Pipeline Architecture

February 28, 2022

Brae Webb

Presented for the Software Architecture course
at the University of Queensland

THE UNIVERSITY
OF QUEENSLAND
AUSTRALIA

Pipeline Architecture

Software Architecture

February 28, 2022

Brae Webb

1 Introduction

Pipeline architectures take the attribute of modularity of a system to the extreme. Pipeline architectures are composed of small well-designed components which can ideally be combined interchangeably. In a pipeline architecture, input is passed through a sequence of components until the desired output is reached. Almost every developer will have been exposed to software which implements this architecture. Some notable examples are bash, hadoop, some older compilers, and most functional programming languages.

Definition 1. Pipeline Architecture
Components connected in such a way that the output of one component is the input of another.

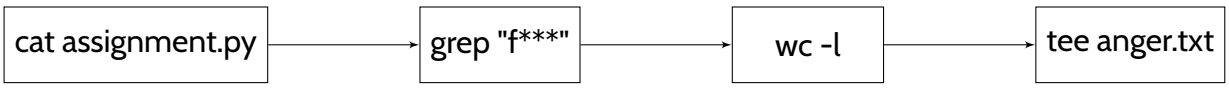

Figure 1: An example of using bash's pipeline architecture to perform statistical analysis.

The de-facto example of a well-implemented pipeline architecture is bash, we'll explore the philosophy that supports the architecture shortly. The above diagram represents the bash command,

```
$ cat assignment.py | grep "f***" | wc -l | tee anger.txt
```

.

If you're unfamiliar with unix processes (start learning quick!),

cat Send the contents of a file to the output.

grep Send all lines of the input matching a pattern to the output.

wc -l Send the number of lines in the input to the output.

tee Send the input to stdout and a file.

2 Terminology

As illustrated by Figure 2, a pipeline architecture consists of just two elements;

Filters modular software components, and

Pipes the transmission of data between filters.

Filters themselves are composed of four major types:

Figure 2: A generic pipeline architecture.

Producers Filters where data originates from are called producers, or source filters.

Transformers Filters which manipulate input data and output to the outgoing pipe are called transformers.

Testers Filters which apply selection to input data, allowing only a subset of input data to progress to the outgoing pipe are called testers.

Consumers The final state of a pipeline architecture is a consumer filter, where data is eventually used.

The example in Figure 1 shows how bash's pipeline architecture can be used to manipulate data in unix files. Figure 3 labels the bash command using the terminology of pipeline architectures.

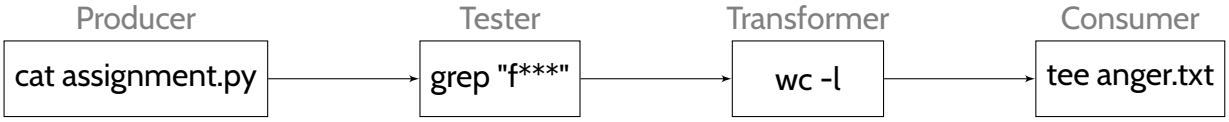

Figure 3: Figure 1 with labelled filter types.

3 Pipeline Principles

While the concept of a pipeline architecture is straightforward, there are some principles which should be maintained to produce a well-designed and re-usable architecture.

Definition 2. One Direction Principle
Data should flow in one direction, this is the *downstream*.

The data in a pipeline architecture should all flow in the same direction. Pipelines should not have loops nor should filters pass data back to their *upstream* or input filter. The data flow *is* allowed to split into multiple paths. For example, Figure 3 demonstrates a potential architecture of a software which processes the stream of user activity on a website. The pipeline is split into a pipeline which aggregates activity on the current page and a pipeline which records the activity of this specific user.

The One Direction Principle makes the pipeline architecture a poor choice for applications which require interactivity, as the results aren't propagated back to the input source. However, it is a good choice when you have data which needs processing with no need for interactive feedback.

Definition 3. Independent Filter Principle
Testers and transformers should not rely on specific upstream or down-stream filters.

In order to maintain the reusability offered by the pipeline architecture, it is important to remove dependencies between individual filters. Where possible, filters should be able to move around freely. In the example architecture in Figure 3,

TODO: Filters needed between tags and databases

the EventCache component should be able to work fine without the TagTime component. Likewise, EventCache should be able to process data if the Anonymize filter is placed before it.

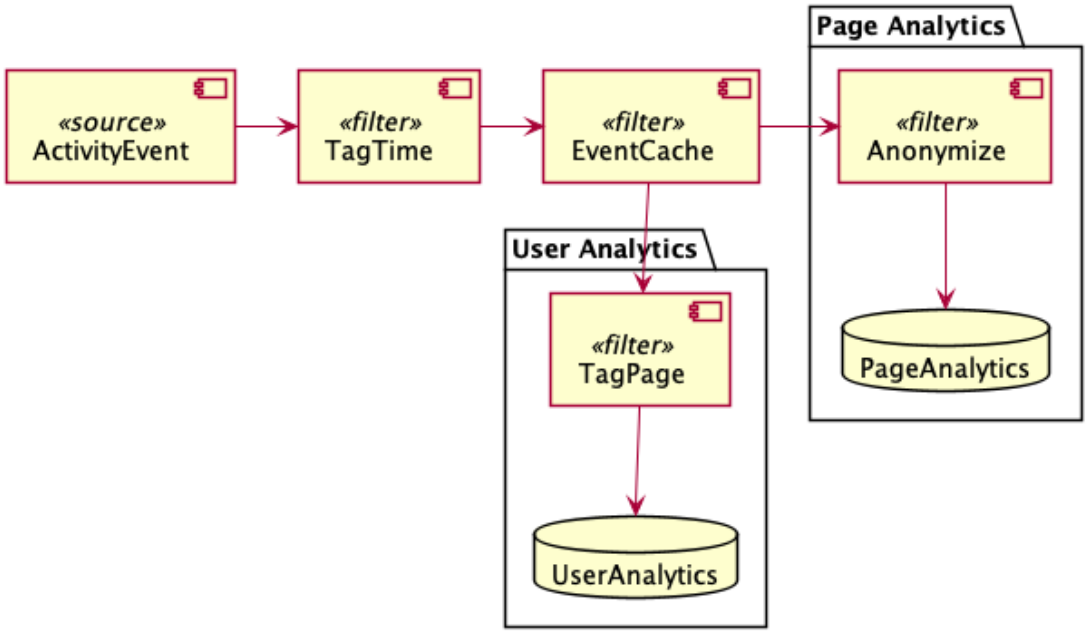

Figure 4: Pipeline architecture for processing activity on a website for later analytics.

4 Case Study: MapReduce

One of the more prevalent uses of the pipeline architecture is the MapReduce pattern. The MapReduce pattern was discovered in 2004 as a solution to the challenges which Google faced managing their search index [1].[1]MapReduce affords impressive parallelism inherit to the programming pattern.

The two key ideas of MapReduce, *map* and *reduce*, come from functional programming.[2] Below are the generic types of the *map* and *reduce* functions in functional programming.

```
map : (τ1 → τ2) → τ1Seq → τ2Seq
map f xs
reduce : (τ1 → τ1 → τ1) → τ1Seq → τ1 → τ1Seq
reduce f xs initial
```

If you're unfamiliar with this notation, the rough English translation is:

map The parameters of the *map* function are:

(a) A function, f, which takes a parameter of type τ_1 and returns a type τ_2.

(b) A sequence of elements of type τ_1.

The return type of the *map* function is a sequence of elements of type τ_2.

reduce The parameters of the *reduce* function are:

(a) A function, f, which takes two parameters both of type τ_1 and returns a type τ_1.

(b) A sequence of elements of type τ_1.

(c) An initial accumulator value of type τ_1

[1]Although the pattern was in use prior to their work[2]

[2]I *think*? Will consult with history textbook (Ian)

The return type of the *reduce* function is a sequence of elements of type τ_1.

The code snippet below uses the map and $reduce$ functions to perform the operations of the above bash example. One important thing to note about the example below is the map operation on line 11. Each application of the lambda function within the map operation is completely independent and could, in theory, be executed simultaneously.

```
1  contents = read("assignment.py")
2
3  # filter relevant lines by rebuilding the list
4  contents = reduce(λ xs x →
5                    if x.contains("f***")
6                        then x + xs
7                        else xs,
8                  contents)
9
10 # use map to count occurrences of word
11 contents = map(λ line →line.count("f***"), contents)
12
13 # use reduce to sum list of counts
14 contents = reduce(λ total curr →total + curr, contents, 0)
15
16 write("anger.txt", contents)
```

So by design, code written in this pattern can process data simultaneously. Tools such as Hadoop[3] are able to take advantage of this to distribute computation automatically.

> Using the terminology of a pipeline architecture, what filters do the *map* and *reduce* operators correspond to?

> How would you improve the efficiency of the code snippet above?

5 Case Study: Compilers

An interesting case study of the pipeline architecture is a compiler.[4] As a foundational technology, compilers have undergone rigorous refinement and are perhaps the most well studied type of software. Modern compilers have well-defined modular phases as illustrated by Figure 5, each phase of a compiler transforms the representation of the program until the target program is produced.

[3] https://hadoop.apache.org/

[4] You don't need to understand the phases of a compiler — two data structures, the Symbol Table and AST, are transformed in each compiler phase.

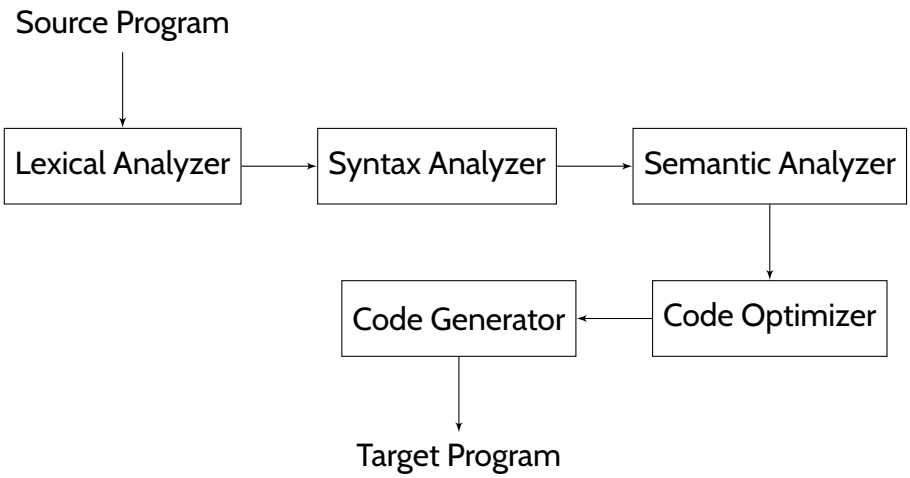

Figure 5: Typical phases of a compiler.

However, a compiler is not well suited to use a pipeline architecture. In general, the modules of a pipeline architecture should be independent of their input source. This is not the case in compilers, as each phase relies on the completion of the previous phase. As a result, the input dependencies of a compiler make it too restrictive for a true pipeline architecture.

Instead, compilers are often built as a hybrid of a pipeline architecture and the *Blackboard Architecture*. The blackboard architecture consists of;

- a knowledge base, the 'blackboard',
- knowledge sources which use and update the knowledge base, and
- a control component to coordinate the operation of knowledge sources.

In modern compilers, the data which would be passed through pipes, the Symbol Table and AST, are used and updated by each phase. They are subsequently used as the 'blackboard'. Each phase is considered a knowledge source which uses the knowledge base to transform and update the knowledge base. Finally, in this hybrid, the control component is not required as the sequence of phase execution in a pipeline coordinates operation. Figure 6 illustrates this proposed architecture. Of course, there are many compilers out there, many of them deviate from this architectural hybrid.

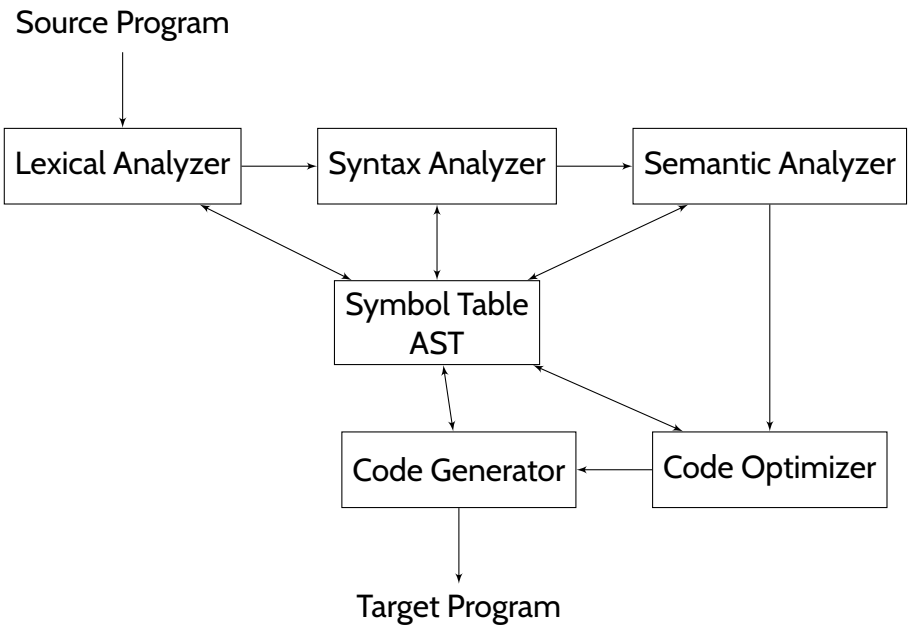

Figure 6: Modern phases of a compiler.

References

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, (San Francisco, CA), pp. 137–150, 2004.

[2] D. J. DeWitt and M. Stonebraker, "Mapreduce: A major step backwards." https://dsf.berkeley.edu/cs286/papers/backwards-vertica2008.pdf, January 2008.