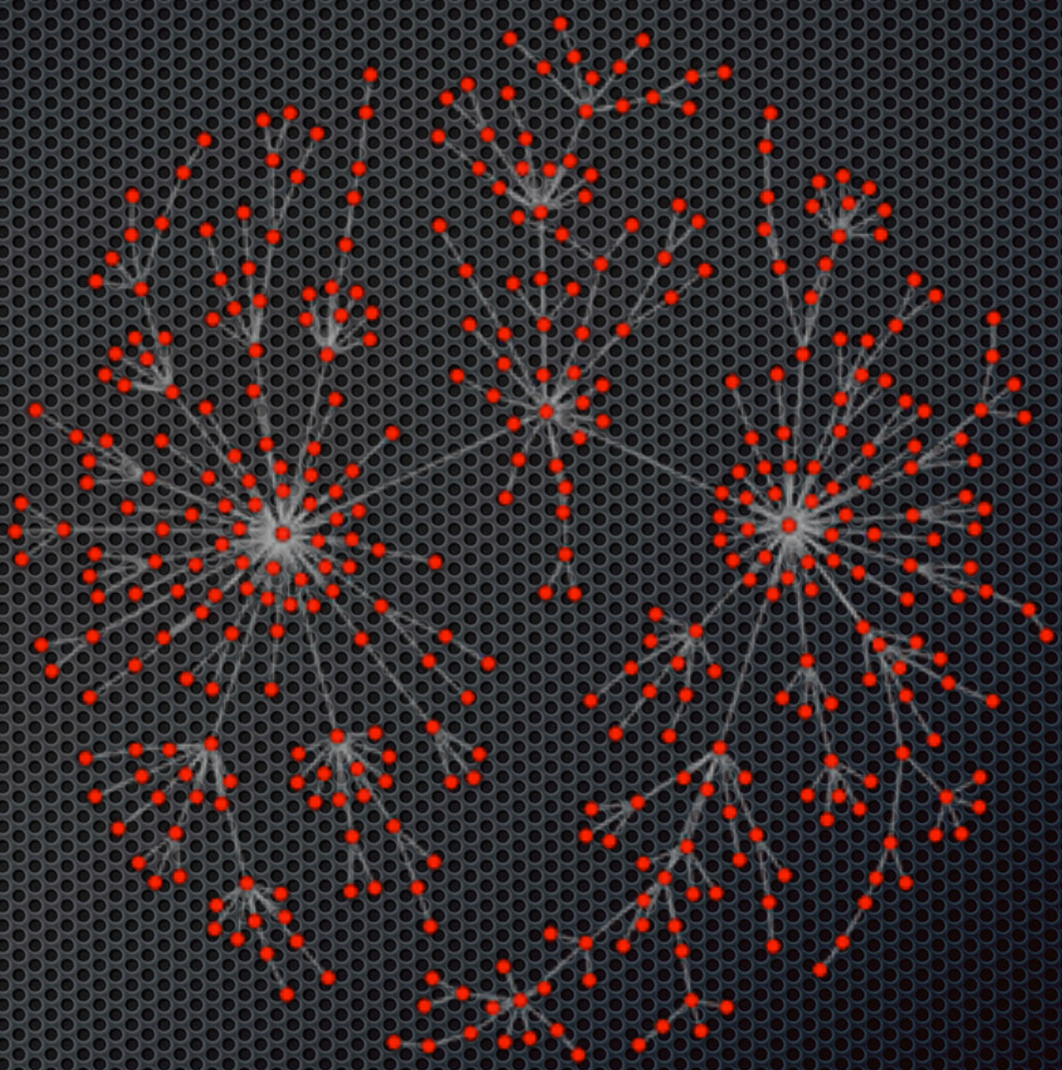
# Graph4Scala

brad miller



# Intro

- Library home page:  www.scala-graph.org
- My homepage: www.bradfordmiller.org
- My email: bfm@bradfordmiller.org
- I’m looking for cool Scala side projects to hack on
- About me
- Thanks - LeadiD and Chariot Solutions
- Shout-outs - IMS Health and Walnut Street Labs

# Agenda

- Survey of the API
- My use case
- Q & A
- Feel free to interrupt

# Why use Scala4Graph?

- Simplicity - Intuitive API for creating, manipulating, and querying a graph
- Consistency - consistent state of nodes and edges, duplicate prevention, smart add/remove
- Conformity - same look and feel as members of Scala collection framework
- Flexibility - Mixed Graphs, Multi-Graphs, HyperGraphs
- Functional/Extendable - Concise, functional syntax and easily customizable

# sbt and imports

- libraryDependencies ++= Seq("com.assembla.scala-incubator" % "graph-core_2.10" % "1.7.3")
- import scalax.collection.Graph // or scalax.collection.mutable.graph
- import scalax.collection.GraphPredef._

# Graph Types

| Graph Type | Definition | Edge Type |
|---|---|---|
| Simple | only undirected edges w/o multi-edges | UnDiEdge |
| Mixed | directed and undirected edges w/o multi-edges | UnDiEdge/DiEdge |
| Weighted | edges w/ the special weight label | WUndiEdge/WDiEdge |
| Multi | one or more edge types w/ multi-edges | Any pre-defined Edge classes containing K in prefix |

# Edge Factory Examples

| Shortcut | Named Factory | Meaning |
|---|---|---|
| 1 ~ 2 ~3 | HyperEdge(1, 2, 3) | Undirected hyperedge btw 1, 2, and 3 |
| 1 ~> 2 ~>3 | DiHyperEdge(1, 2, 3) | directed hyperedge from 1 via 2 to 3 |
| “A” ~ “B” | UnDiEdge(“A”, “B”) | undirected edge between A and B |
| “A” ~> “B” | DiEdge(“A”, “B”) | Any pre-defined Edge classes containing K in |
| 1 ~ 2 % 5 | WUnDiEdge(1, 2, 5) | undirected edge btw 1 and 2 w/ a weight of 5 |
| 1 ~> 2 % 0 | WDiEdge(1, 2, 0) | directed edge from 1 to 2 w/ a weight of 0 |

# Instantiating Graphs

- val g1 = Graph(3~1, 5)
- val g2 = Graph(UnDiEdge(3, 1), 5)
- val gA = Graph(3~>1.2)
- val h = Graph(1~1, 1~2~3)

```
//Graph[Int, UnDiEdge](1, 3, 5, 3 ~ 1)
//same as above

//Graph[AnyVal, DiEdge](3, 1.2, 3~>1.2)

//Graph[Int, HyperEdge](1, 2, 3, 1~1, 1~2~3)
```

# Live Code

- val nodes = List(5)
  val edges = List(3~1)
  val g3 = Graph.from(nodes, edges)
- var n, m = 0;  val f = Graph.fill(100)({n = m; m+=1; n~m})

# Type Parameter Inference

- val g = Graph()  //Graph[Nothing, Nothing]
- val g = Graph(1)  //Graph[Int, Nothing]
- var g = Graph(1~2)  //Graph[Int,Nothing]
  g += 1.2  //compiler error
- Graph(1~>2) + (2~3)  //compiler error

# Inner and Outer Objects

- val g = Graph(1 ~ 2)
- //Graph[Int, UnDiEdge](1, 2, 1~2)

Int and UniDiEdge are the **types** of outer nodes and outer edges of g

- val n1 = g.nodes.head
- //g.NodeT = 1 or 2

The type of the inner node is g.NodeT. Head does not guarantee order

- val e1 = g.edges.head
- //g.EdgeT = 1~2

Similarly, the inner edge e1 is of the type g.EdgeT

- e1._1
- //g.NodeT = 1

First incident node with e1 is an inner node of the type g.NodeT

# Looking up Nodes and Edges

- val g = Graph(1~2)
- g find 1
- //Option[g.NodeT] = Some(1)
- g find 3
- //Option[g.NodeT] = None
- g get 1
- //g.NodeT = 1
- g get 3
- //NoSuchElementException
- g find 1~2
- //Option[g.EdgeT] = Some(1~2)
- g addAndGet 5
- //g.NodeT = 5

# Addition of Nodes

- val g = Graph(1, 2~3)
- g + 1
- g + 0
- g + 0~1
- g ++ List(1~2, 2~3)

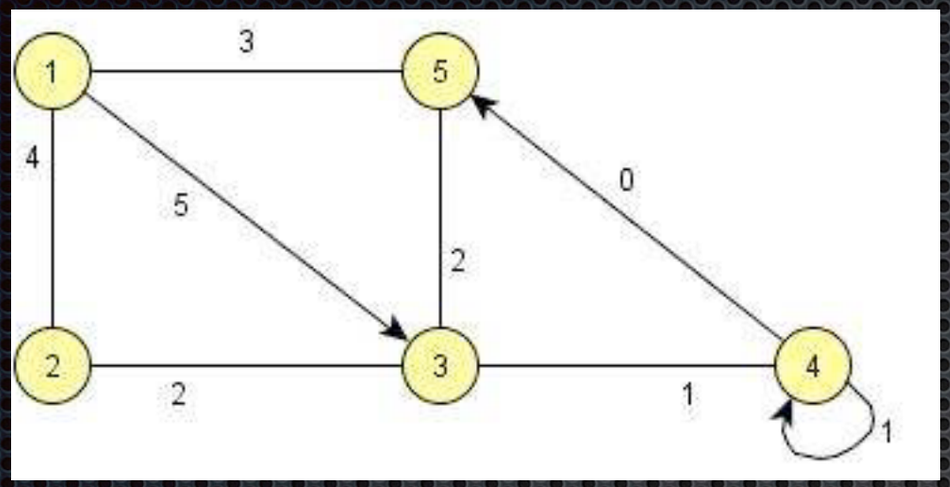
- //Graph(1, 2, 3, 2~3)
- //==g (1 already exists)
- //Graph(0,1,2,3, 2~3)
- //Graph(0, 1, 2, 3, 0~1, 2~3)
- //Graph(1, 2, 3, 1~2, 2~3)

Subtraction Ops are similar

# Unions, Diffs, Intersections

- val g = Graph(1~2, 2~3, 2~4, 3~5, 4~5)
- //Graph(1, 2, 3, 4, 5, 1~2, 2~3, 2~4, 3~5, 4~5)
- val h = Graph(3~4, 3~5, 4~6, 5~6)
- //Graph(3, 4, 5, 6, 3~4, 3~5, 4~6, 5~6)
- g union h
- //Graph(1, 2, 3, 4, 5, 6 1~2, 2~3, 2~4, 3~5, 4~5, 3~4, 4~6, 5~6)
- g diff h
- //Graph(1, 2, 1~2)
- g intersect h
- Graph(3, 4, 5, 3~5)

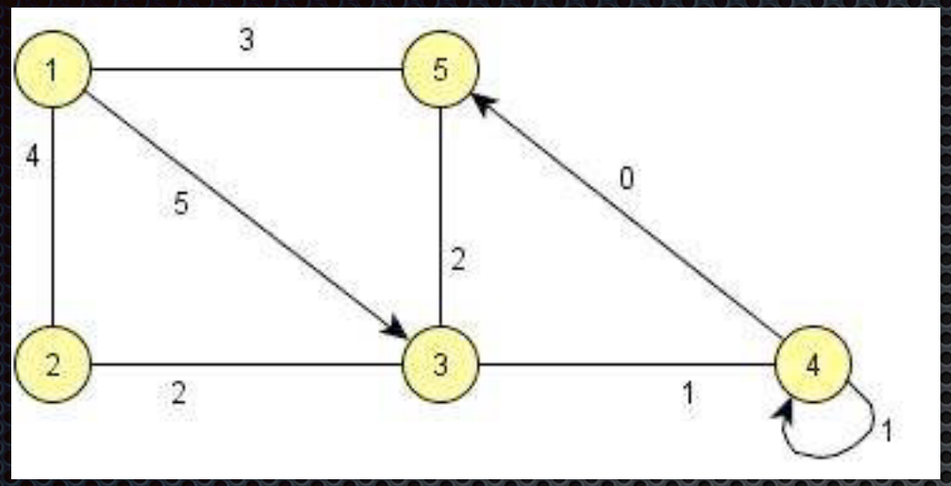
# Finding Paths




```
import scalax.collection.edge.WDiEdge
import scalax.collection.edge.Implicits._
val g = Graph( 1~2 % 4, 2~3 % 2, 1~>3 % 5,
1~5 % 3, 3~5 % 2, 3~4 % 1, 4~>4 % 1, 4~>5
% 0)
def n(outer: Int) = g get outer
```

```
n(1) findSuccessor (_.outDegree > 3)                  // Option[g.NodeT] = None
n(1) findSuccessor (_.outDegree >= 3)                 // Option[g.NodeT] = Some(3)
n(4) findSuccessor (_.edges forall (_.undirected))    // Option[g.NodeT] = Some(2)
n(4) isPredecessorOf n(1)                             // true
n(1) pathTo n(4)                                      // Some(Path(1, 1~>3 %5, 3, 3~4 %1, 4))
n(1) pathUntil (_.outDegree >= 3)                     // Some(Path(1, 1~>3 %5))
n(3) shortestPathTo n(1)                            //Some(Path(3, 3~4 %1, 4, 4~>5 %0, 5, 1~5 %3, 1))
```

# General Useful Metadata




val g = Graph( 1~2 % 4, 2~3 % 2, 1~>3 % 5, 1~5 % 3, 3~5 % 2, 3~4 % 1, 4~>4 % 1, 4~>5 % 0)

a) g.order // Int = 5 (Number of Nodes)
b) g.graphSize // Int = 8 (Number of Edges)
c) g.size // Int = 13 (Nodes + Edges)
d) g.totalDegree // Int = 16

# Classification/Finding Cycles

- val g = Graph(1~>2, 1~>3, 2~>3, 3~>4, 4~>2)
- g findCycle
- g isCyclic
- //Graph(1, 2, 3, 4, 1~>2, 1~>3, 2~>3, 3~>4, 4~>2)
- //Some(Cycle(2, 2~>3, 3, 3~>4, 4, 4~>2, 2))
- //true

- val g = Graph(1, 2~>3)
- (g + 2~>1).isConnected // true
- g.isConnected // false

# Other things to check out

- Serialize Graphs to JSON
- http://www.scala-graph.org/guides/json.html
- Translate Graphs to the DOT Language
- http://www.scala-graph.org/guides/dot.html
- Scala-graph Github
- https://github.com/scala-graph/scala-graph

# My use case for scala-graph

- Extract complete schema from MongoDB Collection
- Field names, BSON Types, 1-1 and 1-M relationships
- Recursively deconstruct sub-documents into their BSON types.
- Extract arrays as 1 - M relationships with their parent document

# Algorithm

- Perform map reduce operation on the collection which grabs field keys and types
- Store extracted types as nodes in a graph
- Any fields discovered of type Object, recursively decompose into BSON types using MR until all objects have been broken down
- Use directed graph to distinguish establish parent-child relationship between the nodes

# Example

```
case class BSONField(
  parentName: String = "",
  field: String = "",
  isArray: Boolean = false,
  bsonType: Byte = 0,
  id: UUID = UUID.randomUUID()
  )
```

```
val g = Graph[BSONField, DiGraph]()
```

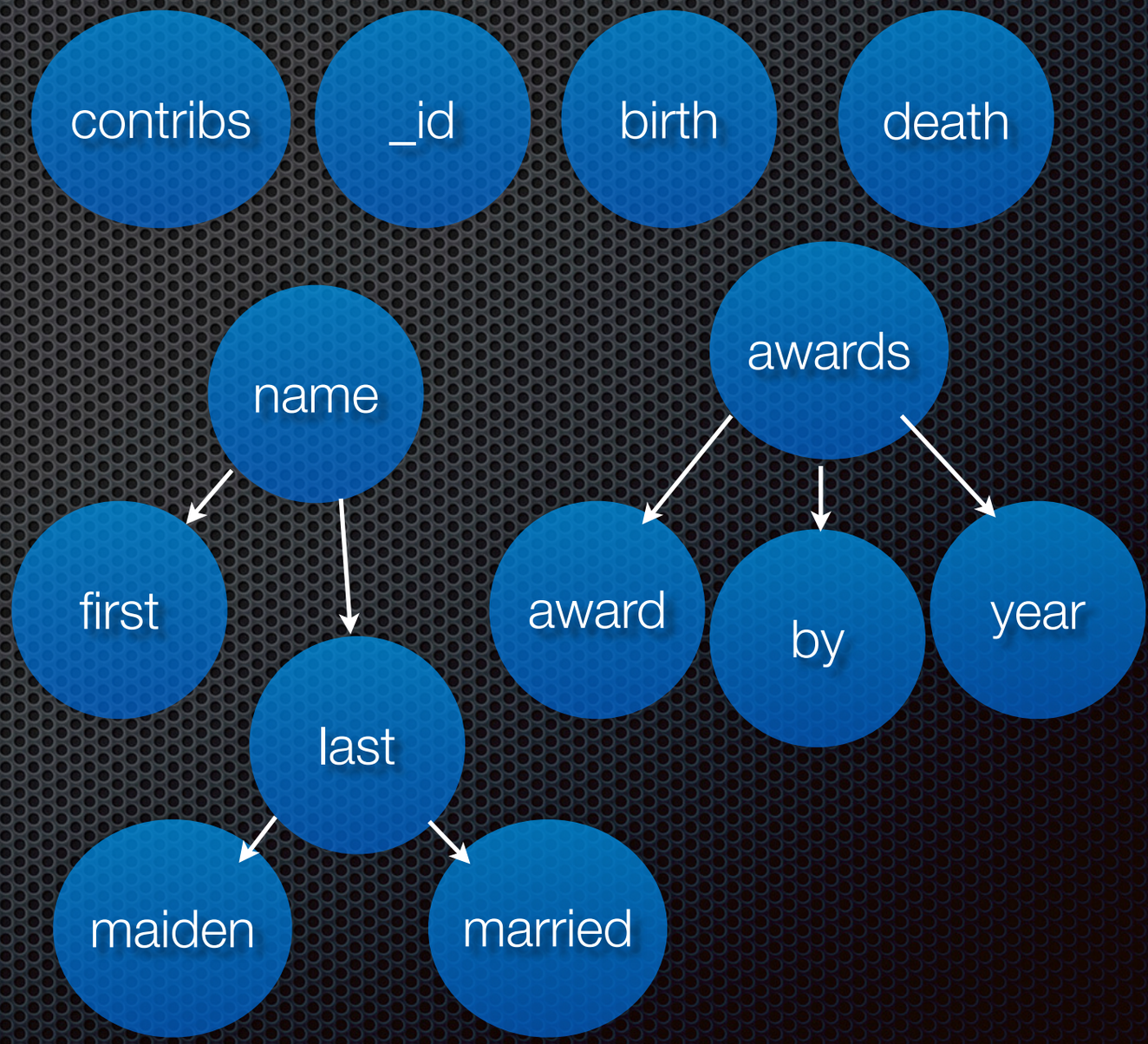
## Sample Document:

```
{
"_id" : 20,
"name" : { "first" : "Sara", "last" : { "maiden" : "Ritchie", "married" : "Scritchie" } },
"birth" : ISODate("1989-09-09T04:00:00Z"),
"death" : ISODate("2022-10-12T04:00:00Z"),
"contribs" : [ "HVAC", "Happiness" ],
"awards" : [
{ "award" : "Turing Award", "year" : 1983, "by" : "ACM" },
{ "award" : "National Medal of Technology", "year" : 1998, "by" : "United States" },
{ "award" : "Japan Prize", "year" : 2011, "by" : "The Japan Prize Foundation" } ]
}
```

# Visualizing the process


```
{
"_id" : 20,
"name" : { "first" : "Sara", "last" : { "maiden" :
"Ritchie", "married" : "Scritchie" } },
"birth" : ISODate("1989-09-09T04:00:00Z"),
"death" : ISODate("2022-10-12T04:00:00Z"),
"contribs" : [ "HVAC", "Happiness" ],
"awards" : [
{ "award" : "Turing Award",
"year" : 1983, "by" : "ACM" },
{ "award" : "National Medal of Technology",
"year" : 1998, "by" : "United States" },
{ "award" : "Japan Prize", "year" : 2011,
"by" : "The Japan Prize Foundation" } ]
}
```


1st MR =>2nd MR=>3rd MR




**Note: “Edgeless” nodes can be accessed by filtering the node list: g.nodes.filter(n => n.isIsolated)**

- Starting point for unknown schema of Mongo collection
- Could be used to see if data is being loaded correctly (IE, should an array have two different data types stored in it)
- My use case: Graph was first stage in extracting insight from mongo collection using only meta data. Ultimate goal, sync'ing the data in an RDBMS system.

# Questions?