Collections for Concurrency

Venkat Subramaniam
venkats@agiledeveloper.com
@venkat_s

Topics

JDK Collections

Synchronized Collections

Concurrent Collections

Immutable Collections

Google Guava

Practicality of Immutability

Design of data structures for immutability

Tries

Concurrency & Collections

It’s hard to realize a OO app without using collections

Collections were introduced in JDK 1.0, but has gone through quite some evolution

So, fundamental, yet evolving, why?

What's Wrong?

Remember JDK 1.0 collections like Vector?

They were provided for thread-safety

That is good, but did not consider performance in mind

Overly conservative locking resulted in poor performance

Newer Collections

Then a new wave of collections were introduced in JDK 1.2

ArrayList instead of Vector

What's different?

ArrayList

Faster than Vector, but did not provide thread-safety by default

Totally unsynchronized

Vector vs. ArrayList

```java
import java.util.ArrayList;
import java.util.List;
import java.util.Vector;

public class VecVsArray {
  public static void addElements(List<Integer> list) {
    for(int i = 0; i < 1000000; i++) {
      list.add(i);
    }
  }

  public static void main(String[] args) {
    final long start1 = System.nanoTime();
    addElements(new Vector<Integer>());
    final long end1 = System.nanoTime();

    final long start2 = System.nanoTime();
    addElements(new ArrayList<Integer>());
    final long end2 = System.nanoTime();

    System.out.println("Vector    time " + (end1 - start1));
    System.out.println("ArrayList time " + (end2 - start2));
  }
}
```

```
Vector    time 511674000
ArrayList time 285836000
```

Synchronized Collection

You can wrap unsynchronized collections through a synchronized wrapper

Collections.synchronizedList(...);

```
ArrayList     time 306899000
Syn ArrayList time 453112000
```

Concurrency Violation

```
public class UseHashMap {
  public static void sleep(int time) {
    try { Thread.sleep(time); } catch(Exception ex) {}
  }

  private Map<String, Integer> scores = new HashMap<String, Integer>();

  public void printScores() {
    for(Map.Entry entry : scores.entrySet()) {
      System.out.println(
        String.format("Score for name %s is %d",
          entry.getKey(), entry.getValue()));
      sleep(1000); // simulate computation delay
    }
  }

  public void addScore(String name, int score) {
    scores.put(name, score);
  }

  public static void main(String[] args) {
    final UseHashMap useHashMap = new UseHashMap();
    useHashMap.addScore("Sara", 14);
    useHashMap.addScore("John", 12);

    new Thread(new Runnable() {
      public void run() {
        useHashMap.printScores();
      }
    }).start();

    sleep(1000);
    useHashMap.addScore("Bill", 13);
    System.out.println("Added Bill");
  }
}
```

```
Score for name Sara is 14
Added Bill
Exception in thread "Thread-0" java.util.ConcurrentModificationException
```

Explicit Synchronization

Safe, no exception, but blocking and slow

```
private Map<String, Integer> scores = new HashMap<String, Integer>();

public void printScores() {
  synchronized(scores) {
    for(Map.Entry entry : scores.entrySet()) {
      System.out.println(
        String.format("Score for name %s is %d",
          entry.getKey(), entry.getValue()));
      sleep(1000); // simulate computation delay
    }
  }
}

public void addScore(String name, int score) {
  synchronized(scores) { scores.put(name, score); }
}
```

```
Score for name Sara is 14
Score for name John is 12
Added Bill
```

Thread-Safety vs. Scalability

Synchronized collections provided thread-safety at the expense of scalability or performance

If you're willing to compromise just a little on semantics you can enjoy concurrency and scalability with Concurrent collections

ConcurrentHashMap

You can iterate over the collection and change it at the same time

Be willing to accept slight change in semantics

Does not bend over back to show you concurrent updates

Guarantees you'll never visit same element twice in iteration

No ConcurrentModificationException

Using ConcurrentHashMap

```
private Map<String, Integer> scores
  = new ConcurrentHashMap<String, Integer>();

public void printScores() {
  for(Map.Entry entry : scores.entrySet()) {
    System.out.println(
      String.format("Score for name %s is %d",
        entry.getKey(), entry.getValue()));
    sleep(1000); // simulate computation delay
  }
}

public void addScore(String name, int score) {
  scores.put(name, score);
}
```

```
Score for name Sara is 14
Added Bill
Score for name John is 12
```

Throughput

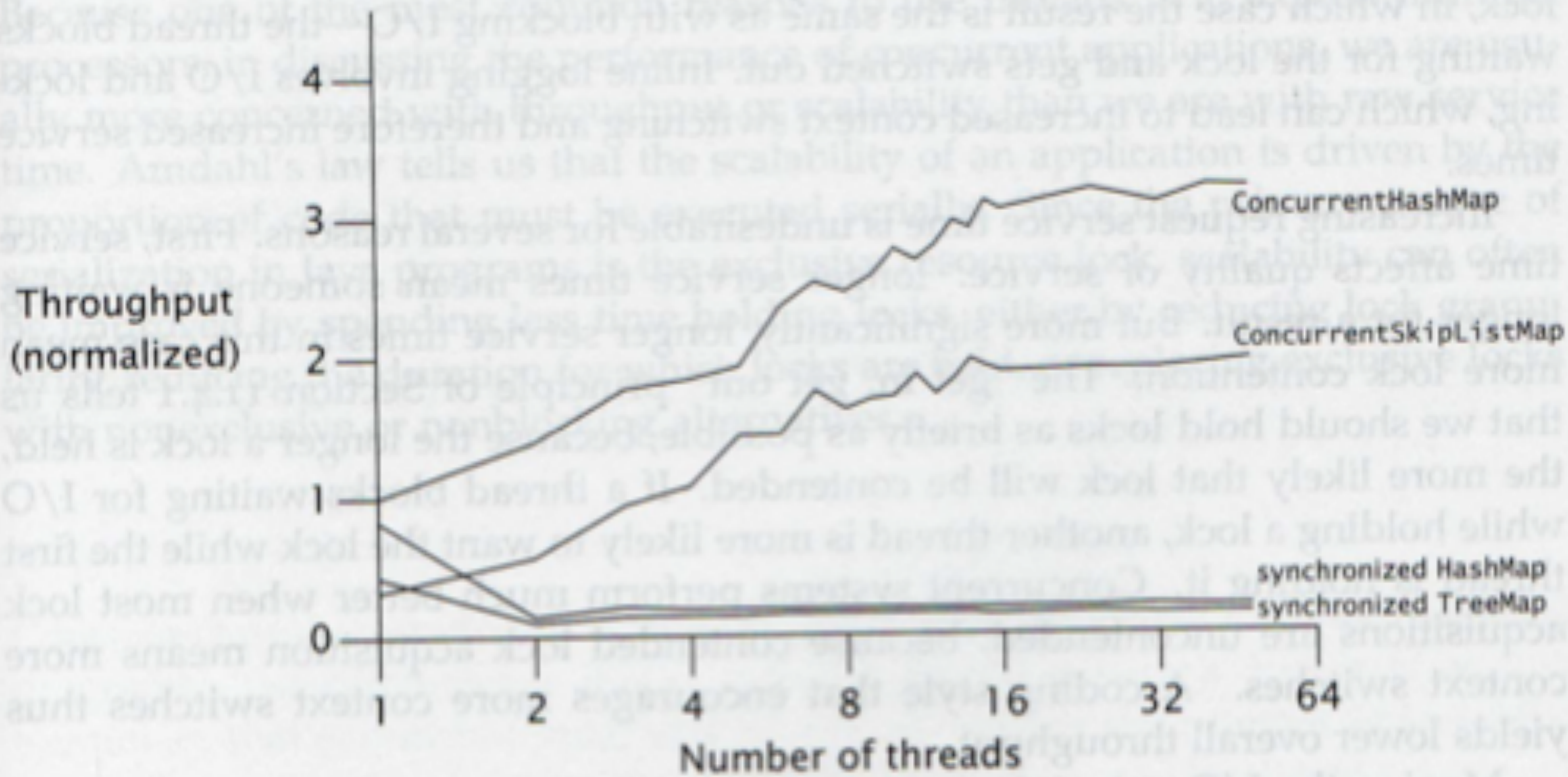
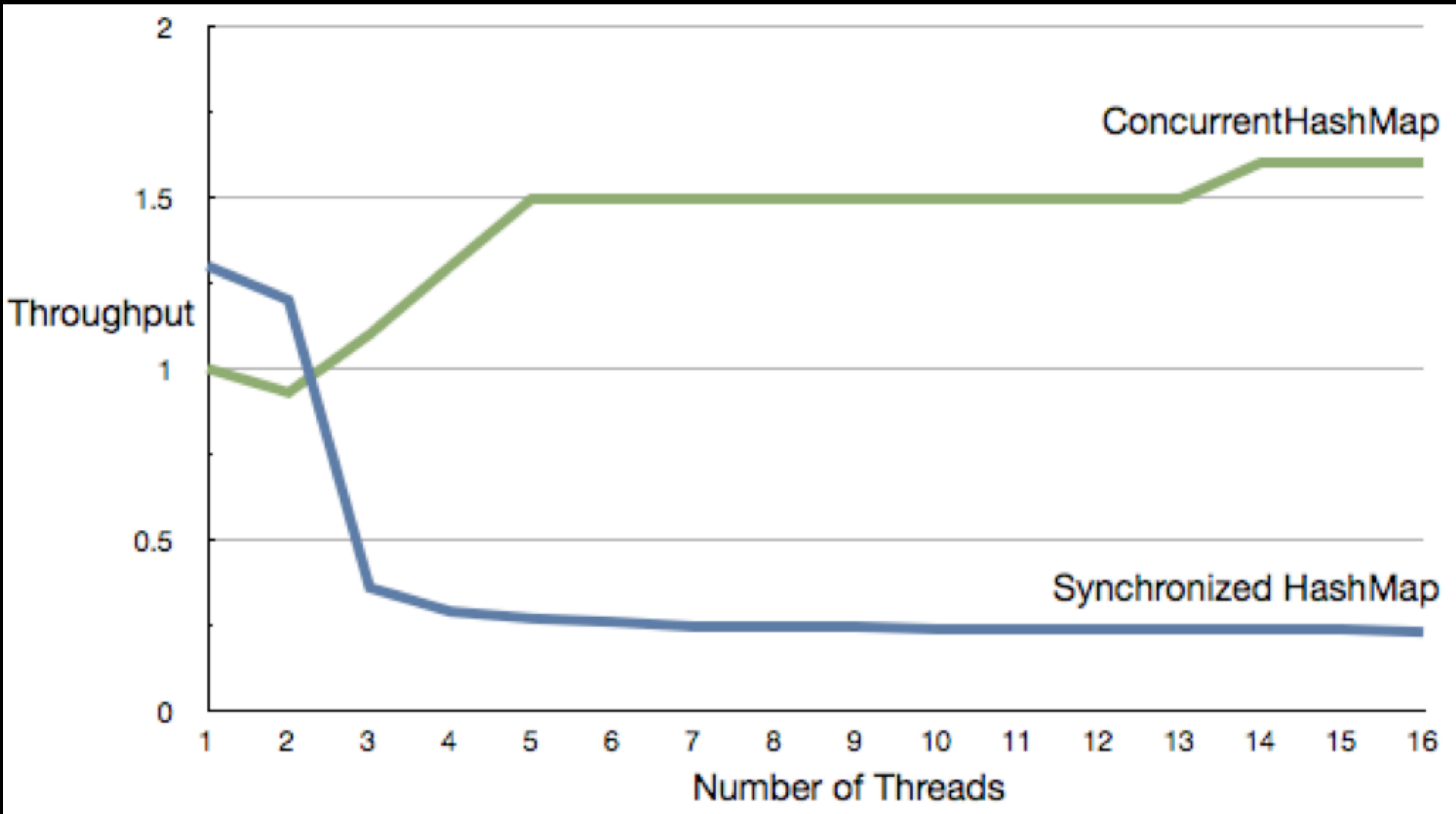


FIGURE 11.3. Comparing scalability of Map implementations.

Source: Java Concurrency in Practice by Brian Goetz, Addison-Wesley

Performance



Source: Programming Concurrency by Venkat Subramaniam, Pragmatic Programmers

Queue Interface

Allows you to peek, poke, remove

Doesn't support blocking operations

For that you can use BlockingQueue

BlockingQueue

Blocks for events with option to timeout

If space not available, block on insert

If element not present, block for arrival on call to remove

Different implementations

- ArrayBlockingQueue (FIFO, bounded)
- DelayQueue
- LinkedBlockingQueue
- PriorityBlockingQueue
- SynchronousQueue (like CSP/ADA rendezvous channel)

BlockingQueue

```
private static BlockingQueue<Integer> scores = new
SynchronousQueue<Integer>();

public static void publisher() throws InterruptedException {
  for(int i = 0; i < 5; i++) {
    System.out.println("putting value " + i);
    scores.put(i);
  }
}

public static void processor() throws InterruptedException {
  while(true) {
    System.out.println("Getting " + scores.take());
    Thread.sleep(1000);
  }
}
```

```
putting value 0
Getting 0
putting value 1
Getting 1
putting value 2
Getting 2
...
```

Dealing With Concurrency

There are two approaches to deal with concurrency

You can take hard measures to provide thread-safety

or

You can remove the problem at the root—make your data structure immutable

Return Immutable Collection

- You don't have to worry about change to your collection outside of your control
- No need to deal with thread-safety issues (internally)
- Good performance

```
public class Car {
  List<Wheel> wheels = new ArrayList<Wheel>();

  Iterator<Wheel> getWheels() {
    return wheels.iterator();
  }
```

```
Iterator<Wheel> getWheels() {
  return Collections.unmodifiableList(wheels).iterator();
}
```

Google Guava

Written as an extension to the Java Collections

Provides greater convenience of use

Greatly favors immutability

Greatly favors concurrency

Very customizable and extensible

Promotes functional style though pure Java API

Google Guava

- Convenience to create instances using factories
- Specialized Collections with MultiMap and MultiSet to hold multiple values
- Promotes Functional Style with Iterable and Predicates

Google Guava

ImmutableSet<E>

ImmutableList<E>

ImmutableMap<K, V>

ImmtableMultiMap<K, V>

ImmutableMultiSet<E>

Using ImmutableList

```
ImmutableList<Integer> numbers =
  ImmutableList.of(1, 5, 3, 6, 8, 9, 6, 4, 7);

System.out.println("Number of elements: " + numbers.size());
System.out.println("Has 6? " + numbers.contains(6));
System.out.println("First index of 6 is " + numbers.indexOf(6));
System.out.println("Last index of 6 is " + numbers.lastIndexOf(6));

System.out.print("Iterating over the list: ");
for(int i : numbers) { System.out.print(i + " "); }
System.out.println("");
```

Using ImmutableList

```
System.out.print("Getting only even numbers: ");

Iterable<Integer> evenNumbers = Iterables.filter(numbers, new
Predicate<Integer>() {
  public boolean apply(@Nullable Integer number) {
    return number % 2 == 0;
  }
});

for(int evenNumber : evenNumbers) {
  System.out.print(evenNumber + " "); }
System.out.println("");

System.out.print("Let's get list with values doubled: ");
List<Integer> doubledList = Lists.transform(numbers, new
Function<Integer, Integer>() {
  public Integer apply(@Nullable Integer number) {
    return number * 2;
  }
});

System.out.println(doubledList);
```

Using ImmutableList...

```
Number of elements: 9
Has 6? true
First index of 6 is 3
Last index of 6 is 6
Iterating over the list: 1 5 3 6 8 9 6 4 7
Getting only even numbers: 6 8 6 4
Let's get list with values doubled: [2, 10, 6, 12, 16, 18, 12, 8, 14]
```

Using MultiSet

```
Multiset<Integer> scores = HashMultiset.create();
for(int i = 0; i < 10; i++) {
  scores.add((int)(Math.random() * 10));
}

System.out.println("Number of scores: " + scores.size());
System.out.println("Number of 5's: " + scores.count(5));

scores.add(5, 6);
System.out.println("Number of 5's after adding six more: " +
scores.count(5));

scores.remove(5, 3);
System.out.println("Number of 5's after removing three of them: " +
  scores.count(5));
```

```
Number of scores: 10
Number of 5's: 1
Number of 5's after adding six more: 7
Number of 5's after removing three of them: 4
```

Immutability?

You may wonder if immutable data structures are really useful

It's about how we design our algorithms to use them

Using an Immutable List

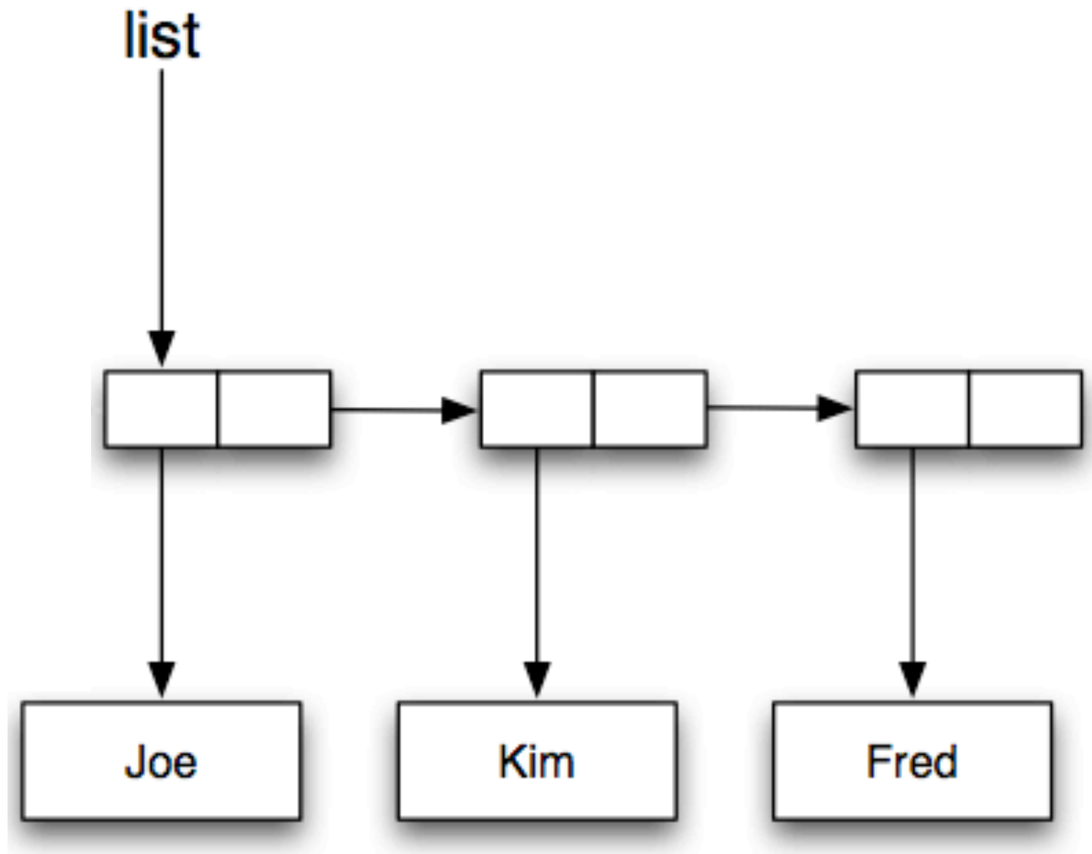


Figure 3.1: Persistent List Processing

Using an Immutable List

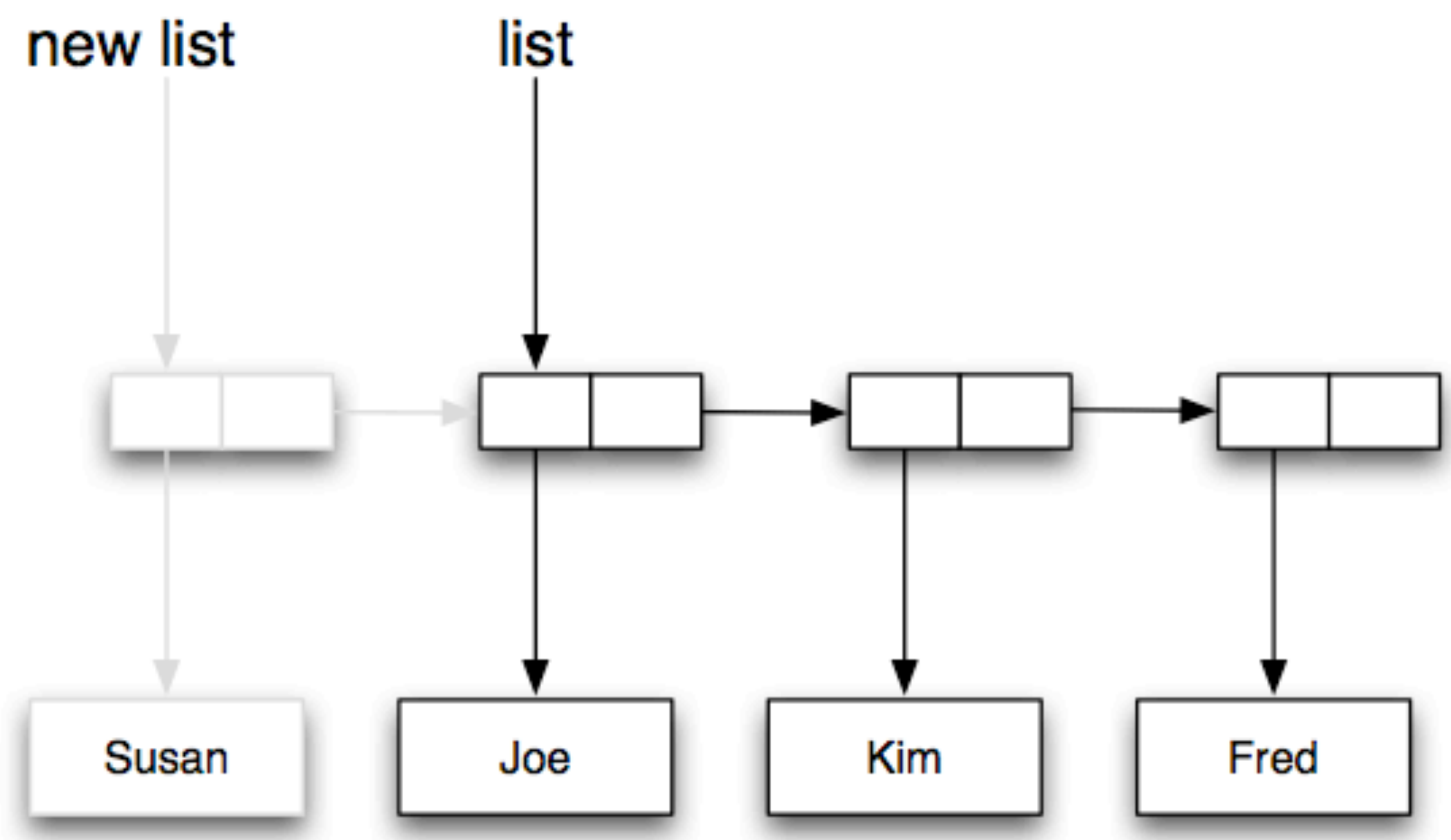
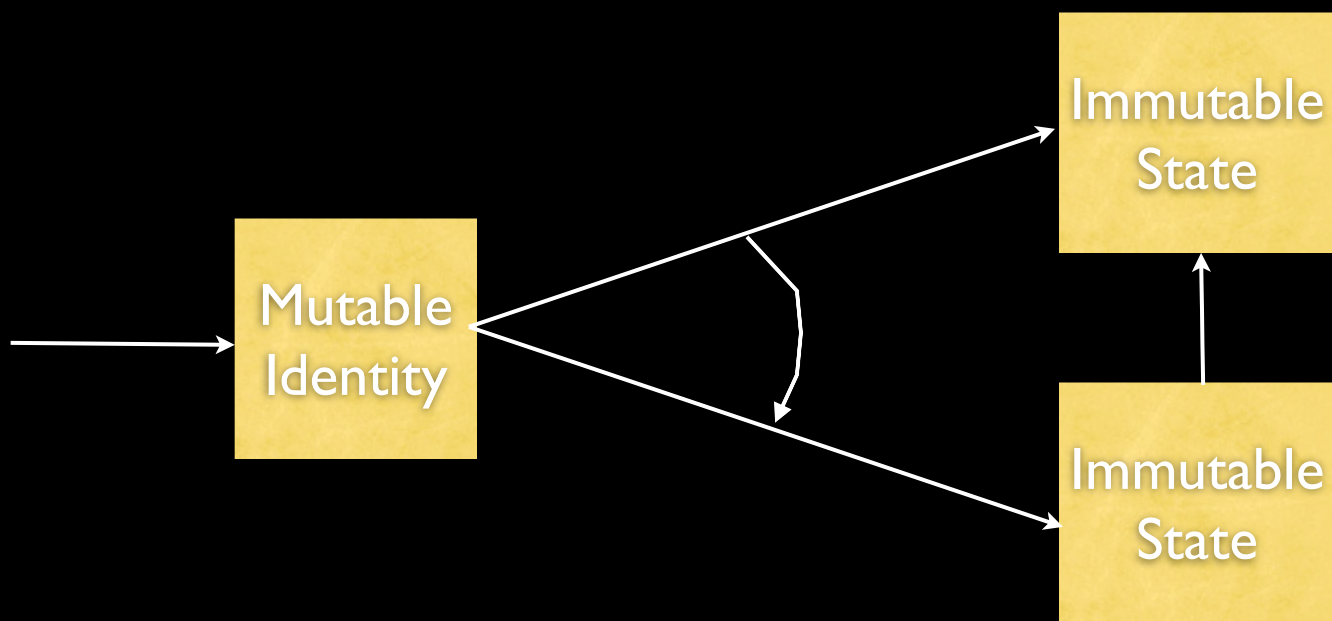


Figure 3.1: Persistent List Processing

Clojure's Approach

Clojure has an interesting separation of State and Identity



Clojure Example

Clojure has an interesting separation of State and Identity

```
(defn addItem [wishlist item]
  (dosync (alter wishlist conj item)))

(def familyWishList (ref '("iPad")))
(def originalWishList @familyWishList)

(println "Original wish list is" originalWishList)

(.start (Thread. (fn[] (addItem familyWishList "MBP"))))
(.start (Thread. (fn[] (addItem familyWishList "Bike"))))

(. Thread sleep 1000)

(println "Original wish list is" originalWishList)
(println "Updated wish list is" @familyWishList)
```

List vs. Vector

Scala Lists allowed manipulation at the head (just like Clojure's list)

But what if you want to modify something in the middle and yet use immutable collection?

Both Scala and Clojure have an answer, and that comes from Bagwell

Scala Vector uses Tries to provide constant time ops

Performance with Tries

High branching factor—32 children per node

Almost constant time inserts, deletes anywhere in the collection

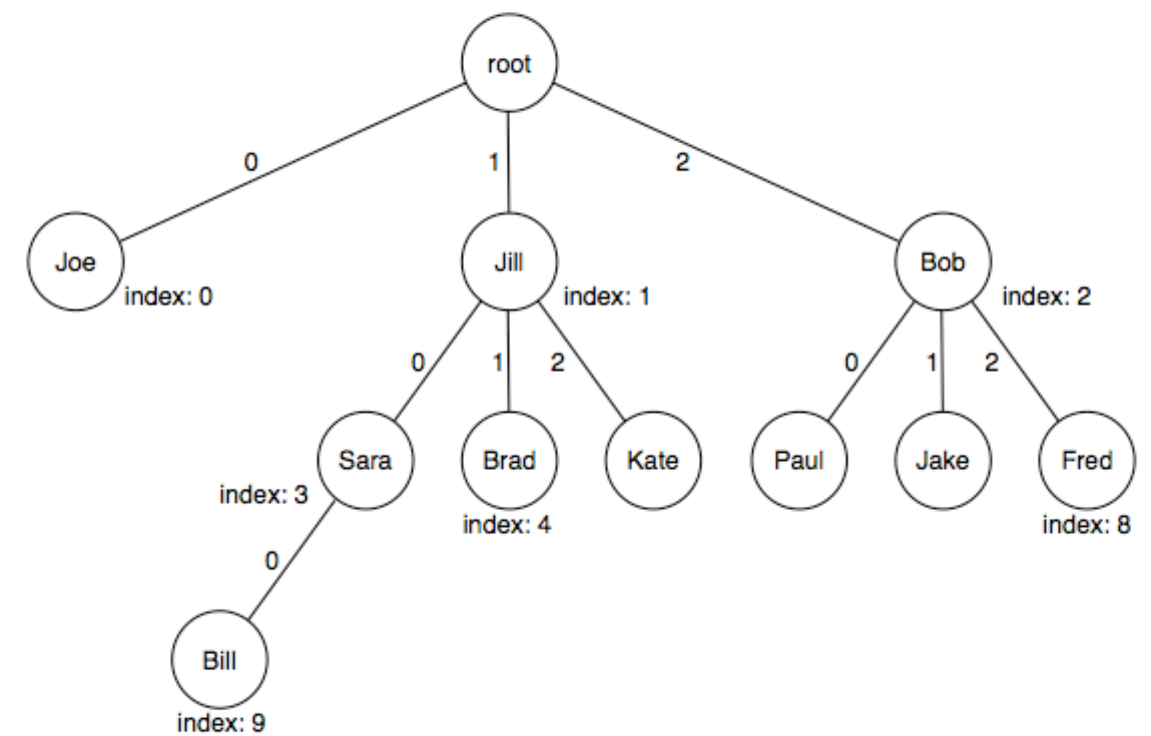


Figure 3.2: Using tries to store a list of people

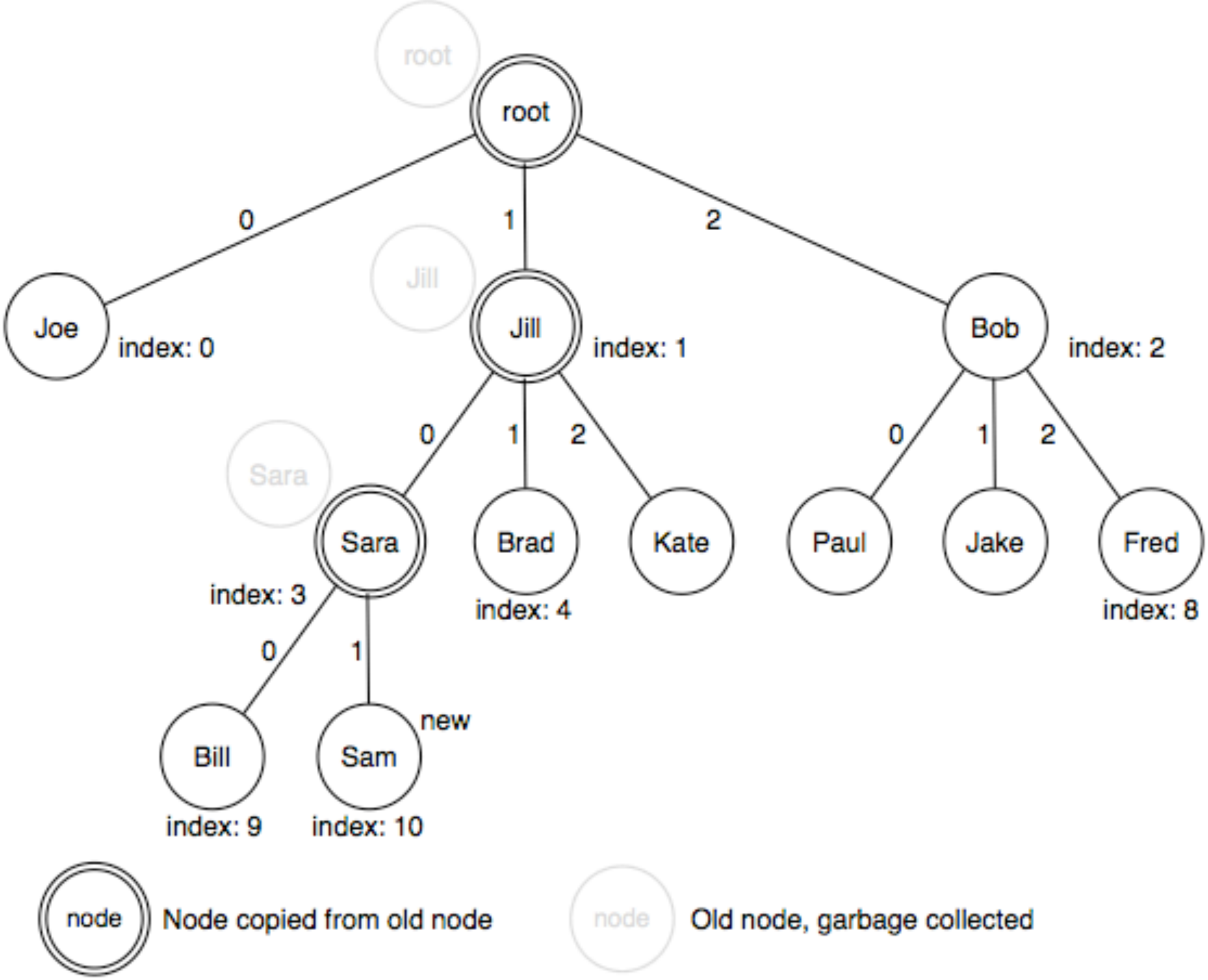


Figure 3.3: "Changing" Persistent List

Thank You!

Venkat Subramaniam
venkats@agiledeveloper.com
twitter: venkat_s