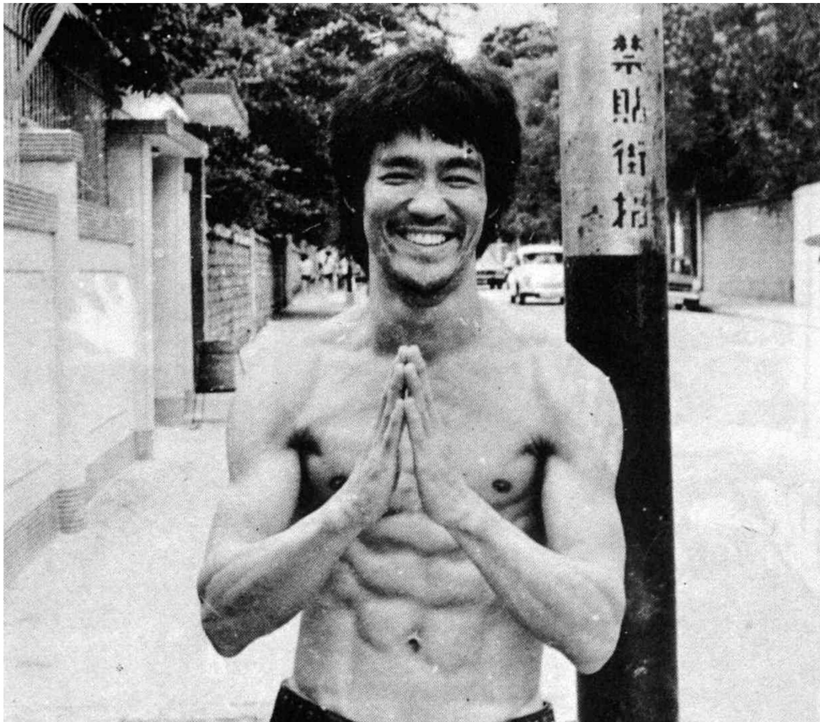
and support a class of constraints

there’s a library that explores this space

shapeless : supercharged generic coding

Empty your mind, be formless. Shapeless, like water.

— Bruce Lee



let’s build a well typed Generic N-Dim R-Tree

shapeless Sized

```
import shapeless._, ops.nat._

trait DataSizedNDim {
  case class Point[T, N <: Nat](terms: Sized[Seq[T], N])
  case class Entry[V, T, N <: Nat](value: V, point: Point[T, N])
  case class Interval[T](l: T, u: T)
  case class Box[T, N <: Nat](intervals: Sized[Seq[Interval[T]], N])
  sealed trait RTree[T, N <: Nat]
  case class Leaf[T, N <: Nat](point: Point[T, N]) extends RTree[T, N]
  case class Node[T, N <: Nat](
    bound: Box[T, N], left: RTree[T, N], right: RTree[T, N])
    extends RTree[T, N]
}
```

perhaps…

size constraint looking good

```
import shapeless.syntax.sized._, shapeless.test._

object sizedNDim extends DataSizedNDim {
  def initBox[T : Order, N <: Nat : ToInt](p1: Point[T, N], p2: Point[T, N]) =
    (p1.terms.unsized.zip(p2.terms.unsized).map {
      case (i, j) => Interval(min(i, j), max(i, j)) }).toList.ensureSized[N]
}
```

```
scala> illTyped("""sizedNDim.initBox(
     |   sizedNDim.Point(Sized(0, 0)), sizedNDim.Point(Sized(2, 3, 5, 7)))""")
```

what if we want heterogeneous types for our dimensions?

shapeless HList

```scala
trait DataHListNDim {
  case class Point[T <: HList](terms: T)
  case class Entry[V, T <: HList](value: V, point: Point[T])
  case class Box[T <: HList](lowerBounds: T, upperBounds: T) // inclusive
  sealed trait RTree[V, T <: HList]
  case class Empty[V, T <: HList]() extends RTree[V, T]
  case class Leaf[V, T <: HList](entry: Entry[V, T]) extends RTree[V, T]
  case class Node[V, T <: HList](
    box: Box[T], left: RTree[V, T], right: RTree[V, T])
    extends RTree[V, T]
}

object dataHListNDim extends DataHListNDim
```

looks promising

note: intentionally postponed a few items

- balancing
- splitting
- heterogeneous distance function (*similarity*)

```scala
import dataHListNDim._, shapeless.ops.hlist._

import spire.math.{ min, max }, spire.implicits.{eqOps => _, _}

trait OpsHListFunctions {
  object minimum extends Poly2 {
    implicit def default[T : Order] = at[T, T](implicitly[Order[T]].min)
  }
  object maximum extends Poly2 {
    implicit def default[T : Order] = at[T, T](implicitly[Order[T]].max)
  }
  object lte extends Poly2 {
    implicit def default[T : Order] = at[T, T](_ <= _)
  }
  object gte extends Poly2 {
    implicit def default[T : Order] = at[T, T](_ >= _)
  }
  object and extends Poly2 {
    implicit def caseBoolean = at[Boolean, Boolean](_ && _)
  }
}
```

```scala
trait OpsHListTypes { self: OpsHListFunctions =>
  type ZWMin[T <: HList] = ZipWith.Aux[T, T, minimum.type, T]
  type ZWMax[T <: HList] = ZipWith.Aux[T, T, maximum.type, T]
  type ZWLB[T <: HList] = {
    type λ[U <: HList] = ZipWith.Aux[T, T, lte.type, U]
  }
  type ZWUB[T <: HList] = {
    type λ[U <: HList] = ZipWith.Aux[T, T, gte.type, U]
  }
  type LFLB[T <: HList] = {
    type λ[U <: HList] =
      LeftFolder.Aux[
        ZipWith.Aux[T, T, lte.type, U]#Out,
        Boolean,
        and.type,
        Boolean]
  }
  type LFUB[T <: HList] = {
    type λ[U <: HList] =
      LeftFolder.Aux[
        ZipWith.Aux[T, T, gte.type, U]#Out,
        Boolean,
        and.type,
        Boolean]
  }
}
```

```
trait OpsHList extends OpsHListFunctions with OpsHListTypes {
  def initBox
    [T <: HList : ZWMin : ZWMax]
    (point1: Point[T], point2: Point[T])
    : Box[T] = Box(
    point1.terms.zipWith(point2.terms)(minimum),
    point1.terms.zipWith(point2.terms)(maximum))
  def withinBox
    [T <: HList, L <: HList : ZWLB[T]#λ : LFLB[T]#λ, U <: HList : ZWUB[T]#λ : LFUB[T]#
    (box: Box[T], point: Point[T])
    : Boolean =
    box.lowerBounds.zipWith(point.terms)(lte).foldLeft(true)(and) &&
    box.upperBounds.zipWith(point.terms)(gte).foldLeft(true)(and)
  def overlaps
    [T <: HList, L <: HList : ZWLB[T]#λ : LFLB[T]#λ, U <: HList : ZWUB[T]#λ : LFUB[T]#
    (box1: Box[T], box2: Box[T])
    : Boolean =
    box1.lowerBounds.zipWith(box2.upperBounds)(lte).foldLeft(true)(and) &&
    box1.upperBounds.zipWith(box2.lowerBounds)(gte).foldLeft(true)(and)
}
```

incomplete implementation for for sake of brevity

a quick look at distance

```
trait Distance[T <: HList] {
  def distance(a: Point[T], b: Point[T]): Number
}

trait PointOps {
  def distance[T <: HList : Distance](a: Point[T], b: Point[T]): Number =
    implicitly[Distance[T]].distance(a, b)
}
```

still with me? good, here's some of our work in action.

```
import ndimrtree._, NDimRTree._, NDimRTreeOps._
```

```
scala> illTyped("""
     | initBox(Point(1 :: HNil), Point(1 :: 2 :: HNil))
     | """)

scala> illTyped("""
     | initBox(Point(1 :: HNil), Point("a" :: HNil))
     | """)

scala> illTyped("""
     | expandBox(
     |  initBox(Point(1 :: HNil), Point(3 :: HNil)),
     |  initBox(Point("a" :: HNil), Point("z" :: HNil)))
     | """)

scala> RTree(List(Entry("z", Point(3 :: 1.7 :: HNil)))).find(
     |   Point(3 :: 1.7 :: HNil)).map(_.value)
res5: Option[String] = Some(z)

scala> illTyped("""
     | RTree(List(
     |   Entry("z", Point(3 :: 1.7 :: HNil)),
     |   Entry("ζ", Point(42 :: HNil))))
     | """)
```

initial benchmarking:

room for improvement. just a first naive pass. confident that reasonable performance is achievable.

fun things explored

- generic programming with shapeless
- leveraging the type system to guarantee certain invariants at compile time
- indexes
- R-Trees (but you already know that)

libraries utilized

- shapeless : Generic programming for Scala
- spire : Powerful new number types and numeric abstractions for Scala
- scalacheck : Property-based testing for Scala
- scalacheck-shapeless : Generation of arbitrary case classes / ADTs with scalacheck and shapeless
- archery : 2D R-Tree implementation in Scala
  - leveraged for scalacheck tests and benchmarking
- thyme - microbenchmark utility for Scala
- tut - doc/tutorial generator for scala
  - sent the code used in this presentation to scalac

future explorations

- heterogenous distance functions
- visualization with d3, perhaps leverage scala-js
- R-Tree variants, e.g. M-Tree, X-Tree, Hilbert R-tree
- distributed implementation
- utilize similar techniques for other data structures

Thanks!

repo: http://github.com/drostron/ndim-rtree

twitter: @yastero