Speeding up distributed graph analysis with probabilistic sketches

Odnoklassniki
Social Network

2019

Dmitry Bugaychenko

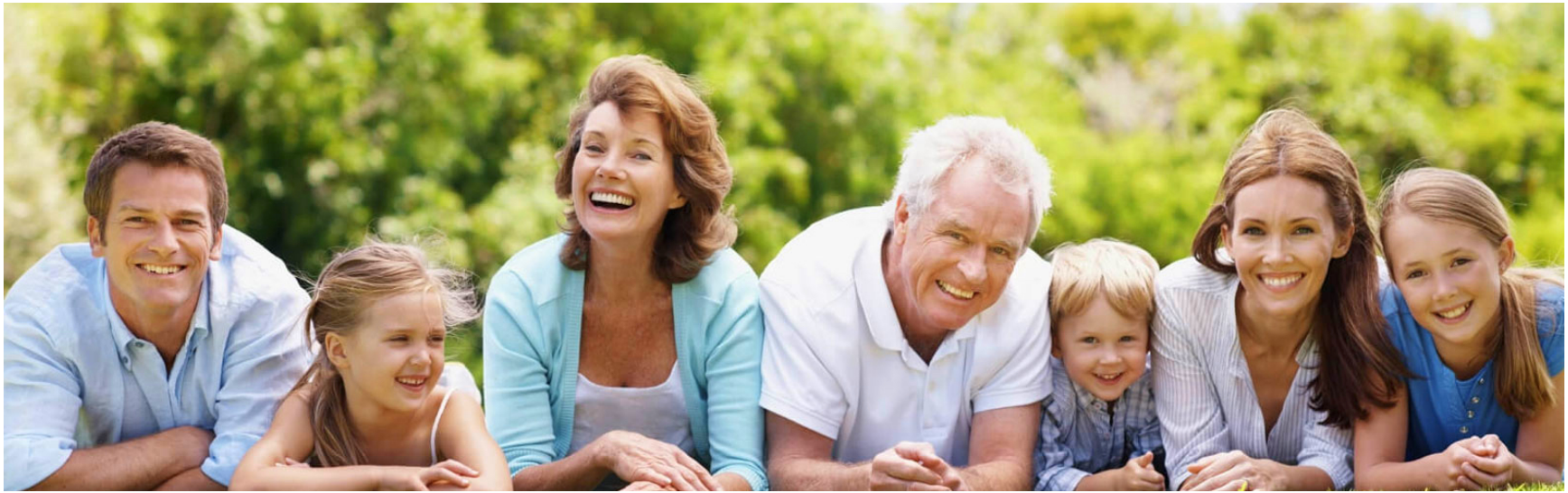
@mail.ru group

OK is...

44.7% of Russian people use OK*

* Mediascope, Monthly Reach, Desktop&Mobile, Russia 100k+, February 2019, 12-64 years

OK is...

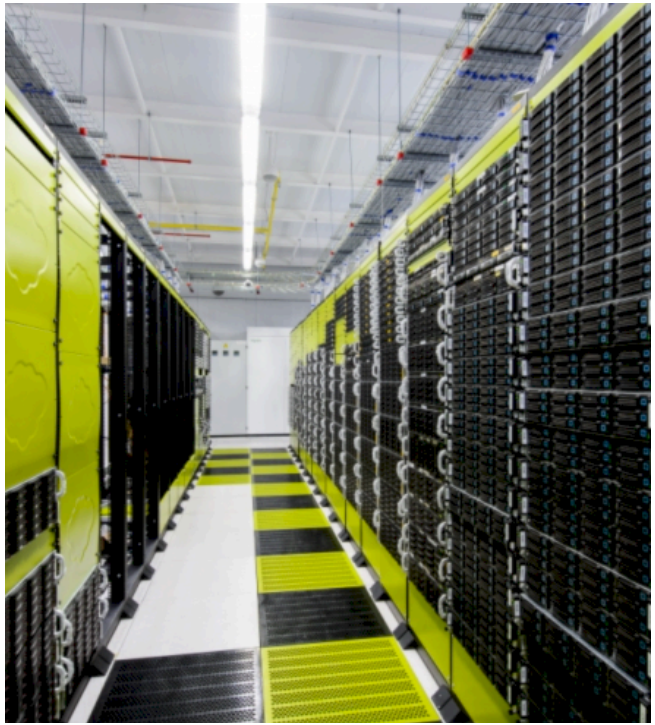


800 000 000+ family connections

OK is...

The place to share true feelings

OK is...

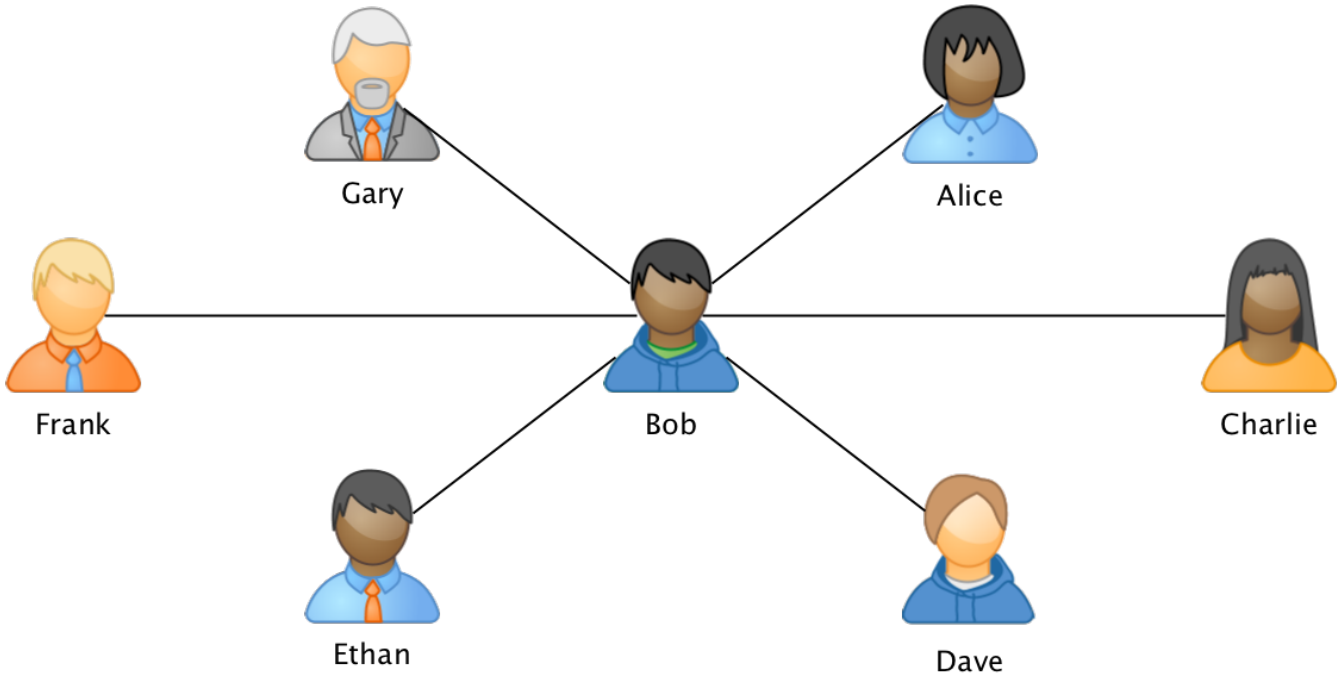


- 10000+ servers around the globe
- 1.8+Tb/s of outgoing trafic
- 400+ software components
- High Load, Big Data, Fault Tolerance…

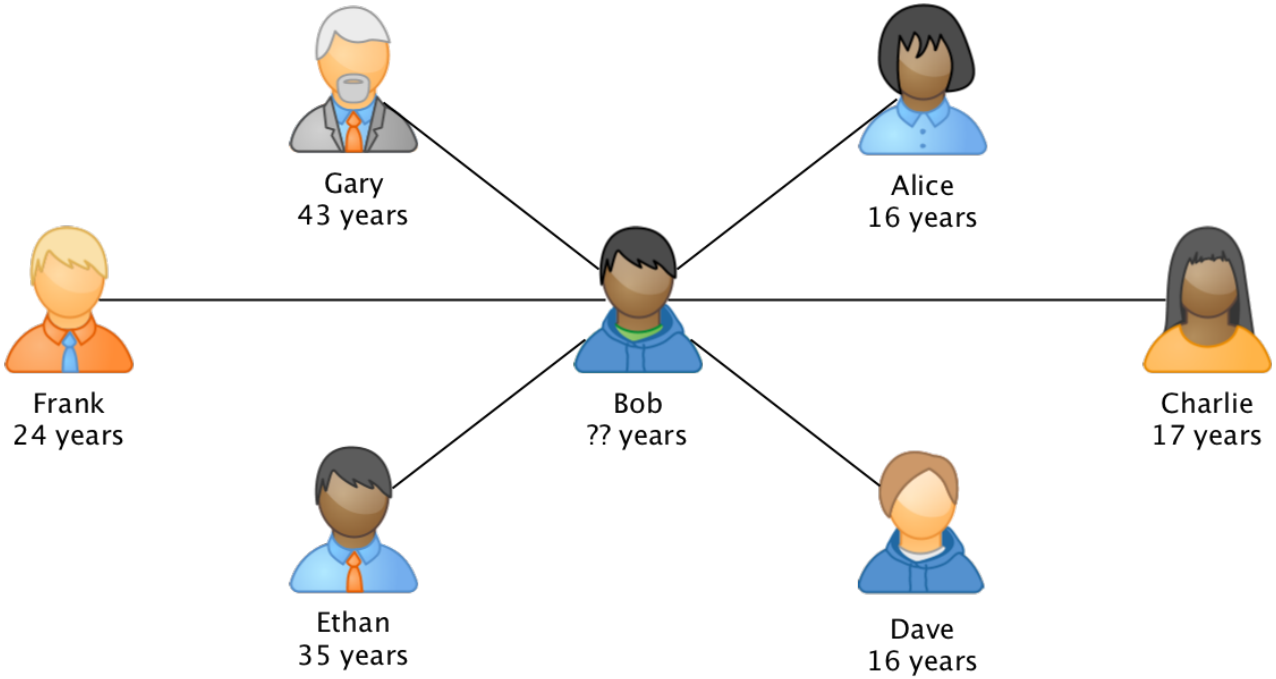
OK is...

- 3 Hadoop clusters
- 60+ petabytes of data (+20Tb daily)
- 10000+ cores
- 80+ TB RAM
- 400+ daily jobs

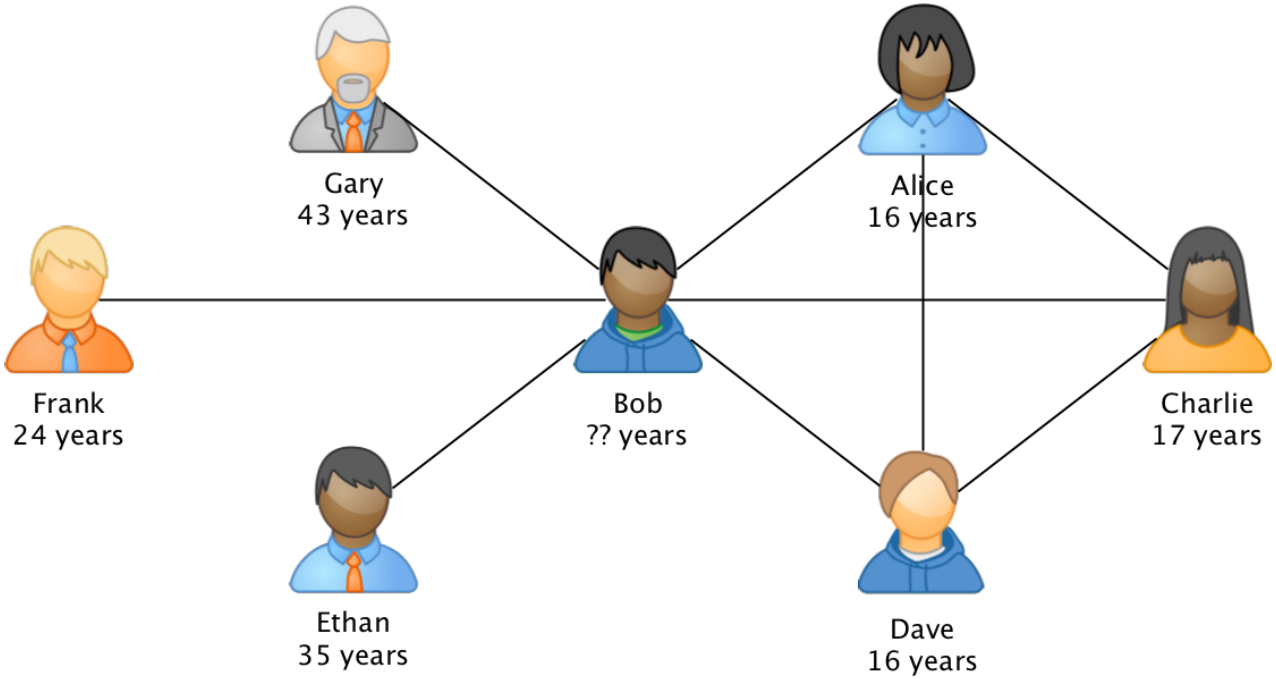
You can judge of a person by his friends



You can judge of a person by his friends



You can judge of a person by his ~~friends~~ ego-subgraph



More questions to answer

- What are Bob’s tastes in music/video/books/politics and etc.?
- Is Bob a human being?
- Whom else Bob might be friend with?
- …

The “hard math”

- For a user with 150 friends we can get up to 11 400 links in the ego-subgraph
  - 2 orders of magnitude rise

The “hard math”

- For a user with 150 friends we can get up to 11 400 links in the ego-subgraph
  - 2 orders of magnitude rise
- For a user with 5000 friends we can get up to 12 505 000 links in the ego-subgraph
  - 4 orders of magnitude rise

The “hard tech”

- Graph is partitioned by user ID and distributed

The “hard tech”

- Graph is partitioned by user ID and distributed
- In order to get information for a single user ego-subgraph we need to fetch data from many partitions

The “hard tech”

- Graph is partitioned by user ID and distributed
- In order to get information for a single user ego-subgraph we need to fetch data from many partitions
  - In most cases from ALL partitions

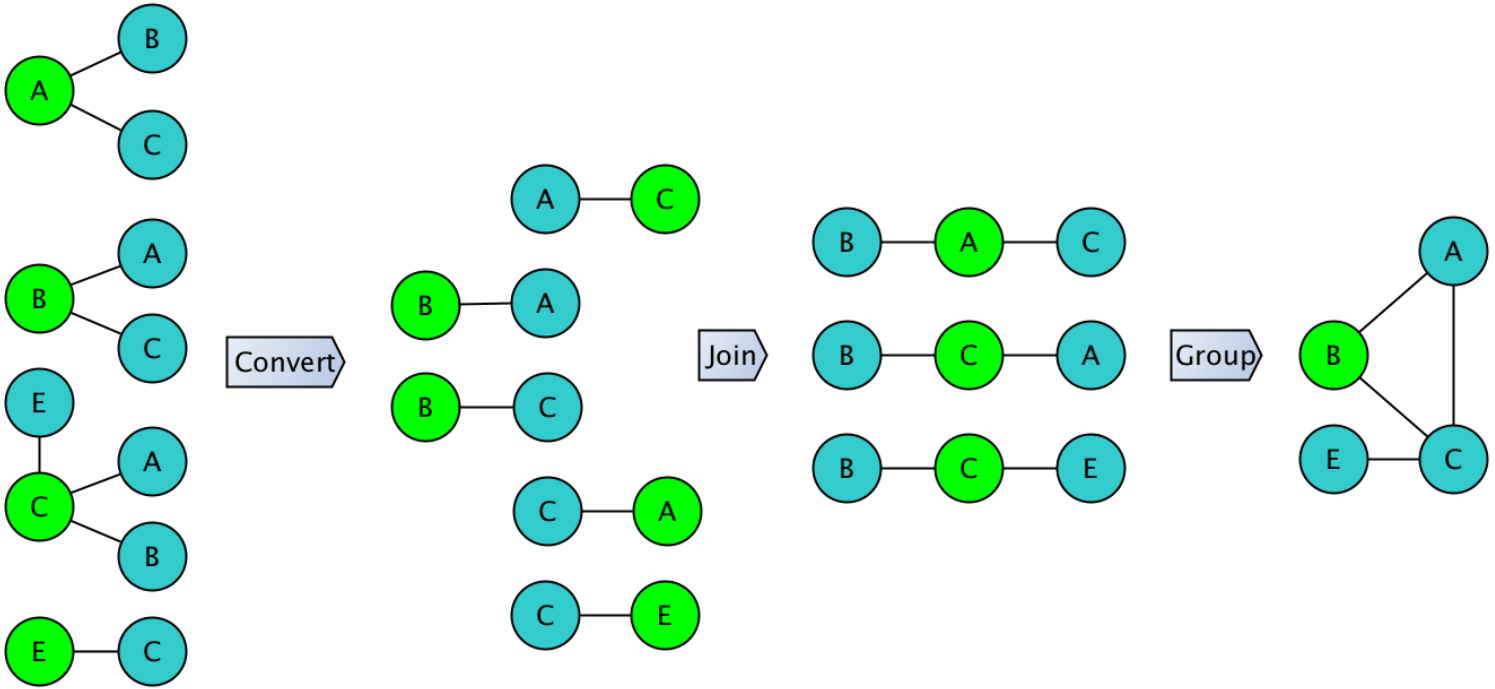
The “hard tech”

- Graph is partitioned by user ID and distributed
- In order to get information for a single user ego-subgraph we need to fetch data from many partitions
  - In most cases from ALL partitions
- High network utilization and long running time for analytics

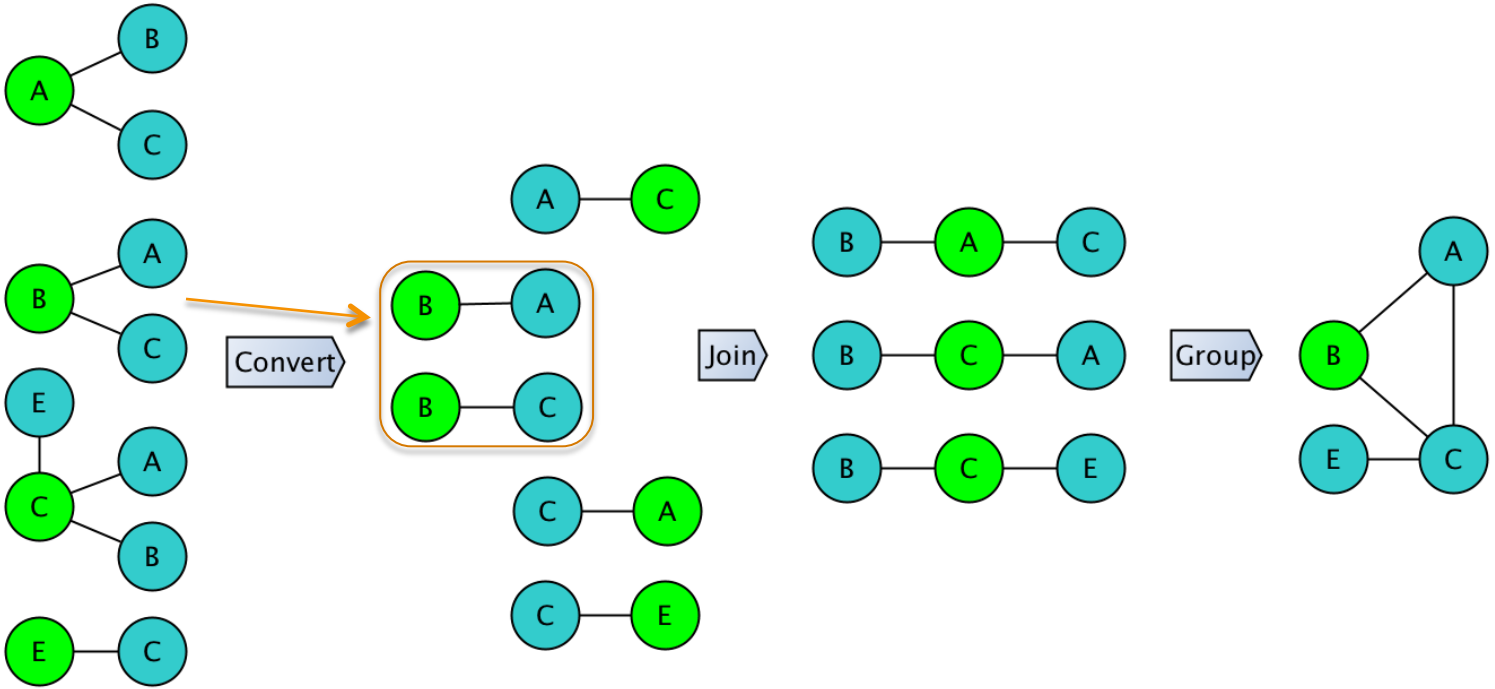
The “hard tech”

- Graph is partitioned by user ID and distributed
- In order to get information for a single user ego-subgraph we need to fetch data from many partitions
  - In most cases from ALL partitions
- High network utilization and long running time for analytics
- High latency with unacceptable extremes in run-time

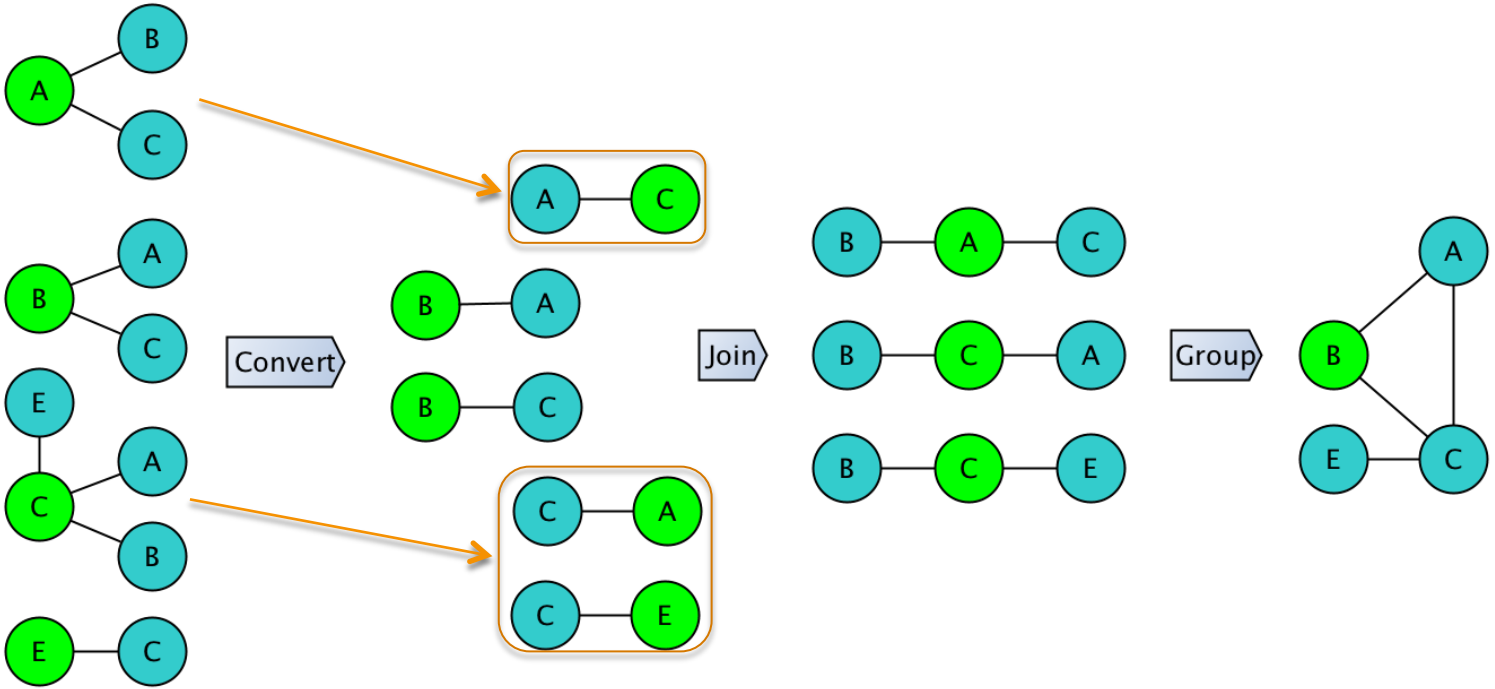
Collecting ego-subgraph: the straight way



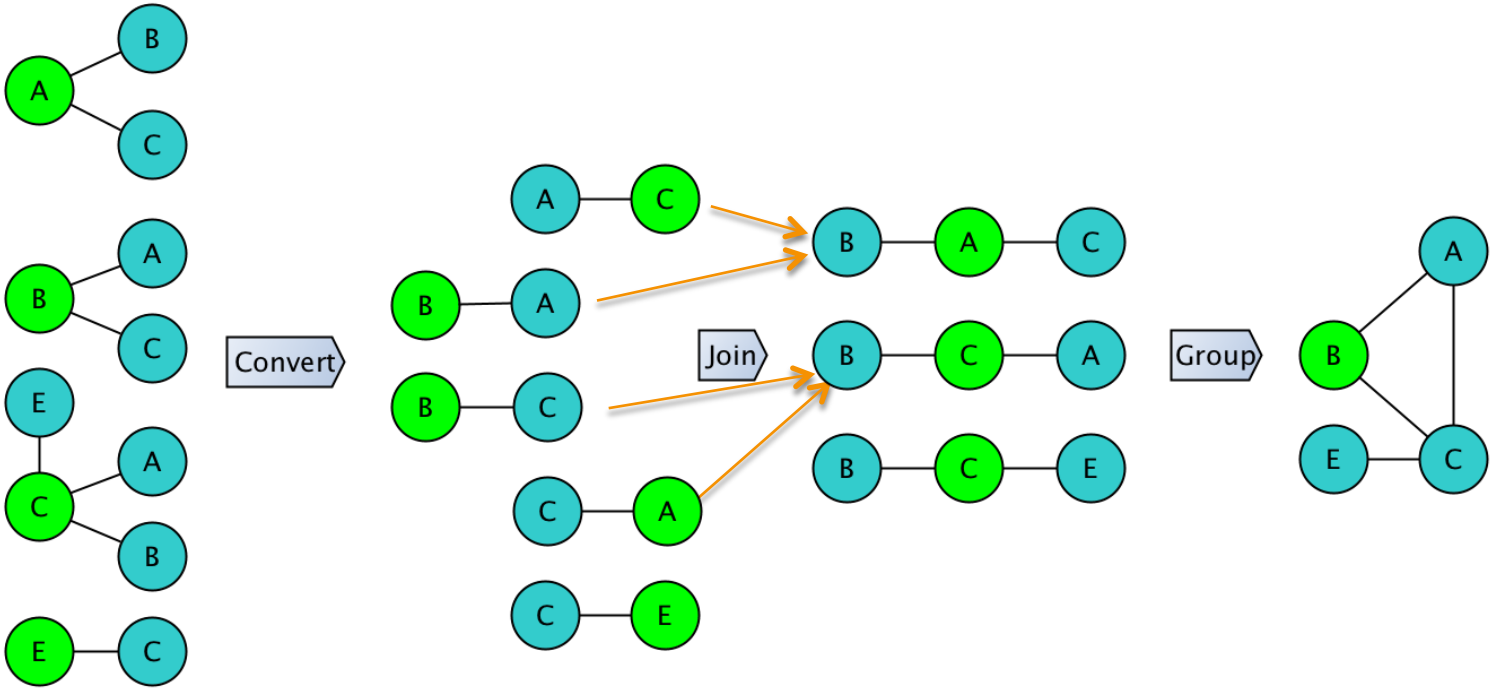
Collecting ego-subgraph: the straight way



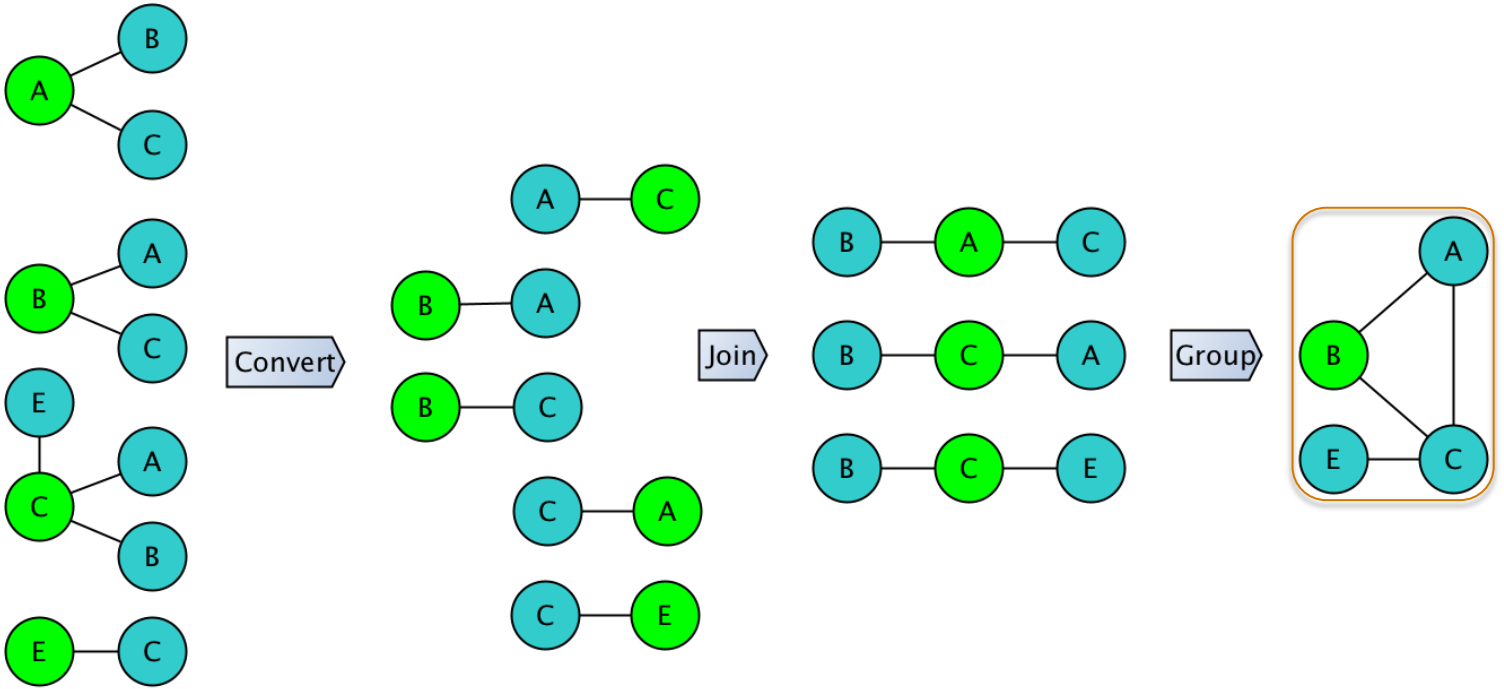
Collecting ego-subgraph: the straight way



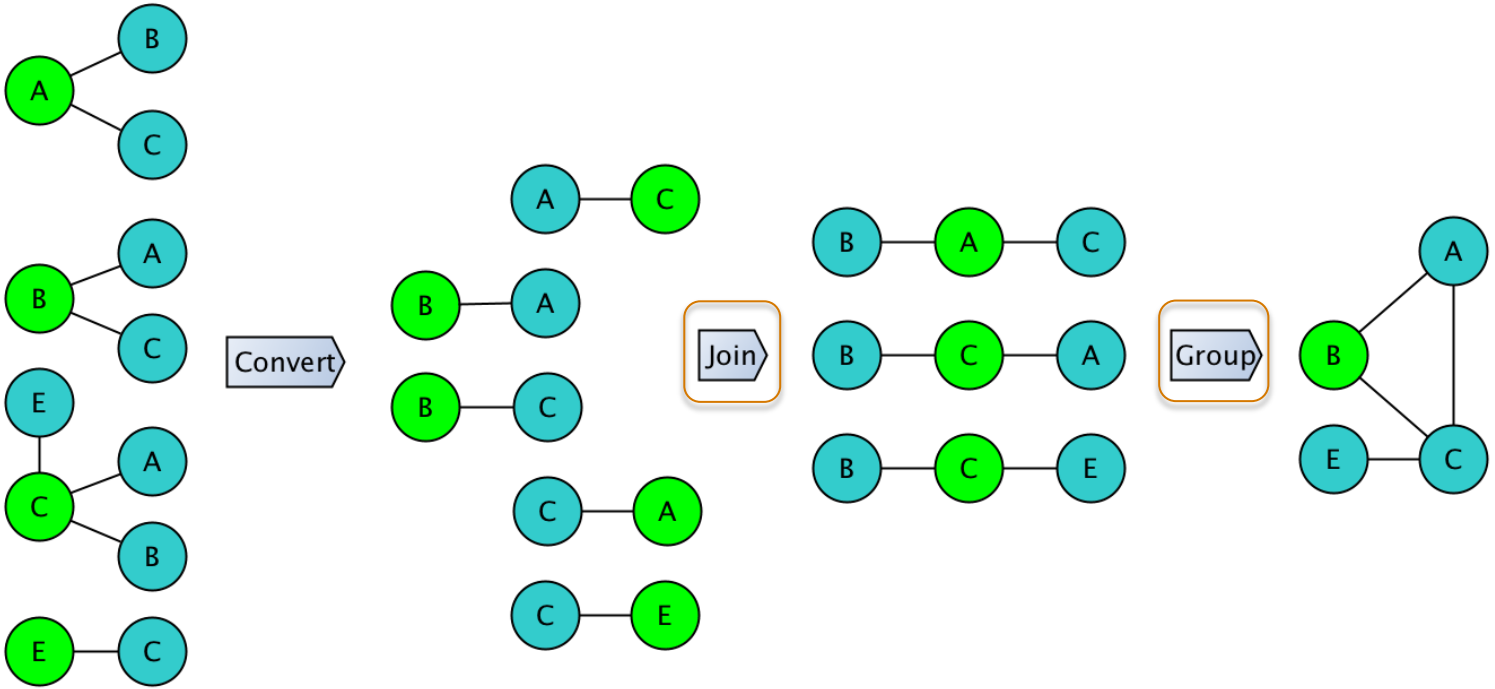
Collecting ego-subgraph: the straight way



Collecting ego-subgraph: the straight way



Collecting ego-subgraph: the straight way



Lets try: reading the graph

```
val graph = sqlContext.read.parquet(
                        s"/graph/friendships/date=2019-06-04")
graph.printSchema
root
 |-- userId: long (nullable = true)
 |-- friendships: array (nullable = true)
 |    |-- element: struct (containsNull = true)
 |    |    |-- friendId: long (nullable = true)
 |    |    |-- types: long (nullable = true)
```

Lets try: reading the graph

```
val graph = sqlContext.read.parquet(
                        s"/graph/friendships/date=2019-06-04")
graph.printSchema
root
 |-- userId: long (nullable = true)
 |-- friendships: array (nullable = true)
 |    |-- element: struct (containsNull = true)
 |    |    |-- friendId: long (nullable = true)
 |    |    |-- types: long (nullable = true)
```

Lets try: reading the graph

```
val graph = sqlContext.read.parquet(
                        s"/graph/friendships/date=2019-06-04")
graph.printSchema
root
 |-- userId: long (nullable = true)
 |-- friendships: array (nullable = true)
 |    |-- element: struct (containsNull = true)
 |    |    |-- friendId: long (nullable = true)
 |    |    |-- types: long (nullable = true)
```

Lets try: joining

```
val links = graph.select($"userId", explode($"friendships.friendId").as("friendId"))
val egoGraph = links
   .join(links.withColumnRenamed("userId", "nextFriendId"), Seq("friendId"))
   .groupBy("userId").agg(collect_list(struct($"friendId",$"nextFriendId")).as("graph"))
egoGraph.printSchema
```

```
root
 |-- userId: long (nullable = true)
 |-- graph: array (nullable = true)
 |    |-- element: struct (containsNull = true)
 |    |    |-- friendId: long (nullable = true)
 |    |    |-- nextFriendId: long (nullable = true)
```

Lets try: joining

```
val links = graph.select($"userId", explode($"friendships.friendId").as("friendId"))
val egoGraph = links
   .join(links.withColumnRenamed("userId", "nextFriendId"), Seq("friendId"))
   .groupBy("userId").agg(collect_list(struct($"friendId",$"nextFriendId")).as("graph"))
egoGraph.printSchema
```

```
root
 |-- userId: long (nullable = true)
 |-- graph: array (nullable = true)
 |    |-- element: struct (containsNull = true)
 |    |    |-- friendId: long (nullable = true)
 |    |    |-- nextFriendId: long (nullable = true)
```

Lets try: joining

```
val links = graph.select($"userId", explode($"friendships.friendId").as("friendId"))
val egoGraph = links
   .join(links.withColumnRenamed("userId", "nextFriendId"), Seq("friendId"))
   .groupBy("userId").agg(collect_list(struct($"friendId",$"nextFriendId")).as("graph"))
egoGraph.printSchema
```

```
root
 |-- userId: long (nullable = true)
 |-- graph: array (nullable = true)
 |    |-- element: struct (containsNull = true)
 |    |    |-- friendId: long (nullable = true)
 |    |    |-- nextFriendId: long (nullable = true)
```

Lets try: joining

```
val links = graph.select($"userId", explode($"friendships.friendId").as("friendId"))
val egoGraph = links
   .join(links.withColumnRenamed("userId", "nextFriendId"), Seq("friendId"))
   .groupBy("userId").agg(collect_list(struct($"friendId",$"nextFriendId")).as("graph"))
egoGraph.printSchema
```

```
root
 |-- userId: long (nullable = true)
 |-- graph: array (nullable = true)
 |    |-- element: struct (containsNull = true)
 |    |    |-- friendId: long (nullable = true)
 |    |    |-- nextFriendId: long (nullable = true)
```

Lets try: joining

```
val links = graph.select($"userId", explode($"friendships.friendId").as("friendId"))
val egoGraph = links
   .join(links.withColumnRenamed("userId", "nextFriendId"), Seq("friendId"))
   .groupBy("userId").agg(collect_list(struct($"friendId",$"nextFriendId")).as("graph"))
egoGraph.printSchema
```

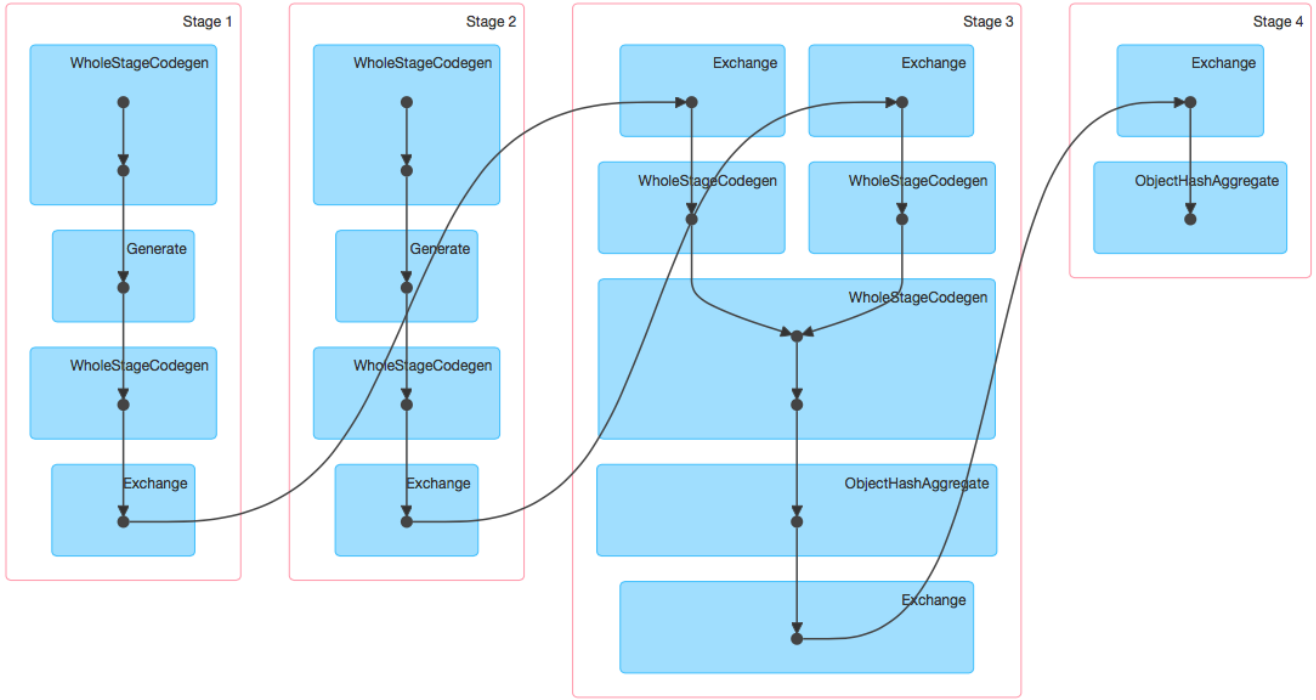
```
root
 |-- userId: long (nullable = true)
 |-- graph: array (nullable = true)
 |    |-- element: struct (containsNull = true)
 |    |    |-- friendId: long (nullable = true)
 |    |    |-- nextFriendId: long (nullable = true)
```

Lets try: joining

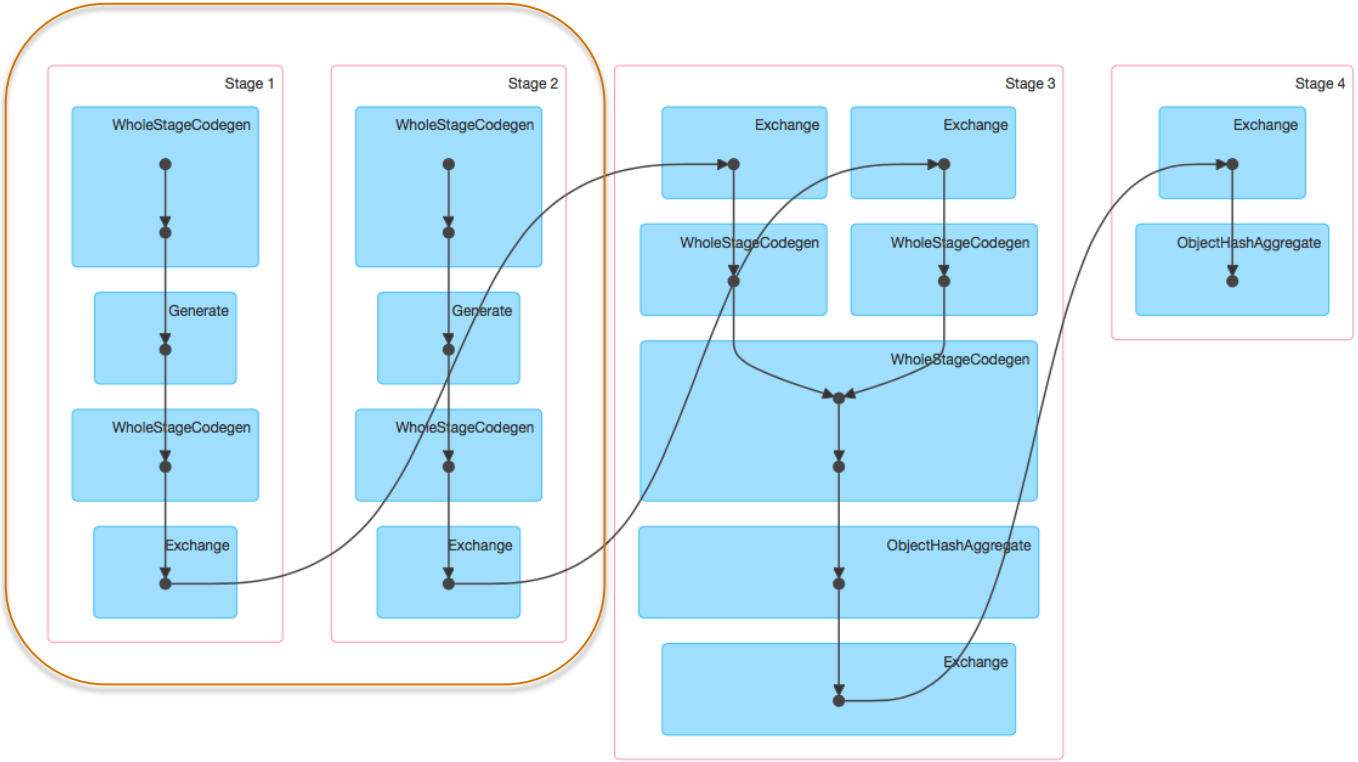
```
val links = graph.select($"userId", explode($"friendships.friendId").as("friendId"))
val egoGraph = links
   .join(links.withColumnRenamed("userId", "nextFriendId"), Seq("friendId"))
   .groupBy("userId").agg(collect_list(struct($"friendId",$"nextFriendId")).as("graph"))
egoGraph.printSchema
```

```
root
 |-- userId: long (nullable = true)
 |-- graph: array (nullable = true)
 |    |-- element: struct (containsNull = true)
 |    |    |-- friendId: long (nullable = true)
 |    |    |-- nextFriendId: long (nullable = true)
```

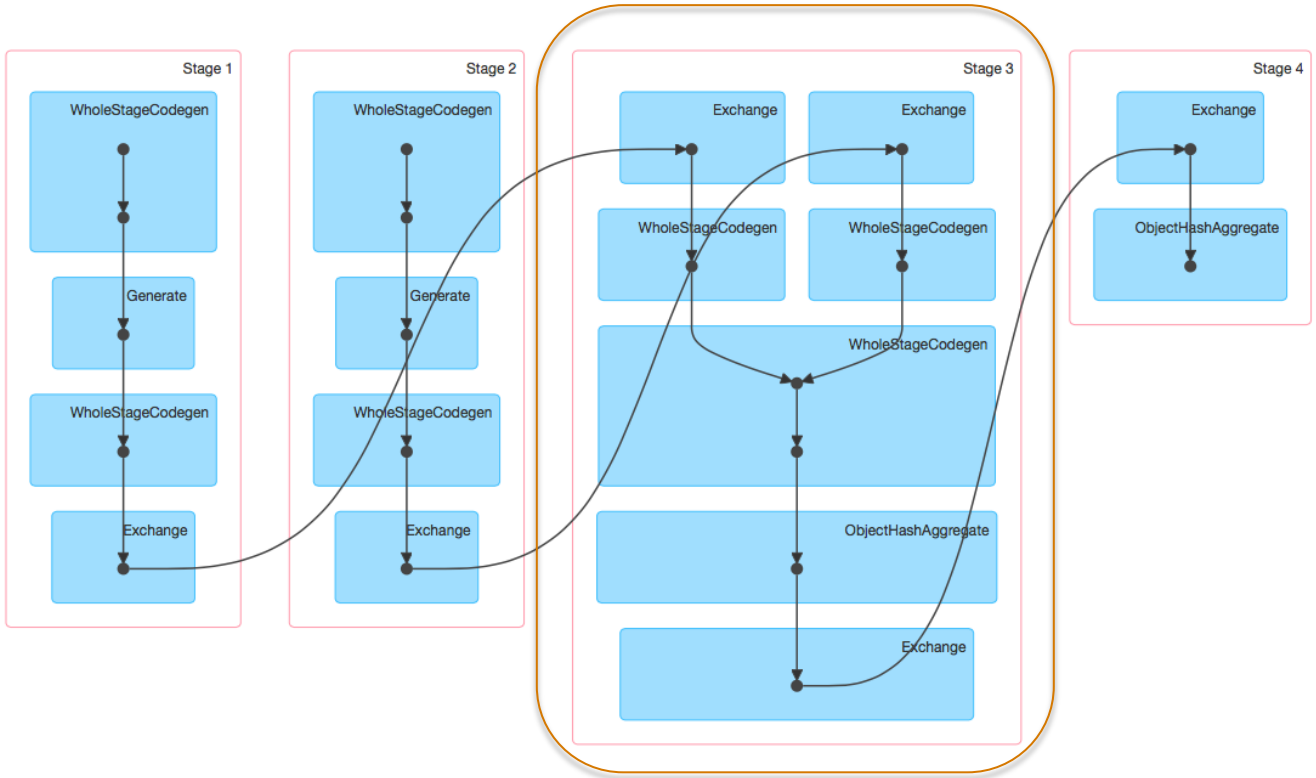
Lets try: Execution plan



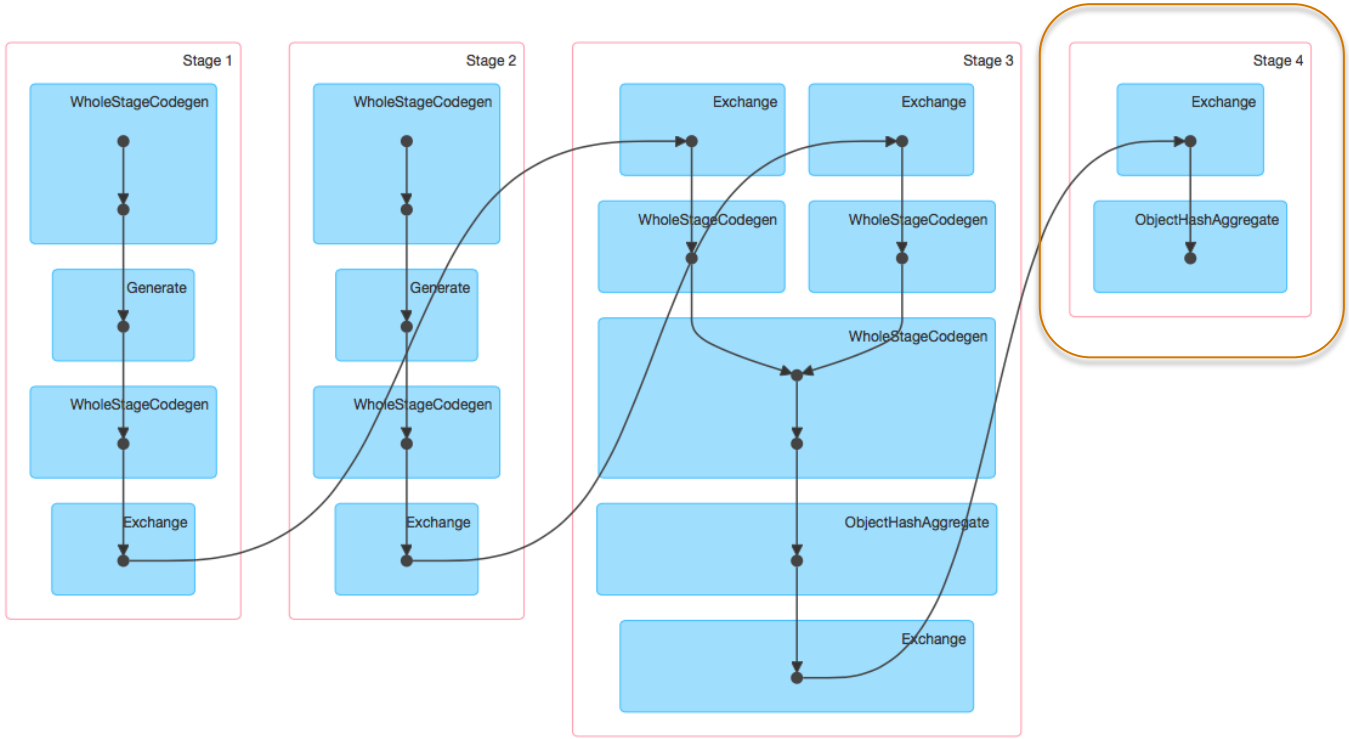
Lets try: Execution plan



Lets try: Execution plan



Lets try: Execution plan



And after few hours…

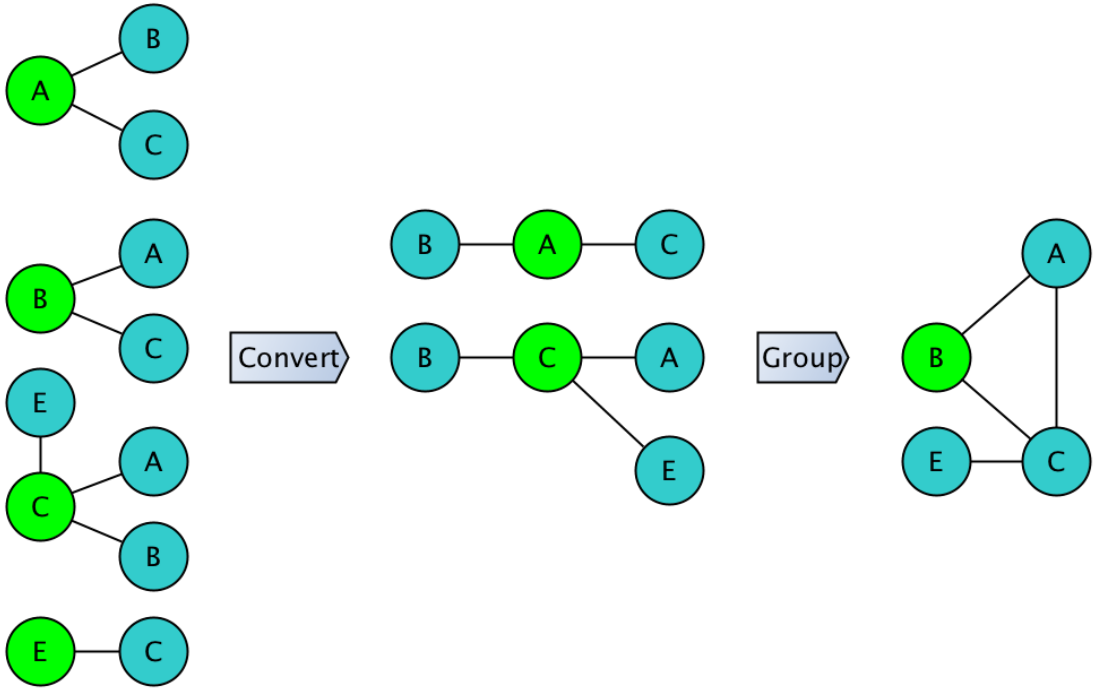
How to go further?

- Social graph in OK is symmetric
  - If A has a friend B, then B has a friend A

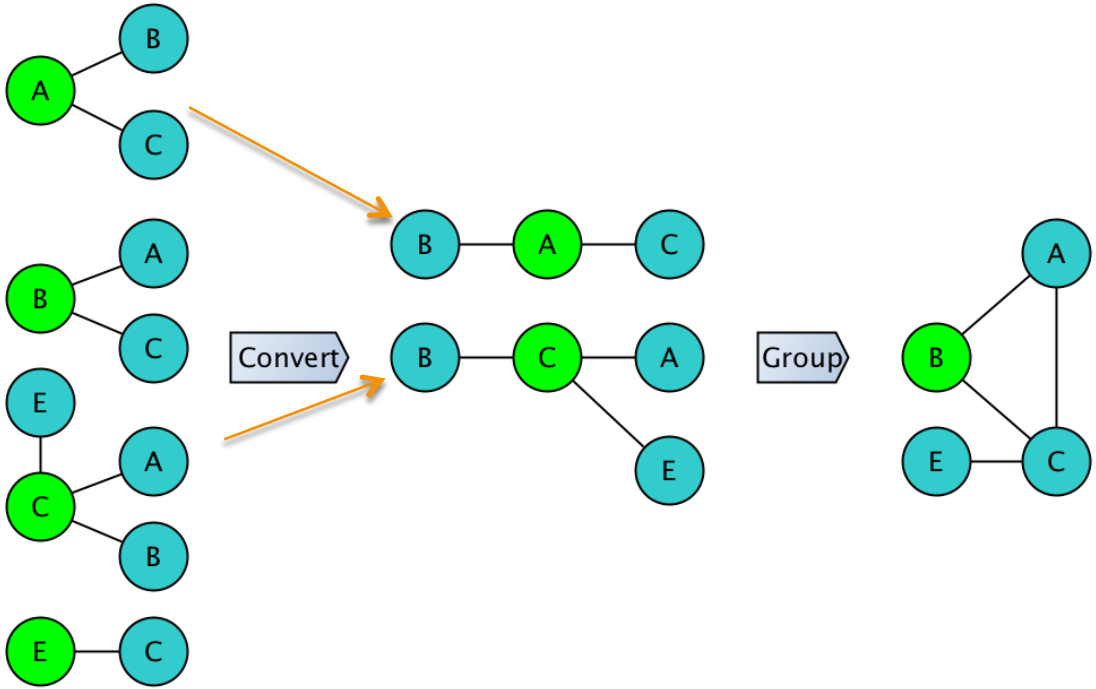
How to go further?

- Social graph in OK is symmetric
  - If A has a friend B, then B has a friend A
- Given the symmetry we can
  - Reconstruct a part of ego-subgraph for B looking at his friend A
  - Emit twice less edges (only for C->A but not for A->C)

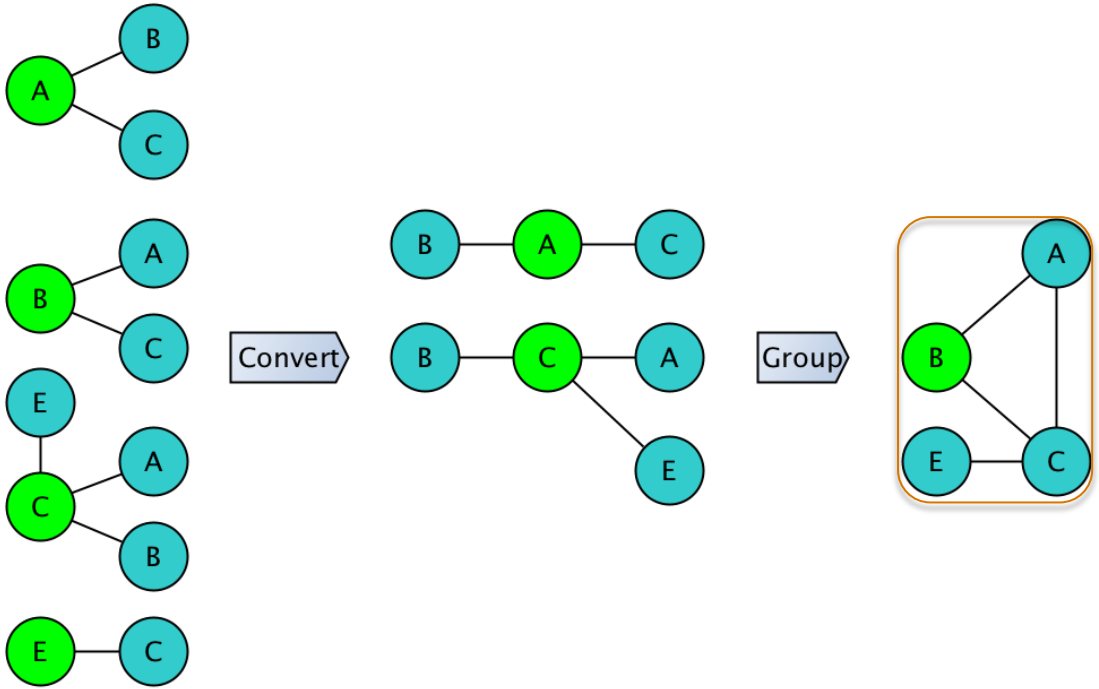
Collecting ego-subgraph: optimized



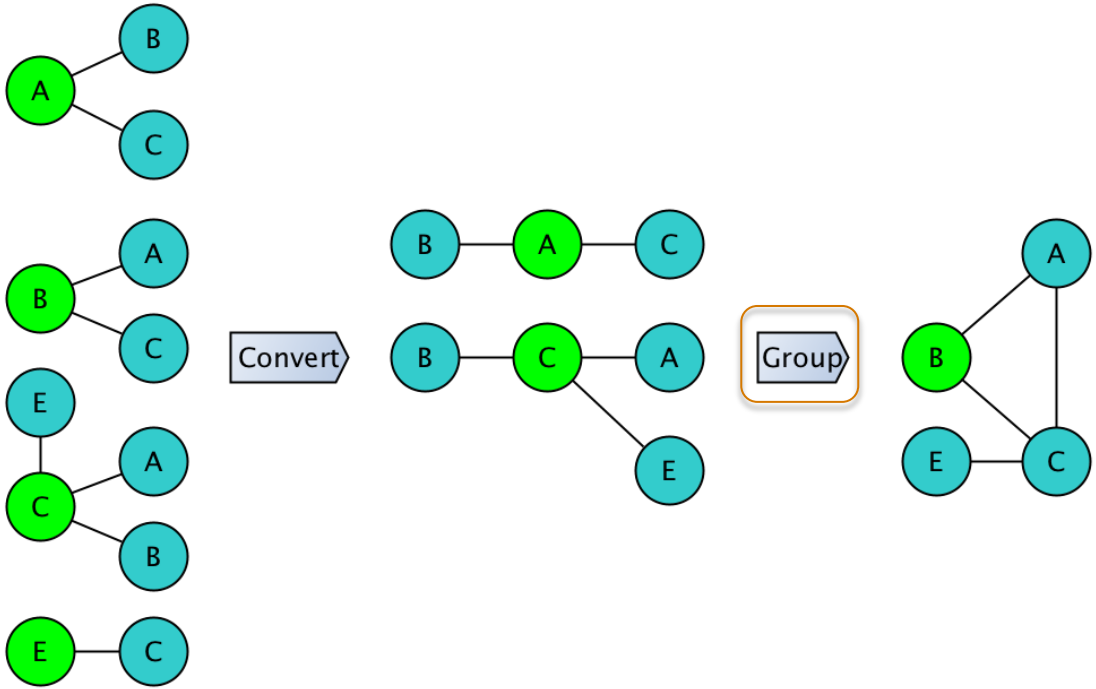
Collecting ego-subgraph: optimized



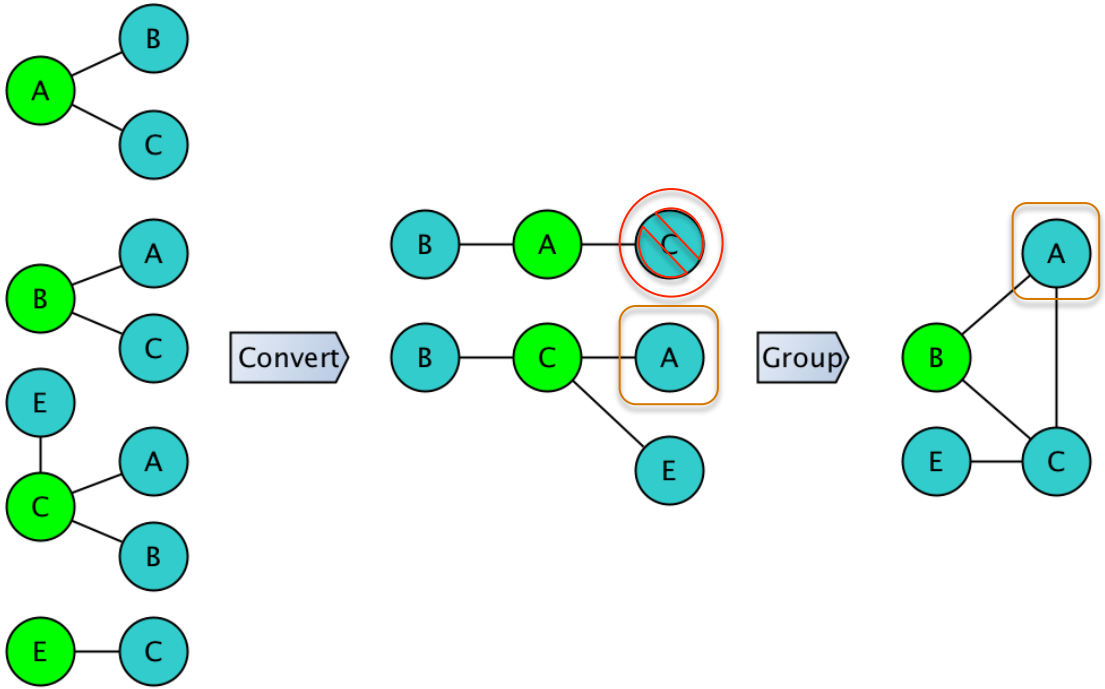
Collecting ego-subgraph: optimized



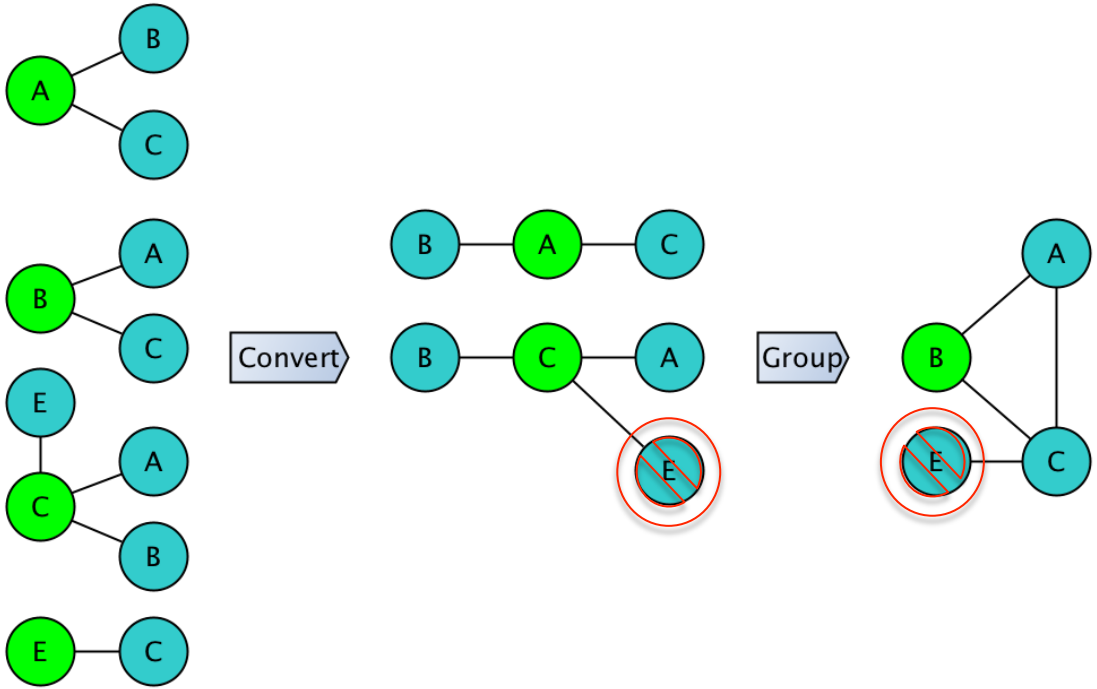
Collecting ego-subgraph: optimized



Collecting ego-subgraph: optimized



Collecting ego-subgraph: optimized



Trial 2: emitting cross edges

```
def crossGraph(userId: Long, friends: Seq[Long]) : Iterator[((Long,Long),Seq[Long])] = {
   val sorted = friends.view.filter(_ < userId).sorted.toArray
   friends.iterator.map(i => (i, userId) ->
      (if (i < userId) sorted.view.filterNot(_ == i).toArray else sorted))
}

crossGraph(20, Seq(12,1,34,56)).foreach(println)
```

```
((12,20),WrappedArray(1))
((1,20),WrappedArray(12))
((34,20),WrappedArray(1, 12))
((56,20),WrappedArray(1, 12))
```

Trial 2: emitting cross edges

```
def crossGraph(userId: Long, friends: Seq[Long]): Iterator[((Long,Long),Seq[Long])] = {
  val sorted = friends.view.filter(_ < userId).sorted.toArray
  friends.iterator.map(i => (i, userId) ->
    (if (i < userId) sorted.view.filterNot(_ == i).toArray else sorted))
}

crossGraph(20, Seq(12,1,34,56)).foreach(println)
```

```
((12,20),WrappedArray(1))
((1,20),WrappedArray(12))
((34,20),WrappedArray(1, 12))
((56,20),WrappedArray(1, 12))
```

Trial 2: emitting cross edges

```
def crossGraph(userId: Long, friends: Seq[Long]) : Iterator[((Long,Long),Seq[Long])] = {
   val sorted = friends.view.filter(_ < userId).sorted.toArray
   friends.iterator.map(i => (i, userId) ->
      (if (i < userId) sorted.view.filterNot(_ == i).toArray else sorted))
}

crossGraph(20, Seq(12,1,34,56)).foreach(println)
```

```
((12,20),WrappedArray(1))
((1,20),WrappedArray(12))
((34,20),WrappedArray(1, 12))
((56,20),WrappedArray(1, 12))
```

Trial 2: emitting cross edges

```
def crossGraph(userId: Long, friends: Seq[Long]) : Iterator[((Long,Long),Seq[Long])] = {
   val sorted = friends.view.filter(_ < userId).sorted.toArray
   friends.iterator.map(i => (i, userId) ->
      (if (i < userId) sorted.view.filterNot(_ == i).toArray else sorted))
}

crossGraph(20, Seq(12,1,34,56)).foreach(println)
```

```
((12,20),WrappedArray(1))
((1,20),WrappedArray(12))
((34,20),WrappedArray(1, 12))
((56,20),WrappedArray(1, 12))
```

Trial 2: emitting cross edges

```
def crossGraph(userId: Long, friends: Seq[Long]) : Iterator[((Long,Long),Seq[Long])] = {
   val sorted = friends.view.filter(_ < userId).sorted.toArray
   friends.iterator.map(i => (i, userId) ->
      (if (i < userId) sorted.view.filterNot(_ == i).toArray else sorted))
}

crossGraph(20, Seq(12,1,34,56)).foreach(println)
```

```
((12,20),WrappedArray(1))
((1,20),WrappedArray(12))
((34,20),WrappedArray(1, 12))
((56,20),WrappedArray(1, 12))
```

Trial 2: emitting cross edges

```
def crossGraph(userId: Long, friends: Seq[Long]) : Iterator[((Long,Long),Seq[Long])] = {
   val sorted = friends.view.filter(_ < userId).sorted.toArray
   friends.iterator.map(i => (i, userId) ->
      (if (i < userId) sorted.view.filterNot(_ == i).toArray else sorted))
}

crossGraph(20, Seq(12,1,34,56)).foreach(println)
```

```
((12,20),WrappedArray(1))
((1,20),WrappedArray(12))
((34,20),WrappedArray(1, 12))
((56,20),WrappedArray(1, 12))
```

Trial 2: emitting cross edges

```
def crossGraph(userId: Long, friends: Seq[Long]) : Iterator[((Long,Long),Seq[Long])] = {
   val sorted = friends.view.filter(_ < userId).sorted.toArray
   friends.iterator.map(i => (i, userId) ->
      (if (i < userId) sorted.view.filterNot(_ == i).toArray else sorted))
}

crossGraph(20, Seq(12,1,34,56)).foreach(println)
```

```
((12,20),WrappedArray(1))
((1,20),WrappedArray(12))
((34,20),WrappedArray(1, 12))
((56,20),WrappedArray(1, 12))
```

Trial 2: emitting cross edges

```
def crossGraph(userId: Long, friends: Seq[Long]) : Iterator[((Long,Long),Seq[Long])] = {
   val sorted = friends.view.filter(_ < userId).sorted.toArray
   friends.iterator.map(i => (i, userId) ->
      (if (i < userId) sorted.view.filterNot(_ == i).toArray else sorted))
}

crossGraph(20, Seq(12,1,34,56)).foreach(println)
```

```
((12,20),WrappedArray(1))
((1,20),WrappedArray(12))
((34,20),WrappedArray(1, 12))
((56,20),WrappedArray(1, 12))
```

Trial 2: emitting cross edges

```
def crossGraph(userId: Long, friends: Seq[Long]) : Iterator[((Long,Long),Seq[Long])] = {
  val sorted = friends.view.filter(_ < userId).sorted.toArray
  friends.iterator.map(i => (i, userId) ->
    (if (i < userId) sorted.view.filterNot(_ == i).toArray else sorted))
}
```

```
crossGraph(20, Seq(12,1,34,56)).foreach(println)
```

```
((12,20),WrappedArray(1))
((1,20),WrappedArray(12))
((34,20),WrappedArray(1, 12))
((56,20),WrappedArray(1, 12))
```

Trial 2: emitting cross edges

```
def crossGraph(userId: Long, friends: Seq[Long]) : Iterator[((Long,Long),Seq[Long])] = {
   val sorted = friends.view.filter(_ < userId).sorted.toArray
   friends.iterator.map(i => (i, userId) ->
      (if (i < userId) sorted.view.filterNot(_ == i).toArray else sorted))
}

crossGraph(20, Seq(12,1,34,56)).foreach(println)
```

```
((12,20),WrappedArray(1))
((1,20),WrappedArray(12))
((34,20),WrappedArray(1, 12))
((56,20),WrappedArray(1, 12))
```

Trial 2: emitting cross edges

```
def crossGraph(userId: Long, friends: Seq[Long]) : Iterator[((Long,Long),Seq[Long])] = {
   val sorted = friends.view.filter(_ < userId).sorted.toArray
   friends.iterator.map(i => (i, userId) ->
      (if (i < userId) sorted.view.filterNot(_ == i).toArray else sorted))
}

crossGraph(20, Seq(12,1,34,56)).foreach(println)
```

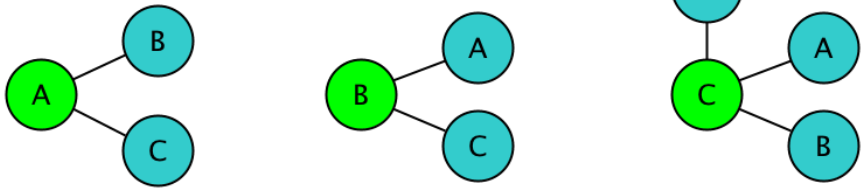
```
((12,20),WrappedArray(1))
((1,20),WrappedArray(12))
((34,20),WrappedArray(1, 12))
((56,20),WrappedArray(1, 12))
```

Trial 2: emitting cross edges

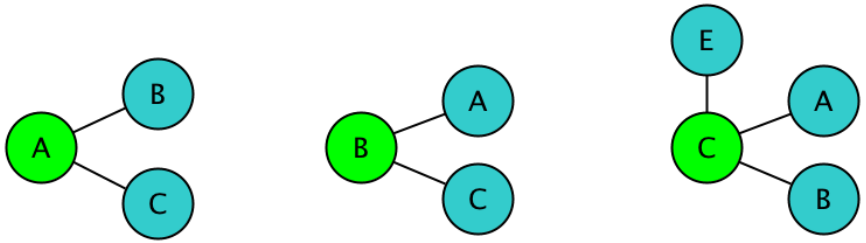
```
def crossGraph(userId: Long, friends: Seq[Long]) : Iterator[((Long,Long),Seq[Long])] = {
   val sorted = friends.view.filter(_ < userId).sorted.toArray
   friends.iterator.map(i => (i, userId) ->
      (if (i < userId) sorted.view.filterNot(_ == i).toArray else sorted))
}

crossGraph(20, Seq(12,1,34,56)).foreach(println)
```

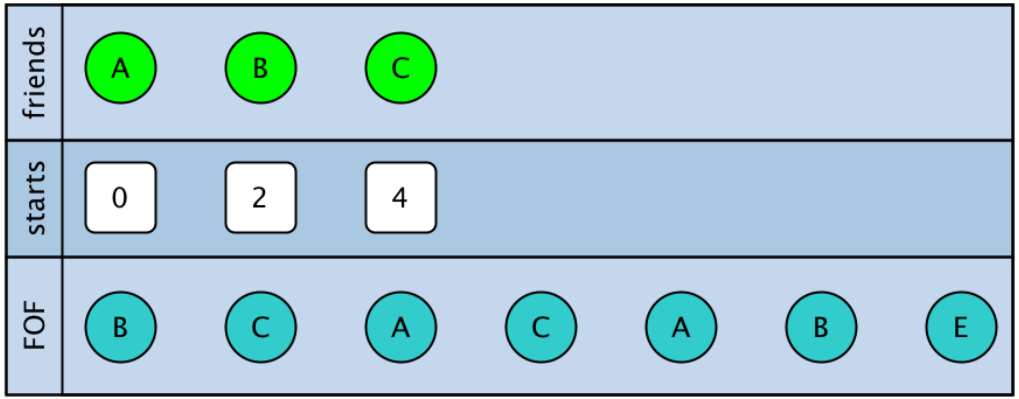
```
((12,20),WrappedArray(1))
((1,20),WrappedArray(12))
((34,20),WrappedArray(1, 12))
((56,20),WrappedArray(1, 12))
```

Trial 2: How to store graph?

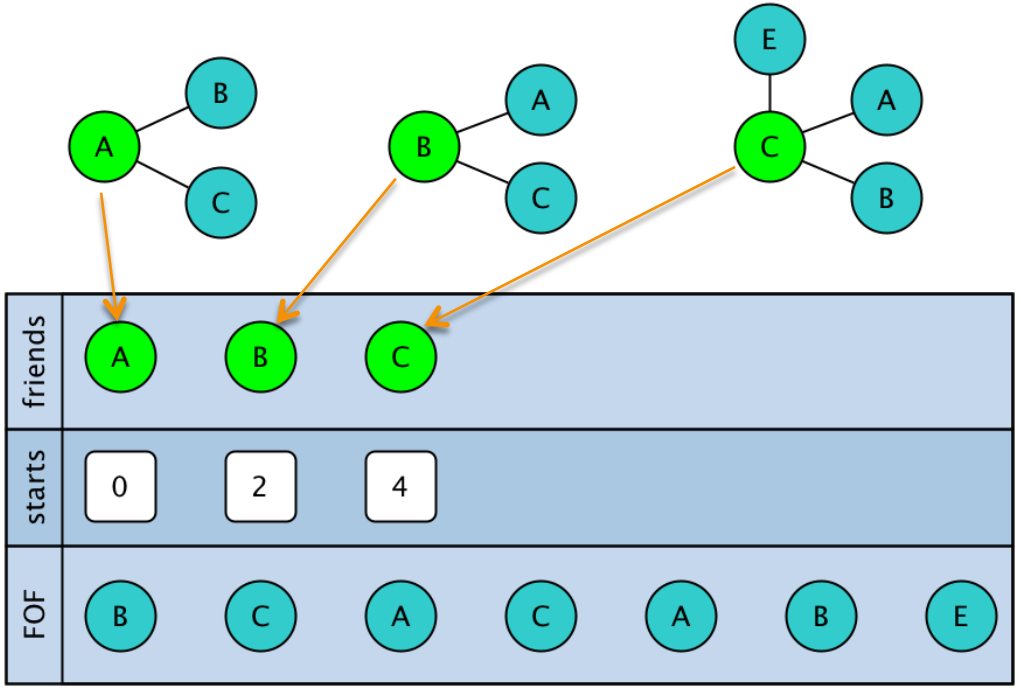


Trial 2: How to store graph?

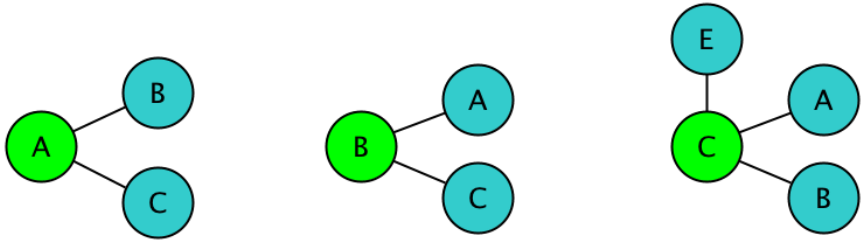


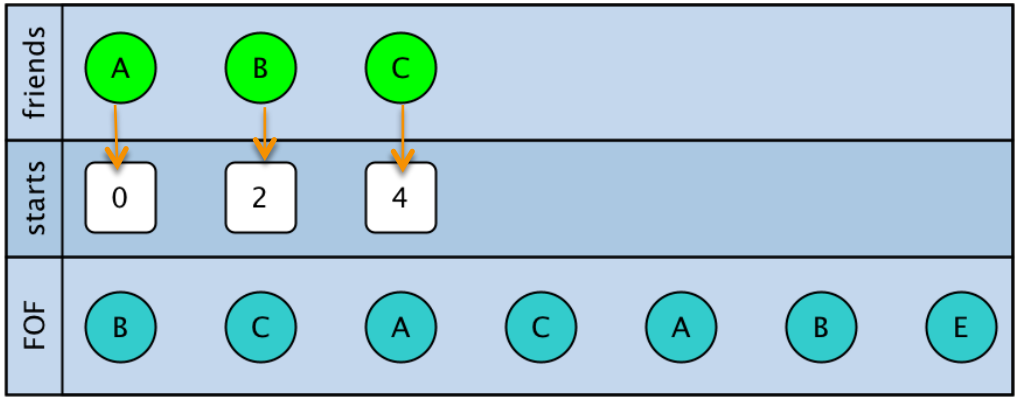


Trial 2: How to store graph?

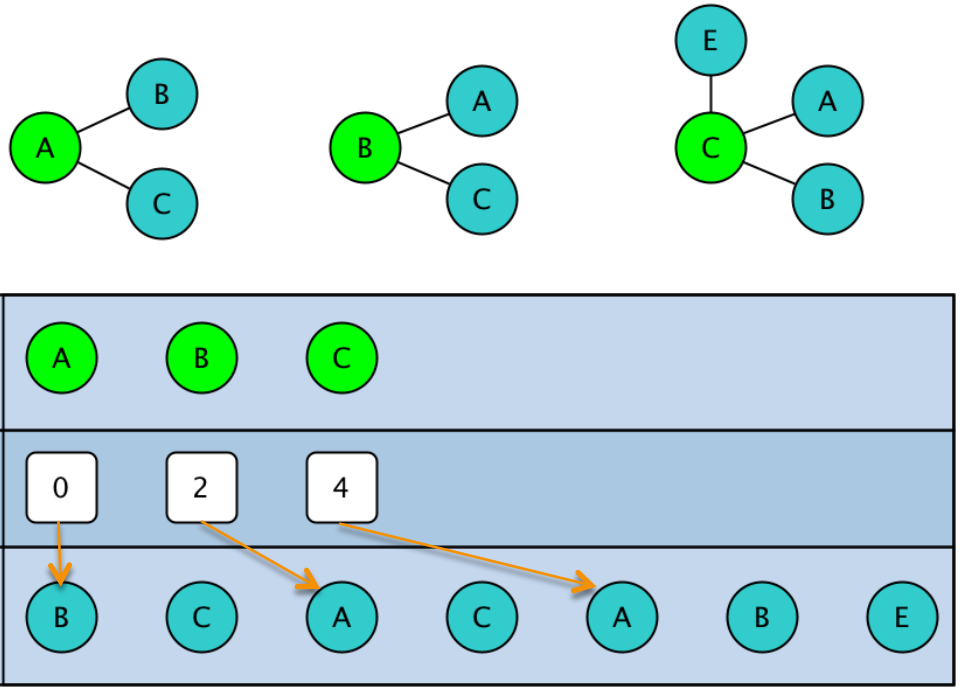


Trial 2: How to store graph?





Trial 2: How to store graph?



Trial 2: Merging and filtering

```
def mergeEgoGraph(graph : Iterator[(Long,Seq[Long])]) : (Seq[Long], Seq[Int], Seq[Long]) = {
  val myFriends = new scala.collection.mutable.ArrayBuffer[Long]()
  val linksStart = scala.collection.mutable.ArrayBuilder.make[Int]
  val friendFriends = scala.collection.mutable.ArrayBuilder.make[Long]
  var links = 0

  graph.foreach(record => {
    var index = 0
    linksStart += links.toShort

    record._2.foreach(friendOfFriend => {
      index = myFriends.view(index, myFriends.length).takeWhile(_ <= friendOfFriend).foldLeft(index)((a,_) => a + 1)
      if (index > 0 && index <= myFriends.length && myFriends(index - 1) == friendOfFriend) {
        index += 1
        links += 1
        friendFriends += friendOfFriend
      }
    })

    myFriends += record._1
  })
  (myFriends.toArray, linksStart.result, friendFriends.result)
}
```

Trial 2: Merging and filtering

```
def mergeEgoGraph(graph : Iterator[(Long,Seq[Long])]) : (Seq[Long], Seq[Int], Seq[Long]) = {
  val myFriends = new scala.collection.mutable.ArrayBuffer[Long]()
  val linksStart = scala.collection.mutable.ArrayBuilder.make[Int]
  val friendFriends = scala.collection.mutable.ArrayBuilder.make[Long]
  var links = 0

  graph.foreach(record => {
    var index = 0
    linksStart += links.toShort

    record._2.foreach(friendOfFriend => {
      index = myFriends.view(index, myFriends.length).takeWhile(_ <= friendOfFriend).foldLeft(index)((a,_) => a + 1)
      if (index > 0 && index <= myFriends.length && myFriends(index - 1) == friendOfFriend) {
        index += 1
        links += 1
        friendFriends += friendOfFriend
      }
    })

    myFriends += record._1
  })
  (myFriends.toArray, linksStart.result, friendFriends.result)
}
```

Trial 2: Merging and filtering

```
def mergeEgoGraph(graph : Iterator[(Long,Seq[Long])]) : (Seq[Long], Seq[Int], Seq[Long]) = {
  val myFriends = new scala.collection.mutable.ArrayBuffer[Long]()
  val linksStart = scala.collection.mutable.ArrayBuilder.make[Int]
  val friendFriends = scala.collection.mutable.ArrayBuilder.make[Long]
  var links = 0

  graph.foreach(record => {
    var index = 0
    linksStart += links.toShort

    record._2.foreach(friendOfFriend => {
      index = myFriends.view(index, myFriends.length).takeWhile(_ <= friendOfFriend).foldLeft(index)((a,_) => a + 1)
      if (index > 0 && index <= myFriends.length && myFriends(index - 1) == friendOfFriend) {
        index += 1
        links += 1
        friendFriends += friendOfFriend
      }
    })

    myFriends += record._1
  })
  (myFriends.toArray, linksStart.result, friendFriends.result)
}
```

Trial 2: Merging and filtering

```
val (myFriends, linksStart, friendFriends) = mergeEgoGraph(Iterator(
    1L -> Seq(),
    2L -> Seq(),
    3L -> Seq(1L,2L),
    5L -> Seq(1L,2L,3L,4L),
    12L -> Seq(1L,4L,5L,7L)))
```

myFriends: Seq[Long] = WrappedArray(1, 2, 3, 5, 12)

linksStart: Seq[Int] = WrappedArray(0, 0, 0, 2, 5)

friendFriends: Seq[Long] = WrappedArray(1, 2, 1, 2, 3, 1, 5)

Trial 2: Merging and filtering

```
val (myFriends, linksStart, friendFriends) = mergeEgoGraph(Iterator(
  1L -> Seq(),
  2L -> Seq(),
  3L -> Seq(1L,2L),
  5L -> Seq(1L,2L,3L,4L),
  12L -> Seq(1L,4L,5L,7L)))
```

myFriends: Seq[Long] = WrappedArray(1, 2, 3, 5, 12)

linksStart: Seq[Int] = WrappedArray(0, 0, 0, 2, 5)

friendFriends: Seq[Long] = WrappedArray(1, 2, 1, 2, 3, 1, 5)

Trial 2: Merging and filtering

```
val (myFriends, linksStart, friendFriends) = mergeEgoGraph(Iterator(
    1L -> Seq(),
    2L -> Seq(),
    3L -> Seq(1L,2L),
    5L -> Seq(1L,2L,3L,4L),
    12L -> Seq(1L,4L,5L,7L)))
```

myFriends: Seq[Long] = WrappedArray(1, 2, 3, 5, 12)

linksStart: Seq[Int] = WrappedArray(0, 0, 0, 2, 5)

friendFriends: Seq[Long] = WrappedArray(1, 2, 1, 2, 3, 1, 5)

Trial 2: Merging and filtering

```
val (myFriends, linksStart, friendFriends) = mergeEgoGraph(Iterator(
    1L -> Seq(),
    2L -> Seq(),
    3L -> Seq(1L,2L),
    5L -> Seq(1L,2L,3L,4L),
    12L -> Seq(1L,4L,5L,7L)))
```

```
myFriends: Seq[Long] = WrappedArray(1, 2, 3, 5, 12)
linksStart: Seq[Int] = WrappedArray(0, 0, 0, 2, 5)
friendFriends: Seq[Long] = WrappedArray(1, 2, 1, 2, 3, 1, 5)
```

Trial 2: Merging and filtering

```
val (myFriends, linksStart, friendFriends) = mergeEgoGraph(Iterator(
    1L -> Seq(),
    2L -> Seq(),
    3L -> Seq(1L,2L),
    5L -> Seq(1L,2L,3L,4L),
    12L -> Seq(1L,4L,5L,7L)))
```

myFriends: Seq[Long] = WrappedArray(1, 2, 3, 5, 12)

linksStart: Seq[Int] = WrappedArray(0, 0, 0, 2, 5)

friendFriends: Seq[Long] = WrappedArray(1, 2, 1, 2, 3, 1, 5)

Trial 2: Merging and filtering

```
val (myFriends, linksStart, friendFriends) = mergeEgoGraph(Iterator(
    1L -> Seq(),
    2L -> Seq(),
    3L -> Seq(1L,2L),
    5L -> Seq(1L,2L,3L,4L),
    12L -> Seq(1L,4L,5L,7L)))
```

myFriends: Seq[Long] = WrappedArray(1, 2, 3, 5, 12)

linksStart: Seq[Int] = WrappedArray(0, 0, 0, 2, 5)

friendFriends: Seq[Long] = WrappedArray(1, 2, 1, 2, 3, 1, 5)

Trial 2: Assembling

```
val egoGraph = graph.select($"userId", $"friendships.friendId”)
   .rdd.mapPartitions(
      _.flatMap(row => crossGraph(row.getLong(0), row.getAs[Seq[Long]](1))))
   .repartitionAndSortWithinPartitions(new org.apache.spark.HashPartitioner(2057) {
      override def getPartition(key: Any): Int = super.getPartition(key.asInstanceOf[(Long,Long)]._1)
   })
   .mapPartitions(
       iterator => IteratorUtils.groupByKey(iterator.map(x => x._1._1 -> (x._1._2, x._2)))
           .map(x => x._1 -> mergeEgoGraph(x._2)))
   .toDF("userId", "egoGraph")
   .select($"userId", $"egoGraph._1".as("friends"), $"egoGraph._2".as("linksStart"),
      $"egoGraph._3".as("friendFriends"))
```

Trial 2: Assembling

```
val egoGraph = graph.select($"userId", $"friendships.friendId”)
  .rdd.mapPartitions(
    _.flatMap(row => crossGraph(row.getLong(0), row.getAs[Seq[Long]](1))))
  .repartitionAndSortWithinPartitions(new org.apache.spark.HashPartitioner(2057) {
    override def getPartition(key: Any): Int = super.getPartition(key.asInstanceOf[(Long,Long)]._1)
  })
  .mapPartitions(
     iterator => IteratorUtils.groupByKey(iterator.map(x => x._1._1 -> (x._1._2, x._2)))
       .map(x => x._1 -> mergeEgoGraph(x._2)))
  .toDF("userId", "egoGraph")
  .select($"userId", $"egoGraph._1".as("friends"), $"egoGraph._2".as("linksStart"),
    $"egoGraph._3".as("friendFriends"))
```

Trial 2: Assembling

```
val egoGraph = graph.select($"userId", $"friendships.friendId”)
   .rdd.mapPartitions(
      _.flatMap(row => crossGraph(row.getLong(0), row.getAs[Seq[Long]](1))))
   .repartitionAndSortWithinPartitions(new org.apache.spark.HashPartitioner(2057) {
      override def getPartition(key: Any): Int = super.getPartition(key.asInstanceOf[(Long,Long)]._1)
   })
   .mapPartitions(
       iterator => IteratorUtils.groupByKey(iterator.map(x => x._1._1 -> (x._1._2, x._2)))
           .map(x => x._1 -> mergeEgoGraph(x._2)))
   .toDF("userId", "egoGraph")
   .select($"userId", $"egoGraph._1".as("friends"), $"egoGraph._2".as("linksStart"),
      $"egoGraph._3".as("friendFriends"))
```

Trial 2: Assembling

```
val egoGraph = graph.select($"userId", $"friendships.friendId”)
   .rdd.mapPartitions(
      _.flatMap(row => crossGraph(row.getLong(0), row.getAs[Seq[Long]](1))))
   .repartitionAndSortWithinPartitions(new org.apache.spark.HashPartitioner(2057) {
      override def getPartition(key: Any): Int = super.getPartition(key.asInstanceOf[(Long,Long)]._1)
   })
   .mapPartitions(
       iterator => IteratorUtils.groupByKey(iterator.map(x => x._1._1 -> (x._1._2, x._2)))
           .map(x => x._1 -> mergeEgoGraph(x._2)))
   .toDF("userId", "egoGraph")
   .select($"userId", $"egoGraph._1".as("friends"), $"egoGraph._2".as("linksStart"),
      $"egoGraph._3".as("friendFriends"))
```

Trial 2: Assembling

```
val egoGraph = graph.select($"userId", $"friendships.friendId”)
   .rdd.mapPartitions(
      _.flatMap(row => crossGraph(row.getLong(0), row.getAs[Seq[Long]](1))))
   .repartitionAndSortWithinPartitions(new org.apache.spark.HashPartitioner(2057) {
      override def getPartition(key: Any): Int = super.getPartition(key.asInstanceOf[(Long,Long)]._1)
   })
   .mapPartitions(
       iterator => IteratorUtils.groupByKey(iterator.map(x => x._1._1 -> (x._1._2, x._2)))
           .map(x => x._1 -> mergeEgoGraph(x._2)))
   .toDF("userId", "egoGraph")
   .select($"userId", $"egoGraph._1".as("friends"), $"egoGraph._2".as("linksStart"),
      $"egoGraph._3".as("friendFriends"))
```

Trial 2: Assembling

```
val egoGraph = graph.select($"userId", $"friendships.friendId”)
   .rdd.mapPartitions(
      _.flatMap(row => crossGraph(row.getLong(0), row.getAs[Seq[Long]](1))))
   .repartitionAndSortWithinPartitions(new org.apache.spark.HashPartitioner(2057) {
      override def getPartition(key: Any): Int = super.getPartition(key.asInstanceOf[(Long,Long)]._1)
   })
   .mapPartitions(
       iterator => IteratorUtils.groupByKey(iterator.map(x => x._1._1 -> (x._1._2, x._2)))
           .map(x => x._1 -> mergeEgoGraph(x._2)))
   .toDF("userId", "egoGraph")
   .select($"userId", $"egoGraph._1".as("friends"), $"egoGraph._2".as("linksStart"),
      $"egoGraph._3".as("friendFriends"))
```

Trial 2: Assembling

```
val egoGraph = graph.select($"userId", $"friendships.friendId”)
   .rdd.mapPartitions(
      _.flatMap(row => crossGraph(row.getLong(0), row.getAs[Seq[Long]](1))))
   .repartitionAndSortWithinPartitions(new org.apache.spark.HashPartitioner(2057) {
      override def getPartition(key: Any): Int = super.getPartition(key.asInstanceOf[(Long,Long)]._1)
   })
   .mapPartitions(
       iterator => IteratorUtils.groupByKey(iterator.map(x => x._1._1 -> (x._1._2, x._2)))
           .map(x => x._1 -> mergeEgoGraph(x._2)))
   .toDF("userId", "egoGraph")
   .select($"userId", $"egoGraph._1".as("friends"), $"egoGraph._2".as("linksStart"),
      $"egoGraph._3".as("friendFriends"))
```

Trial 2: Assembling

```
val egoGraph = graph.select($"userId", $"friendships.friendId”)
   .rdd.mapPartitions(
      _.flatMap(row => crossGraph(row.getLong(0), row.getAs[Seq[Long]](1))))
   .repartitionAndSortWithinPartitions(new org.apache.spark.HashPartitioner(2057) {
      override def getPartition(key: Any): Int = super.getPartition(key.asInstanceOf[(Long,Long)]._1)
   })
   .mapPartitions(
       iterator => IteratorUtils.groupByKey(iterator.map(x => x._1._1 -> (x._1._2, x._2)))
           .map(x => x._1 -> mergeEgoGraph(x._2)))
   .toDF("userId", "egoGraph")
   .select($"userId", $"egoGraph._1".as("friends"), $"egoGraph._2".as("linksStart"),
      $"egoGraph._3".as("friendFriends"))
```

Trial 2: Assembling

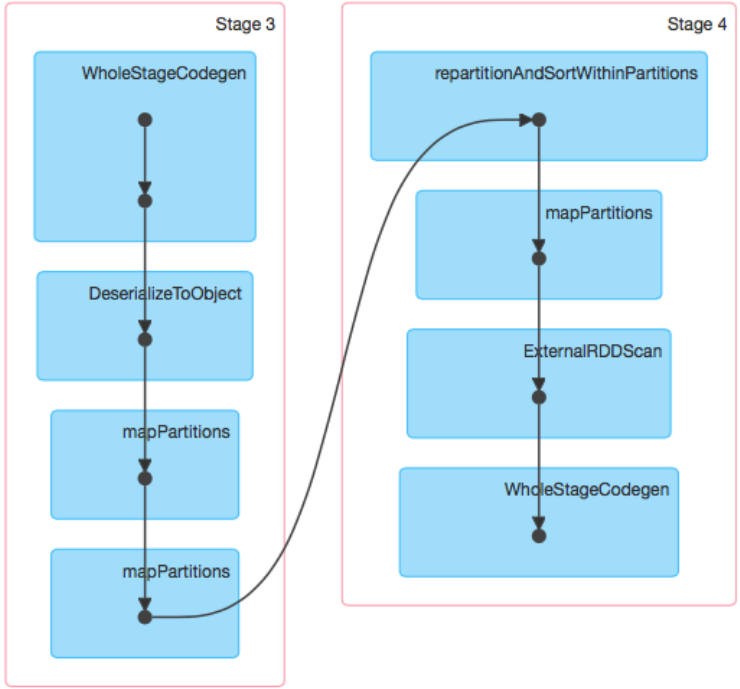
```
val egoGraph = graph.select($"userId", $"friendships.friendId”)
   .rdd.mapPartitions(
      _.flatMap(row => crossGraph(row.getLong(0), row.getAs[Seq[Long]](1))))
   .repartitionAndSortWithinPartitions(new org.apache.spark.HashPartitioner(2057) {
      override def getPartition(key: Any): Int = super.getPartition(key.asInstanceOf[(Long,Long)]._1)
   })
   .mapPartitions(
       iterator => IteratorUtils.groupByKey(iterator.map(x => x._1._1 -> (x._1._2, x._2)))
           .map(x => x._1 -> mergeEgoGraph(x._2)))
   .toDF("userId", "egoGraph")
   .select($"userId", $"egoGraph._1".as("friends"), $"egoGraph._2".as("linksStart"),
      $"egoGraph._3".as("friendFriends"))
```

Trial 2: Assembling

egoGraph.printSchema

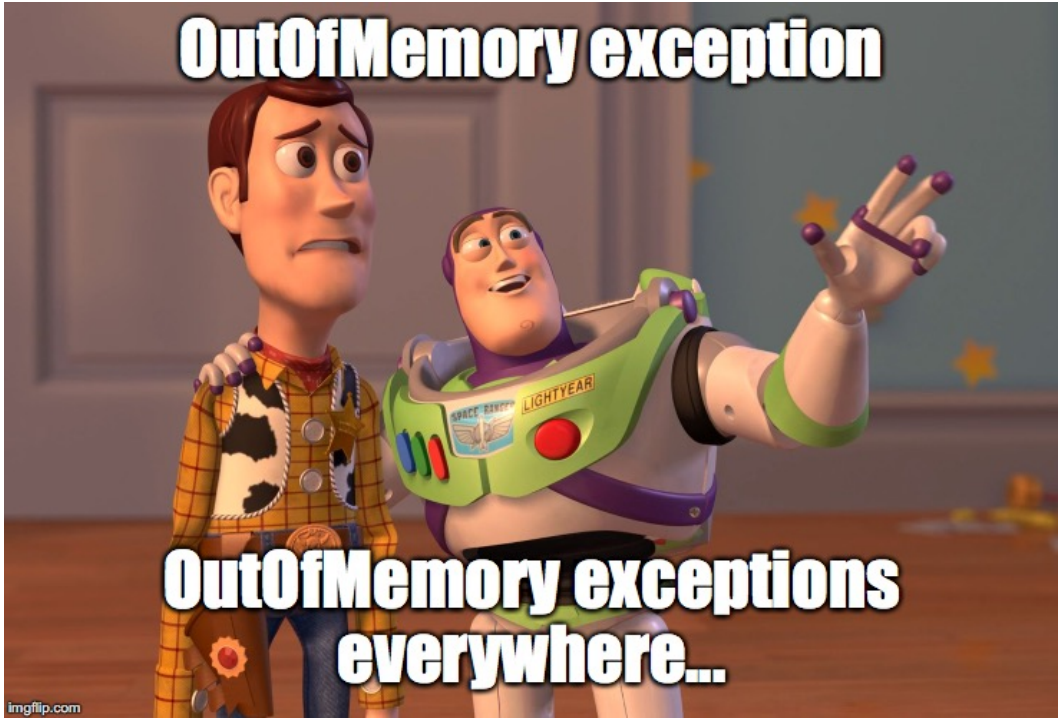
```
root
 |-- userId: long (nullable = false)
 |-- friends: array (nullable = true)
 |    |-- element: long (containsNull = false)
 |-- linksStart: array (nullable = true)
 |    |-- element: short (containsNull = false)
 |-- friendFriends: array (nullable = true)
 |    |-- element: long (containsNull = false)
```

Trial 2: execution plan

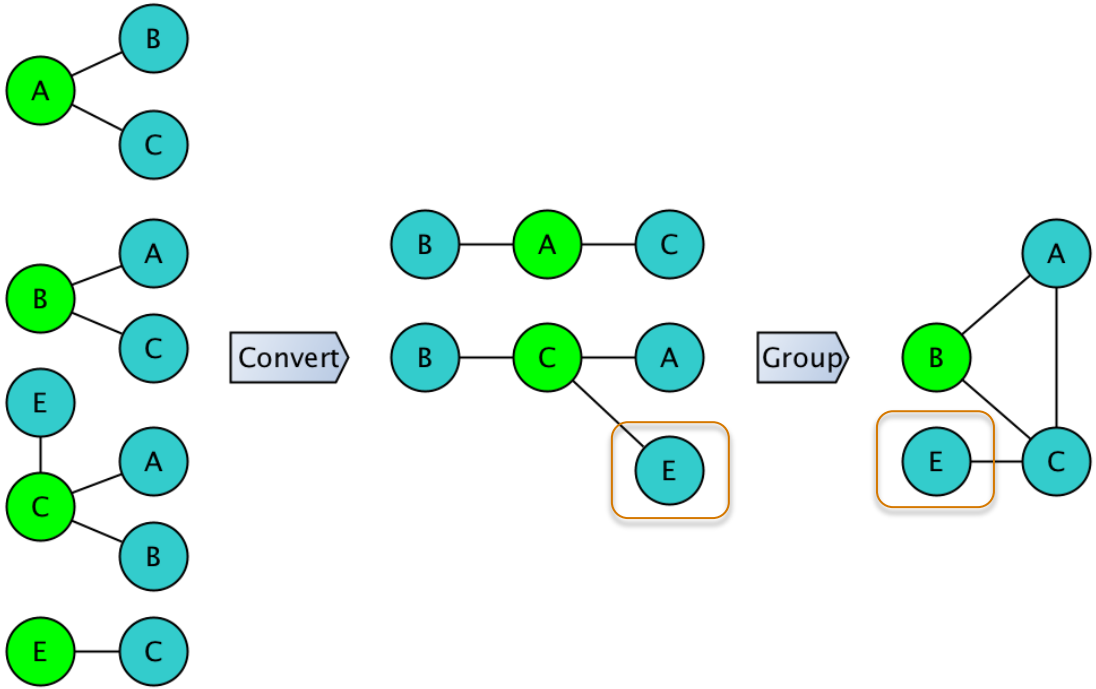


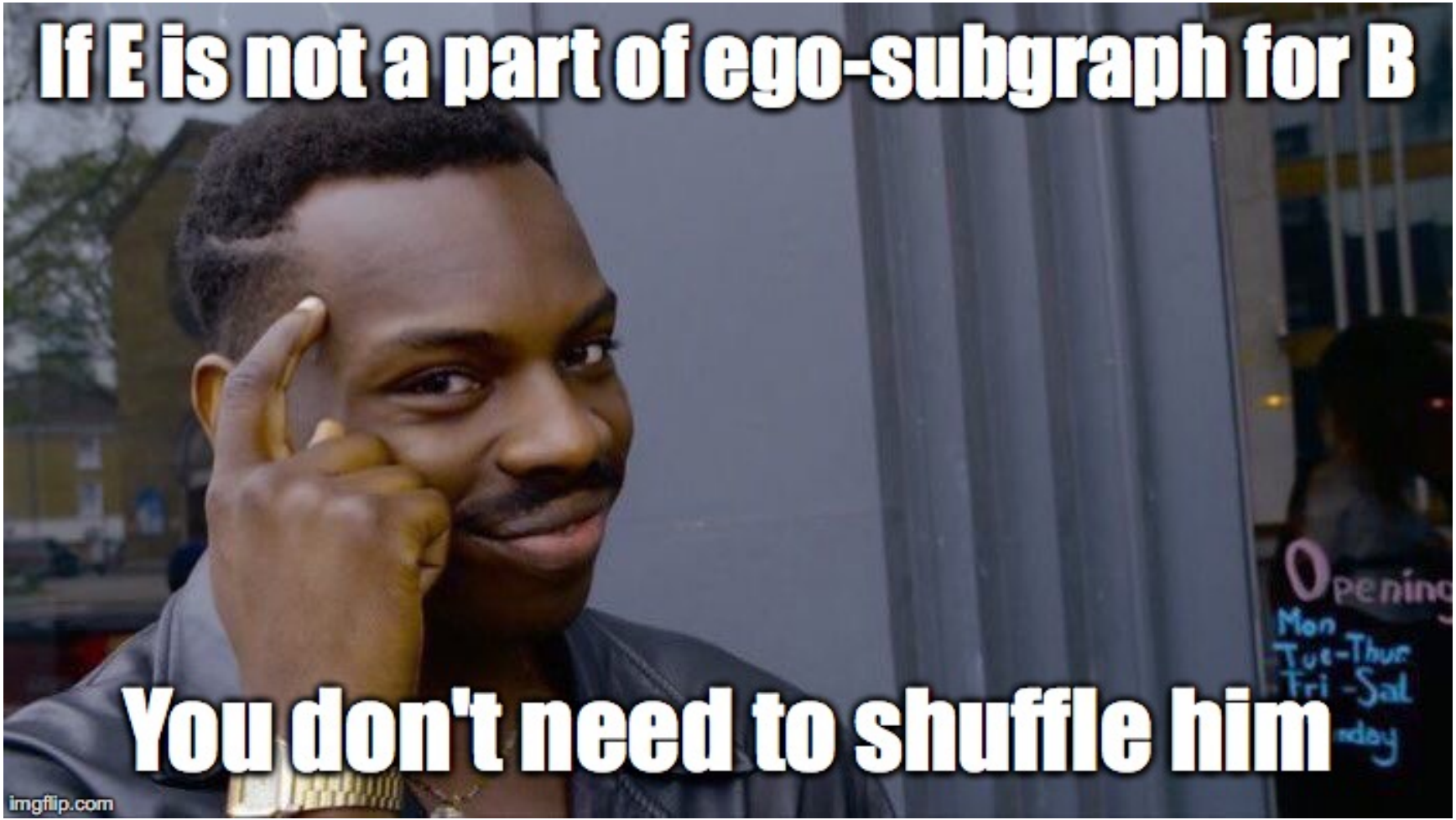
Trial 2: Stage 3 completed ☺

Trial 2: After an hour Stage 4 fails ☹

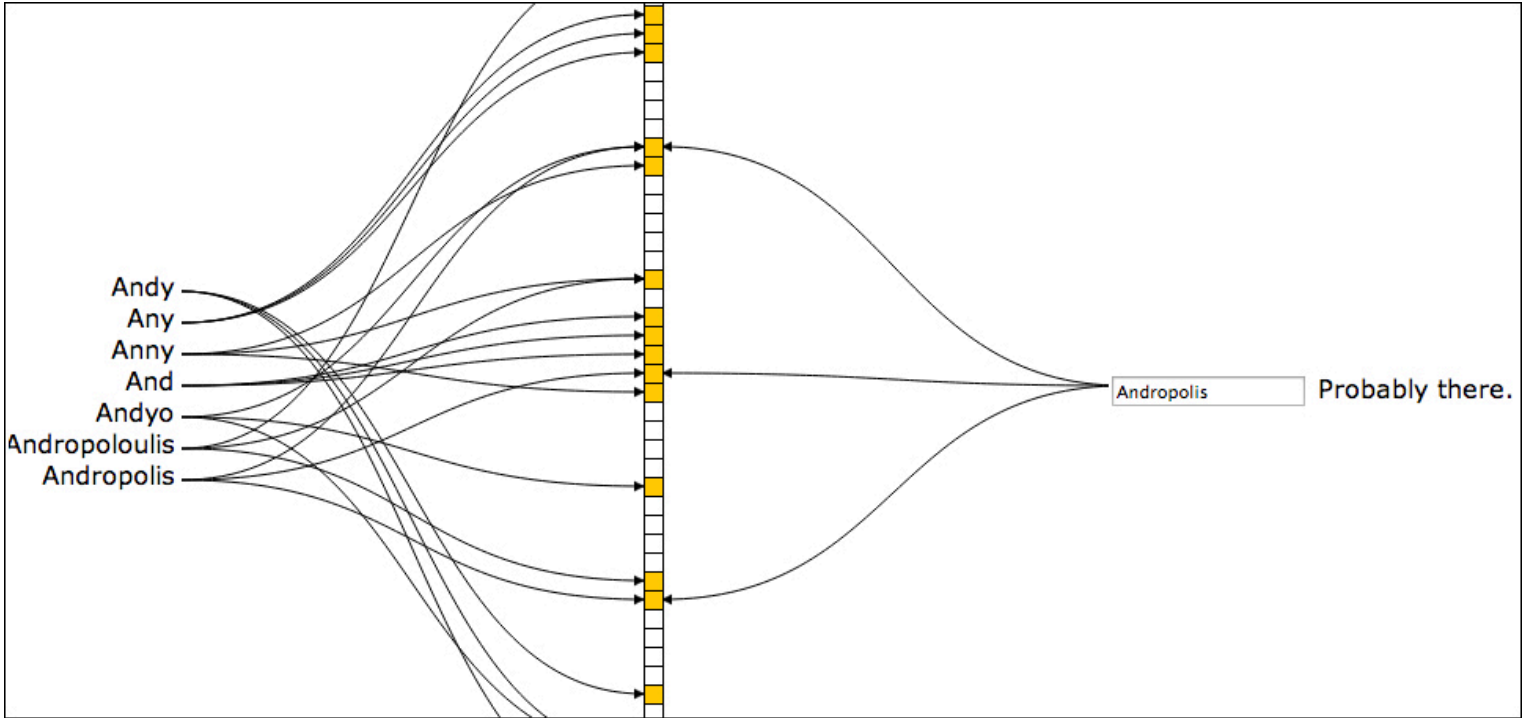


What to do next?

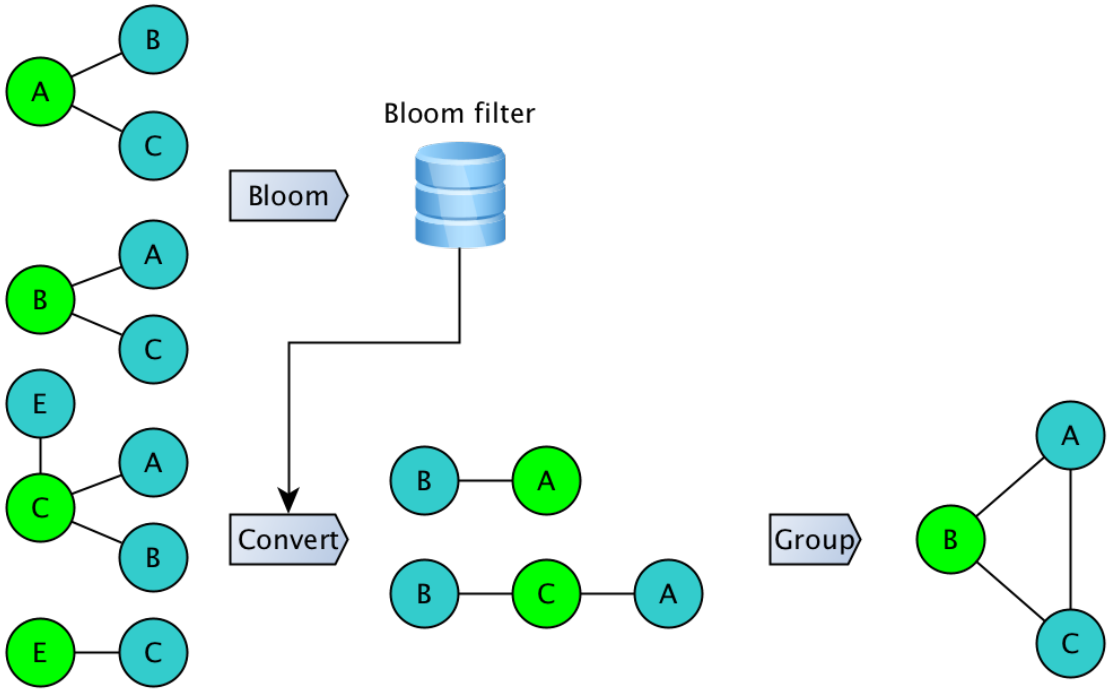



If E is not a part of ego-subgraph for B
You don't need to shuffle him
imgflip.com

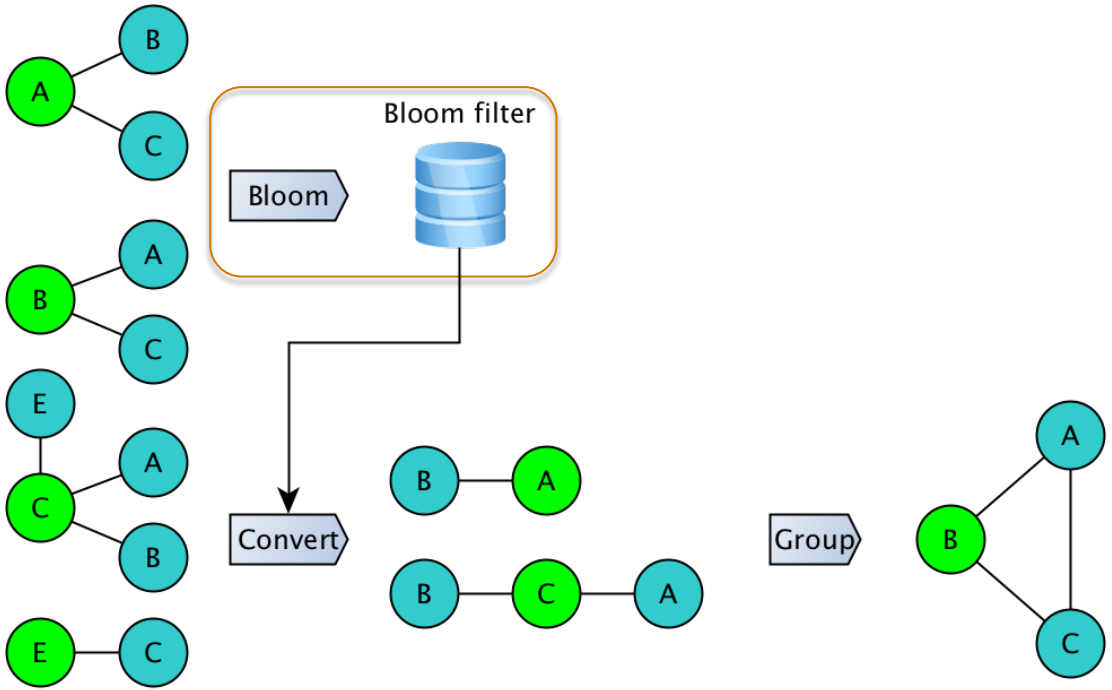
Bloom them all!



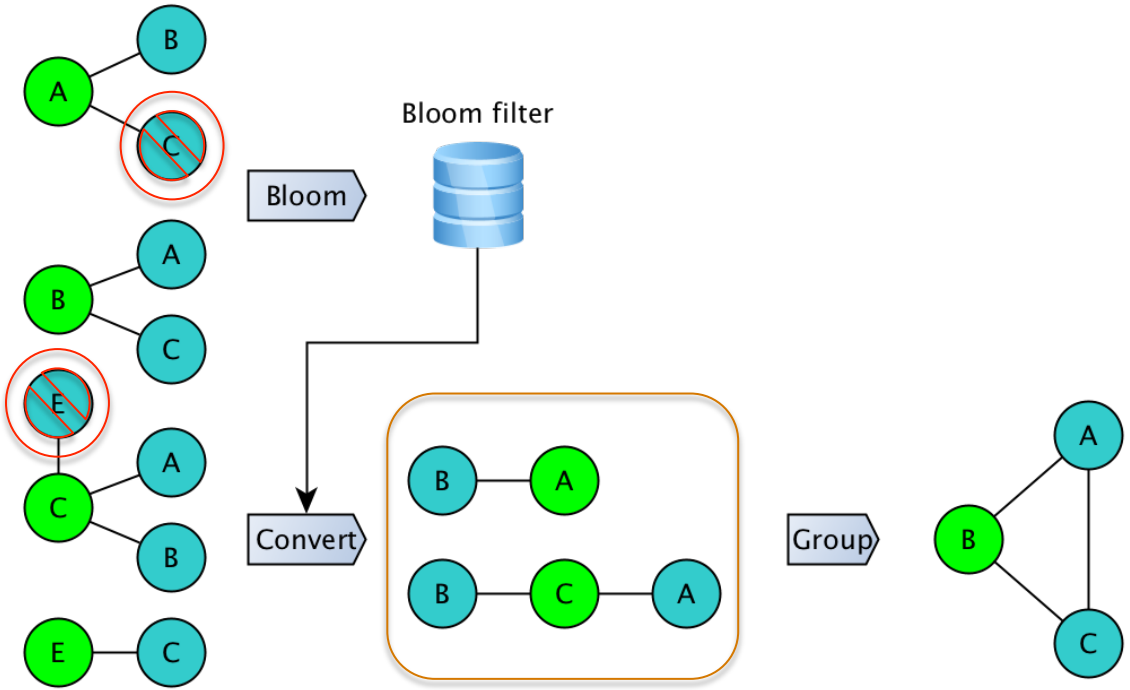
Using Bloom filter to reduce shuffle



Using Bloom filter to reduce shuffle



Using Bloom filter to reduce shuffle



Trial 3: Load and apply filter

```
@transient lazy val filter = BloomFilterHolder.getModel(
        "/graph/bloomfilter")


def crossGraphFiltered(userId: Long, friends: Seq[Long]) = {
   val sorted = friends.view.filter(_ < userId).sorted.toArray

   friends.iterator.map(i => (i, userId) ->
      sorted.view.filter(x => x != i &&
         filter.isPresent(Utils.pack(i,x))).toArray)
}
```

Trial 3: Load the filter

```
@transient lazy val filter = BloomFilterHolder.getModel(
        "/graph/bloomfilter")
```

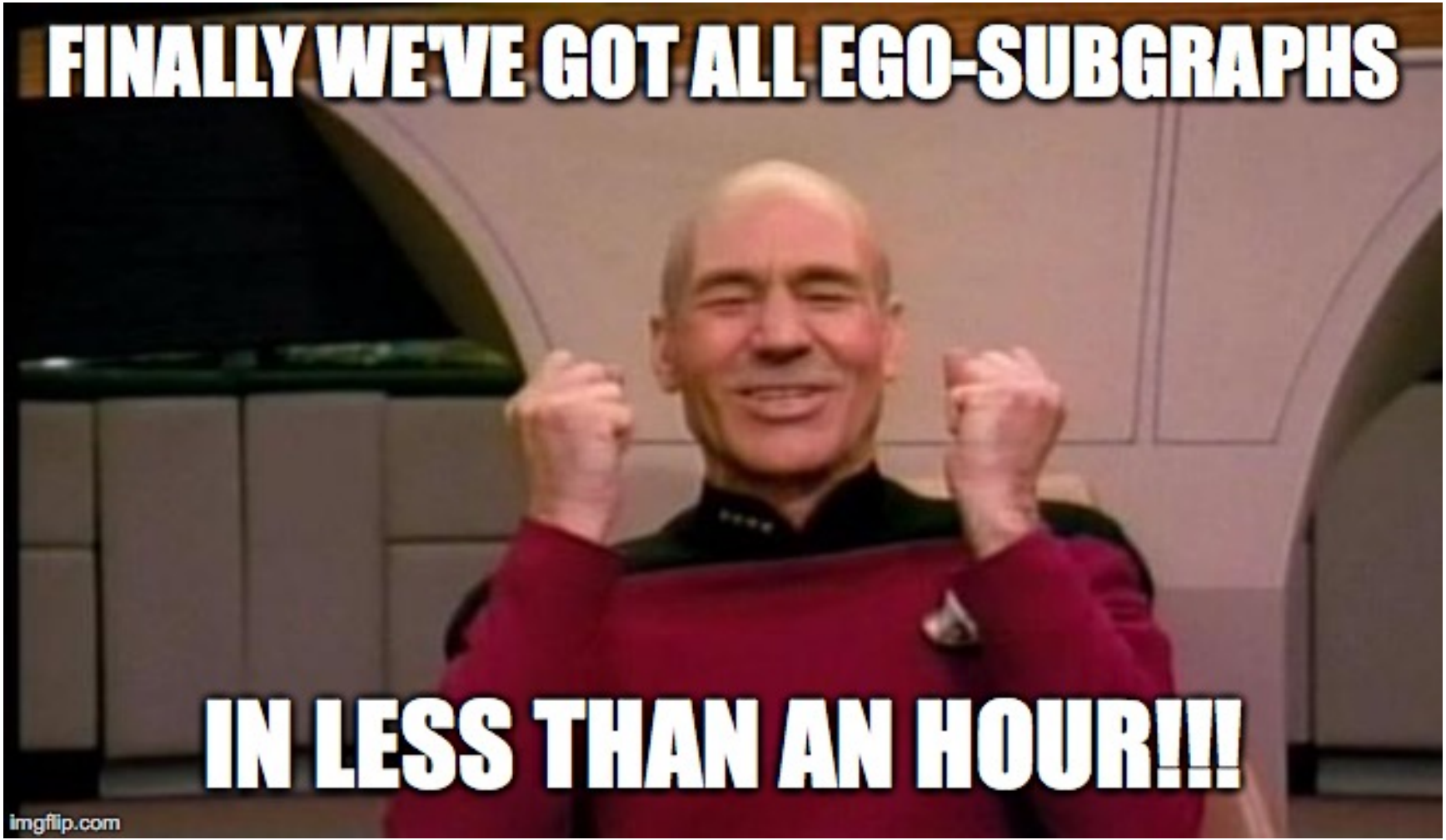
```
def crossGraphFiltered(userId: Long, friends: Seq[Long]) = {
  val sorted = friends.view.filter(_ < userId).sorted.toArray

  friends.iterator.map(i => (i, userId) ->
    sorted.view.filter(x => x != i &&
      filter.isPresent(Utils.pack(i,x))).toArray)
}
```

Trial 3: Load the filter

```
@transient lazy val filter = BloomFilterHolder.getModel(
        "/graph/bloomfilter")


def crossGraphFiltered(userId: Long, friends: Seq[Long]) = {
   val sorted = friends.view.filter(_ < userId).sorted.toArray

   friends.iterator.map(i => (i, userId) ->
      sorted.view.filter(x => x != i &&
         filter.isPresent(Utils.pack(i,x))).toArray)
}
```

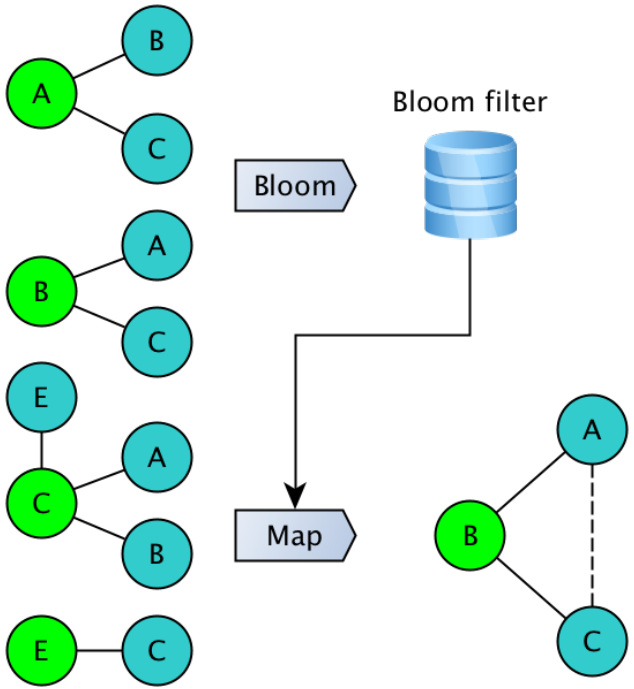
Trial 3: Load the filter

```
@transient lazy val filter = BloomFilterHolder.getModel(
        "/graph/bloomfilter")


def crossGraphFiltered(userId: Long, friends: Seq[Long]) = {
   val sorted = friends.view.filter(_ < userId).sorted.toArray

   friends.iterator.map(i => (i, userId) ->
      sorted.view.filter(x => x != i &&
         filter.isPresent(Utils.pack(i,x))).toArray)
}
```

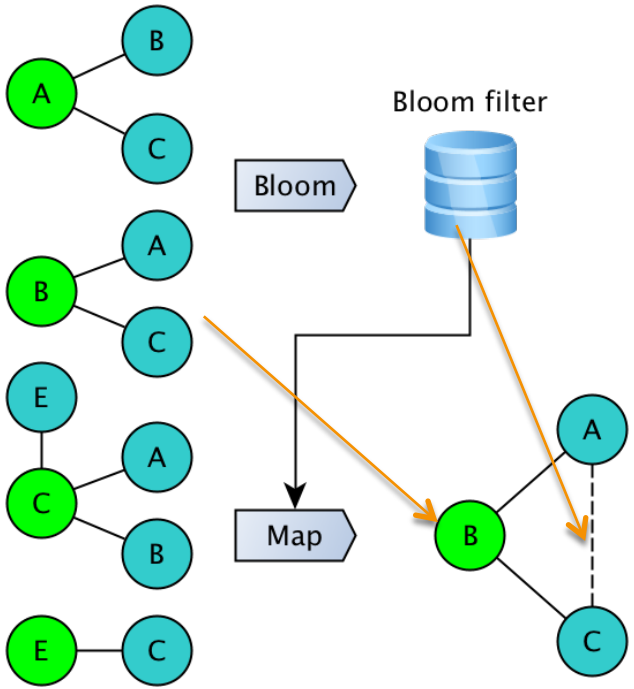

FINALLY WE'VE GOT ALL EGO-SUBGRAPHS
IN LESS THAN AN HOUR!!!
imgflip.com


WHAT IF I TOLD YOU
YOU DON'T NEED TO SHUFFLE ANYTHING AT ALL?
imgflip.com

No shuffle approach



No shuffle approach



Where it works and it doesn’t?

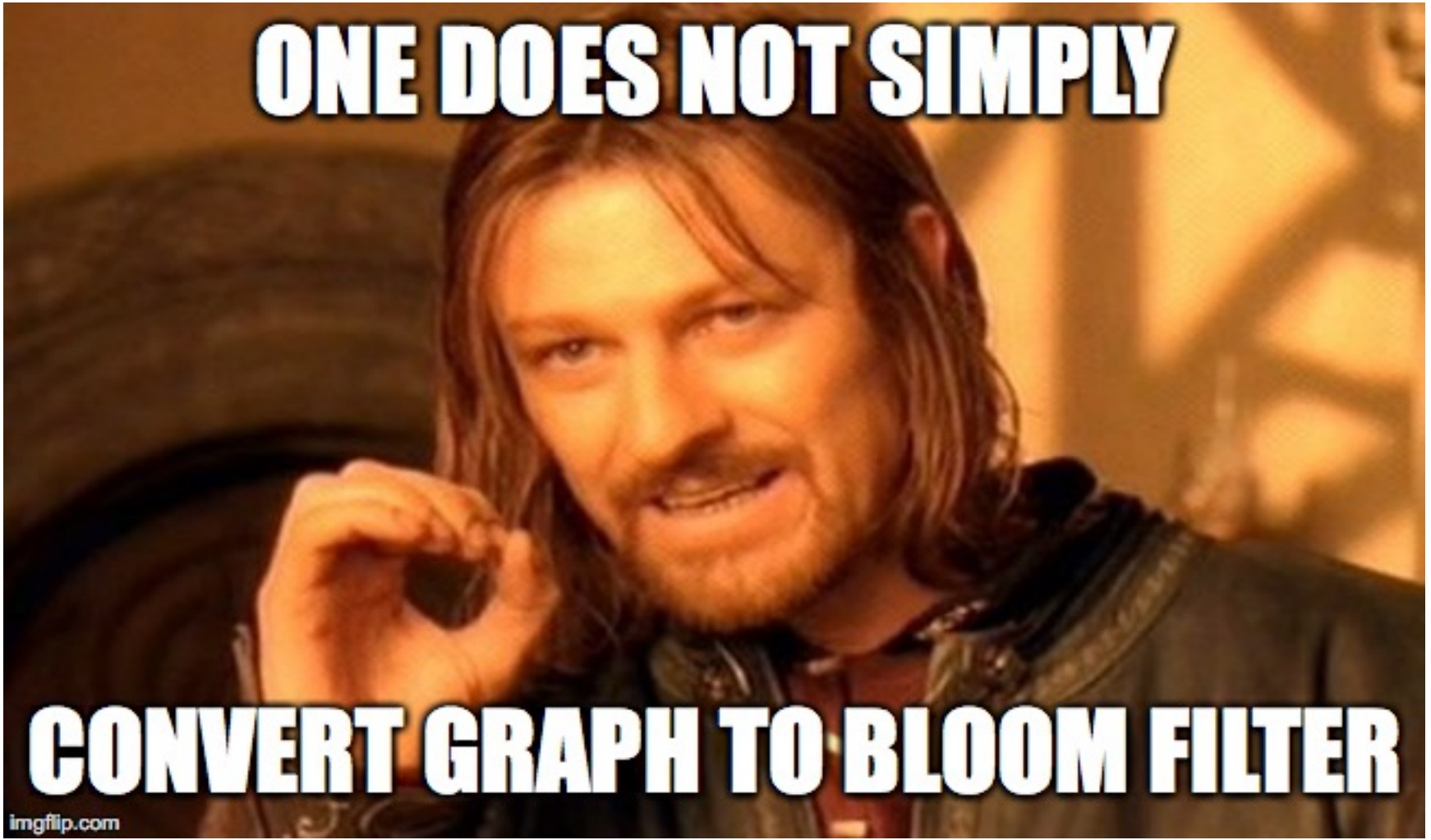
- Triangle counting, clustering coefficient, density estimation etc. ✔

Where it works and it doesn’t?

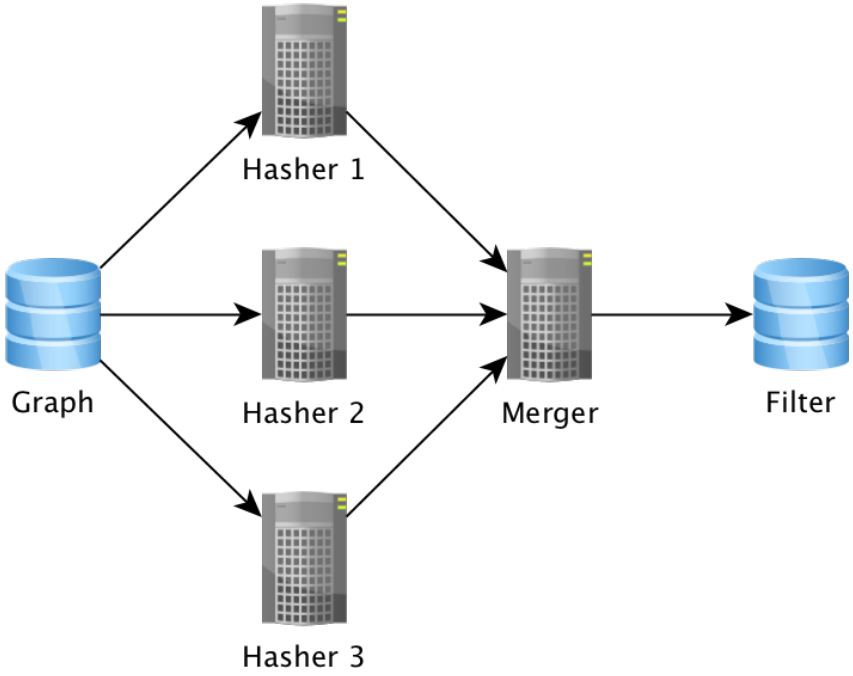
- Triangle counting, clustering coefficient, density estimation etc. ✔
- Connected components ✖

Where it works and it doesn’t?

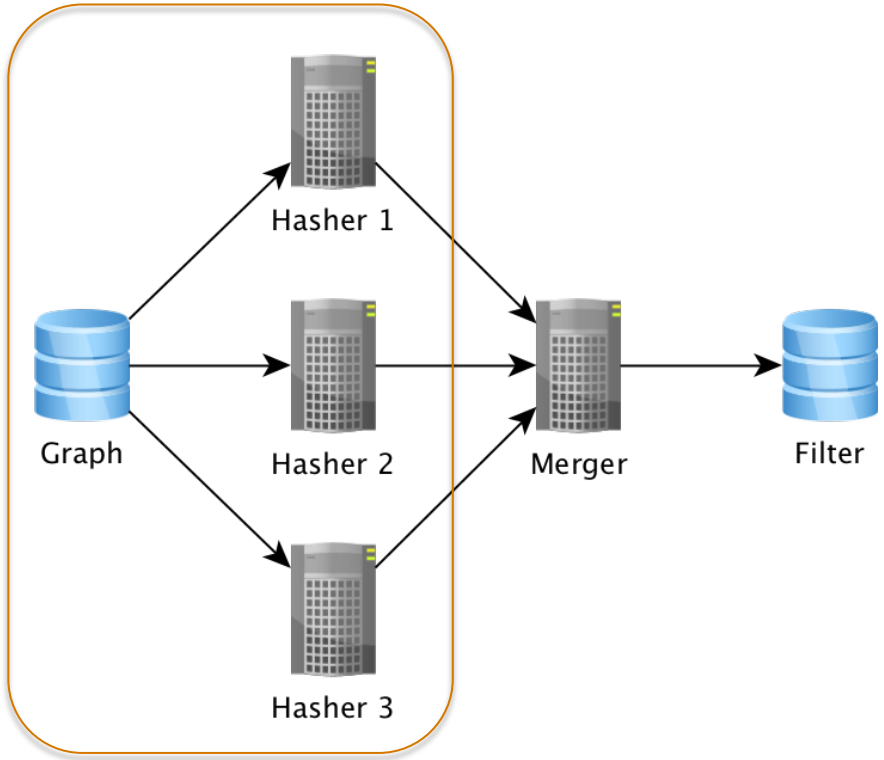
- Triangle counting, clustering coefficient, density estimation etc. ✔
- Connected components ✖
- Clustering, community detection ?


ONE DOES NOT SIMPLY
CONVERT GRAPH TO BLOOM FILTER
imgflip.com

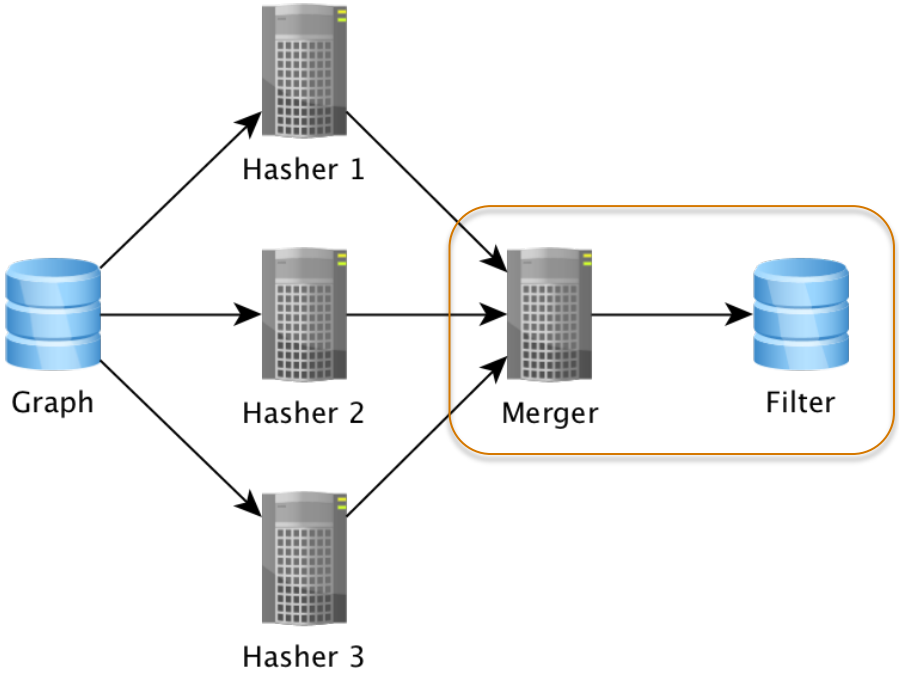
How to construct a filter: current schema



How to construct a filter: current schema



How to construct a filter: current schema



Problems with current schema

- Merger is a bottleneck

Problems with current schema

- Merger is a bottleneck
- Single file for a filter is a bottleneck

Problems with current schema

- Merger is a bottleneck
- Single file for a filter is a bottleneck
  - File is created by a single worker

Problems with current schema

- Merger is a bottleneck
- Single file for a filter is a bottleneck
  - File is created by a single worker
  - Default HDFS replication policy stores first replica on the worker writing it

Problems with current schema

- Merger is a bottleneck
- Single file for a filter is a bottleneck
  - File is created by a single worker
  - Default HDFS replication policy stores first replica on the worker writing it
  - Default HDFS reading policy choose replica at random

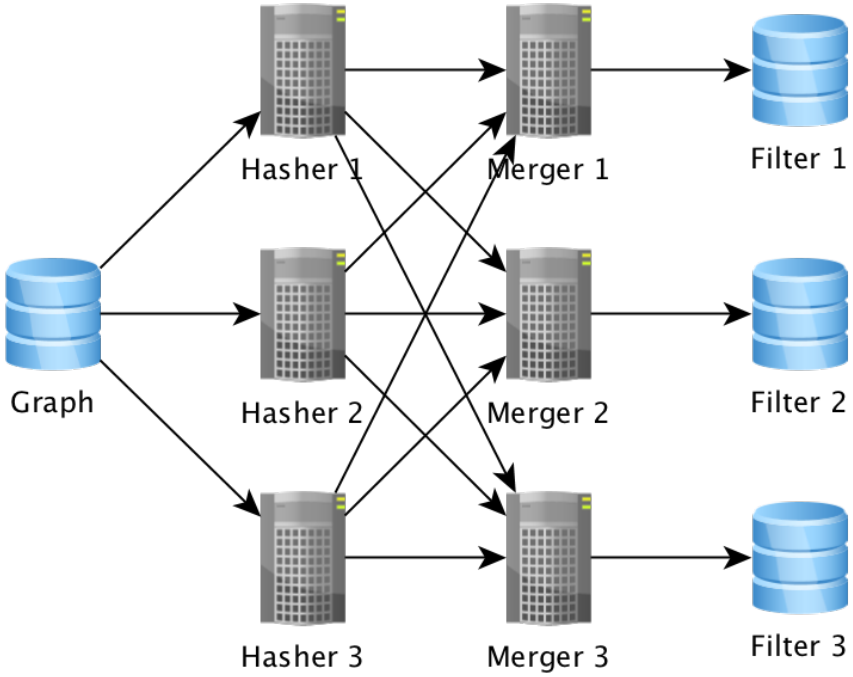
Problems with current schema

- Merger is a bottleneck
- Single file for a filter is a bottleneck
  - File is created by a single worker
  - Default HDFS replication policy stores first replica on the worker writing it
  - Default HDFS reading policy choose replica at random
  - 1/3 of executors use the same server to read filter from

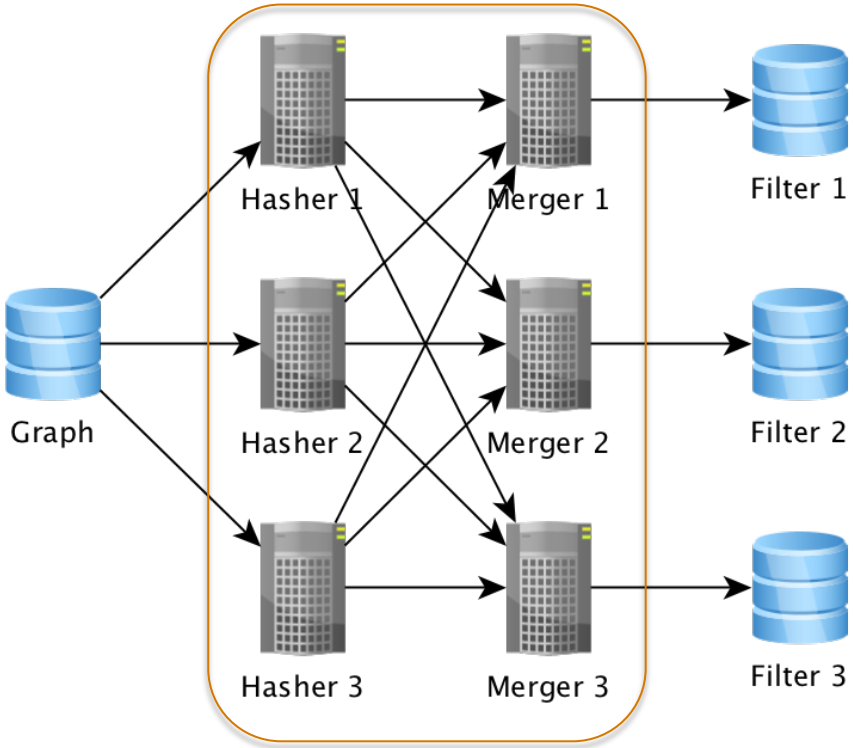
Problems with current schema

- Merger is a bottleneck
- Single file for a filter is a bottleneck
  - File is created by a single worker
  - Default HDFS replication policy stores first replica on the worker writing it
  - Default HDFS reading policy choose replica at random
  - 1/3 of executors use the same server to read filter from
  - Could be mitigated by increasing RF

Distributed schema



Distributed schema



Wrap-up

- Graph is an important source of information
- Graph can explode vastly in analysis
- Using graph properties you can optimize the analytics data flow (symmetry is a good candidate)
- Using probabilistic data structures (sketches) you can achieve an order of magnitude improvement

Thank you for your attention!