CMSC 621
Case Study on Programming Languages

Naveen Bansal, Vineet Ahirkar, Shantanu Hirlekar
Department of Computer Science and Electrical Engineering
University of Maryland Baltimore County
Maryland, USA
nbansal1@umbc.edu

Abstract - Choosing the correct programming language is very critical for the success of a software project. If the decision is not taken carefully, it can prove out disastrous to the organization. The benchmarks present over the internet lacks insight and may not be relatable to a project. This paper proposes the idea of identifying the domains to which a project can belong and benchmark programs developed in those domains. Benchmarks like Time and Memory are used to compare the performance of each language. The languages are further compared on some of the theoretical aspects. A project can be aligned with the domains identified and the language for the project can be decided based on the benchmarks results for that domain. Some other tradeoffs mentioned in result section should also be accounted while making the decision.

Keywords - Go, Erlang, Elixir, Scala, JVM, fault tolerance, garbage collector, concurrency, dynamic typing, message passing, database.

1. Introduction

While starting a new project the most important thing that belongs to the preparation phase of the project seems to be choosing a language for the project. The importance of this step can be analyzed from the situation in which a company is forced to move to a new language because of picking a bad language earlier. Moving a project to a new project is always painful for the company because of effort required to develop the same components in a different language and it adds a lot of cost to the company. Apart from these, compatibility of your product and data with the new language also becomes an issue. For example, say you are using a particular encryption library in Java and you decide to move the project to a new language and you find out that the same encryption algorithm is not available in the new language. The reason to move to a new language may be anyone from the following.

1) The product is not able to handle the increased user base (not scalable).
2) The language chosen is not efficient in performing the required tasks (eg - chose Python instead of Perl for log processing).
3) A particular problem with the chosen language(for eg. memory leaks in C++).
4) It's a legacy system written in a very old language and it has become very difficult to maintain or extend the product further(hard to find developers of old language).

The decision to choose a language may not be important in certain cases where the performance is not a bottleneck, like a college project or a software system to cater the small user base.

Let's take a look at one of the approach suggested in "Selecting the optimal programming language" [1] .

Some of the factor to consider while choosing a programming language can be

The targeted platform - Since certain programming languages are native, they require platform compiler to produce the executables for that particular platform.

So a program developed in C if compiled on a windows platform would provide a ".exe" executable which can not be run on Linux. Similarly, Gcc compiled C programs can not be run in Windows. This is not the case with languages like Java which are platform independent. Java code is converted to bytecode by java compiler which then can be interpreted on any operating system. This compatibility problem is also applicable in the web development field where a particular browser may not be compatible with new versions of CSS or HTML.

Elasticity - Elasticity is defined as the ability of the language to easily support the addition of new functionalities to the project. The support can be from the language itself or from the libraries written for that language. The problem that often arises with the new languages is that it does not have a large number of libraries in the beginning, which poses certain challenges for organizations in the situations where they need some specific functionality from the new language to maintain the compatibility with the old system. There are two solutions to this problem, one being writing the functionality and exposing the library yourself and the other being outsourcing the missing functionality to a plugin in some languages.

The first solution seems to be good but many organizations hesitate to take this step due to the fear of facing production issues. The second solution is a kind of a workaround and is always slower than the complete system built in just one language. Some of the following questions can help you to measure the elasticity of the programming language.

Can you complete your requirement form the core language? i.e without using libraries

If the functionality is not available in the core language, is it available in any library of the language?

If the functionality is not available even in the libraries, what is the effort required to write the functionality on your own from scratch?

Before making the functionality from scratch, you should be aware of the upcoming improvement in the language or the new features coming in the new version. It may be the case that your requirement might be there in the next version of the language.

Time to production - It is the time required to deploy the software on production after the software is ready to Go live. There are certain dependencies that needs to be resolved while deploying a product. For example, various libraries mentioned in #include directives are resolved and with large scale systems having millions of lines of code, it can hours to just compile the whole source code after making any change.

Performance - Performance of a language can be measured by several factors like time taken to complete the job, total memory used, lines of code etc. These performance benchmarks should be carried out in the same environment.

Support and community - This is a very important factor to look for before choosing a particular programming language. There are numerous problems faced by developers during the software development life cycle, if a large community of developers is using the same programming language then it's highly probable that someone else might have faced the problem same as yours. If the community maintains a page or group to share the problems and solutions then it becomes very easy to quickly fix the problems and finish the development of the project. Moreover, languages like Go with the support of big companies like Google ensures the developer that the problems faced by the community will be addressed in a timely manner.

2. Related Work:

Similar concept of language comparison is used in Computer Language Benchmarks game[14]. The website is informative and gives a detailed analysis of language performance on various benchmarks. However, it fails to show which language would be good for commonly used applications or projects requiring the use of database or network programming, etc.

3. Our Approach

We are considering the performance factor to compare the programming languages. There are numerous programming benchmarks present over the internet for several programming languages but they often do not give you any idea about the performance of language when used to perform the common tasks required in organizations. We tried to identify the domains to which a particular project can belong. The domain we identified are as follows

1) File Input/Output and data parsing - This domain refers to the domain of programs which takes the files as input and perform some computation to produce the output. The data can be in any format eg - JSON, XML etc.

2) Intensive lookup operations - This is the domain of programs which require a huge amount of data in memory and perform numerous data lookup.

3) Concurrency - This is the domain of programs which can produce the result fast by performing several independent tasks in parallel.

4) Networks - This is the domain of programs which performs communication over a network.

5) Pattern matching - This is the domain of programs which require searching a pattern in a file to perform some analysis or take some decisions. For e.g log analyzing tools, email validators.

6) Database Operation - This is the domain of programs which interact with the database to look for some information or store some information.

7) Graphics - this is the domain of programs which plots the data for the purpose of visual representation.

Programs corresponding to domains

1) JSON Parser - This programs loads a JSON file consisting of thousand entries of "id" and "price" and find the maximum priced item.

2) In Memory Cache - This program maintains a map of thousand key value pairs. A request with the request key is served from the map if present else the value is brought and stored in the map. Eviction of a random entry is made when the map is full and a new key value pair needs to be inserted.

3) Concurrency - In this program, the Fibonacci of nth term is computed in parallel.

4) Network programming - In this program, a client and server are created. The client sends a request message to the server and the server computes the length and sends the length of the message.

5) Pattern Matching - In this program, an email validator function is written and two requests are made, one with a valid email and the other with the invalid email.

6) Database operation - In this program, a query with student name is made to the database and all the details of the student are fetched and returned as a string.

7) Graph plot - In this program, a histogram of hundred data items is plotted. The data items on the x axis are all the numbers from the range [1,100] and the data on the y axis is the frequency of the corresponding items in the x axis. The frequencies are randomly generated from the range [1,100]

2.1. Languages chosen for comparison

2.1.1. Go

Go is a programming language created at Google. The project was made public in 2009. Let's see why the creators of Go felt the need of adding another language to the pile of existing programming languages. Some of the reason they mentioned on the official documentation website of Go are -

• There has been a lot of development in making computer faster but the pace of software development is still not fast.

• Dependency management is a very critical part any software project and directly affects the speed with which it can be compiled. The header files in languages like C do not seem to provide a clean dependency management and fast compilation.

• The complex type system in languages like C, C++, Java becomes difficult to manage which is why the developers are moving towards dynamically typed languages.

• The garbage collectors in other programming languages are not efficient as they can be.

• There is not much support in other languages to program on modern day multicore systems.

Targeting the above-mentioned shortcomings of other languages, the creators of Go tried their hand in creating a new language to overcome the problems.

• Large Go programs can be compiled within a few seconds and on a single system compared to some large C++ programs which may require a distributed system.

• Go provides easy dependency management

• Go doesn't have type hierarchy and the static types it has are lighter than typical OO programming languages

• Go has an efficient garbage collector with support for concurrent execution.

It can be seen from the above points that the purpose of creating Go was making the software development easy. Today the software has reached the size of millions of code which is maintained by thousands of people in parallel inside the organization. Each update requires recompilation the code which might take hours to complete. Thus it can be cumbersome to work on large projects with the traditional programming languages as most of the time will be wasted in just compiling the project and it may require several compilations to successfully make and test a change.

Go is written with the approach of built in concurrency to meet the demands modern software systems consisting of web applications, networking applications etc. Another good point about Go is that its syntax is very similar to C which makes it very easy to pick for a developer as most of the developers are already familiar with the C syntax, thus a developer can begin writing efficient code in Go in no time.

How Go makes the dependency management efficient?

Go gives you a compile time error if your code has some unused dependencies compared to warnings given by other languages such as java. This forces the developer to remove the unused dependencies making the dependency tree of the code precise. This feature also results in the less

compilation time as the compiler will not have to process the extra code.

Go doesn't have transitive includes which means that if a package A imports package B which in turn imports package C, does not make A import B. To achieve this, when the project is compiled, A will use the object file of package B rather than the source code. The object file of B has all the necessary information required to support the tasks in the interface of B. This strategy solves the problem of multiple includes faced in C or C++.

Go doesn't allow circular imports. Thus the compiler throws an error if it finds any circular imports. Circular imports make the compiler process a large number of source code files at once and results in a slower build process. Go believes that it is better to copy a small amount of code than to use a big library for just a small thing.

Concurrency in Go

Concurrency model in Go is a variant of Communicating Sequential Process(CSP) model with first class channels. CSP can be added to a procedural language easily and without affecting the core model of the language. This allows us to execute several independent functions parallelly with each function being powerful enough to carry out even massive computational tasks.

Go achieves concurrency using goroutines. A normal function can be made a Go routine by just writing Go before the function name. Thus a call like

```
go Sum(x, y)
```

will carry out the computation of the sum of x and y concurrently. If we want to receive the value returned by Go routine then we can create a channel to communicate with the Go routine. This is how we can channel to get the sum.

```
func sum(x int, y int, c chan int){
    c<-x+y
}
func main() {
    ch := make(chan int, 1)
    sum(1,2,ch)
    res := <-ch
    fmt.Println(res)
}
```

Garbage collection in Go

Creators of Go felt that too much programming effort is required to manage memory in languages like C and C++ which burdens the developer and reduces the speed of development. This is why they decided to keep a garbage collector in Go. In a concurrent programming environment, memory management can be really challenging and thus favour the demand for automatic memory management.

Go has a concurrent, tri-color, mark-sweep garbage collector which was originally proposed by Dijkstra in 1978. Go's garbage collector has a single knob known as GOGC. Thus you can adjust the value of this knob to vary the duration of GC pause. If you want short GC pause then increase the GOGC value otherwise, you can lower it to get more memory but at the cost of greater GC time.

Packages

Go package system's design is a combination of libraries, modules, and namespaces. A Go source file like /shapes/triangle. Go will have a package definition like

package triangle.

Package names in Go are kept concise, for example, the package name above is 'triangle'. To use a package, we can import it using the statement like

```
import "path_to_package/package_name"
```

In this way, the path is defined by the actual directory structure of the repository and not the language. To maintain clarity, Go requires you to use the package name before using anything from the package in the source file which is importing the package. For ex. If you want to use Abs function from math package then you will use it like

```
var absx = math.Abs(x)
```

Go does not require package name to be unique but it requires package path to be unique. Therefore if we need to use two log packages, one being standard and other being your organization specific then you can import them as

```
import "log"
import locallog "organization_name/go/log"
```

Here we are providing the local name to the log package of organization and we can use it like

```
log_one := locallog.Print(err)
```

Remote packages

The package path for remote packages is the URL of the repository serving that package. Thus the package dependency “github.com/gonum/plot” can be fetched with the ‘go get’ command like

```
go get "github.com/gonum/plot"
```

This command fetches the dependency “github.com/gonum/plot” and installs it. After the dependency is installed we can import like the way we include local packages. Thus the above installed dependency will be included as

```
import "github.com/gonum/plot"
```

Syntax

As mentioned earlier the syntax of Go is very similar to that of C. An integer variable a is defined as

```
var a int = 22
```

Function is defined as

```
func sum(x int, y int) int {
        return x+y;
}
```

We can return multiple arguments functions in Go. Thus a program to swap two integers can be written as

```
a, b = swap(a, b)
```

Where swap function can be defined as

```
func swap(x int, y int) (int, int) {
        return y, x;
}
```

A method in Go can be defined in the same way as we define functions by just writing the receiver before the function name. Here is an example

```
type Vehicle struct {
    wheelCount int
}

func main() {
    fmt.Println("Hello, playground")
    a := Vehicle{4}
    fmt.Println(a.getWheelCount())
}
```

```
func (a Vehicle) getWheelCount() (int) {
        return a.wheelCount
}
```

Naming

The approach of Go is very different in defining the visibility of an identifier. Visibility of an identifier is determined from the case of the initial letter. An entity is exportable(public) if the initial letter of its name is written in uppercase otherwise it is not.

Names in Go are local which means that we can define a function with the same name in different packages. Thus if a function name “Sum” is defined in package A and package B and package A imports package B, there is no conflict. If a call to Sum of package B needs to be made, then it has to be made like this

```
result := B.Sum()
```

On the other hand call to local Sum function will be like this

```
result := Sum()
```

Semantics

The semantics of Go are like that of C. It is a compiled language, statically typed, supports pointers. It is a procedural language.

To make Go more robust there are some changes in the semantics of Go compared to C.

• It does not have pointer arithmetic. Which means we can not increment or decrement a pointer to get next or previous address respectively

• There are no implicit type conversions. If we perform some arithmetic operation of two variable of different types, the compiler will give us an error. Thus we have to explicitly cast the variables as per requirement.

• Unary increment or decrement operators are statements are statements, not expressions. Thus an operation like y := x++ is invalid whereas x++ is not.

•Assignment is not an expression thus a statement like while(c! = io.EOF) are invalid.

Composition instead of Inheritance

Go has methods different than java in the sense that they can be called with any type rather than just classes.

Thus there is no type hierarchy in go. Go achieves inheritance using interfaces. Go interface is a set of methods and all the types which implements all the methods of the interface are said to satisfy that interface implicitly. Which means there is no declaration like Implements. Interface can be defined as

```
type Vehicle interface {
        start()
        stop()
}
```

Thus any type which implements start and stop methods is said to satisfy 'Vehicle interface'. A good point about Go is that any change in the interface just affects the direct clients of the interface as there is no subclassing. This saves the effort of updating the whole hierarchy tree as in other languages.

Errors

Go does not have exceptions, it has an error interface to handle errors.

```
type error interface {
    Error() string
}
```

Thus Errors method returns the description of the error in string format. Makers of Go designed it in such a way that errors are treated values and programs compute them just like the value is computed for other types. Thus if no error occurs then function will return nil as value of error otherwise, the value of error is returned

Tools

Tools in Go can be written with great ease as its syntax, package system and other features were designed this way. One of the most popular tools written in Go is 'gofmt' which formats the Go source code. This solves the problem of different indentation convention by different programmers by letting machine format your code.

Another popular tool that is written in Go is 'gofix' which propagates changes to your code base, based on the changes made in the language. Thus it upgrades your application to the new Go version with great ease.

2.2.2 Elixir

History

Elixir is a general purpose language created in 2012 by Jose Valim. It is compiled on the Erlang Virtual Machine and has thus reaped the benefits of the Erlang language. So, although it is quite a new language it has a solid base architecture.[3]

Erlang and its benefits

Erlang was made by 'Joe Armstrong, Robert Virding, and Mike Williams' in 1986 at 'Ericsson'. Erlang was developed for the telecom industry for efficiently managing switches and is thus built to be very scalable and fault tolerant. At the same time, it is very fast, concurrent & functional. Features like these were rarely seen in a language created 30 years ago.
The famous mobile messenger application 'Whatsapp' uses Erlang for a major part of their application. On several occasions they showcased how they were able to scale so efficiently using Erlang. There are a huge contributor to the Erlang VM and have brought out the best in the out of it.[6]

Why Elixir, why not Erlang?

Erlang syntax is quite different from the usual programming languages such as C, Java, Python, etc, making it very difficult to read for the average programmer. This causes a major setback in drawing people's attention to this language. Explaining why, being so old, isn't very popular. Productivity also takes a major toll due to this. Due to less popularity, it is hard to find people efficient in this language. It misses the cool features of the new languages such as -

- Efficient build tools
- Rapid prototyping
- Easy syntax

Elixir over Erlang

The creator of Elixir, 'Jose Valim', was previously a ruby developer and part of the 'Rails Core Team'. He started exploring newer technologies when he found out the flaws of Ruby in terms of concurrency. It was then Erlang which caught his attention. His past experience in Ruby is reflected in the design of the Elixir. Not only is the syntax quite similar to Ruby, he has also brought in many amazing features such as Meta Programming, Rich set of core APIs. The best part is that Erlang syntax is accepted in Elixir. So all the code from erlang can directly be copied over to Elixir without any worries. Also, all the third party libraries written in erlang can be used in Elixir applications.

Basic Conventions

Datatypes

```
# atoms
:language_comparison

# char lists
'Language Comparison'
```

```
# string
"Language Comparison"

# list
[1,2,3]

# tuple
{:ok, 'hello'}

# map
%{"name" => "language_comparison"}
```

No Iterators (Only recursion)

Pipeline Operator

Like we have the '_' variable in Linux terminal, we have the pipeline operator in Linux. It can be used for chaining methods to each other leading to concise code.

```
# Pipeline Operator
"Elixir rocks" |> String.split
```

Protocols

Protocols are similar to what we call Interfaces in Java. We can use it to declare some functions which will then be implemented by some other class. It is used to create multiple inheritance patterns in Elixir.

```
# Protocols
defprotocol MathOp do
  def max(data)
end

defimpl MathOp, for: List do
  def max(tuple), do: Enum.max(tuple)
end

IO.puts MathOp.max([1,2,3])
```

Dynamic & Weakly Typed

Being a compiled language, Elixir is dynamically typed, means that data types are not checked at compile time. This feature has both plus and minus points. Plus point is that syntax not becomes simpler, putting the programmer at ease.

Minus point is that it increases the chance of a breakdown in the system, if the data types are not neatly handled.
This is because all the errors are now shifted from compile time to runtime, so unless the buggy part of the code is executed, the programmer might not be able to identify the problem.

Pattern Matching

Everything is an expression in Elixir.

```
defmodule Greetings do
        def greet(%{"first_name" => fname,
"last_name" => ""}) do
                IO.puts "Hii #{fname} !"
        end
```

```
        def greet(%{"first_name" => fname,
"last_name" => lname}) do
                IO.puts "Hello #{fname} #{lname}
!"
        end
end

Greetings.greet(%{"first_name" => "Vineet",
"last_name" => ""})
Greetings.greet(%{"first_name" => "Vineet",
"last_name" => "Ahirkar"})
```

Functional Programming

Elixir takes its functional programming roots from Erlang. One of the most important features of Elixir which makes it purely functional is that only Immutable data structures exist. This also helps in concurrency and parallel processing.

Functional programming constructs supported in Elixir are as follows

1) Anonymous functions - These are unnamed functions also called First class functions which are nothing but (blocks of code, similar to lambdas) functions which can be passed around and reused.

2) Composing (currying) - Executing a set of functions by passing them as parameters to each other.

3) Closures - A first-class function which remembers the values of all the variables in scope at the time of creation.

4) Higher-order functions - A function which accepts one or more functions as parameters and returns a function.

Concurrency

Many languages are based on the traditional way of thread-based concurrency. But, in Elixir we have the Actor model of concurrency.

Erlang Process (Actors)

Erlang Processes or Actors are nothing but lighter version of the usual threads which are independent of the underlying Operating System. They are commonly called as 'Green threads' in other languages. Switching between such threads takes around 20 nanosec as opposed to 500 nanosec taken by usual threads. Due to the granularity of the Erlang process, we can ensure that failure of one process doesn't affect the entire process. This helps in creating a loosely coupled architecture.

Spawning a new process in Elixir is done using the 'spawn' command. Each of these processes has a PID by which they are addressed and scheduled by the Erlang Virtual Machine using a preemptive scheduling mechanism. Also, the Erlang Runtime Environment allows Symmetric Multi-Processing (SMP) which schedules the

processes in parallel on multiple CPUs thus giving true parallelism.

Message Passing via Channels

There are two models of communication between processes -

• Message Passing
• Shared Memory

Elixir is based on the Message Passing model.

Message Passing is a scheme by which two processes communicate by sending messages to each other instead of accessing shared state. All communications in this pattern are asynchronous, ie, senders do not need to wait for receivers to receive and process the message. This results in a significant boost in performance. This kind of a concurrency model has been highly critiqued by many people in the industry because of the high volume of data flow it causes. It is due to this reason there is no pass by reference in Elixir.

In Elixir, message passing is done using two main commands - 'send' & 'receive'.

```
defmodule MathOp do
        def multiply(pid, a,b) do
                send(pid, a*b)
        end

        def square(n) do
                spawn(MathOp, :multiply, [self,
n, n])

            receive do
                        x ->
                                IO.puts "Square
of #{n} is #{x}"
                end
        end
end

MathOp.square(5)
```

Process Linking

When two processes spawn another process there is a medium of communication between them. We can link two processes by using the 'spawn_link' command by which they will receive exit notifications.

Process Monitoring

We can also set up a process monitor for a spawned process. When the process fails we get a message about the details of the incident.

Agents

Agents in Elixir are an abstraction to store state in a process which can be accessed by other processes.

It can be thought of as a key value store, which has two commands, 'Update' to update the value and 'Get' to fetch the value.

```
defmodule MyApplicationState do
  def start do
    Agent.start_link(fn -> %{} end)
  end

  def put(pid, key, value) do
    Agent.update(pid, &Map.put(&1, key, value))
  end

  def get(pid, key) do
    Agent.get(pid, &Map.get(&1, key))
  end
end
```

Tasks

Task is an async piece of code that gets spawned into a new process and sends back a message to the calling process once the operation is performed.[5]

It is also possible to spawn a Task under supervision or dynamically add supervisors to existing Tasks.

```
# call the task asynchronously
task = Task.async(fn -> do_some_work() end)

# perform other operations
do_some_other_work()

# send response as return value whenever the task
is finished
res + Task.await(task)
```

Fault Tolerance

OTP (Open Telecom Platform)

OTP is Erlang's built-in library which provides standards for smooth inter-process communication resulting high fault tolerance support.

Elixir processes have a client-server kind of a relationship, i.e, one process requests information from another process. These processes cannot store state, due to which state has to be maintained by the These services may reside on one node or may even be on separate nodes communicating via network.

• Supervisors & Supervision Trees
• Behaviors
• Gen server

Gen server standing for 'Generic Server' is nothing but a container of functions/macros which give a beautiful API to manage communication between processes.[7]

In Elixir, since the state cannot be maintained by a sharing variables, the state can be maintained a continuous mutable state by running an endless recursion in a separate Elixir process.

```
def loop(state) do
```

```
  message = receive
  new_state = f(state, message)
  loop(new_state)
end
```

This task of creating a new process is done for us by the gen_server also taking care of many aspects such as asynchronous calls, timeouts, deadlocks, mailbox errors, etc.

Although it is built in Erlang, the 'gen_server' module has been completely ported to Elixir for easy usage.

```
defmodule Stack do
  use GenServer

  def handle_call(:pop, _from, []) do
    {:reply, nil, []}
  end

  def handle_call(:pop, _from, state) do
    [head|new_state] = state

    {:reply, head, new_state}
  end

  def handle_cast({:push, value}, state) do
    {:noreply, [value|state]}
  end
end

# start_link params - module name, initial state
{:ok, pid} = :gen_server.start_link(Stack, [],
[])
# synchronous
IO.inspect :gen_server.call(pid, :pop)

# asynchronous
IO.inspect :gen_server.cast(pid, {:push, 1})
IO.inspect :gen_server.cast(pid, {:push, 2})
IO.inspect :gen_server.call(pid, :pop)
IO.inspect :gen_server.call(pid, :pop)
```

• Finite State Machine (FSM)
• Gen Event
Spawn handlers where the response is not expected.
Eg - logging
• Gen TCP
Robust and fault tolerant socket programming. Have to write in erlang

Dependency Management

Dependency Management comes built-in with the Elixir language in the form of a tool called 'mix'.
With easy commands and configuration options is by far one of the best dependency management tools.

Some of the commonly required commands are as follows -

```
# create a new project with the name -
'my_project'
mix new my_project
```

```
# download all the dependencies from the
specified sources
mix deps.get

# compile the downloaded dependencies (can be
skipped as automatically compiles when running
the project first time)
mix.deps.compile

# compiles and executes the 'my_program.exs' file
mix run my_program.exs
```

A file 'mix.exs' is the configuration file in which we can specify the dependencies -

```
defmodule MyProject.Mixfile do
  use Mix.Project

  defp deps do
    [
      {:benchfella, "~> 0.3.0"},
      {:socket, "~> 0.3"}
    ]
  end
end
```

Garbage Collection

Based on the Erlang garbage collector, Elixir uses a tracing garbage collector with a 'Mark and Sweep' strategy to manage its dynamic memory. Usually, in any language, all the working threads are first stopped before running the GC, but in Elixir each thread is halted and run separately. So the complete program is never halted in Elixir while collecting garbage unlike other languages like Go which have a 'Stop The World' approach. Thus, the overall performance of the Elixir Garbage Collector is very good.

Hot code reloading

Hot code reloading is one thing which makes deployment simpler by reducing the time required to compile builds. This feature of Elixir is again inherited from Erlang from the OTP library. In Elixir, one can specify a certain module to be reloaded which then gets reloaded on each process one by one causing a zero downtime of the complete project. This internally uses 'gen_server' to pause the ongoing process, change the code and resume the process again.

Meta-Programming

Meta-Programming is nothing but 'code which writes code' and is very similar to metaprogramming in the Ruby language. It gives the ability to add functionality to the existing code on the go. The internal representation of Elixir code is an Abstract Search Tree (AST). Internally, metaprogramming directly modifies the AST at compile time.[4]

There are three ways to create quoted expressions in Elixir -

1) Manually construct it
2) Macro.escape
3) quote/unquote to compose AST

```
# dynamic generation of functions and nested
macros using Unquote fragment.

defmodule MyModule do
  Enum.each [foo: 1, bar: 2, baz: 3], fn { k, v }
->
    def unquote(k)(arg) do
      unquote(v) + arg
    end
  end
end

IO.puts MyModule.foo(1)
```

Web development

Elixir as a language is suitable for web development because of its easy syntax, concise code which leads to rapid development. There are many web development frameworks in Elixir but the one that is gaining the most attention is the Phoenix framework.

Phoenix

Phoenix is a fast concurrent framework having a productive stack and an emerging community. It is very similar to the Ruby on Rails web framework but is extremely fast compared to it. It follows the MVC architecture - Models, Views & Controllers. Phoenix absorbs out all the best parts out of Elixir, but its ecosystem still needs to develop.

2.3.3 Scala

Introduction of Scala

Scala stands for scalable language. That is, it can grow with the programmer over the time. Scala can be modified to suit a person's needs. We can type a one line code or many lines of code and play with Scala due to its flexible nature. Many people feel that Scala is a scripting language due to the fact that compilers can infer the types and the code syntax is concise. Many developers liked Scala so much that the language won the ScriptBowl contest at the 2012 JavaOne conference. There's a very well and careful integration of functional programming and object oriented programming at the root of scala. [9]

Scala is pure-bred object oriented language. In Scala, every value is an object with all operations being method calls. Using implicit classes, Scala allows to even use the functionalities of Java.

Scala has all the features of a functional programming language. It has various data structures which are mutable and immutable. Scala also supports first class functions as well. At first, Scala can be used just like Java but without semicolons. Gradually, one can get the feel for the functional programming features of scala. Also, one notable feature of Scala that led to its popularity is the fact that Scala runs on Java Virtual Machine. Therefore, Java and Scala classes can easily be mixed. All the popular Java libraries and frameworks can be used with Scala. Examples being Eclipse, NetBeans, Spring, Hibernate, etc. Scala is very useful in the case when it comes to scalable architecture which makes use of concurrency and distributed processing in the cloud. It's functional nature makes it easier and flexible to introduce concurrency and multithreading.

Scala compiler is flexible and proven highly reliable over the years. Martin Odersky has written the compiler as well as is called the founder of Scala. Odersky has also written the Java reference compiler. Therefore, we can be very sure about the reliability of the compiler of Scala.

The History & Concept of Scala:

Martin Odersky who is known as the creator of Scala decided to develop a new programming language after facing various problems in Java. As he was a professor at École Polytechnique Fédérale de Lausanne (EPFL), Odersky knew about the frustrations various programmers faced. He firstly decided to implement the Java Generics which became very successful and popular. People use Java collections all the time for various purposes. However, Martin Odersky wanted to achieve full scalable concurrent programming. He found Java to be very inefficient in that concept. Thus, Scala was born in 2001 with the fusion of object-oriented concepts as well as functional programming concepts. [10]

Programming with Scala:

Some programmers at first find it very gruesome to program in Scala, but they realize soon that it is just Java without semicolons. Sooner they start picking up the language very well and grow in parallel with the language. Some people even go all the way to create their own domain-specific language. Many programmers love the fact that everything is an object in Scala and we can pass functions around easily and use Object Orientation in Scala as well. Following is a basic example of a Scala Object.

```
object Hello {

    /*This prints Hello World. We can see how an
object is used in place of a Java class.*/
     def main(args: Array[String]) {
      println("Hello, world!") // prints Hello
World
   }
}
```

From the above example, this appears to be the same Java program without semicolons and Classes being replaced by Objects. This is what Scala is in the initial stages of programming.

Variable Declaration in Scala:

Variables & data types are a key part of any programming language. Variable declaration plays a crucial role when we want to limit variables access to some environment. In Java, we can do that with the help of keywords such as public, private or protected. Scala supports these keywords as well but is unique in the following way.

```
var myHello: String = "Hello"
```

In the above table, a keyword "var" is used followed by a variable name and the data type. This states that the variable is mutable. That is the string can take any value it likes later on in the code. But, what if I want my variable to be immutable? That can be easily achieved in Scala by replacing the keyword "var" with "val" as follows.

```
val myHello: String = "Hello"
```

Here the value of "myHello" can not be changed later on in the execution of the program.

Functions:

Functions in Scala appear to be similar to both Java and Python. Both of the language programmers can feel comfortable using functions in Scala. A function has parameters with a return type. The syntax of a Scala function is as follows:

```
def functionName ([list of parameters]) : [return
type] = {
   function body
   return [expr]
}
```

Now when we dive deeper into functional programming, we need to understand the basics of functional programming and why it is special. We have pure functions. They do not modify any outside code or state and return a result of a particular input.

Example: Taking cosine of something.

```
cos(myVar: Double)
```

But, in a functional language such as Scala, the functions are "first class citizens". This means that the functions can be passed around within functions. We can use functions as a data type and even store them in any data structures whenever we want. A basic example of this amazing coding style is as follows. [11]

```
val firstFive = Seq(1, 2, 3, 4, 5)
ints.filter(n => n % 2 == 1)
```

The above code prints out the odd integers in the sequence. Now think about how much code would be required in the same case in Java. Scala makes everything shorter.

To stress the use of functional programming more we can see the following example that uses "map", "reduce", and "filter" functions in Scala to perform such tedious tasks. Also, as mentioned above the code size is reduced substantially.

```
import scala.collection.immutable.List

Object SumOfEvenSquares
{
    def main(args:Array[String]):Unit=
        {
            var myList= List(10,11,12,13,14,15)

            def sum = intList.filter(x => x % 2
== 0).map(x => x * x).reduce((x,y) => x+y)
            println(sum)
}
```

Now we can see the power of functional programming, this function takes input as a list of integers and adds the sum of the squares of even numbers. Java code for this would be humongous! Therefore functional programming is very beneficial when it comes to reducing the code size.

Currying in Scala:

Scala has a key feature of currying, which is found in many other functional programming languages. Currying is the technique of translation of a function that takes multiple arguments and evaluates them into a sequence of functions. That is we can have functions within functions from the parameters. This is very useful in the case where functions taking multiple arguments have to be used in frameworks requiring functions to take a single argument. An example of this would be the case where some analytical techniques are applied to a function having a single parameter. A basic curried Scala program to add two strings would like as follows. We can use this concept to carry out more advanced tasks.

```
object Curry{

    def concat(first: String)(second: String) = {
      first + second
```

```
   }

   def main(args: Array[String]) {
      val first:String = "This is "
      val second:String = "Currying!"

      //This is a currying call
      println( "Combining both the strings! " +
concat(first)(second) )
   }
}
```

In the above program, we can see how currying is used to concatenate two strings and print the result.

Closures

One of the key functionalities of Scala stressing on its functional programming ability is the use of closures. Considering a scenario where I want to pass a function around like a variable but I want the function to refer to fields that were not part of the function or were not in the scope of the function. How can I do that? Some would say the use of global variables. That is always an option. However, global variables can suffer from the possibility of a race condition. It is always a good programming practice to avoid or to minimize the use of global variables. Closure comes to our rescue in such scenarios. An example of a closure is as follows.

```
object Closure {

   def main(args: Array[String]) {
      println( "add(1) value = " +  add(1) )
      println( "add(2) value = " +  add(2) )
   }

   val num = 10
   val add = (i:Int) => i + num
}
```

In the above example, we can see the example of closure, where the variable num is used even when it is outside the scope of the function add. This is a basic example of a closure in Scala.

Concurrency with Akka Actors

Scala is sort of an extended version of Java thus, it supports multithreading similar to Java to handle concurrency issues. This is done in the way of synchronous programming. However, in Scala, we can also handle concurrency with the help of Asynchronous programming using Akka Actors. According to Wikipedia, "Akka is a free and open-source toolkit and runtime simplifying the construction of concurrent and distributed applications on the JVM." Akka supports multiple programming models, but it draws inspiration for Actors from Erlang. Akka is written in Scala and Scala 2.10 and greater versions have Akka as part of their standard library. A basic example of a Scala Actor is as follows: [12]

```
import akka.actor.Actor
import akka.actor.ActorSystem
import akka.actor.Props

class HelloCountry extends Actor {

    def receive = {
        case "USA" => println("So you are an
American!")
        case _     => println("Sorry, I do not
know that country.")
    }
}

object Main extends App {
  val system = ActorSystem("HelloSystem")
  val helloCountry =
system.actorOf(Props[HelloCountry], name =
"hellocountry")
  helloCountry ! "USA"
  helloCountry ! "ABCD"
}
```

• In the above example, we create an Actor with it's behaviour defined in the "receive" class.
• An Actor system is needed to start with the code.
• An actor instance is created with the actorOf. After the instance is created, we can send it 2 messages.

In this example, if the country is the USA, the user will be greeted with a message as that is the only country which is recognized by the software. This program will run continuously unless and until we stop it manually. Scala Actors are encapsulated entities. The only way their internal state can be changed it through, passing of known messages. Since this is an asynchronous process, when an Actor sends a message, it does not have to wait for a reply. The actor can continue performing various other tasks. In the case of multiple messages being sent to the Actor, they will be executed in Queue. Therefore, an Actor implements single threading internally. If the message changes the internal state of an Actor, the change is reflected immediately. Thus we can see how easy and powerful the actor library in Scala is. We can even use classes such as Future, Promise, etc. for better concurrent programming in Scala.

Scala Build Tool (SBT)

SBT is an open source build tool for Scala or Java projects. It is similar to Ant or Maven used with Java. By SBT we can compile, run, and test code. It provides external libraries needed to make our program run efficiently. Following shell script can be used to set up the

initial environment required for SBT. It is taken from the Scala cookbook.

```
#!/bin/sh
mkdir -p src/{main,test}/{java,resources,scala}
mkdir lib project target
# create an initial build.sbt file
echo 'name := "MyProject"
version := "1.0"
scalaVersion := "2.10.0"' > build.sbt
```

The “build.sbt” is the most important file. It is used to add all the configuration and settings needed to run the SBT. All the library dependencies are specified in this file. After setting up the initial environment, we can use Giter8 to create the tree structure needed by the specific project. For an example, if we need to use the GSON library to parse my JSON data in a Scala program. We can do this by adding the following line in “build.sbt” as follows.

```
libraryDependencies += "com.google.code.gson" %
"gson" % "2.8.0"
```

Similarly, all other library dependencies can be added in the “build.sbt” file to run the code. After adding the dependencies we can compile and run the code in the following manner from the terminal.

```
$sbt compile
```

After which the program can be run with the following command.

```
$sbt run
```

These are the basic steps of SBT and how it used in Scala. Therefore, we can see how efficient Scala language is and can not be surprised by the growing popularity of this language for various features it provides. Scala is rich with above features and has even many more fantastic things to explore and code with. It won’t be surprising if many people shift to Scala from Java in the near future. [13]

4. Results

As described above, we had 6 programs each belonging to the domains identified by us. We ran all the programs on a single machine giving us the following results -

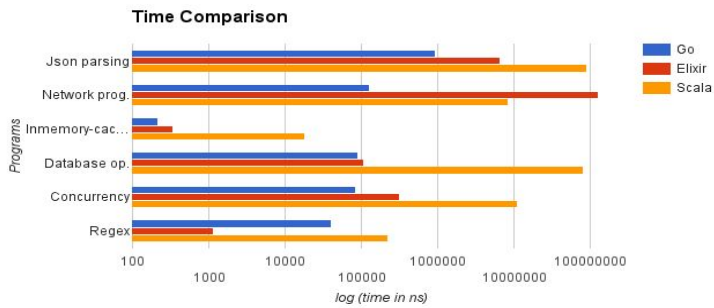
Time benchmarks (nanoseconds)

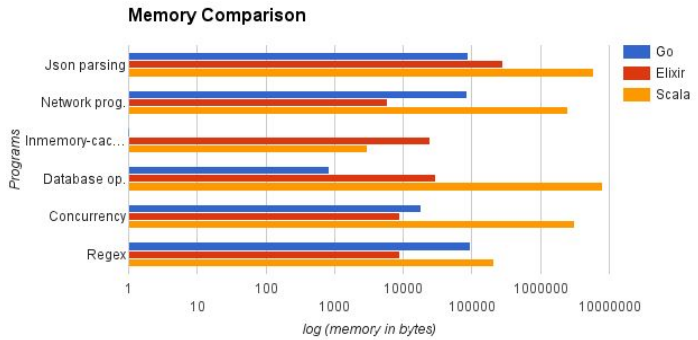
Programs	Go	Elixir	Scala
Json parsing	929,529	6,560,010	88,911,085
Network prog.	127,486	126,804,300	8,396,568
In Memory-cache	218	340	18,002
Database op.	89,452	105,520	79,983,526
Concurrency	83,544	313,940	10,926,355
Regex	41,054	1160	220,918

Memory benchmarks (bytes)

Programs	Go	Elixir	Scala
Json parsing	88,393	284,576	5,918,048
Network prog.	84,024	5978	2,504,824
In Memory-cache	0	24,664	3009
Database op.	840	30,092	8,093,344
Concurrency	18,414	8768	3,105,008
Regex	94,964	8768	211,168

System Configuration
Operating System: Ubuntu 16.04.1 LTS (64-bit)
RAM - 11.6 GB
Processor: Intel® Core™ i7-6500U CPU @ 2.50GHz × 4





From the results, it is evident that Go performed better than the other two in terms of time and memory both. This is because it is a lightweight language and is designed to get the performance of C or C++ combined with features of dynamically typed languages. It compacts its data structures and utilizes the cache really well which reduces the time and memory consumption. Threads in other languages tend to depend on the kernel to be scheduled whereas Goroutines are more advanced and are cooperatively scheduled.

Elixir has an average time and memory consumption. It does offer optimum parallelism, concurrency and fault tolerance, but there is no efficient way to capture it. In the case of pattern matching, Elixir performs really well because of the Erlang base and the way Erlang is designed.

As we can see Scala has an enormous memory footprint. Scala being based on Java has many heavyweight libraries which need to be loaded for a basic program to run. This is evidently visible in smaller programs like the ones we have done benchmarks on. Developers tend to load the complete package instead of loading only the required subpart. They don't realize that such mistakes which they take for granted might take a huge toll on the performance of the application.

5. Conclusion

Go, Elixir and Scala each have different architectures and internal data structures. Due to this, they don't perform the best in all aspects. Go is good to use while building microservices, while Elixir along with the phoenix web framework is good to use in web technologies. Scala, on the other hand, is good in highly secure and tightly bound applications. Thus, no language is better than the other. The technology to be used totally depends on what type of application is to be created.

6. Each Person's Role in the Project

Timelines	Taks	Performed by
Oct 20, 2016 - Oct 27, 2016	Decide the languages to compare.	Shantanu, Naveen, Vineet
Oct 28, 2016 - Dec 5, 2016	Study languages, write programs, and finalize the domains.	Shantanu - Scala Vineet - Elixir Naveen - Go
Dec 6, 2016 - Dec 20, 2016	Finish the code and write the documentation.	Vineet, Shantanu, Naveen

7. References

[1] https://www.ibm.com/developerworks/library/wa-optimal
[2] https://talks.Go.org/2012/splash.article
[3] Elixir official docs http://elixir-lang.org
[4] Meta Programming in Elixir, https://dockyard.com/blog/2016/08/16/The-minumum-knowledge-you-need-to-start-metaprogramming-in-Elixir
[5] Elixir tasks - http://learningelixir.joekain.com/learning-elixir-task
[6] Whatsapp Erlang success story http://highscalability.com/blog/2014/2/26/the-whatsapp-architecture-facebook-bought-for-19-billion.html
[7] Elixir GenServer https://medium.com/@StevenLeiva1/understanding-elixir-s-genserver-a8d5756e6848#.fab8wcksf
[8] Phoenix framework http://blog.carbonfive.com/2016/04/19/elixir-and-phoenix-the-future-of-web-apis-and-apps
[9]What is Scala https://www.scala-lang.org/what-is-scala.html
[10] Scala Programming Language: http://www.scala-lang.org/old/node/25
[11] Java Code Geeks: https://examples.javacodegeeks.com/jvm-languages/scala/functional-programming-scala/
[12] Akka Wikipedia, https://en.wikipedia.org/wiki/Akka_(toolkit)
[13]Alvin Alexander, How to create SBT Project Directory Structure http://alvinalexander.com/scala/how-to-create-sbt-project-directory-structure-scala
[14] The Benchmarks Game https://benchmarksgame.alioth.debian.org/u64q/scala.html