Goldman Sachs

Scala Collections Performance

GS.com/Engineering
March, 2015
Craig Motlin

we BUILD

Who am I?

Goldman Sachs

- Engineer at Goldman Sachs
- Technical lead for GS Collections
  - A replacement for the Java Collections Framework

Goldman Sachs

Goals

- Scala programs ought to perform as well as Java programs, but Scala is a little slower

Goals

From RedBlackTree.scala

```
/*
 * Forcing direct fields access using the @inline annotation
 * helps speed up various operations (especially
 * smallest/greatest and update/delete).
 *
 * Unfortunately the direct field access is not guaranteed to
 * work (but works on the current implementation of the Scala
 * compiler).
 *
 * An alternative is to implement the these classes using plain
 * old Java code...
 */
```

Goldman Sachs

Goals

- Scala programs ought to perform as well as Java programs, but Scala is a little slower
- Highlight a few performance problems that matter to me
- Suggest fixes, borrowing ideas and technology from GS Collections

Goldman Sachs

Goals

GS Collections and Scala's Collections are similar

- Mutable and immutable interfaces with common parent
- Similar iteration patterns at hierarchy root `Traversable` and `RichIterable`
- Lazy evaluation (`view`, `asLazy`)
- Parallel-lazy evaluation (`par`, `asParallel`)

Goldman Sachs

Agenda

GS Collections and Scala's Collections are different

- Persistent data structures
- Hash tables
- Primitive specialization
- Fork-join

Goldman Sachs

Persistent Data Structures

Goldman Sachs

Persistent Data Structures

- Mutable collections are similar in GS Collections and Scala – though we'll look at some differences
- Immutable collections are quite different
- Scala's immutable collections are **persistent**
- GS Collections' immutable collections are not

Goldman Sachs

Persistent Data Structures

Wikipedia:

> In computing, a **persistent data structure** is a data structure that always preserves the previous version of itself when it is modified.

Goldman Sachs

Persistent Data Structures

Wikipedia:

> In computing, a **persistent data structure** is a data structure that always preserves the previous version of itself when it is **modified**.

Operations yield new updated structures when they are "modified"

Goldman Sachs

Persistent Data Structures

- Typical examples: List and Sorted Set

Goldman Sachs

Persistent Data Structures

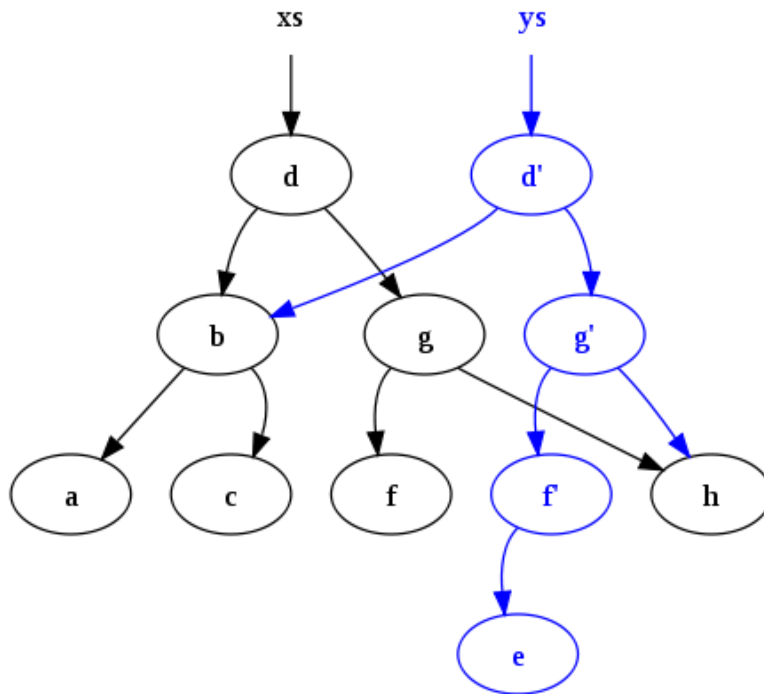
- Typical examples: List and **Sorted Set**
- "Add" by constructing O(log n) new nodes from the leaf to a new root.

- Scala sorted set → binary tree
- GSC immutable sorted set → covered later

Goldman Sachs

Persistent Data Structures

Wikipedia:



Goldman Sachs

Persistent Data Structures

- Typical examples: **List** and Sorted Set
- “Prepend” by constructing a new head in O(1) time. LIFO structure.

- Scala immutable list → linked list or stack
- GSC immutable list → array backed

Goldman Sachs

Persistent Data Structures

- Extremely important in purely functional languages
- All collections are immutable
- Must have good runtime complexity
- No one seems to miss them in GS Collections

Goldman Sachs

Persistent Data Structures

- Proposal: Mutable, **Persistent Immutable**, and **Plain Immutable**
- Mutable: Same as always
- Persistent: Use when you want structural sharing
- (Plain) Immutable: Use when you're done mutating or when the data never mutates

Goldman Sachs

Persistent Data Structures

Plain old immutable data structures

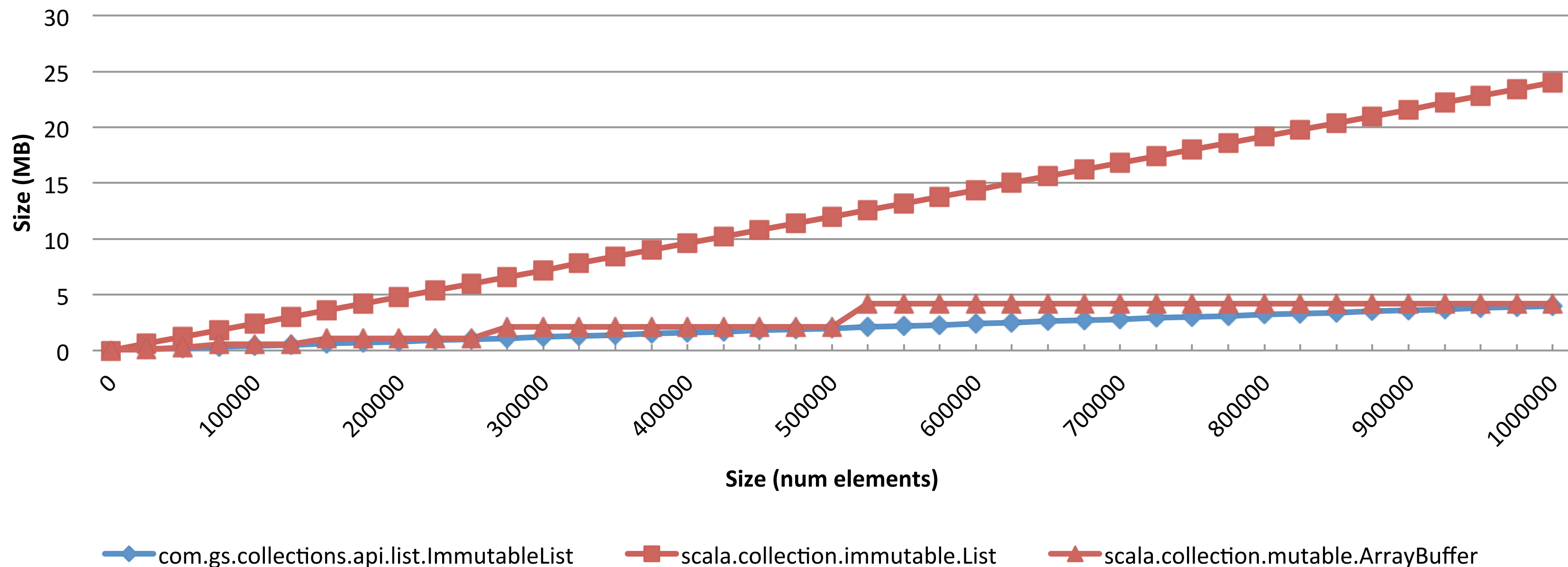
- Not persistent (no structural sharing)
- “Adding” is much slower: O(n)
- Speed benefits for everything else
- Huge memory benefits

Goldman Sachs

Persistent Data Structures

- Immutable array-backed list
- Immutable → trimmed array
- No nodes → 1/6 memory
- Array backed → cache locality, should parallelize well

Persistent Data Structures



Measured with Java 8 64-bit and compressed oops

Goldman Sachs

Persistent Data Structures

Performance assumptions:

- Iterating through an array should be faster than iterating through a linked list
- Linked lists won't parallelize well with `.par`
- No surprises in the results – so we'll skip

github.com/goldmansachs/gs-collections/tree/master/jmh-tests

github.com/goldmansachs/gs-collections/tree/master/memory-tests

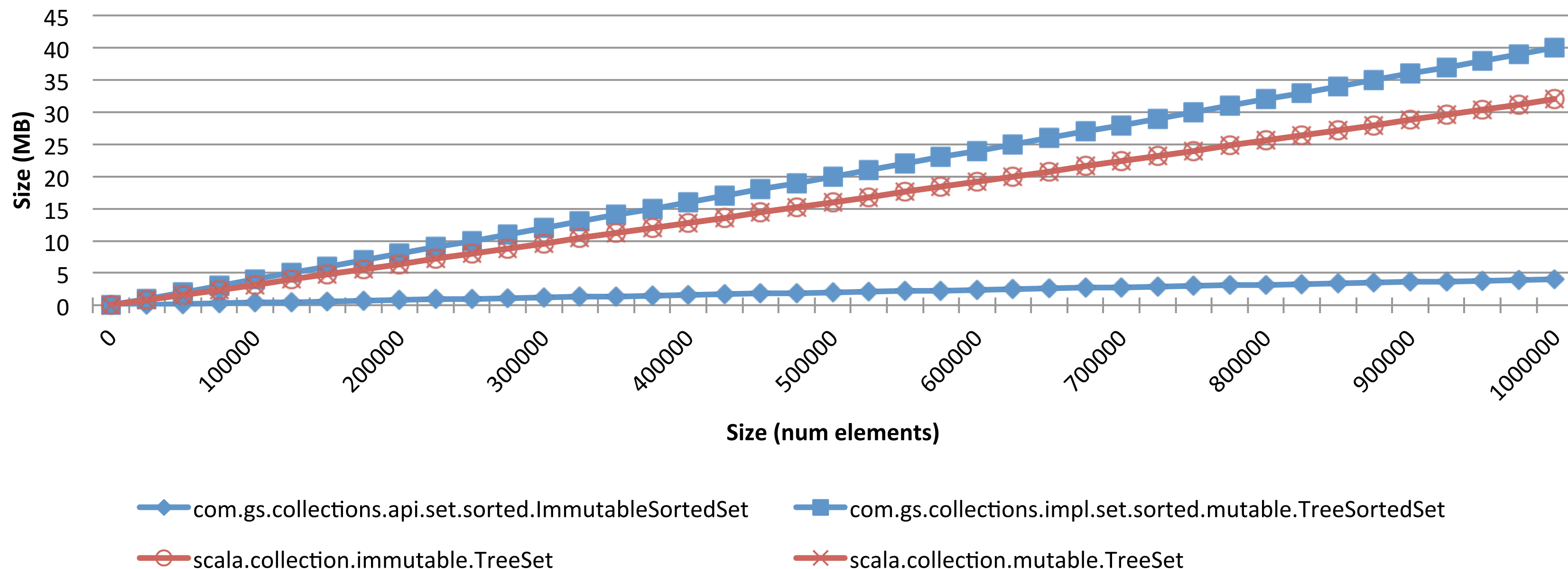
Goldman Sachs

Persistent Data Structures

- Immutable array-backed sorted set
- Immutable → trimmed, sorted array
- No nodes → ~1/8 memory
- Array backed → cache locality

Goldman Sachs

Persistent Data Structures



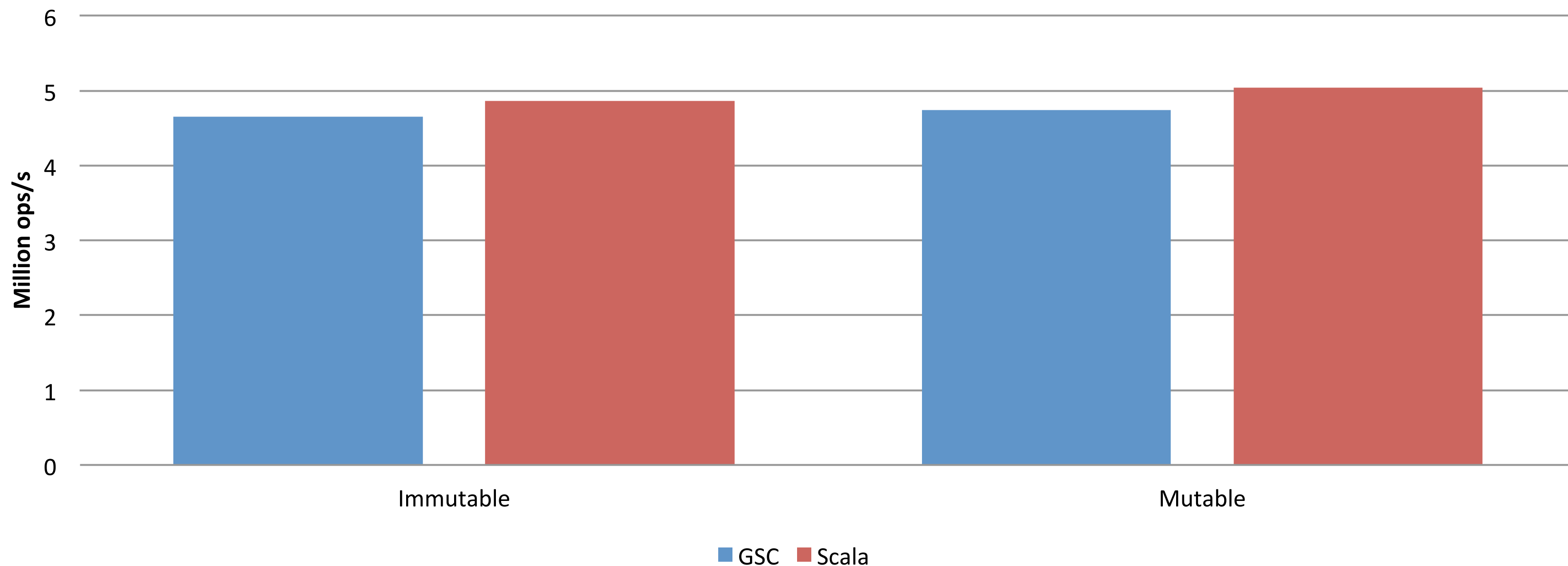
Goldman Sachs

Persistent Data Structures

- Assumptions about contains:
- May be faster when it's a binary search in an array (good cache locality)
- Will be about the same in mutable / immutable tree (all nodes have same structure)

Goldman Sachs

Persistent Data Structures



Goldman Sachs

Iteration

Goldman Sachs

Persistent Data Structures

```
val scalaImmutable: immutable.TreeSet[Int] =
  immutable.TreeSet.empty[Int] ++ (0 to SIZE)

def serial_immutable_scala(): Unit =
{
  val count: Int = this.scalaImmutable
    .view
    .filter(each => each % 10000 != 0)
    .map(String.valueOf)
    .map(Integer.valueOf)
    .count(each => (each + 1) % 10000 != 0)
  if (count != 999800)
  {
    throw new AssertionError
  }
}
```

Goldman Sachs

Persistent Data Structures

```
val scalaImmutable: immutable.TreeSet[Int] =
  immutable.TreeSet.empty[Int] ++ (0 to SIZE)

def serial_immutable_scala(): Unit =
{
  val count: Int = this
    .view
    .filter(each => each % 10000 != 0)
    .map(String.valueOf)
    .map(Integer.valueOf)
    .count(each => (each + 1) % 10000 != 0)
  if (count != 999800)
  {
    throw new AssertionError
  }
}
```

Lazy evaluation so we don't create intermediate collections

Goldman Sachs

Persistent Data Structures

```
val scalaImmutable: immutable.TreeSet[Int] =
  immutable.TreeSet.empty[Int] ++ (0 to SIZE)

def serial_immutable_scala(): Unit =
{
  val count: Int = this
    .view
    .filter(each => each % 10000 != 0)
    .map(String.valueOf)
    .map(Integer.valueOf)
    .count(each => (each + 1) % 10000 != 0)
  if (count != 999800)
  {
    throw new AssertionError
  }
}
```

Lazy evaluation so we don't create intermediate collections

Terminating in .toList or .toBuffer would be dominated by collection creation

Goldman Sachs

Persistent Data Structures

```
private final ImmutableSortedSet<Integer> gscImmutable =
  SortedSets.immutable.withAll(Interval.zeroTo(SIZE));
@Benchmark
public void serial_immutable_gsc()
{
  int count = this.gscImmutable
    .asLazy()
    .select(each -> each % 10_000 != 0)
    .collect(String::valueOf)
    .collect(Integer::valueOf)
    .count(each -> (each + 1) % 10_000 != 0);
  if (count != 999_800)
  {
    throw new AssertionError();
  }
}
```

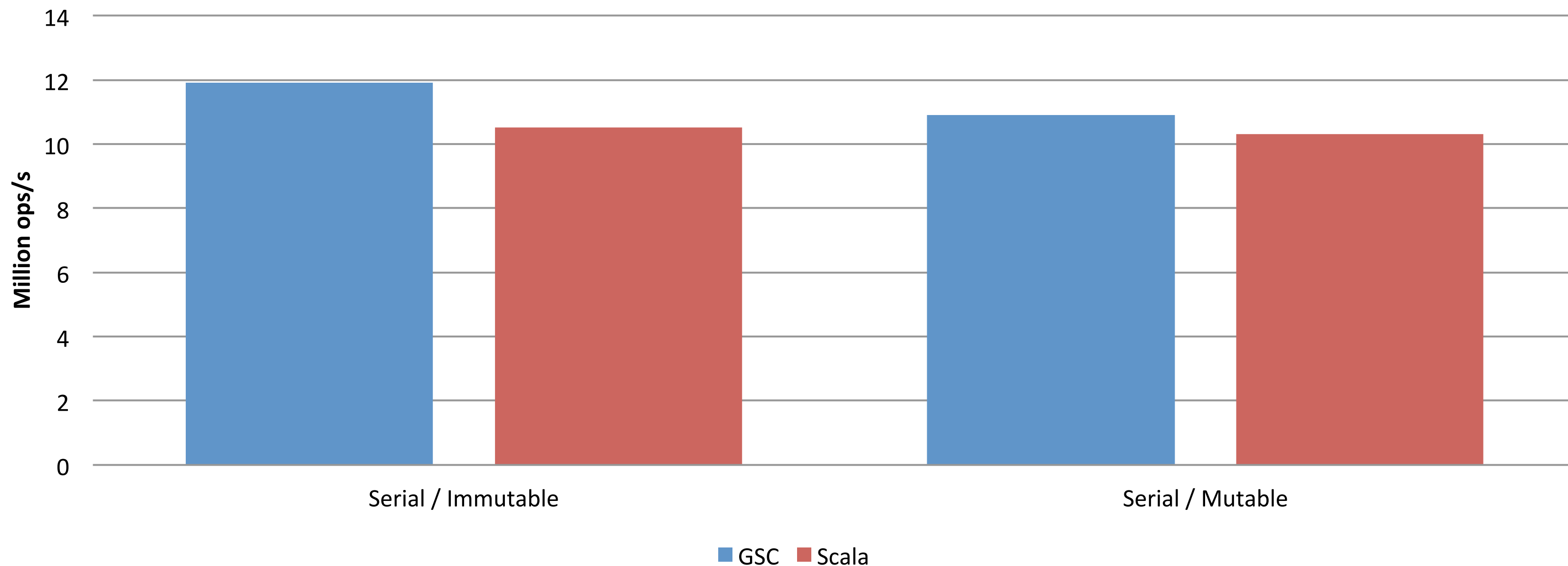
Goldman Sachs

Persistent Data Structures

- Assumption: Iterating over an array is somewhat faster

Goldman Sachs

Persistent Data Structures



Goldman Sachs

Persistent Data Structures

Next up, parallel-lazy evaluation

```
this.scalaImmutable.view →
this.scalaImmutable.par

this.gscImmutable.asLazy() →
this.gscImmutable.asParallel(this.executorService, BATCH_SIZE)
```

Goldman Sachs

Persistent Data Structures

- Assumption: Scala's tree should parallelize moderately well
- Assumption: GSC's array should parallelize very well

Goldman Sachs

Persistent Data Structures

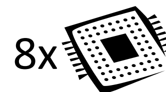
Sorted Set Iteration throughput (higher is better)

Million ops/s

100
90
80
70
60
50
40
30
20
10
0

Serial / Immutable
Serial / Mutable
Parallel / Immutable

GSC
Scala

8x 

Measured on an 8 core Linux VM
Intel Xeon E5-2697 v2

Goldman Sachs

Persistent Data Structures

- Scala's `immutable.TreeSet` doesn't override `.par`, so parallel is slower than serial
- Some tree operations (like `filter`) are hard to parallelize
- `TreeSet.count()` should be easy to parallelize using fork/join with some work
- New assumption: Mutable trees won't parallelize well either

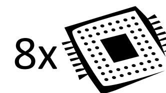
Goldman Sachs

Persistent Data Structures

Sorted Set Iteration throughput (higher is better)

	GSC	Scala
Serial / Immutable		
Serial / Mutable		
Parallel / Immutable		
Parallel / Mutable		

Million ops/s (axis 0–100)



Measured on an 8 core Linux VM
Intel Xeon E5-2697 v2

Goldman Sachs

Persistent Data Structures

- GS Collections follow up:
  - `TreeSortedSet` delegates to `java.util.TreeSet`. Write our own tree and override `.asParallel()`
- Scala follow up:
  - Override `.par`

Goldman Sachs

Persistent Data Structures

- Proposal: Mutable, Persistent, and **Immutable**
- All three are useful in different cases
- How common is it to share structure, really?

Goldman Sachs

Hash Tables

Goldman Sachs

Hash Tables

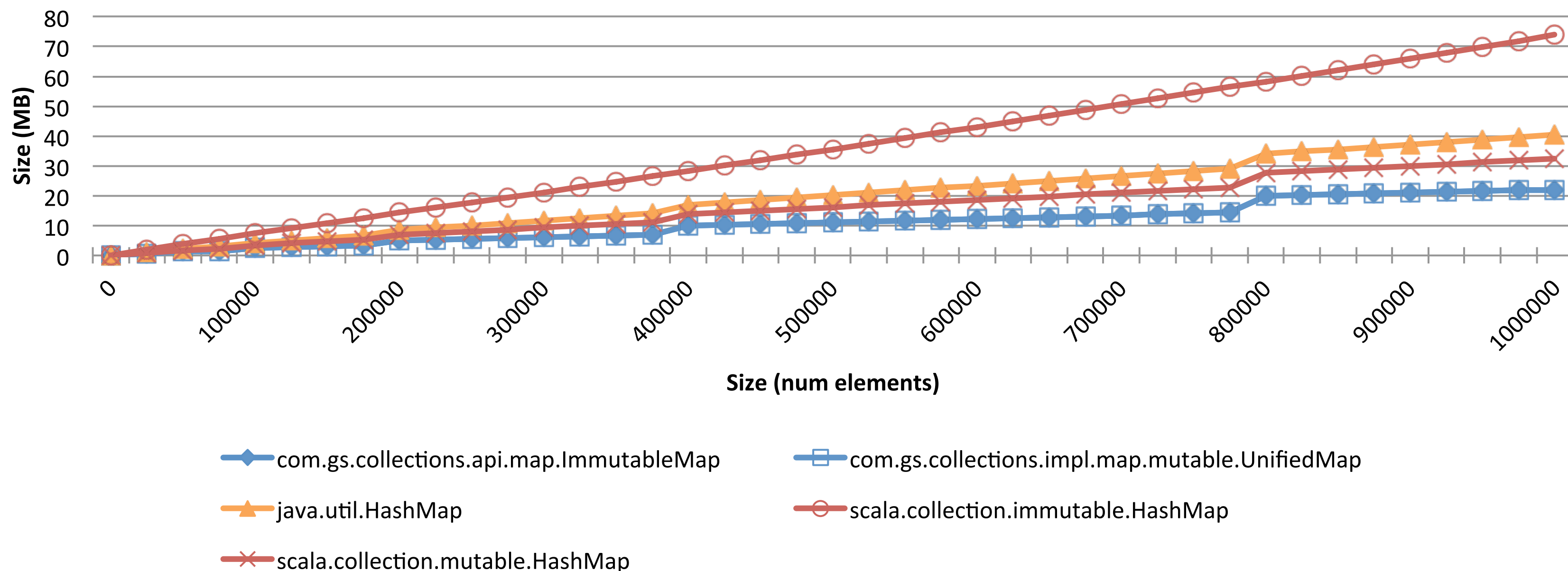
- Scala's `immutable.HashMap` is a hash array mapped trie (persistent structure)
- Wikipedia: "achieves almost hash table-like speed while using memory much more economically"
- Let's see

Goldman Sachs

Hash Tables

- Scala's mutable.HashMap is backed by an array of Entry pairs
- java.util.HashMap.Entry caches the hashcode
- UnifiedMap<K, V> is backed by Object[], flattened, alternating K and V
- ImmutableUnifiedMap is backed by UnifiedMap

Hash Tables

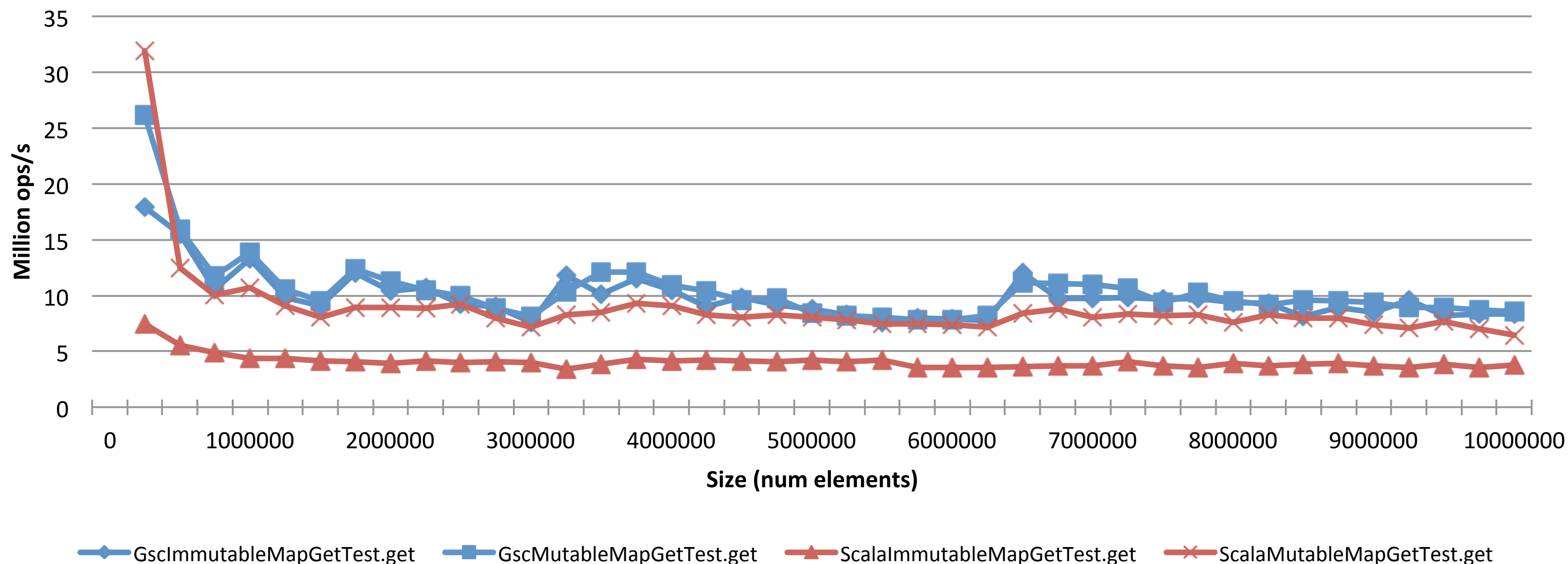


Hash Tables

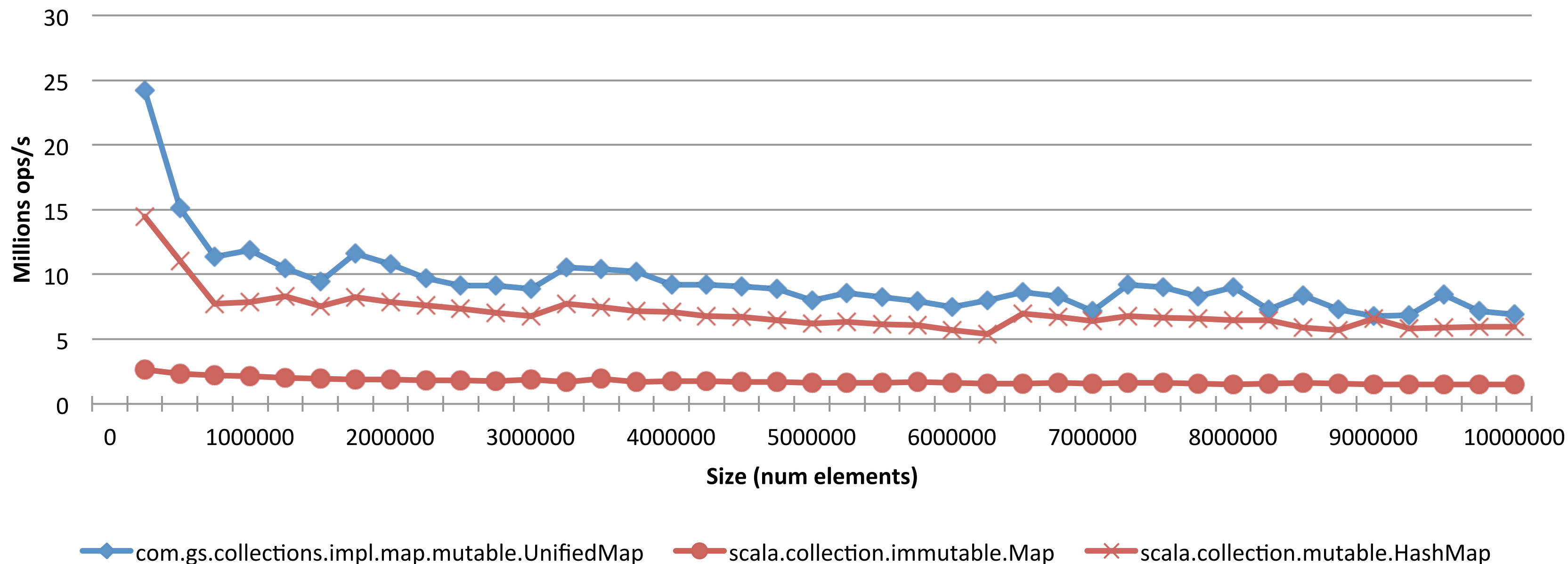
```
@Benchmark
public void get()
{
  int localSize = this.size;
  String[] localElements = this.elements;
  Map<String, String> localScalaMap = this.scalaMap;

  for (int i = 0; i < localSize; i++)
  {
    if (!localScalaMap.get(localElements[i]).isDefined())
    {
      throw new AssertionError(i);
    }
  }
}
```

Goldman Sachs

Hash Tables



Hash Tables



Goldman Sachs

Hash Tables

```
@Benchmark
public mutable.HashMap<String, String> mutableScalaPut()
{
    int localSize = this.size;
    String[] localElements = this.elements;

    mutable.HashMap<String, String> map =
        new PresizableHashMap<>(localSize);

    for (int i = 0; i < localSize; i++)
    {
        map.put(localElements[i], "dummy");
    }
    return map;
}
```

Goldman Sachs

Hash Tables

HashMap ought to have a constructor for pre-sizing

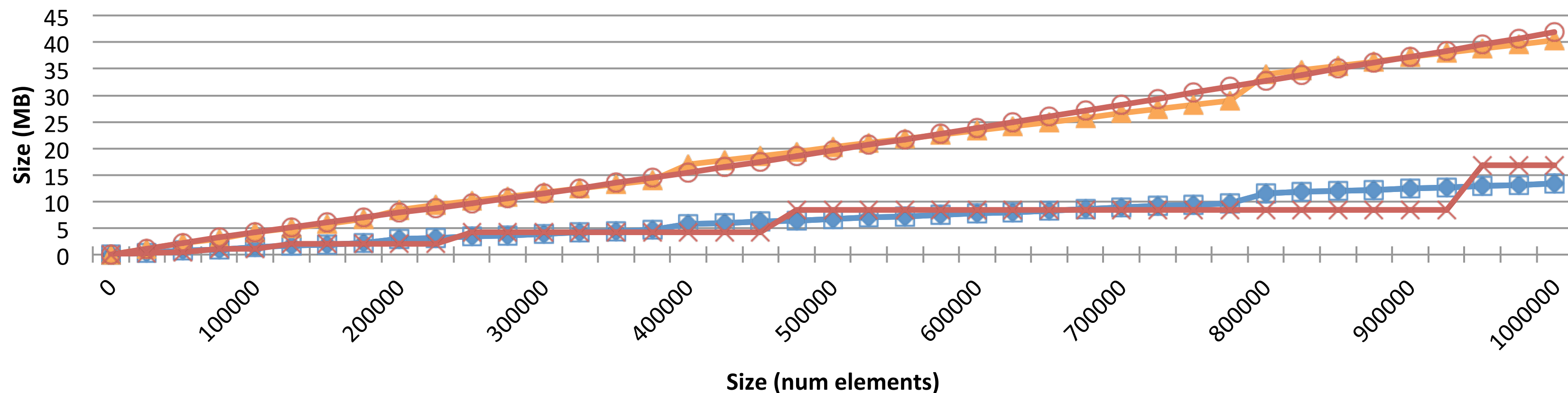
```
class PresizableHashMap[K, V](val _initialSize: Int)
  extends scala.collection.mutable.HashMap[K, V]
{
  private def initialCapacity =
    if (_initialSize == 0) 1
    else smallestPowerOfTwoGreaterThan(
      (_initialSize.toLong * 1000 / _loadFactor).asInstanceOf[Int])

    private def smallestPowerOfTwoGreaterThan(n: Int): Int =
      if (n > 1) Integer.highestOneBit(n - 1) << 1 else 1

    table = new Array(initialCapacity)
    threshold = ((initialCapacity.toLong * _loadFactor) / 1000).toInt
}
```

Goldman Sachs

Hash Tables

- `scala.collection.mutable.HashSet` is backed by an array using open-addressing
- `java.util.HashSet` is implemented by delegating to a `HashMap`.
- `UnifiedSet` is backed by `Object[]`, either elements or arrays of collisions
- `ImmutableUnifiedSet` is backed by `UnifiedSet`

Hash Tables



Goldman Sachs

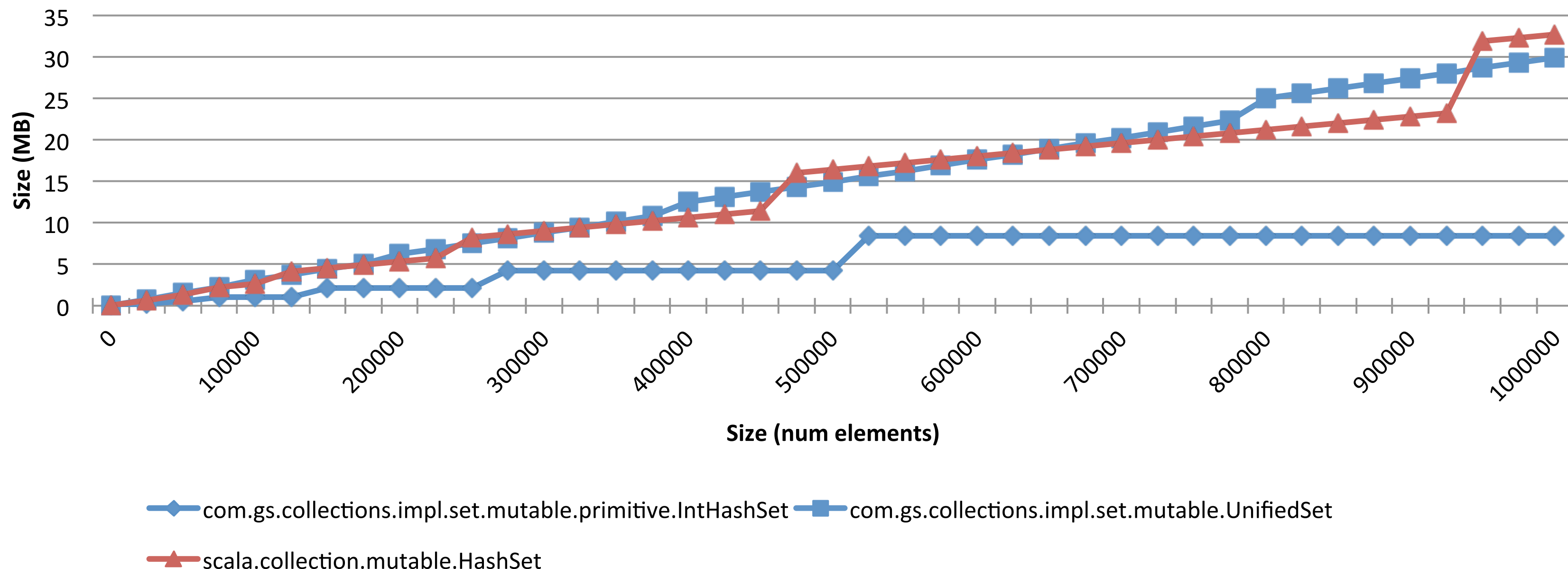
Primitive Specialization

Goldman Sachs

Primitive Specialization

- Boxing is expensive
  - Reference + Header + alignment
- Scala has specialization, but most of the collections are not specialized
- If you cannot afford wrappers you can:
  - Use primitive arrays (only for lists)
  - Use a Java Collections library

Primitive Specialization



Goldman Sachs

Primitive Specialization

- Proposal: Primitive lists, sets, and maps in Scala
- Not Traversable – outside the collections hierarchy
- Fix Specialization on Functions (lambdas)
  - Want for-comprehensions to work well

Goldman Sachs

Fork/Join

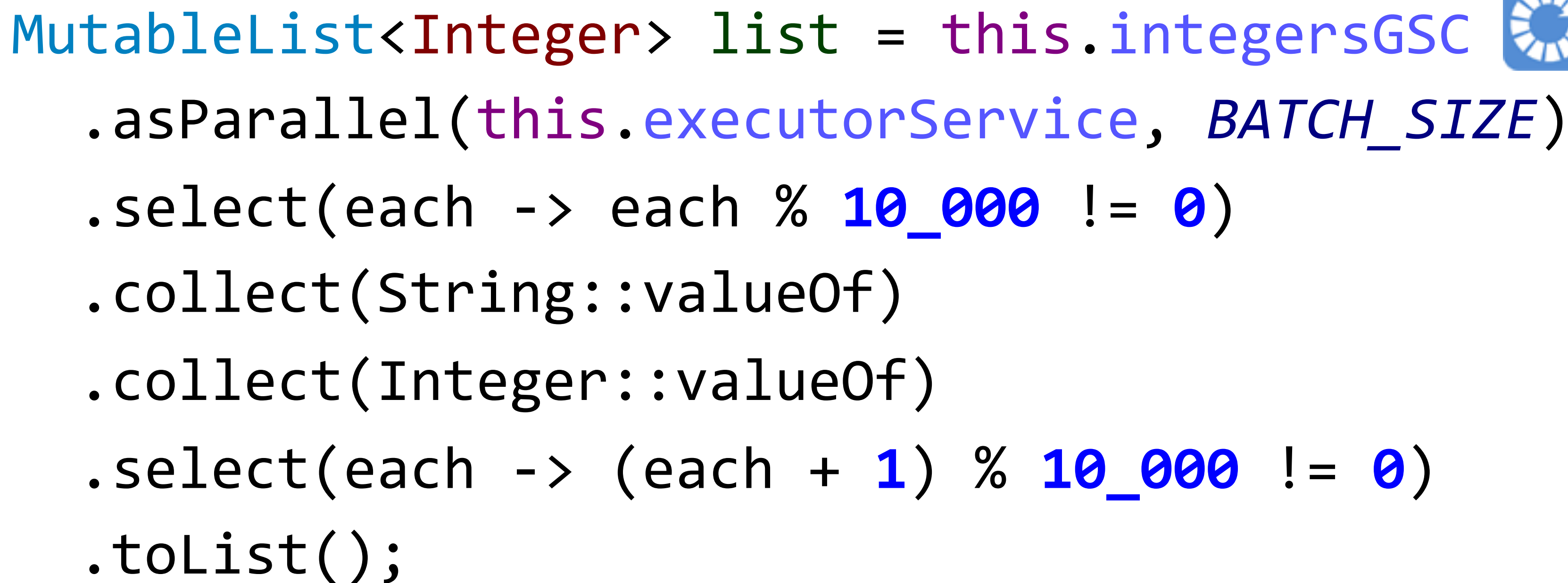
Goldman Sachs

Parallel / Lazy / Scala

```
val list = this.integers
    .par
    .filter(each => each % 10000 != 0)
    .map(String.valueOf)
    .map(Integer.valueOf)
    .filter(each => (each + 1) % 10000 != 0)
    .toBuffer

Assert.assertEquals(999800, list.size)
```

Goldman Sachs

Parallel / Lazy / GSC




```
Verify.assertSize(999_800, list);
```

Goldman Sachs

Parallel / Lazy / JDK

```
List<Integer> list = this.integersJDK
    .parallelStream()
    .filter(each -> each % 10_000 != 0)
    .map(String::valueOf)
    .map(Integer::valueOf)
    .filter(each -> (each + 1) % 10_000 != 0)
    .collect(Collectors.toList());

Verify.assertSize(999_800, list);
```

Goldman Sachs

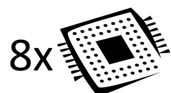
Stacked computation ops/s (higher is better)

60
50
40
30
20
10
0

Serial Lazy

Parallel Lazy

Java 8 GS Collections Scala



Goldman Sachs

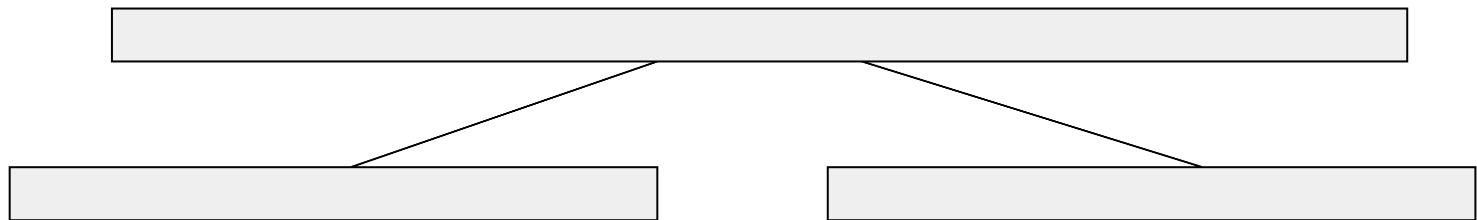
Fork-Join Merge

- Intermediate results are merged in a tree
- Merging is O(n log n) work and garbage

Goldman Sachs

Fork-Join Merge

- Amount of work done by last thread is O(n)



Goldman Sachs

Parallel / Lazy / GSC

```java
MutableList<Integer> list = this.integersGSC
    .asParallel(this.executorServi
    .select(each -> each % 10_000
    .collect(String::valueOf)
    .collect(Integer::valueO
    .select(each -> (   ach + 1) % 10_000 != 0)
    .toList();

Verify.assertSize(999_800, list);
```

ParallelIterable.toList() returns a CompositeFastList, a List with O(1) implementation of addAll()

Goldman Sachs

Parallel / Lazy / GSC

```java
public final class CompositeFastList<E> {
  private final FastList<FastList<E>> lists =
    FastList.newList();

  public boolean addAll(Collection<? extends E> collection) {
    FastList<E> collectionToAdd = collection instanceof FastList
      ? (FastList<E>) collection
      : new FastList<E>(collection);
    this.lists.add(collectionToAdd);
    return true;
  }
  ...
}
```

Goldman Sachs

CompositeFastList Merge

- Merging is O(1) work per batch

CFL

Goldman Sachs

Fork/Join

- Fork-join is general purpose but always requires merge work
- We can get better performance through specialized data structures meant for combining

Goldman Sachs

Summary

GS Collections and Scala's Collections are different

- Persistent data structures
- Hash tables
- Primitive specialization
- Fork-join

Goldman Sachs

More Information

@motlin

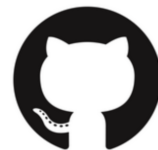
@GoldmanSachs

stackoverflow.com/questions/tagged/gs-collections

github.com/goldmansachs/gs-collections/tree/master/jmh-tests

github.com/goldmansachs/gs-collections/tree/master/memory-tests

github.com/goldmansachs/gs-collections-kata



infoq.com/presentations/java-streams-scala-parallel-collections

we
BUILD

Learn more at GS.com/Engineering

© 2015 Goldman Sachs. This presentation should not be relied upon or considered investment advice. Goldman Sachs does not warrant or guarantee to anyone the accuracy, completeness or efficacy of this presentation, and recipients should not rely on it except at their own risk. This presentation may not be forwarded or disclosed except with this disclaimer intact.

Goldman Sachs

Q&A