Stream Programming

Wojciech Macyna, Ph.D.

Contents

1 Introduction to Scala

The lecture contains an introduction to Scala language. It is based on the following materials: [2], [3], [4] and [5].

1.1 What is Scala

Scala is a type-safe JVM language that incorporates both object oriented and functional programming into an extremely concise, logical, and extraordinarily powerful language.

The name Scala stands for scalable language. The language is so named because it was designed to grow with the demands of its users. Scala may be applied to the wide ranges of tasks. Thanks to its scalability, it may be used for writing small scripts or building large systems.

Scala is a mixture of object-oriented and functional programming concepts in a statically typed language. Scalas functional programming constructs make it easy to build interesting things quickly. On the other hand, its object-oriented constructs facilities to structure larger systems and to adapt them to new demands. The combination of both styles in Scala makes it possible to express new kinds of programming patterns and component abstractions. It also leads to a legible and concise programming style.

1.2 Scala as an object oriented language

Many languages admit values that are not objects. A good example for that are the primitive values in Java. Such languages often allow static fields and methods that are not members of any object. Scala is a pure object-oriented language: every value is an object and every operation is a method call. For example, when you write 4 + 2 in Scala, you are actually invoking a method named + defined in class `Int`.

1.3 Scala is functional

Scala is also a functional language. Functions are first-class values. In a functional language, a function is a value of the same status as, for example, an integer or a string. You can pass functions as arguments to other functions, return them as results from functions, or store them in variables. You can also define a function inside another function, just as you can define an integer value inside a function.

1.4 Installation and the first use of Scala

To get a standard Scala installation, you should to http://www.scalalang.org/downloads and follow the directions for your platform. Scala plugin may also be incorporated in the standard Java tools like Eclipse, IntelliJ, or NetBeans.

The easiest way to get started with Scala is by using the Scala interpreter, an interactive shell for writing Scala expressions and programs. Simply type an expression into the interpreter and it will evaluate the expression and print the resulting value. The interactive shell for Scala is simply called `scala`. You use it by typing `scala` at a command prompt:

```
> scala
Welcome to Scala version 2.8.1.
Type in expressions to have them evaluated.
Type :help for more information.
scala>
```

After you type an expression, such as 6 + 2, and hit enter:

```
scala> 6 + 2
```

The interpreter will print:

```
res0: Int = 8
```

1.5 Compiling and running the example

To compile the example, the Scala compiler command `scalac` can be used. `scalac` works like most compilers: it takes a source file as argument and produces one or several object files. The object files it produces are standard Java class files.

If we save the above program in a file called `HelloWorld.scala`, we can compile it using the following command:

```
> scalac HelloWorld.scala
```

This will generate a few class files in the current directory. One of them will be called `HelloWorld.class`, and contains a class which can be directly executed.

A Scala program can be run using the `scala` command. Its usage is very similar to the java command used to run Java programs, and accepts the same options. The above example can be executed using the following command, which produces the expected output:

```
> scala HelloWorld
Hello, world!
```

1.6 Main differences between Java and Scala

Java is often too verbose. In Scala, the developer may not pay attention to specifying many unnecessary things which can be inferred by the compiler.

```
public class HelloJava {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

```
object HelloScala {
    def main(args: Array[String]): Unit = {
        println("Hello World!")
    }
}
```

In the listing above, we provide the standard "Hello world" application in Java and Scala. The most similar thing is the `main` method which is the entry point for the Java and Scala program. It takes as input an array of String arguments and contains a call to the pre-defined method with a String greeting.
Instead of the `public void static` definition for the `main` method, we use def keyword to define a Scala function. Also, there is nothing like static in Scala.
This object definition (`def`) defines a class called `HelloWorld` and an instance with the same name.

The second example of conciseness in Scala programming is a class definition.
In Java, a class with a constructor often looks like this:

```
class MyClass {
private int index;
private String name;
public MyClass(int index, String name) {
this.index = index;
this.name = name;
}
}
```

In Scala, you would likely write this instead:

```
class MyClass(index: Int, name: String)
```

1.7 Scala as a high level language

Scala is high level. For example, imagine you have a String variable name, and you want to find out whether or not that String contains an upper case character. In Java, you might write this:

```
boolean nameHasUpperCase = false;
for (int i = 0; i < name.length(); ++i) {
if (Character.isUpperCase(name.charAt(i))) {
nameHasUpperCase = true;
break;
}
}
```

Whereas in Scala, you could write this:

```
val nameHasUpperCase = name.exists(_.isUpper)
```

The definition of the array list in both languages may be specified like this.

In Java:

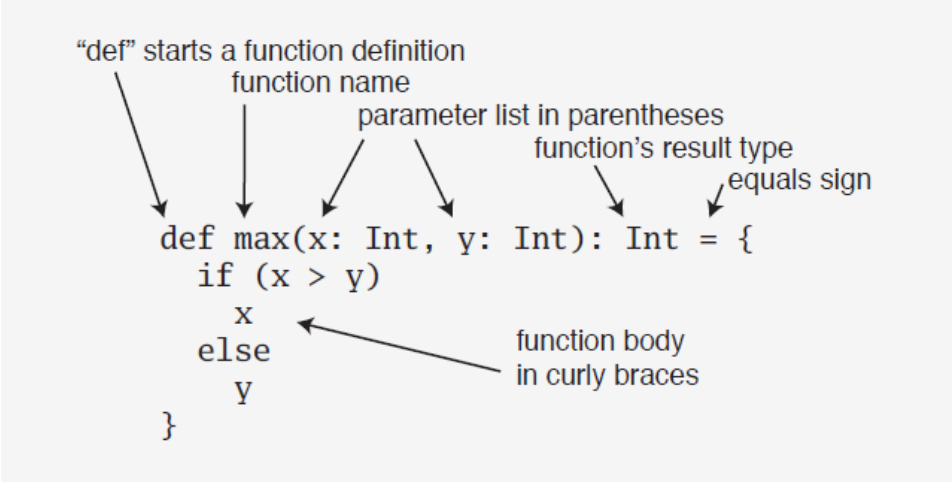

Figure 1: Method example

```
List<String> list = new ArrayList<String>();
list.add("1");
list.add("2");
list.add("3");
```

In Scala:

```
val list = List("1", "2", "3")
```

1.8 Define some variables

Scala has two kinds of variables, `val` and `var`. A `val` is similar to a `final` variable in Java. That means that once initialized, a `val` can never be reassigned. A `var`, by contrast, can be reassigned throughout its lifetime. Here is a `val` example:

```
scala> val msg = "Hello, world!"
msg: java.lang.String = Hello, world!
```

This statement introduces `msg` as a name for the string `"Hello, world!"`. The type of `msg` is `java.lang.String`, because Scala strings are implemented by Javas String class.

1.9 Scala functions and loops

The figure 1 is a illustration of the method signature.

The last example in this lecture presents the language difference as far as the loop `for` is concerned.

The loop `for` in Java:

```
public static void main(String [] args){
for (int i = 0; i <= 10; i = i + 1) {
```

```
   System.out.println(i);
}
}
```

The loop `for` in Scala:

```
object MainObject {
   def main(args: Array[String]) {
        for( a <- 1 to 10 ){
         println(a);
      }
   }
}
```

2 Object oriented aspects of Scala

In this lecture, you will learn more about classes, fields, and methods.

2.1 Defining a class

A minimal class definition is simply the keyword class and an identifier.

```
class Car
val c = new Car
```

The keyword `new` is used to create an instance of the class. The class `Car` has a default constructor with no arguments. However, you will often define a constructor within the class body:

```
class Car(var carType: String, var capacity : Int) {
        def getCapacity() : Int ={
        return capacity
        }
}
```

Now, you can create an object and use the methods:

```
val c = new Car1("Opel", 2000)
val d = c.getCapacity()
```

The class `Car` has one constructor and one method `getCapacity`. In Scala, the primary constructor is in the class signature: `var carType: String, var capacity : Int`. The `getCapacity` method takes one `String` and one `Int` argument. It returns a `Int` value.

2.2 Constructors

Constructors can have optional parameters by providing a default value:

```
class Car(var carType: String, var capacity : Int, var year : Int = 2010)
}
```

In this version of the class `Car`, `year` has the default value 2010 so the argument `year` may be omitted required. You may define the objects as follows:

```
val c = new Car("Opel", 2000)
val c1 = new Car("Opel", 2000, 2012)
```

In the first version, the argument `year` of c has the value 2010.

2.3 Attributes and Methods

Members are `public` by default. Use the `private` access modifier to hide them from outside of the class.

```
class Car (val carType: String, val capacity : Int) {
private val year = 2010
private def getCapacity() : Int ={
        return capacity
}
def printType() ={
        println(carType)
}
}
```

In this version , the class `Car` has one private attribute `year`, one private method `getCapacity` and one public method `printType`. Note that we cannot invoke the private method `v.getCapacity`, since the private class members are not accessed from outside the class.

```
val v = new Car
// v.getCapacity() - error: private access
v.printType()
```

Parameters without `val` or `var` are private values, visible only within the class.

2.4 Auxiliary constructors

Sometimes you need multiple constructors in a class. In Scala, constructors other than the primary constructor are called auxiliary constructors. It is defined on the basis of the main constructor with the keyword `this`

```
class Car (val carType: String, val capacity : Int) {
  // Auxiliary constructor
  def this(carType: String) = this(carType, 1600)
  def this() = this("BMW", 1600)
}
```

Now, we have three constructors and three ways of object construction:

```
val c = new Car("Opel", 2000)
val c1 = new Car("Opel")
val c2 = new Car
```

2.5 Method's overloading

The class may have many attribute. In the example below, the class `Car` has three attributes: `carType`, `capacity` and `year`. Two of them are private, one is public. The class has two versions of the method `setCapacity`. They differ in the number of arguments: the first version has one argument `newCapacity: Int`. The second one has two arguments `newCapacity: Int, typeName: String`. Both versions return nothing (`Unit`).

```
class Car {
var carType = "Opel"
```

```
private var capacity = 2000
private val year = 2010
def setCapacity(newCapacity: Int): Unit = {
  capacity = newCapacity
}
def setCapacity(newCapacity: Int, typeName: String): Unit = {
  capacity = newCapacity
}
}
```

The public attributes of an object can be changed but the private not:

```
val car1 = new Car
car1.carType="Fiat";
// car1.capacity=2100 // Does not compile: private access
```

The method `setCapacity` can be used as follows:

```
val car1 = new Car
car1.setCapacity(2500)
car1.setCapacity(2500, "Opel")
```

2.6 Class inheritance

In object-oriented programming, inheritance is the mechanism of basing an object or class upon another object or class, retaining similar implementation.

If a class `A` (subclass) inherit from the class `B` (superclass), the class `A` retains all the method and attributes of class `B` and adds its own specific methods. In this way, the classes may form class hierarchies. The simplest form of inheritance looks like this:

```
class Van3 extends Car4
```

The more difficult example is shown on the following listings:

```
class Car (val carType: String, val capacity : Int) {
private val year = 2010
private def getCapacity() : Int ={
  return capacity
}
def printType() ={
  println(carType)
}
protected def printType() ={
  println(carType)
}
}

class Van(override val carType: String) extends Car (carType: String){
def printType1() ={
```

```
    printType()
}
def getCapacity1()={
  // getCapacity() does not compile
}
}
```

We present two classes: `Car` and `Van`. The class `Car` has one private attribute `year` and three methods: `getCapacity`, `printType` and `printType` with the access `private`, `public` and `protected`, respectively. The class `Van` inherits from the class `Car`. It means that the class `Van` can use all not private attributes and methods from `Car`. Additionally, the class `Van` implements two new methods: `printType1` and `getCapacity1`. In class `Van` the method `printType` is used inside `printType1`. It comes from the fact that `printType` is defined as protected. Thus, it may be used in the class where is defined and in all inherited classes. On the other hand, the method `getCapacity` cannot be used in `getCapacity1`, because is defined as private and is not accessible in the inherited classes.

2.7 Constructor inheritance

In the above listings the constructor inheritance is shown.

```
class Van(override val carType: String) extends Car (carType: String){
}
```

Please note that `Car` has only one constructor. So, the parameters in `Van` constructor must extend the parameters of `Car` ones.

3 Functional Objects

A key concept of this lecture are the description of functional objects. Please see chapter 6 of [5] for more detail.

3.1 Constructing Rational

This initial Rational example highlights a difference between Java and Scala. In Java, classes have constructors, which can take parameters, whereas in Scala, classes can take parameters directly.

In this way, the notation of Scala is more concise: class parameters can be used directly in the body of the class. There's no need to define fields and write assignments that copy constructor parameters into fields. Such a situation in Java is called getter/setter definition.

The Scala compiler will compile any code you place in the class body, which isn't part of a field or a method definition, into the primary constructor. For example, you could print a debug message like this:

```
class Rational(n: Int, d: Int) {
    println("Created "+ n +"/"+ d)
}
```

Given this code, the Scala compiler would place the call to `println` into Rational's primary constructor. The `println` call will, therefore, print its debug message whenever you create a new Rational instance:

```
scala> new Rational(1, 2)
Created 1/2
res0: Rational = Rational@90110a
```

3.2 Reimplementing the toString method

By default, class Rational inherits the implementation of `toString` defined in class `java.lang.Object`. You can override the default implementation by adding a method `toString` to class Rational, like this:

```
class Rational(n: Int, d: Int) {
  override def toString = n +"/"+ d
}
```

3.3 Operation Overloading

Now, we'll define a public add method on class Rational that takes another Rational as a parameter. To keep Rational immutable, the add method must not add the passed rational number to itself. Rather, it must create and return a new Rational that holds the sum. You might think you could write `add` this way:

```
class Rational(n: Int, d: Int) {
  require(d != 0)
  val numer: Int = n
  val denom: Int = d
  override def toString = numer +"/"+ denom
  def add(that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )
}
```

In the version of Rational shown above, we added two fields named `numer` and `denom`, and initialized them with the values of class parameters n and d. We also reimplemented the implementation of `toString` and the method `add` so that they use the fields, not the class parameters.

Here, `that.numer` refers to the numerator of the object on which `add` was invoked.

3.4 Auxiliary constructors

Sometimes you need multiple constructors in a class. In Scala, constructors other than the primary constructor are called auxiliary constructors. For example, a rational number with a denominator of 1 can be written simply the numerator. Instead of 5/1, you can just write 5. This would require adding an auxiliary constructor to Rational that takes only one argument, the numerator, with the denominator predefined to be 1 (for example Relational(5)). Please note that `require` is a condition statement, so that the condition $d! = 0$ must be fulfilled for further processing:

```
class Rational(n: Int, d: Int) {
   require(d != 0)
   val numer: Int = n
   val denom: Int = d
   def this(n: Int) = this(n, 1) // auxiliary constructor
   override def toString = numer +"/"+ denom
   def add(that: Rational): Rational =
     new Rational(
       numer * that.denom + that.numer * denom,
       denom * that.denom
     )
}
```

Auxiliary constructors in Scala start with `def this(...)`. The body of Rational's auxiliary constructor merely invokes the primary constructor. Here is the example of using the auxiliary constructor:

```
scala> val y = new Rational(3)
y: Rational = 3/1
```

The invoked constructor is either the primary constructor (as in the Rational example), or another auxiliary constructor that comes textually before the calling constructor. The primary constructor is thus the single point of entry of a class.

3.5 Private fields and methods

In the listing below, we define a private function `gcd` which estimates the greatest common divisor of two integers.

```
class Rational(n: Int, d: Int) {
      require(d != 0)
      private val g = gcd(n.abs, d.abs)
      val numer = n / g
      val denom = d / g
      def this(n: Int) = this(n, 1)
      def add(that: Rational): Rational =
        new Rational(
          numer * that.denom + that.numer * denom,
          denom * that.denom
        )
      override def toString = numer +"/"+ denom
      private def gcd(a: Int, b: Int): Int =
        if (b == 0) a else gcd(b, a % b)
    }
```

In this version of Rational, we added a private field, g, and modified the initializers for `numer` and `denom`. Because g is private, it can be accessed inside the body of the class, but not outside. To make a field or method private you simply place the `private` keyword in front of its definition.

3.6 Defining operators

To make Rational even more convenient, we add new methods to the class that perform mixed addition and multiplication on rational numbers and integers.

```
class Rational(n: Int, d: Int) {
      require(d != 0)
      private val g = gcd(n.abs, d.abs)
      val numer = n / g
      val denom = d / g
      def this(n: Int) = this(n, 1)
      def + (that: Rational): Rational =
        new Rational(
          numer * that.denom + that.numer * denom,
          denom * that.denom
        )
```

```
  def + (i: Int): Rational =
    new Rational(numer + i * denom, denom)
  def - (that: Rational): Rational =
    new Rational(
      numer * that.denom - that.numer * denom,
      denom * that.denom
    )
  def - (i: Int): Rational =
    new Rational(numer - i * denom, denom)
  def * (that: Rational): Rational =
    new Rational(numer * that.numer, denom * that.denom)
  def * (i: Int): Rational =
    new Rational(numer * i, denom)
  def / (that: Rational): Rational =
    new Rational(numer * that.denom, denom * that.numer)
  def / (i: Int): Rational =
    new Rational(numer, denom * i)
  override def toString = numer +"/"+ denom
  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)
}
```

4 Advanced function declarations and closures

In this lecture, we present the advanced way of function declarations (see chapter 8 of [5]). Additionally, we show some examples of a very powerful mechanism called closure (examples are from [6]).

4.1 Local functions

In the functional programming, programs should be decomposed into many small functions that each do a well-defined task. Individual functions are often quite small. The advantage of this style is that it gives a programmer many building blocks that can be flexibly composed to do more difficult things. Each building block should be simple enough to be understood individually.

One problem with this approach is that all the helper function names can pollute the program namespace. They often do not make sense individually, and you often want to keep enough flexibility to delete the helper functions if you later rewrite the class a different way.

```
import scala.io.Source
    object LongLines {
      def processFile(filename: String, width: Int) {
        def processLine(line: String) {
          if (line.length > width)
            print(filename +": "+ line)
        }
        val source = Source.fromFile(filename)
        for (line <- source.getLines)
          processLine(line)
      }
    }
```

In this example, we push the private method, $processLine$, into a local function $processFile$. To do so we omit the private modifier, which can only be applied (and is only needed) for methods, and placed the definition of $processLine$ inside the definition of $processFile$. As a local function, $processLine$ is in scope inside $processFile$, but inaccessible outside.

4.2 First-class functions

Scala offers first-class functions. You can not only define functions and call them, but you can write down functions as unnamed literals and then pass them around as values. Here is a simple example of a function literal that adds one to a number of integer type:

```
(x: Int) => x + 1
```

The $=>$ designates that this function converts the thing on the left (any integer x) to the thing on the right $(x + 1)$. So, this is a function mapping any integer x to $x + 1$. Function values are objects, so you can store them in variables if you like. They are functions, too, so you can invoke them using the usual parentheses function-call notation. Here is an example of both activities:

```
scala> var increase = (x: Int) => x + 1
increase: (Int) => Int = <function>

scala> increase(10)
res0: Int = 11
```

If you want to have more than one statement in the function literal, surround its body by curly braces and put one statement per line, thus forming a block. Just like a method, when the function value is invoked, all of the statements will be executed, and the value returned from the function is whatever the expression on the last line generates.

```
scala> increase = (x: Int) => {
            println("We")
            println("are")
            println("here!")
            x + 1
          }
increase: (Int) => Int = <function>

scala> increase(10)
      We
      are
      here!
  res4: Int = 11
```

$Foreach$ method is available for all collections. It takes a function as an argument and invokes that function on each of its elements. Here is how it can be used to print out all of the elements of a list:

```
val someNumbers = List(-11, -10, -5, 0, 5, 10)
someNumbers.foreach((x: Int) => println(x))
```

As another example, collection types also have a $filter$ method. This method selects those elements of a collection that pass a test the user supplies. That test is supplied using a function. For example, the function $(x : Int) => x > 0$ could be used for filtering. This function maps positive integers to true and all others to false. Here is how to use it with filter:

```
scala> someNumbers.filter((x: Int) => x > 0)
scala> someNumbers.filter(x => x > 0)
```

To make a function literal even more concise, you can use underscores as placeholders for one or more parameters, so long as each parameter appears only one time within the function literal. For example, $\_ > 0$ is very short notation for a function that checks whether a value is greater than zero:

```
scala> someNumbers.filter(_ > 0)
```

You can think of the underscore as a "blank" in the expression that needs to be "filled in." This blank will be filled in with an argument to the function each time the function is invoked. For example, given that $someNumbers$ was initialized here to the value List(-11, -10, -5, 0,

5, 10), the filter method will replace the blank in $\_ > 0$ first with -11, as in $-11 > 0$, then with -10, as in $-10 > 0$, then with -5, as in $-5 > 0$, and so on to the end of the List.

You can specify the types using a colon, like this:

```
scala> val f = (_: Int) + (_: Int)
f: (Int, Int) => Int = <function>

scala> f(5, 10)
res11: Int = 15
```

Note that $\_ + \_$ expands into a literal for a function that takes two parameters. This is why you can use this short form only if each parameter appears in the function literal at most once. Multiple underscores mean multiple parameters, not reuse of a single parameter repeatedly. The first underscore represents the first parameter, the second underscore the second parameter, the third underscore the third parameter, and so on.

4.3 Partially applied functions

When you use an underscore in this way, you are writing a partially applied function. In Scala, when you invoke a function, passing in any needed arguments, you apply that function to the arguments. For example, given the following function:

```
scala> def sum(a: Int, b: Int, c: Int) = a + b + c
scala> sum(1, 2, 3)
```

A partially applied function is an expression in which you don't supply all of the arguments needed by the function. Instead, you supply some, or none, of the needed arguments. For example, to create a partially applied function expression involving sum, in which you supply none of the three required arguments, you just place an underscore after sum. The resulting function can then be stored in a variable. Here's an example:

```
scala> val a = sum _
```

Given this code, the Scala compiler instantiates a function value that takes the three integer parameters missing from the partially applied function expression, sum _, and assigns a reference to that new function value to the variable a. When you apply three arguments to this new function value, it will turn around and invoke sum, passing in those same three arguments:

```
scala> a(1, 2, 3)
```

Thus, a(1, 2, 3) is a short form for:

```
scala> a.apply(1, 2, 3)
```

Here's what just happened: The variable named a refers to a function value object. This function value is an instance of a class generated automatically by the Scala compiler from sum _, the partially applied function expression. The class generated by the compiler has an apply method that takes three arguments. The generated class's apply method takes three arguments because three is the number of arguments missing in the sum _ expression. The Scala compiler translates the expression a(1, 2, 3) into an invocation of the function value's apply method, passing in the three arguments 1, 2, and 3.

You can also express a partially applied function by supplying some but not all of the required arguments. Here's an example:

```
scala> val b = sum(1, _: Int, 3)
```

In this case, you've supplied the first and last argument to sum, but the middle argument is missing. Since only one argument is missing, the Scala compiler generates a new function class whose apply method takes one argument. When invoked with that one argument, this generated function's apply method invokes sum, passing in 1, the argument passed to the function, and 3. Here's an example:

```
scala> b(2)
```

4.4 Closure

So far, all the examples of function literals have referred only to passed parameters. The following example is a closure function:

```
scala> var more = 1
scala> val addMore = (x: Int) => x + more
scala> addMore(10)
```

The function value (the object) that's created at runtime from this function literal is called a closure. A function literal with no free variables, such as $(x : Int) => x+1$, is called a closed term, where a term is a bit of source code. Thus a function value created at runtime from this function literal is not a closure in the strictest sense, because $(x : Int) => x + 1$ is already closed as written. But any function literal with free variables, such as $(x : Int) => x+more$, is an open term. Therefore, any function value created at runtime from $(x : Int) => x + more$ will by definition require that a binding for its free variable, $more$, be captured. The resulting function value, which will contain a reference to the captured more variable, is called a closure, therefore, because the function value is the end product of the act of closing the open term, $(x : Int) => x + more$.

In the second example, we define a $isOfVotingAge$ function which for given age determines if the condition $age >= votingAge$ holds. Then, the function $printResult$ is defined. It takes two argument. The first argument must be a function with one integer parameter, which returns Boolean. The second parameter must be integer. The function $printResult$ is invoked with $isOfVotingAge$ as a first argument and 20 as the second one.

```
var votingAge = 18
val isOfVotingAge = (age: Int) => age >= votingAge
def printResult(f: Int => Boolean, x: Int) {
  println(f(x))}
printResult(isOfVotingAge, 20)
```

5 Collections in Scala

In this lecture, we provide the basic collection types of Scala (see the tutorial [7] for more detail).

5.1 Set

Set is a collection which contains distinct elements. The set can hold various types of elements. In the example below, the set s stores the integer values. There are two ways of defining a set:

```
var s : Set[Int] = Set(1,3,5,7,7)
var s1 = Set(1,3,5,7,9)
s.foreach((z) => println(z))
```

This collection has many useful methods. To print the first element of the set s, you can write:

```
println( "Head of set : " + s.head )
```

Two sets (s and $s2$) can be concatenated using $++$:

```
var s2 = s ++ s1
```

5.2 List

The second collection is a list. In contrast to a set, the list can contain repeatable values. We distinguish two methods of declaring of the list (see code below):

```
val fruit: List[String] = List("apples", "oranges", "pears")
 val fruit1 = "apples" :: ("oranges" :: ("pears" :: Nil))
```

Lists are immutable, which means elements of a list cannot be changed by assignment. Once assigned, it is impossible to reassign like this:

```
// fruit(0) = "banana"
```

5.3 Map

The next collection is a map. It can hold the items. Each item is a pair: $< key,value >$. The map can contain the set of items with distinct keys. The following listing creates two maps. After that, it assigns the values to $myMap$ and prints the items

```
val colors = Map("red" -> "#FF0000", "azure" -> "#F0FFFF")
var myMap:Map[Char,Int] = Map()

myMap += ('I' -> 1)
myMap += ('J' -> 5)
myMap += ('K' -> 10)
```

```
myMap += ('L' -> 100)

myMap.keys.foreach{ i =>
  print( "Key = " + i )
  println(" Value = " + myMap(i) )}
```

5.4 Iterator

An iterator is not a collection, but rather a way to access the elements of a collection one by one. The following code creates an iterator, checks if it has the next element. If it is true, it print this element

```
val it = Iterator("a", "number", "of", "words")
while (it.hasNext){
  println(it.next())
}
```

5.5 Example

In this subsection we demonstrate an example of the list which holds the object of type Car defined as a separate class. In the $main$ method three cars are created and put into the list. We show very useful functions like: filter, map and find. The first function filers the set and returns the cars of type $Opel$. Then, the result set is sorted using $sortBy$ by $capacity$. The function map converts the elements of the map to a form : $< carType, capacity, year >$.If you use $filter$ instead of $find$ (like in the last instruction), it will compute all elements in the collection because Scala collection is strict.

```
class Car(var carType: String, var capacity: Int, var year : Int) {
}
object Demo1 {
   def main(args: Array[String]) {

   val c = new Car("Opel", 2000, 2010)
   val c1 = new Car("Opel", 1800, 2012)
   val c2 = new Car("BMW", 1800, 2014)

   val cars = c :: (c1:: (c2 :: Nil))

   val carsF = cars.filter{i => i.carType == "Opel"}
        .sortBy{i => i.capacity}
   for (i <- carsF) println(i.carType + " "+ i.capacity)

    val carsM = cars.map{ i => s"{i.carType} - capacity:
         {i.capacity}, year:    {i.year}"}
    val carsFi = cars.find{ i => i.carType == "Opel" }
```

```
    for (i <- carsFi) println(i.carType + " "+ i.capacity)
  }
}
```

6 Main assumptions of stream processing

This lecture contains the hot topics and problems of stream processing (see [8]). In the second part of the lecture, we present some examples, how to deal with streams in Scala.

6.1 What is a Data Stream

Large Data Set which is hard to:

- Process (by classic algorithms)
- Transfer
- Store (in a single location)

It is due to the following reasons:

- Short processing time for each element in the stream
- No random data access
- Unbounded number of element

6.2 Streaming in Scala

In this topic, the methods in the Stream object are looked at a little closer. For additional information see: [9] and [10].

The stream may be declared like this:

```
val stream = 1 #:: 2 #:: 3 #:: Stream.empty
stream take 3 foreach println
```

The empty stream is good for terminating a stream. The instruction prints the first 3 element of the string.

The other way of declaring of the stream is below. The last instruction prints the first element of the stream.

```
import scala.collection.immutable.Stream.cons

val stream2: Stream[Int] = cons(1, cons(2, cons(3, Stream.empty)))
println(stream2.head)
```

Here are the examples of infinite stream declaration. The $inifiniteNumberStream$ and i are the functions of stream declaration. The last line returns: 6, 18, 48.

```
def inifiniteNumberStream(number: Int): Stream[Int] =
cons(number, inifiniteNumberStream(number + 1))
inifiniteNumberStream(1).take(20).print
```

```
def i(x:Int,y:Int):Stream[Int] = (x*y) #:: i(x+1,y*2)
i(2,3) take 3 foreach println
```

In the next declaration, we create the infinite stream, but take only 20 elements from it.

```
val stream3: Stream[Int] = Stream.from(1)
stream3.take(20).print
```

An infinite stream starting at 10 and increasing by 3

```
Stream.from (10,3) take 3 foreach println
```

The next instruction creates a stream starting at 1 increased by 3 with elements less than 20.

```
Stream.range(1, 20, 3) foreach println
```

As we can observe now, a Scala Stream works like a List, except that its elements are computed lazily. Because Stream elements are computed lazily, a Stream can be infinitely long. For example, after invoking the following line:

```
val stream = (1 to 100000000).toStream
```

We receive the output:

```
stream: scala.collection.immutable.Stream[Int] = Stream(1, ?)
```

This is because the end of the stream has not been evaluated yet.

You can attempt to access the head and tail of the stream. The head is returned immediately:

```
scala> stream.head
res0: Int = 1
```

but the tail is not evaluated yet:

```
scala> stream.tail
res1: scala.collection.immutable.Stream[Int] = Stream(2, ?)
```

The ? symbol is the way a lazy collection shows that the end of the collection has not been evaluated yet.

The following transformer methods are computed lazily, so when transformers are called, you see the familiar ? character that indicates the end of the stream has not been evaluated yet:

```
scala> stream.take(5)
res0: scala.collection.immutable.Stream[Int] = Stream(1, ?)

scala> stream.filter(_ < 300)
res1: scala.collection.immutable.Stream[Int] = Stream(1, ?)

scala> stream.filter(_ > 300)
res2: scala.collection.immutable.Stream[Int] = Stream(201, ?)

scala> stream.map { _ * 5 }
res3: scala.collection.immutable.Stream[Int] = Stream(2, ?)
```

However, be careful with methods that arent transformers. Calls to the following methods can easily cause `java.lang.OutOfMemoryError` errors: `stream.max`, `stream.size` and `stream.sum`.

Using a Stream gives you a chance to specify a huge list, and begin working with its elements:

```
val list = (1 to 100000000).toStream

list(0)  // returns 1
list(1)  // returns 2
// ...
list(10)  // returns 11
```

The following code calculates the Fibonacci sequence. As the sequence is infinitive, the evaluation is done in the last:

```
def fibFrom(a: Int, b: Int): Stream[Int] = a #:: fibFrom(b, a + b)
val fibs = fibFrom(1,1)
fibs take 10 foreach println
```

7 Frequency counter algorithms

Counter-based algorithms track a subset of items from the inputs, and monitor counts associated with these items. For each new arrival, the algorithms decide whether to store this item or not, and if so, what counts to associate with it.

7.1 Applications of frequency counter algorithms

- Packets on the Internet

  Frequent items: most popular destinations or most heavy bandwidth users

- Queries submitted to a search engine:

  Frequent items: most popular queries

7.2 Boyer Moore majority algorithm

In this subsection, the Boyer Moore majority algorithm and some examples are presented.

`Task:` Given a list of m elements. Is there an absolute majority (an element occurring more than $m/2$ times)?

The Boyer Moore algorithm [11] maintains in its local variables a sequence element and the counter initially set to 0. It then processes the elements of the sequence, one at a time. When processing an element i, if the counter is zero, the algorithm stores i as its remembered sequence element and sets the counter to 1. Otherwise, it compares i to the stored element and either increments the counter (if they are equal) or decrements the counter (otherwise). At the end of this process, if the sequence has a majority, it will be the element stored by the algorithm. The following pseudocode presents the Boyer Moore majority algorithm.

```
majority algorithm:

counter:=0; v unassigned;
for each i:
if counter=0:
        v:=i;
        counter:=1;
else if v=i:
        counter:=counter+1;
else
        counter:=counter-1;
```

Please note that even when the input sequence has no majority, the algorithm will report one of the sequence elements as its result. However, it is possible to perform a second pass over the same input sequence in order to count the number of times the reported element occurs and determine whether it is actually a majority. This second pass is needed to check whether the element determined in a first pass is a majority element.

The time and space complexity of the algorithm is estimated as:
Time Complexity: O(n)

Space Complexity: O(1)

7.2.1 Example of the Majority algorithm

`Objective`: Given an array of integer values, find the majority element in it (if exist). Let $arrA$ be an array with following elements: 5, 3, 3, 5, 5, 1, 5. `Majority Element`: If an element appears more than $n/2$ times in array where n is the size of the array. From the example, we can see that 5 appears more than 3 times. So the algorithm should output 5 as a result.

For the given array ($arrA$), the algorithm works as follows:

```
First Iteration:

int [] arrA = {5,3,3,5,5,1,5};
i = 0, v = 5
counter  = 1

int [] arrA = {5,3,3,5,5,1,5};
i = 1, current_element=3, v = 5
counter=0 (current element is not same, counter = counter -1)

int [] arrA = {5,3,3,5,5,1,5};
i = 2, current_element=3, v = 3
counter=1 (counter was 0 so, current_element = v)

int [] arrA = {5,3,3,5,5,1,5};
i = 3, current_element=5, v= 3
counter=0(current element is not same, counter = counter-1)

int [] arrA = {5,3,3,5,5,1,5};
i = 4 current_element=5, v= 5
counter = 1(counter was 0 so, current_element = v)

int [] arrA = {5,3,3,5,5,1,5};
i = 5, current_element=1, v= 5,
counter=0(current element is not same, counter = counter-1)

int [] arrA = {5,3,3,5,5,1,5};
i = 6, current_element=5, v= 5
counter = 1(counter was 0 so, current_element = v)
```

The algorithm returns $v = 5$. Hence, 5 is a candidate as a majority element. Now, in the second pass, the counter for 5 should be checked.

7.3 Misra-Gries Algorithm

The Misra-Gries summary is a simple algorithm that solves the frequent items problem (see [12]). It can be viewed as a generalization of Majority to track multiple frequent items. Instead of keeping a single counter and item from the input, the Misra-Gries summary stores $k-1$ (item,counter) pairs. The natural generalization of Majority is to compare each new item against the stored items T, and increment the corresponding counter if it is among them. Else, if there is some counter with count zero, it is allocated to the new item, and the counter set to 1. If all $k-1$ counters are allocated to distinct items, then all are decremented by 1. A grouping argument is used to argue that any item which occurs more than n/k times must be stored by the algorithm when it terminates.

As for all algorithms in the data stream model, the input is a finite sequence of integers from a finite domain. The algorithm outputs an associative array which has values from the stream as keys, and estimates of their frequency as the corresponding values. It takes a parameter k which determines the size of the array, which impacts both the quality of the estimates and the amount of memory used.

The trick is to run the MAJORITY algorithm, but with k counters instead of 1.

```
algorithm misra-gries:

c[1,..., (k-1)]:=0; T:= Empty
for each i:
if i is in T:
        c[i] := c[i] + 1;
else if |T| < k-1:
        T = T + i;
        c[i] := 1;
else for all j in T
        c[j]:=c[j]-1
        if c[j]=0
        T=T\{j}
```

The MisraGries algorithm uses $O(k(log(m)+log(n)))$ space, where m is the maximum value in the stream and n is the length of the stream.

Every item which occurs at least n/k times is guaranteed to appear in the output array. Therefore, in the second pass over the data, the exact frequencies for the $k-1$ items can be computed to solve the frequent items problem, or in the case of k=2, the majority problem. This second pass takes $O(k*log(m))$ space.

The summaries (arrays) output by the algorithm are mergeable, in the sense that combining summaries of two streams s and r by adding their arrays keywise and then decrementing each counter in the resulting array until only k keys remain results in a summary of the same (or better) quality as compared to running the Misra-Gries algorithm over the concatenation of s with r.

7.3.1 Example of the algorithm

Let $arrA = 1, 1, 2, 1, 2, 3, 4, 2, 1, 2, 1, 2$ be the stream with 12 elements. Let $k = 3$. So, we will try to find all elements which appear more than m/k in the stream. In this situation, the result should be: 1 and 2, because only these two items appear more then $12/3 = 4$.

First Iteration:

```
int [] arrA = {1,1,2,1,2,3,4,2,1,2,1,2};
i = 0, current_element = 1
v1 = 1, c1 = 1

int [] arrA = {1,1,2,1,2,3,4,2,1,2,1,2};
i = 1, current_element = 1
v1 = 1, c1 = 2

int [] arrA = {1,1,2,1,2,3,4,2,1,2,1,2};
i = 2, current_element = 2
v1 = 1, c1 = 2
v2 = 2, c2 = 1

int [] arrA = {1,1,2,1,2,3,4,2,1,2,1,2};
i = 3, current_element = 1
v1 = 1, c1 = 3
v2 = 2,c2 = 1

int [] arrA = {1,1,2,1,2,3,4,2,1,2,1,2};
i = 4, current_element = 2
v1 = 1, c1 = 3
v2 = 2, c2 = 2

int [] arrA = {1,1,2,1,2,3,4,2,1,2,1,2};
i = 5, current_element = 3
v1 = 1, c1 = 2
v2 = 2, c2 = 1

int [] arrA = {1,1,2,1,2,3,4,2,1,2,1,2};
i = 6, current_element = 4
v1 = 1, c1 = 1
c2 = 0

int [] arrA = {1,1,2,1,2,3,4,2,1,2,1,2};
i = 7, current_element = 2
v1 = 1, c1 = 1
v2 = 2, c2 = 1
```

```
int [] arrA = {1,1,2,1,2,3,4,2,1,2,1,2};
i = 8, current_element = 1
v1 = 1, c1 = 2
v2 = 2, c2 = 1

int [] arrA = {1,1,2,1,2,3,4,2,1,2,1,2};
i = 9, current_element = 2
v1 = 1, c1 = 2
v2 = 2, c2 = 2

int [] arrA = {1,1,2,1,2,3,4,2,1,2,1,2};
i = 10, current_element = 1
v1 = 1, c1 = 3
v2 = 2, c2 = 2

int [] arrA = {1,1,2,1,2,3,4,2,1,2,1,2};
i = 11, current_element = 2
v1 = 1, c1 = 3
v2 = 2, c2 = 3
```

Now, the second pass is needed to check whether the counters of 1 and 2 appear more than 3.

8 Filtering

Summarizing vs. filtering:

- So far: all data is useful, summarize for lack of space/time
- Now: not all data is useful, some is harmful

Classic example: spam filtering

- Mail servers can analyse the textual content
- Mail servers have blacklists
- Mail servers have whitelists (very effective!)
- Incoming mails form a stream; quick decisions needed (delete or forward)

8.1 Hash functions

The hash functions are used to speed up searching ([1]). Consider the problem of searching an array for a given value. If the array is not sorted, the search might require examining each and all elements of the array. If the array is sorted, we can use the binary search, and therefore reduce the worse-case runtime complexity to O(log n). We could search even faster if we know in advance the index at which that value is located in the array. Suppose we do have that magic function that would tell us the index for a given value. With this magic function our search is reduced to just one probe, giving us a constant runtime O(1). Such a function is called a hash function . A hash function is a function which when given a key, generates an address in the table.

A hash function that returns a unique hash number is called a universal hash function. In practice it is extremely hard to assign unique numbers to objects. The later is always possible only if you know (or approximate) the number of objects to be processed.

Thus, we say that our hash function has the following properties:

- it always returns a number for an object.
- two equal objects will always have the same number
- two unequal objects not always have different numbers

The procedure of storing objects using a hash function is the following:

- Create an array of size M.
- Choose a hash function h, that is a mapping from objects into integers 0, 1, ..., M-1.
- Put these objects into an array at indexes computed via the hash function index = h(object). Such array is called a hash table.

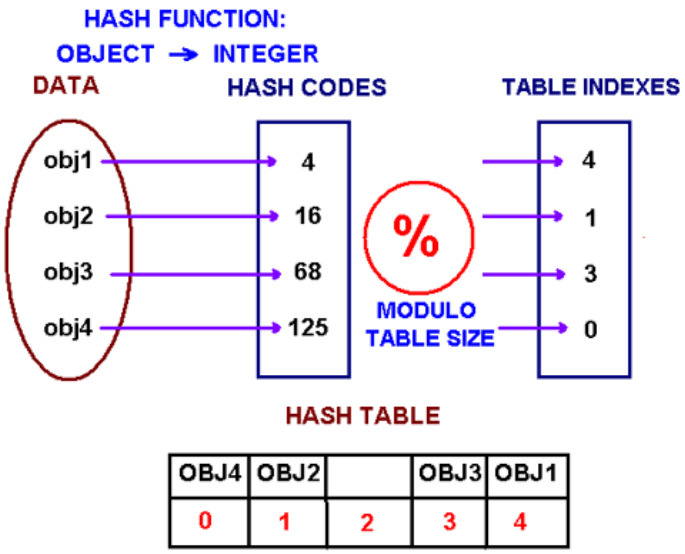

Figure 2: *Hashing example (from [1])*

8.1.1 Collisions

When we put objects into a hashtable, it is possible that different objects (by the equals() method) might have the same hashcode. This is called a collision.

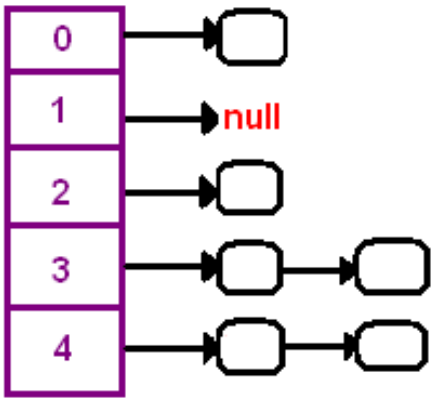

Figure 3: *Collision*

How to resolve collisions? Where do we put the second and subsequent values that hash to this same location? There are several approaches in dealing with collisions. One of them is based on idea of putting the keys that collide in a linked list. A hash table then is an array of lists. This technique is called a separate chaining collision resolution.

The big attraction of using a hash table is a constant-time performance for the basic operations add, remove, contains, size. Though, because of collisions, we cannot guarantee the constant runtime in the worst-case. Why? Imagine that all our objects collide into the same index. Then searching for one of them will be equivalent to searching in a list, that takes a liner runtime. However, we can guarantee an expected constant runtime, if we make sure that

our lists won't become too long. This is usually implemented by maintaining a load factor that keeps a track of the average length of lists. If a load factor approaches a set in advanced threshold, we create a bigger array and rehash all elements from the old table into the new one.

8.2 Standard Bloom Filter.

Problem statement

- A set S containing n values (e.g. IP addresses, email addresses, etc.)
- Working memory of sizem bit
- Goal: data structure that allows fast checking whether the next element in the stream is in W
  - return FALSE with probability 1 if the element is not in S
  - return TRUE with high probability if the element is in S

Bloom filters are very effective structures for filtering items from the streams.

A Bloom filter [13] is a space efficient structure for estimating the membership queries on the set of elements.

Let S be a set of n elements: $S = \{s_1, s_2, ..., s_n\}$. A bloom filter is an array of m bits initially set to 0. It uses k independent hash functions: $h_1, h_2, ..., h_k$ with range $\{1...m\}$. For each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$. To check if an element y is in S, we must calculate $h_i(y)$. If at least one of the bits is set to 0, it means that y is not contained in S. Otherwise, we assume that y is in S although it may be wrong with some probability. Such probability is called a false positive rate.

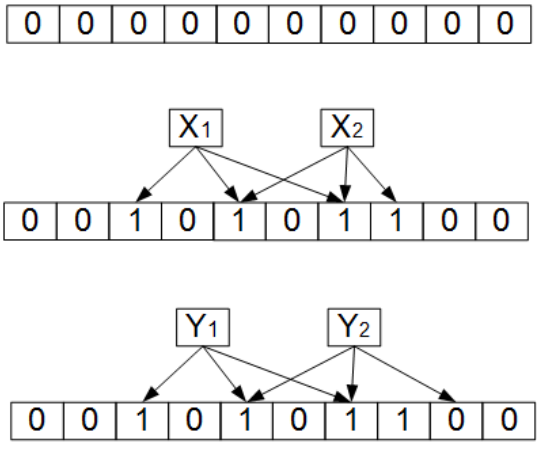

Figure 4: *Bloom Filter Example*

Figure 4 shows a Bloom Filter for which $k = 3$ and $m = 10$. At the beginning, all elements are set to 0. When two elements: x_1 and x_2 are inserted into S, the value of the hash

functions are calculated. As a results, some elements of the Bloom filter are set to 1. To verify the presence of an element in S, the hash functions for the element must be calculated. In the example, the element y_1 probably belongs to S since all hash functions return the Bloom filter element with value 1. In the case of y_2, the bit of one hash function is 0. Thus, this elements cannot belong to S.

8.2.1 False positive

There might be a case where an element can refer to the same array position as that of an another element,in those cases the new element does not change the value as 1 in its array position, but the previous element would have set the value as 1. Hence in these cases even if the element is not present in the set,its existence is returned as 1. This is called False Positives. There is no proper way to distinguish between the two cases of positives,we can just calculate the probability of false positives.

Assume that a hash function selects each array position with equal probability. If m is the number of bits in the array, the probability that a certain bit is not set to 1 by a certain hash function during the insertion of an element is: $1 - \frac{1}{m}$

If k is the number of hash functions and each has no significant correlation between each other, then the probability that the bit is not set to 1 by any of the hash functions is: $(1 - \frac{1}{m})^k$

If we have inserted n elements, the probability that a certain bit is still 0 is: $(1 - \frac{1}{m})^{kn}$

The probability that it is 1 is therefore: $1 - (1 - \frac{1}{m})^{kn}$

Now test membership of an element that is not in the set. Each of the k array positions computed by the hash functions is 1 with a probability as above. The probability of all of them being 1, which would cause the algorithm to erroneously claim that the element is in the set, is often given as:

Hence the probability of False Positives and that all bits of new element is set to 1 is: $P = (1 - [1 - \frac{1}{m}]^{kn})^k \approx (1 - e^{-kn/m})^k$ where P = Probability of False Positives

If m,n is constant, then P is minimum when: $k = log(2)(\frac{n}{m})$

For more details see [14].

Points to be noted:

- The bloom filter provides a faster way to check the existence of an element in a set.It just returns a true(1) or false(0) as results.
- Unlike a standard hash table, a Bloom filter of a fixed size can represent a set with an arbitrarily large number of elements.
- It does not do a exhaustive search to find a element in the array positions which were the result of hash function.

- The Bloom Filters are space efficient and space constant. The complexity is defined as O(k) and since k is a constant, the space complexity of a bloom filter is always a constant.
- As the Number of elements m in a n-bit Bloom filter array increases, the probability of the False Positives P increases.
- False Negative is case wherein the element x exists in the system but its existence is returned as 0(false). The False Negative cases are not permitted in Bloom Filters and hence the removal of an element from a bloom filter is not possible.

8.3 Counting Bloom Filters.

In the standard Bloom it is not possible to delete the elements from the set. It can not be accomplished by changing ones to zeros, because every bit may correspond to multiply elements. To fix this deficiency, the counting bloom filter (CBF) were introduced. It uses a counter array instead of the bit array. Thus, each element inserted into the set increments the relevant counters in the Bloom filter. In the case of deletion, the counters may be simply decremented. The drawback of CBF is a big bloom filter size. The counters must be large enough to avoid overflow.

The more space-efficient counting bloom filter was presented in [15]. However, the schema proposed in this work is more complex than the standard bloom filter, what makes the approach not practical in the implementation. In [16] the simple counting bloom filter based on d-left hashing was proposed. It offers the same functionality as a CBF but saves the space by factor of two or more.

8.4 Bloom filter applications.

- Google Bigtable, Apache HBase and Apache Cassandra, and Postgresql use Bloom filters to reduce the disk lookups for non-existent rows or columns. Avoiding costly disk lookups considerably increases the performance of a database query operation
- The Google Chrome web browser used to use a Bloom filter to identify malicious URLs. Any URL was first checked against a local Bloom filter, and only if the Bloom filter returned a positive result was a full check of the URL performed
- Bitcoin uses Bloom filters to speed up wallet synchronization
- Medium uses Bloom filters to avoid recommending articles a user has previously read
- Resource and packet routing. Bloom filters allow probabilistic algorithms for locating resources and simplify packet routing protocols. Bloom filters are also the appropriate tools for collecting the useful data in routes and other network devices.

8.4.1 Bloomjoin

One of the examples may be a bloomjoin. It is used for joining rows from distributed database relations. Instead of transferring the entire relations between the database servers, the corresponding Bloom filter is sent and after that only the rows which can take part in the join operation are considered. Utilizing the CBF enables performing more complex queries which require the join of the a fixed number of rows.

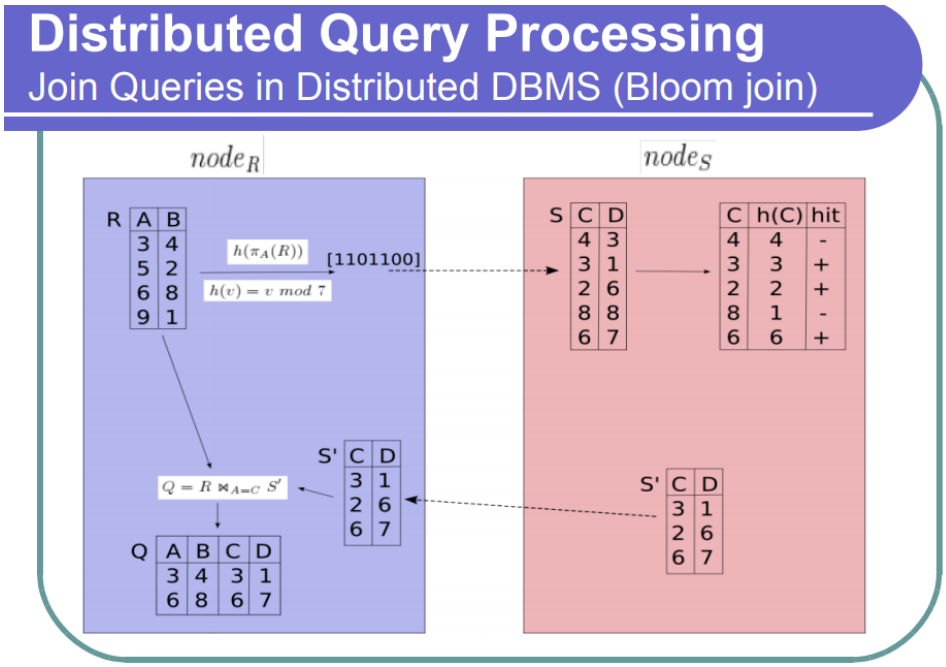

Figure 5: *Bloomjoin*

8.5 Approximate counters.

Approximate counters called also probabilistic counters, were introduced by Morris in 1978 (see [17]) and were intensively studied by Flajolet in [18].

The main idea of the approximate counters may be described as follows. The approximate counter starts with the counter value C initialized to 0. The process of incrementation of the counter C is described by Alg. 1

Algorithm 1 Approximate Counter Increment

Require: $C \geq 0$
  if Random([0,1]) $< 2^{-C}$ **then**
    C:= C+1
  end if

Let C_n denote the value of the counter C after n consecutive increments. In a detailed analysis of the approximate counter done in [18] is shown that the number $2^C - 2$ is an unbiased estimator of the number of increments n. As a consequence, we see that:

1. the random variable C_n is strongly concentrated,
2. we need approximately $\lg_2 \log_2 n$ bits to store the value of the counter C after n increments,
3. the value 2^{C_n} is a quite precise estimator of the number n of increments.

We can conclude that the probabilistic counter are more space efficient than the real counter. To store a value up to 1 million, the probabilistic counter needs approximately $\log_2 \log_2 10^6 \approx 5$ bits. On the contrary the real counter needs $\lfloor \log_2 10^6 \rfloor + 1 \approx 21$ bits.

9 Sampling

9.1 Overview

- Sampling: selection of a subset of items from a large data set. We obtain a smaller data set with the same structure.
- Goal: sample retains the properties of the whole data set.
- Estimating on a sample is often straightforward: run the analysis on the sample that you would on the full data
- Important for drawing the right conclusions from the data

Example:

Figure 6: *Sampling Example*

Real world example:
1. To test the quality of a river's water, samples of the water would be taken from multiple places in the river, on different dates and at different times of the day. All of the samples would be numbered with the numbers entered into a table. As numbers are chosen from the table, quality tests are run on the chosen sample.
2. Presidential election's prediction. After the election, the sample of population is chosen and the score is predicted. It is impractical to poll an entire population: say, all 145 million registered voters in the United States. That is why pollsters select a sample of individuals that represents the whole population.

9.2 Random Sampling

Random sampling is a technique used in selecting people or items for research. There are many techniques that can be used; but, each technique makes sure that each person or item considered for the research has an equal opportunity to be chosen as part of the group.

Suppose we see a sequence of items, one at a time. We want to keep ten items in memory, and we want them to be selected at random from the sequence. If we know the total number of items n, then the solution is easy: select 10 distinct indices i between 1 and n with equal probability, and keep the i-th elements.

Remember, there are many ways to implement random sampling. What they all have in common is that each person or item has an equal chance to be chosen.

```
scala> val r = scala.util.Random
```

```
// returns a value between 0 and 99
scala> r.nextInt(100)

// returns a value between 0.0 and 1.0
scala> r.nextFloat
```

9.3 Sampling framework

- Algorithm A chooses every incoming element with a certain probability
- If the element is sampled, A puts it into memory, otherwise the element is discarded
- Algorithm A may discard some items from memory after having added them
- For every query, A computes some function only based on the in-memory sample p(X).

9.4 Reservoir Sampling with Single Sample

The base case occurs when there is only one element sampled ($t1$). In this case, our random sample is this element. Hence the sample $S = t1$.

Lets suppose that the new element $t2$ is added. The naive approach is to restart the entire sampling process. Instead, in reservoir sampling, we accept the new element as the random sample with probability $\frac{1}{2}$. We toss a coin which returns head with probability 0.5 and if it returns head, then replace $t1$ with $t2$. We can see why S is a uniform sample. The probability that S contains $t1$ or $t2$ remains the same:
1. $Pr(S = t1) = 1 * \frac{1}{2} = \frac{1}{2}$. The random sample is $t1$ when it was selected first into S (with probability 1) and then not rejected by $t2$ with probability $1 - \frac{1}{2}$.
2. $Pr(S = t2) = \frac{1}{2}$. The random sample is $t2$ when it replaces $t1$ in the second step. This occurs with probability $\frac{1}{2}$

Then, lets assume that the new element $t3$ is added. We accept the new element as the random sample with probability $\frac{1}{3}$. We toss a coin which returns head with probability 0.33 and if it returns head, then replace the previous value of S ($t1$ or $t2$) with $t3$. More generally when inspecting the i-th element, accept it with probability $\frac{1}{i}$.

It might look as if we are treating $t3$ unfairly because we only accept it with probability 0.33. But we can show probabilistically that the sample is still uniform. The probability that S is $t1$ or $t2$ or $t3$ remains the same.

1. $Pr(S = t1) = 1 \times \frac{1}{2} \times \frac{2}{3} = \frac{1}{3}$. The only scenario when random sample is still $t1$ occurs when it was selected first into S (with probability 1), not rejected by $t2$ with probability $1 - \frac{1}{2}$ and not rejected by $t3$ with probability $1 - \frac{1}{3} = \frac{2}{3}$.
2. $Pr(S = t2) = \frac{1}{2} \times \frac{2}{3} = \frac{1}{3}$. The random sample is $t2$ when it replaces $t1$ in the second step. This occurs with probability $\frac{1}{2}$. Then in the next step is not replaced by $t3$. This occurs with probability $1 - \frac{1}{3} = \frac{2}{3}$.
3. $Pr(S = t3) = \frac{1}{3}$. The random sample is $t3$ when S contains either $t1$ or $t2$ and it is replaced by $t3$. This occurs with probability $\frac{1}{3}$.

The pseudo code after i-th items looks like this:

```
S = t1
for i = 2 to N
    Replace S with element t_i with probability 1/i
```

9.5 Reservoir Sampling with Multiple Samples

A very similar approach works when the sample size is more than 1. Lets say that we need a sample of size s. Then we initially set the first s elements as the sample. The next steps is a bit different. In the previous case, there was only one sample so we replaced the sample with the selected element. When sample size is more than 1, then this steps splits to two parts :

- Acceptance : For any new element t_i, we need to decide if this element enters the sample. This occurs with probability $\frac{s}{i}$.
- Replacement: Once we decided to accept the new element into the sample, some element already in the sample needs to make way. This is usually done randomly. We randomly pick a element in the sample and replace it with element t_i.

The pseudo code looks like :

```
Store first s elements into S
for i = s+1 to N
    Accept element t_i with probability s/i
    If accepted, replace a random element in S with element t_i
```

A coding trick that avoids the "coin tossing" by generating a random index and then accepts it if it is less than our sample size. The pseudo code looks like :

```
Store first s elements into S
for i = s+1 to N
    randIndex = random number between 1 and i
    if randIndex <= s
        replace element at index "randIndex" in the sample with
t_i
```

We can similarly analyze that the classical reservoir sampling does provide a uniform random sample.

9.6 Example: Sample size 10

As we do not always know the exact n in advance, the possible solution is the following:

- Keep the first ten items in memory
- When the i-th item arrives (for $i > 10$)
  - with probability $10/i$, keep the new item (discard an old one, selecting which to replace at random, each with chance 1/10)

- with probability $1 - 10/i$, keep the old items (ignore the new one)

Thus:

- when there are 10 items or fewer, each is kept with probability 1
- when there are 11 items, each of them is kept with probability 10/11
  - for the new item is 10/11
  - for the old item is (1)(1/11 + (10/11)(9/10)) = 1/11 + 9/11 = 10/11. In other words, the 10 old items are kept either if the new one is not selected 1/11 or the new one is selected to replace one of the other 9 items 10/11 9/10, and since "or" is represented with addition, we derive (1/11 + (10/11)(9/10))
- when there are 12 items, the twelfth item is kept with probability 10/12, and each of the previous 11 items are also kept with probability (10/11)(2/12 + (10/12)(9/10)) = (10/11)(11/12) = 10/12
- by induction, it is easy to prove that when there are n items, each item is kept with probability $10/n$

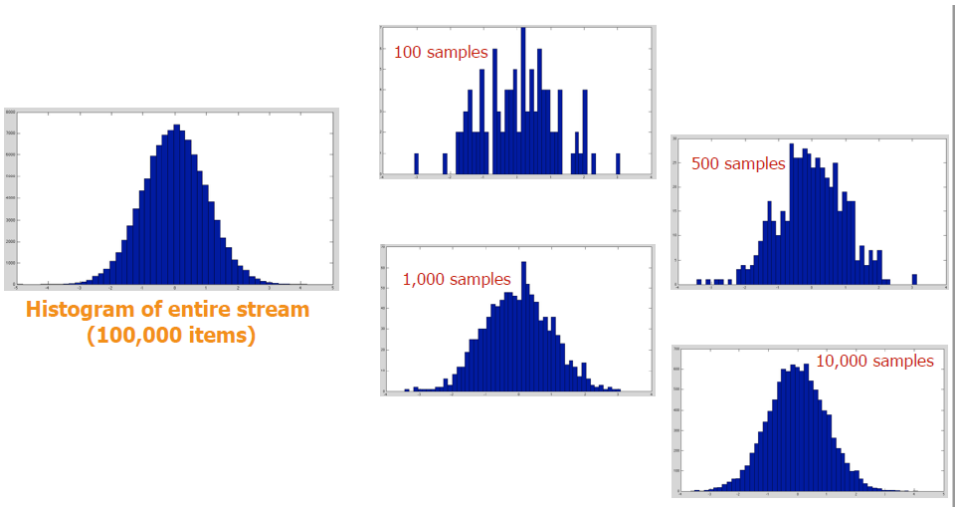

Figure 7: *Reservoir sampling example*

The proof by induction:

- Assume that after n elements, the sample contains each element seen so far with probability s/n
- We need to show that after seeing element n+1 the sample maintains the property: sample contains each element seen so far with probability $s/(n+1)$

Base case:

- After we see n=s elements the sample S has the desired property: Each out of n=s elements is in the sample with probability $s/s = 1$

Proof:

- Inductive hypothesis: After n elements, the sample S contains each element seen so far with probability s/n
- Now element n+1 arrives. It is stored with probability $s/n + 1$
- Inductive step: For elements already in S, probability that the algorithm keeps it in S is: $(1 - \frac{s}{n+1}) + (\frac{s}{n+1}) * (\frac{s-1}{s}) = \frac{n}{n+1}$. The first: element $n + 1$ discarded, the second: element $n + 1$ not discarded, the third: element in the sample not picked.
- So, at time n, tuples in S were there with probability $\frac{s}{n}$.
- Time $n- > n + 1$, tuple stayed in S with probability $\frac{n}{n+1}$.
- So probability tuple is in S at time $n + 1$ is $\frac{s}{n} * \frac{n}{n+1} = \frac{s}{n+1}$

9.7 Distributed/Parallel Reservoir Sampling

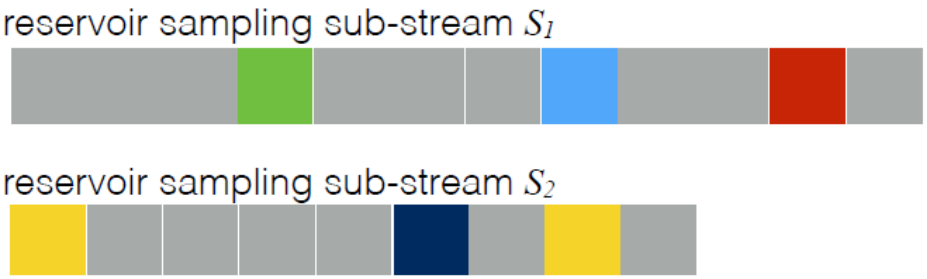

Figure 8: *Reservoir distributing sampling example*

Without loss of generality, let us assume there are two sub-streams of size m1 and m2, respectively. Both m1 and m2 are far greater than k. In the first step of the algorithm, workers work on their own sub-streams in parallel, using the basic algorithm. When both workers finish their sub-stream traversal, two reservoir lists S_1 and S_2 are generated. In addition, both workers also count the number of items in their own sub-streams during the traversal, and thus m1 and m2 are known when S_1 and S_2 are available.

The critical step is to combine the two reservoir lists to get k items out of them. To do this, we assign weights to items according to the sizes of the sub-stream where they were sampled in the first step, and then do a second sampling phase. We run k iterations for this secondary sampling. In each iteration, we flip a random coin such that, with probability $p = \frac{m_1}{m_1+m_2}$, we pick one random sample from reservoir list S_1, and with probability $1 - p$, we pick one random sample from reservoir list S_2. At the end of the k-th iteration, we will get the final reservoir list for the entire stream. This algorithm is described as follows:

Combining sub-stream pairs in 2. sampling phase. For k-iterations do:

- with probability $p = \frac{m_1}{m_1+m_2}$ pick a sample from S_1
- with probability $1 - p$ pick a sample from S_2

It can be shown by induction that in each iteration of the second sampling phase, any item in the entire stream has probability of $\frac{1}{m_1+m_2}$ being chosen. Again, by total probability, any item has probability of $\frac{k}{m_1+m_2}$ being chosen during the execution of the algorithm, and thus the algorithm generates k random samples from the entire stream. The nice part of this algorithm is that it runs in $O(max(m1, m2))$ time and uses $O(k)$ in space.

To generalize the algorithm to cases with more than two sub-streams, one only needs to combine reservoirs lists in pairs, which can also be done in parallel. The proof techniques remain the same and thus is omitted.

9.8 Minwise sampling

Goal: Given a data stream of unknown length, randomly pick k elements from the stream so that each element has the same probability of being chosen.

- For each element in the stream, tag it with a random number in the interval [0,1].
- Keep the k elements with the smallest random tags.

Distributed minwise sampling:

- Can easily be run in a distributed fashion with a merging stage (every subset has the same chance of having the smallest tags)
- Disadvantage: more memory/CPU intensive than reservoir sampling (tags need to be stored as well)

For more detail, see [19].

9.9 Sampling: applications

Q: How do you make critical calculations about the stream using a limited amount of (secondary) memory?

9.9.1 Approximate query processing

Count : $\frac{\sum_{i=1}^{n} T_i p_i}{n} = \frac{\sum_{i=1}^{n} T_i \frac{1}{N}}{n}$ where T_i is an indicator random variable that is 1 when tuple t_i satisfies our clause. p_i is the probability that tuple will be selected into the sample. n and N denote the sample size and the total size, respectively. Intuitively, the formula reweighs each tuple according to the selection probability of the tuple.

10 The Count-Distinct Problem

Suppose stream elements are chosen from some universal set. We would like to know how many different elements have appeared in the stream, counting either from the beginning of the stream or from some known time in the past.

Example:

Web site gathering statistics on how many unique users it has seen in each given month. Web site like Google that does not require login to issue a search query, and may be able to identify users only by the IP address from which they send the query.

Naive solution:

The obvious way to solve the problem is to keep in main memory a list of all the elements seen so far in the stream. Keep them in an efficient search structure such as a hash table or search tree, so one can quickly add new elements and check whether or not the element that just arrived on the stream was already seen. It does not work when the number of distinct items does not fit into the main memory.

Strategy:

We only estimate the number of distinct elements but use much less memory than the number of distinct elements

10.1 The Flajolet-Martin Algorithm

It is possible to estimate the number of distinct elements by hashing the elements of the universal set to a bit-string that is sufficiently long.

The idea behind the Flajolet-Martin Algorithm is that the more different elements we see in the stream, the more different hash-values we shall see.

Let suppose that we have a 4-bits hash function h and two different elements in the stream: s_1 and s_2. The probability that the last bit is 0 equals to $1/2$. If we have four values, the probability that the last two bits are 0 equals to $1/4$

As we see more different hash-values, it becomes more likely that one of these values will be unusual. The particular unusual property we shall exploit is that the value ends in many 0s, although many other options exist.

The process: Whenever we apply a hash function h to a stream element a, the bit string $h(a)$ will end in some number of 0s, possibly none. Call this number the tail length for a and h. Let R be the maximum tail length of any a seen so far in the stream. Then we shall use estimate 2^R for the number of distinct elements seen in the stream.

Explanation: This estimate makes intuitive sense. The probability that a given stream element a has $h(a)$ ending in at least r 0s is 2^{-r}. Suppose there are m distinct elements in the stream. Then the probability that none of them has tail length at least r is $(1 - 2^{-r})^m$. This sort of expression should be familiar by now.

We can rewrite it as $((1-2^{-r})^{2^r})^{m2^{-r}}$

Assuming r is reasonably large, the inner expression is of the form $(1-\epsilon)^{1/\epsilon}$, which is approximately $1/e$. Thus, the probability of not finding a stream element with as many as r 0s at the end of its hash value is $e^{-m2^{-r}}$.

We can conclude:

- If m is much larger than 2^r, then the probability that we shall find a tail of length at least r approaches 1.
- If m is much less than 2^r,then the probability of finding a tail length at least r approaches 0.
- We conclude from these two points that the proposed estimate of m, which is 2^R (recall R is the largest tail length for any stream element) is unlikely to be either much too high or much too low.

10.2 Combining Estimates

Let's consider many hash functions.

10.2.1 Estimate 1

Our first assumption would be that if we take the average of the values 2^R that we get from each hash function, we shall get a value that approaches the true m, the more hash functions we use. However, that is not the case, and the reason has to do with the influence an overestimate has on the average.

Consider a value of r such that 2^r is much larger than m. There is some probability p that we shall discover r to be the largest number of 0s at the end of the hash value for any of the m stream elements. Then the probability of finding $r+1$ to be the largest number of 0s instead is at least $p/2$. However, if we do increase by 1 the number of 0s at the end of a hash value, the value of 2^R doubles. Consequently, the contribution from each possible large R to the expected value of 2^R grows as R grows, and the expected value of 2^R is actually infinite.

10.2.2 Estimate 2

Another way to combine estimates is to take the median of all estimates. The median is not affected by the occasional outsized value of 2^R, so the worry described above for the average should not carry over to the median. Unfortunately, the median suffers from another defect: it is always a power of 2. Thus, no matter how many hash functions we use, should the correct value of m be between two powers of 2, say 400, then it will be impossible to obtain a close estimate

10.2.3 Combining estimates 1 and 2

There is a solution to the problem, however. We can combine the two methods. First, group the hash functions into small groups, and take their average. Then, take the median of the

averages. It is true that an occasional outsized 2^R will bias some of the groups and make them too large. However, taking the median of group averages will reduce the influence of this effect almost to nothing. Moreover, if the groups themselves are large enough, then the averages can be essentially any number, which enables us to approach the true value m as long as we use enough hash functions. In order to guarantee that any possible average can be obtained, groups should be of size at least a small multiple of $log_2 m$.

10.2.4 Space Requirements

Observe that as we read the stream it is not necessary to store the elements seen. The only thing we need to keep in main memory is one integer per hash function; this integer records the largest tail length seen so far for that hash function and any stream element. If we are processing only one stream, we could use millions of hash functions, which is far more than we need to get a close estimate.

11 Distribution of frequencies of different elements

The problem, called computing "moments" involves the distribution of frequencies of different elements in the stream.

Let m_i be the number of occurrences of the ith element for any i. Then the kth-order moment (or just kth moment) of the stream is the sum over all i of $(m_i)^k$.

For example: the second moment is the sum of the squares of the m_i. It is sometimes called the surprise number, since it measures how uneven the distribution of elements in the stream is. To see the distinction, suppose we have a stream of length 100, in which eleven different elements appear. The most even distribution of these eleven elements would have one appearing 10 times and the other ten appearing 9 times each. In this case, the surprise number is $10^2 + 10 * 9^2 = 910$. At the other extreme, one of the eleven elements could appear 90 times and the other ten appear 1 time each. Then, the surprise number would be $90^2 + 10 * 1^2 = 8110$. So, having two extreme values and the value from our stream, we can estimate the distribution. But, how to calculate the second moment of our stream?

11.1 The Alon-Matias-Szegedy Algorithm for Second Moments

For now, let us assume that a stream has a particular length n. Suppose we do not have enough space to count all the m_i for all the elements of the stream. We can still estimate the second moment of the stream using a limited amount of space; the more space we use, the more accurate the estimate will be. We compute some number of variables. For each variable X, we store:

- A particular element of the universal set, which we refer to as X.element
- An integer X.value, which is the value of the variable
- To determine the value of a variable X, we choose a position in the stream between 1 and n, uniformly and at random.

- Set X.element to be the element found there, and initialize X.value to 1. As we read the stream, add 1 to X.value each time we encounter another occurrence of X.element.

Example : Suppose the stream is a, b, c, b, d, a, c, d, a, b, d, c, a, a, b. The length of the stream is n = 15. Since a appears 5 times, b appears 4 times, and c and d appear three times each, the second moment for the stream is $5^2 + 4^2 + 3^2 + 3^2 = 59$. Suppose we keep three variables, X1, X2, and X3. Also,assume that at random we pick the 3rd, 8th, and 13th positions to define these three variables.

When we reach position 3, we find element c, so we set X1.element = c and X1.value = 1. Position 4 holds b, so we do not change X1. Likewise, nothing happens at positions 5 or 6. At position 7, we see c again, so we set X1.value = 2.

At position 8 we find d, and so set X2.element = d and X2.value = 1. Positions 9 and 10 hold a and b, so they do not affect X1 or X2. Position 11 holds d so we set X2.value = 2, and position 12 holds c so we set X1.value = 3. At position 13, we find element a, and so set X3.element = a and X3.value = 1. Then, at position 14 we see another a and so set X3.value = 2. Position 15, with element b does not affect any of the variables, so we are done, with final values X1.value = 3 and X2.value = X3.value = 2.

We can derive an estimate of the second moment from any variable X. This estimate is n(2X.value - 1).

Example : Consider the three variables from Example 4.7. From X1 we derive the estimate $n * (2X1.value - 1) = 15 * (2 * 3 - 1) = 75$. The other two variables, X2 and X3, each have value 2 at the end, so their estimates are $15 * (2 * 2 - 1) = 45$. Recall that the true value of the second moment for this stream is 59. On the other hand, the average of the three estimates is 55, a fairly close approximation.

11.1.1 Infinite stream

Technically, the estimate we used for second and higher moments assumes that n, the stream length, is a constant. In practice, n grows with time. That fact, by itself, does not cause problems, since we store only the values of variables and multiply some function of that value by n when it is time to estimate the moment.

For more detail, see [19].

12 Counting Ones in a Window

12.1 The Datar-Gionis-Indyk-Motwani Algorithm

We shall present the simplest case of an algorithm called DGIM. This version of the algorithm uses $O(log^2 N)$ bits to represent a window of N bits, and allows us to estimate the number of 1s in the window with an error of no more than 50%.

Since we only need to distinguish positions within the window of length N, we shall represent timestamps modulo N, so they can be represented by $log_2 N$ bits. If we also store the total number of bits ever seen in the stream (i.e., the most recent timestamp) modulo N, then we can determine from a timestamp modulo N where in the current window the bit with that timestamp is.

We divide the window into buckets consisting of:

- The timestamp of its right (most recent) end.
- The number of 1s in the bucket. This number must be a power of 2, and we refer to the number of 1s as the size of the bucket.

To represent a bucket, we need $log_2 N$ bits to represent the timestamp (modulo N) of its right end. To represent the number of 1's we only need $log_2 log_2 N$ bits. The reason is that we know this number i is a power of 2, say 2^j, so we can represent i by coding j in binary. Since j is at most $log_2 N$, it requires $log_2 log_2 N$ bits. Thus, $O(log N)$ bits suffice to represent a bucket.

There are six rules that must be followed when representing a stream by buckets:

- The right end of a bucket is always a position with a 1.
- Every position with a 1 is in some bucket.
- No position is in more than one bucket.
- There are one or two buckets of any given size, up to some maximum size.
- All sizes must be a power of 2.
- Buckets cannot decrease in size as we move to the left (back in time).

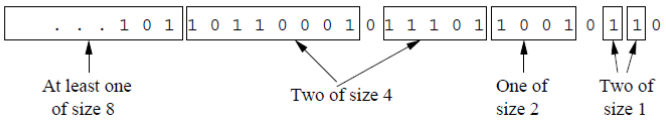

Figure 9: *A bit-stream divided into buckets following the DGIM rules*

Example: Figure 9 shows a bit stream divided into buckets in a way that satisfies the DGIM rules. At the right (most recent) end we see two buckets of size 1. To its left we see one bucket of size 2. Note that this bucket covers four positions, but only two of them are 1.

Proceeding left, we see two buckets of size 4, and we suggest that a bucket of size 8 exists further left. Notice that it is OK for some 0's to lie between buckets. Also, observe from Figure 9 that the buckets do not overlap; there are one or two of each size up to the largest size, and sizes only increase moving left.

12.2 Query Answering in the DGIM Algorithm

Suppose we are asked how many 1's there are in the last k bits of the window, for some $1 <= k <= N$. Find the bucket b with the earliest timestamp that includes at least some of the k most recent bits. Estimate the number of 1's to be the sum of the sizes of all the buckets to the right (more recent) than bucket b, plus half the size of b itself.

Example: Suppose the stream is that of Figure 9, and k = 10. Then the query asks for the number of 1's in the ten rightmost bits, which happen to be 0110010110. Let the current timestamp (time of the rightmost bit) be t. Then the two buckets with one 1, having timestamps $t-1$ and $t-2$ are completely included in the answer. The bucket of size 2, with timestamp $t-4$, is also completely included. However, the rightmost bucket of size 4, with timestamp $t-8$ is only partly included. We know it is the last bucket to contribute to the answer, because the next bucket to its left has timestamp less than $t-9$ and thus is completely out of the window. On the other hand, we know the buckets to its right are completely inside the range of the query because of the existence of a bucket to their left with timestamp $t-9$ or greater. Our estimate of the number of 1s in the last ten positions is thus 6. This number is the two buckets of size 1, the bucket of size 2, and half the bucket of size 4 that is partially within range. Of course the correct answer is 5.

The estimate is at least 50% of c (c is a correct answer).

12.3 Maintaining the DGIM Conditions

Suppose we have a window of length N properly represented by buckets that satisfy the DGIM conditions. When a new bit comes in, we may need to modify the buckets, so they continue to represent the window and continue to satisfy the DGIM conditions. First, whenever a new bit enters:

Check the leftmost (earliest) bucket. If its timestamp has now reached the current timestamp minus N, then this bucket no longer has any of its 1's in the window. Therefore, drop it from the list of buckets.

Now, we must consider whether the new bit is 0 or 1. If it is 0, then no further change to the buckets is needed. If the new bit is a 1, however, we may need to make several changes. First:

Create a new bucket with the current timestamp and size 1.

If there was only one bucket of size 1, then nothing more needs to be done. However, if there are now three buckets of size 1, that is one too many. We fix this problem by combining the leftmost (earliest) two buckets of size 1.

To combine any two adjacent buckets of the same size, replace them by one bucket of twice the size. The timestamp of the new bucket is the timestamp of the rightmost (later in time) of the two buckets.

Combining two buckets of size 1 may create a third bucket of size 2. If so, we combine the leftmost two buckets of size 2 into a bucket of size 4. That, in turn, may create a third bucket of size 4, and if so we combine the leftmost two into a bucket of size 8. This process may ripple through the bucket sizes, but there are at most $log_2 N$ different sizes, and the combination of two adjacent buckets of the same size only requires constant time. As a result, any new bit can be processed in $O(logN)$ time.

Example: Suppose we start with the buckets of Figure 9 and a 1 enters. First, the leftmost bucket evidently has not fallen out of the window, so we do not drop any buckets. We create a new bucket of size 1 with the current timestamp, say t. There are now three buckets of size 1, so we combine the leftmost two. They are replaced with a single bucket of size 2. Its timestamp is $t - 2$, the timestamp of the bucket on the right (i.e., the rightmost bucket that actually appears in Figure 9).

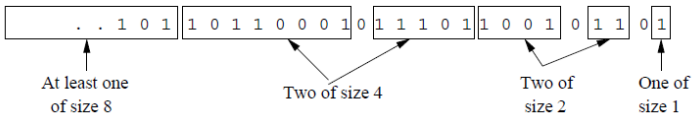

Figure 10: *Modified buckets after a new 1 arrives in the stream*

There are now two buckets of size 2, but that is allowed by the DGIM rules. Thus, the final sequence of buckets after the addition of the 1 is as shown in Figure 9.

12.4 Decaying Windows

We have assumed that a sliding window held a certain tail of the stream, either the most recent N elements for fixed N, or all the elements that arrived after some time in the past. Sometimes we do not want to make a sharp distinction between recent elements and those in the distant past, but want to weight the recent elements more heavily. In this section, we consider exponentially decaying windows, and an application where they are quite useful: finding the most common recent elements.

12.4.1 The Problem of Most-Common Elements

Suppose we have a stream whose elements are the movie tickets purchased all over the world, with the name of the movie as part of the element. We want to keep a summary of the stream that is the most popular movies currently. While the notion of currently is imprecise, intuitively, we want to discount the popularity of a movie like Star WarsEpisode 4, which sold many tickets, but most of these were sold decades ago. On the other hand, a movie that sold n tickets in each of the last 10 weeks is probably more popular than a movie that sold 2n tickets last week but nothing in previous weeks.

One solution would be to imagine a bit stream for each movie. The ith bit has value 1 if the ith ticket is for that movie, and 0 otherwise. Pick a window size N, which is the number of most recent tickets that would be considered in evaluating popularity. Then, use the method of the previous section to estimate the number of tickets for each movie, and rank movies by their estimated counts. This technique might work for movies, because there are only

thousands of movies, but it would fail if we were instead recording the popularity of items sold at Amazon, or the rate at which different Twitter-users tweet, because there are too many Amazon products and too many tweeters. Further, it only offers approximate answers.

An alternative approach is to redefine the question so that we are not asking for a count of 1s in a window. Rather, let us compute a smooth aggregation of all the 1s ever seen in the stream, with decaying weights, so the further back in the stream, the less weight is given. Formally, let a stream currently consist of the elements $a_1, a_2, ..., a_t$, where a_1 is the first element to arrive and at is the current element. Let c be a small constant, such as 10^{-6} or 10^{-9}. Define the exponentially decaying window for this stream to be the sum:
$\sum_{n=1}^{t-1} a_{t-i}(1-c)^i = 1$
However, when a new element a_{t+1} arrives at the stream input, all we need to do is:

- Multiply the current sum by 1 - c.
- Add a_{t+1}.

The reason this method works is that each of the previous elements has now moved one position further from the current element, so its weight is multiplied by 1 - c. Further, the weight on the current element is $(1-c)^0 = 1$, so adding a_{t+1} is the correct way to include the new element's contribution.

12.4.2 Finding the Most Popular Elements

Let us return to the problem of finding the most popular movies in a stream of ticket sales. We shall use an exponentially decaying window with a constant c, which you might think of as 10^{-9}. That is, we approximate a sliding window holding the last one billion ticket sales. For each movie, we imagine a separate stream with a 1 each time a ticket for that movie appears in the stream, and a 0 each time a ticket for some other movie arrives. The decaying sum of the 1's measures the current popularity of the movie. We imagine that the number of possible movies in the stream is huge, so we do not want to record values for the unpopular movies. Therefore, we establish a threshold, say $1/2$, so that if the popularity score for a movie goes below this number, its score is dropped from the counting. For reasons that will become obvious, the threshold must be less than 1, although it can be any number less than 1. When a new ticket arrives on the stream, do the following:

- For each movie whose score we are currently maintaining, multiply its score by $(1-c)$.
- Suppose the new ticket is for movie M. If there is currently a score for M, add 1 to that score. If there is no score for M, create one and initialize it to 1.
- If any score is below the threshold $1/2$, drop that score.

It may not be obvious that the number of movies whose scores are maintained at any time is limited. However, note that the sum of all scores is $1/c$. There cannot be more than $2/c$ movies with score of $1/2$ or more, or else the sum of the scores would exceed $1/c$. Thus, $2/c$ is a limit on the number of movies being counted at any time. Of course in practice, the ticket sales would be concentrated on only a small number of movies at any time, so the number of actively counted movies would be much less than $2/c$.

References

[1] S.Adamchik, V.: Concept of hashing. (https://www.cs.cmu.edu/~adamchik/15-121/lectures/Hashing/hashing.html) 32, 33

[2] Hicks, M.: Scala vs. java: Why should i learn scala? (https://www.toptal.com/scala/why-should-i-learn-scala) 4

[3] Aggarwal, S.: Scala tutorial for java developers. (https://examples.javacodegeeks.com/jvm-languages/scala/scala-tutorial-for-java-developers/) 4

[4] Schinz, M., Haller, P.: A scala tutorial for java programmers. (https://docs.scala-lang.org/tutorials/scala-for-java-programmers.html) 4

[5] Martin, O.: Programming in Scala. 3 edn. Artima Inc (2016) 4, 13, 17

[6] Alexander, A.: How to use closures in scala. (https://alvinalexander.com/scala/how-to-use-closures-in-scala-fp-examples) 17

[7] Point, T.: Scala - collections. (https://www.tutorialspoint.com/scala/scala_collections.htm) 21

[8] Manolache, F.: Data stream algorithms. (http://romania.amazon.com/techon/presentations/DataStreamsAlgorithms_FlorinManolache.pdf) 24

[9] Puripunpinyo, H.: Java 8 vs. scala, part ii: the streams api. (https://dzone.com/articles/java-8-vs-scalapart-ii-streams-api) 24

[10] Eichar, J.: Streams 2: Stream construction. (http://daily-scala.blogspot.com/2010/01/streams-2-stream-construction.html) 24

[11] Boyer, R.S., Moore, J.S.: Mjrty – a fast majority vote algorithm (1982) 27

[12] Misra, J., Gries, D.: Finding repeated elements. Sci. Comput. Program. **2**(2) (1982) 143–152 29

[13] Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM **13**(7) (1970) 422–426. Available from: http://doi.acm.org/10.1145/362686.362692 34

[14] P, R.M.: A case of false positives in bloom filters. (https://medium.com/blockchain-musings/a-case-of-false-positives-in-bloom-filters-da09ec487ff0) 35

[15] Pagh, A., Pagh, R., Rao, S.S.: An optimal bloom filter replacement. In: Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms. SODA '05,

Philadelphia, PA, USA, Society for Industrial and Applied Mathematics (2005) 823–829. Available from: http://dl.acm.org/citation.cfm?id=1070432.1070548 36

[16] Bonomi, F., Mitzenmacher, M., Panigrahy, R., Singh, S., Varghese, G.: An improved construction for counting bloom filters. In: Proceedings of the 14th Conference on Annual European Symposium - Volume 14. ESA'06, London, UK, UK, Springer-Verlag (2006) 684–695 36

[17] Morris, R.: Counting large numbers of events in small registers. Commun. ACM **21**(10) (1978) 840–842 37

[18] Flajolet, P.: Approximate counting: A detailed analysis. BIT **25**(1) (1985) 113–134 37

[19] Ullman, J.: Mining data streams. (http://infolab.stanford.edu/~ullman/mmds/ch4.pdf) 44, 48