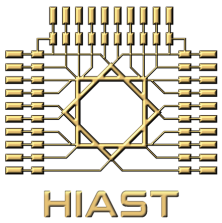
Higher Institute for Applied Sciences and Technology
https://hiast.edu.sy
IT Dept. 2022,



Complex State Management with Immutability

Nassouh AlOlabi
Supervised By: Dr.Yasser Rahal, Fahmi Alammareen

Received: 22-02-2022
Accepted: 25-02-2022
Published: 05-03-2022

Abstract

Growing demand for fault-tolerant, scalable, distributed systems has made some mainstream software architectures and patterns obsolete or rather harder to come by, Thus came the rise of stateless and functional solutions based on data immutability which has already been the cornerstone of Big Data [1].

Keywords: immutable data structures, functional programming, algorithm design

Contents

1. Introduction

Unintended mutations of a program's state cause inconsistent behavior and bugs of the program. These mutations might have been introduced by side-effects of functions that developers were unaware of during the program's implementation. Some programming languages do, for example, allow arguments of a function to be mutated. The fact that a third-party function can mutate the state of its argument can go undetected. The problem of rogue and complicated state mutations can become difficult to handle when states are shared among objects. One way to avoid undesired mutations is to use immutable data instead of mutable data. Immutable data cannot be mutated once created and instead of mutating shared data in memory, data would have to be re-created to include the modifications needed. Programmers can then safely share data without the possibility of having it mutate to something else, which is crucial to avoid, for example, race conditions in concurrent programs.

2. State Management

The term state management has gain traction over the last decade or so with the emergence of wide spread IT solutions and the ease of making one of your own from whatever background you come from. Depending on the underlying scenario the term could stand for: Data persistence management, Information flow, Programming paradigms, I/O (especially networking and caching), Application architecture, Presentation behavior and UI templating. Therefore the term state management has been a catch-all term from data modeling to parallel computing.

2.1 *Data Persistence*

A primary challenge to building reliable and secure computer systems is managing the persistent state of the system: all the executable files, configuration settings and other data that govern how a system functions. The difficulty comes from the sheer volume of this persistent state, the frequency of it's changes, and the variety of workloads and requirements that require customization of persistent state. The cost of not managing a system's persistent state effectively is high: configuration errors are the leading cause of downtime at Internet services, troubleshooting configuration problems is a leading component of total cost of ownership in corporate environments, and malware—effectively, unwanted persistent state—is a serious privacy and security concern on personal computers.

2.2 *Information Flow*

Application State (a.k.a. Program State) represents everything necessary to keep your application running. When we refer to application state we are normally referring to the state of the program as exists in the contents of its memory. What does that mean practically? How am I to understand that? It helps to think in extremes. What happens to information and functionality core to your application if a server goes down and restarts? You lose whatever was residing in memory.

Anti-Patterns or pitfalls includes[2]:

2.2.1 *Missing data*

Its' the situation where some data element needs to be accessed, i.e. read or destroyed, but either it has never been created or it has been deleted without having been created again.

2.2.2 *Inconsistent Data*

Data is inconsistent if a task is using this data while some other task (or another instance of the same task) is writing to this data or is destroying it in parallel.

2.3 *Programming Paradigms*

Programming paradigm is an approach to solve problem using some programming language or also we can say it is a method to solve a problem using tools and techniques that are available to us following some approach. And so most paradigms have their own opinion on data manipulation for example: Procedural & Object Oriented paradigms allows more access to data modifying (destructive updates), deleting. Others like Logic & Functional doesn't allow them.

2.4 *I/O, Network and Disk*

Applications that are I/O heavy often cause bottlenecks. As well as often causing the problem, I/O intensive applications are often more sensitive to a storage latency issue. When you have a large user base trying to access these applications, slowdowns tend to take place. Increased response time in storage I/O causes bottlenecks. When there is a queue in the storage I/O, you would generally see an increase in latency. Network and Disk latency are often times the biggest source of headache if poorly managed, which is often times the case since they're usually treated as if they were function calls which they're clearly not [3].

execute typical instruction	1/1,000,000,000 sec = 1 nanosec
fetch from L1 cache memory	0.5 nanosec
branch misprediction	5 nanosec
fetch from L2 cache memory	7 nanosec
Mutex lock/unlock	25 nanosec
fetch from main memory	100 nanosec
send 2K bytes over 1Gbps network	20,000 nanosec
read 1MB sequentially from memory	250,000 nanosec
fetch from new disk location (seek)	8,000,000 nanosec
read 1MB sequentially from disk	20,000,000 nanosec
send packet US to Europe and back	150 milliseconds = 150,000,000 nanosec

Table 1: Latency numbers every programmer should know[4]

As you can see disk fetches can take 100x more than main memory fetches. And when dealing with distributed systems were network call are an integral part of the system treating network call which 100000x more than main memory calls you end up with catastrophically unpredictable behavior

3. Immutability

When something is immutable, we say that it cannot be changed or that it is unchangeable. The definition of an immutable object in an object-oriented programming language is an object that has a state that cannot be mutated once instantiated (created). Moreover, an immutable class is a class whose instances cannot be mutated, meaning that there are no methods in the class that can mutate an instance of it. Classes that are not immutable can, however, have instances of it that are immutable.

3.1 *Forms of Immutability*

There are different varieties of immutability used in practice today, which includes:

- Object immutability

  An immutable object is an object that cannot be modified (mutated), i.e., its state cannot be mutated

- Class immutability

  When every instantiated object of a class is immutable, then we say that the class itself is immutable.

- Deep and shallow immutability (transitivity)

  Immutability can be deep or shallow, i.e., transitive or non-transitive:

  Deep (transitive): immutability means that all objects referred to by an immutable object must also be immutable.

  Shallow (non-transitive) immutability has a more relaxed constraint, and the immutable object may refer to objects that are mutable, but the fields of the immutable object itself cannot be mutated.

- Reference immutability (read-only references)

  Languages with the support of reference immutability have the notion of read-only references. A readonly reference cannot mutate the object it is referring to. When all references to an object are readonly, then nothing can mutate the object, and the object is immutable. There are, however, often no guarantee that a referred to object is immutable because the object may still be mutated by some other reference that is not read-only, unless there is some analysis that ensures that there only exist readonly references to that object. Reference immutability thus often ensure shallow (non-transitive) immutability and "reference immutability" does not mean that a reference itself is immutable [5].

- Non-assignability

  Non-assignability is a form of immutability and property of a variable that makes sure that it cannot be reassigned. Since fields of objects are variables and if no variable can be reassigned after its initial assignment, the object is effectively immutable if what the fields are referring to is immutable. This make non-assignability give shallow (non-transitive) immutability too. Assignment of an object's field mutates the object, but it does not mutate what was previously on the field or what was assigned to the field.

- Concrete and abstract immutability

  Concrete immutability does not allow any change to the object's state. Abstract immutability, however, allows an immutable object to change its internal state but not the object's "abstract" value. The object is still immutable from the perspective of the object's observer that can only see the abstract value, but the object may mutate internally. This can, for example, be useful to speed up certain operations, lazy initialization and buffering.

3.2 *Properties of Immutability*

The concept of immutability is important and utilizing immutable data is considered to ease software development and reasoning about programs in numerous ways, for example:

3.2.1 *Predictability*

It is harder to understand and reason about programs that have shared mutable states with unclear interactions. Tracking mutations and maintaining the correct state of a program can be difficult. Using immutable data naturally, avoids state changes and forces the programmer to let data flow and be utilized in a different way throughout the program, making the state of the program more predictable. For example, calling the same function twice would yield the same result, and the outcome is predictable. Without the immutability guarantee, the second call could yield another result because of an underlying state mutation making it less predictable.

3.2.2 *Testability*

Because immutable data can only be changed once during construction, they are inherently simpler and easier to unit test. One may reason that by restricting the number of possible mutations in a program; the number of potential errors of the program is also reduced. Testing is essentially to validate that mutations in the program occur correctly and thus having more mutations would require more testing. By restricting the number of mutations in a program, the program has fewer reasons for errors to occur and there are fewer state transitions to test.

3.2.3 *Concurrency*

Immutable data are thread-safe, as data cannot mutate, there is no danger in having multiple threads access the same data at the same time and have synchronization issues.

3.2.4 *Modularity*

Without depending on a local or global state, immutable types and data may be reused in different contexts more easily.

4. Methods

4.1 *Shadowing*

Shadowing is a technique that could represent a changing variable. for instance, an accumulator. the technique is simple and possible in almost every programming language out there. at it's simplest form shadowing looks like this:

```
scala> object Main {
   def main(args: Array[String]) = {
      val i = 1;
      {
         val i = 2;
         {
            val i = 3;
         }
      }
   }
}
```

But that's not really useful and even more confusing and very error prone and I'd agree shadowing in of it's self isn't really useful but it's really at the core of any recursive solution since every function come with it's own block and

```
scala> def factorial (n: BigDecimal):BigDecimal = {
   if (n <= 1) 1
   else n * factorial(n-1)
}
```

Notice here n value range over {n, ... , 1} but it's not really changing each n deffer from the other and has it's own scope if you run this code it'll only go so far (around n = 9613 for this example) until you get a StackOverflowError... not good.

4.2 *Reassignment*

here's a subtle difference, well hidden behind the overloaded use of the symbol '=', that really sets the two apart. In the imperative program, '=' refers to a destructive update, assigning a new value to the left-hand-side variable, whereas in the functional program '=' means true equality, and that both the left-hand-side and the right-hand-side can be used interchangeably. This characteristic of functional programming (known as referential transparency or purity) has a profound influence on the way programs are constructed and reasoned about.

4.3 *Tail Recursion*

Tail recursion provides stack safety to our solutions and prevent Stackoverflows is when you simply return the value of a function call at the end (tail) of your function, in other words your functions has done it's job and handing over the rest of the work to another function

```
def factorial (n: BigDecimal):BigDecimal = {
  def helper(n: BigDecimal, Acc: BigDecimal): BigDecimal = {
    if (n <= 1) Acc
    else helper(n - 1, n * Acc)
  }
  helper(n, 1)
}
```

Notice that as n takes the values {n, n-1, n-2, ...} the accumulator also changes {n, n*(n-1), n*(n-1)*(n-2), ...} which is very similar to a for loop accumulator pattern, so similar that sometimes compilers compile it to an actual loop

4.4 *Pure Functions*

Pure Functions serves as a (possibly infinite) lookup table mapping from one type to another since variable x will always be the same $f(x)$ will too. this is what's known as referential transparency.

```
#include <functional>
#include <iostream>
int sum(const int v[], const int& n) {
   std::function<int(int, int)> helper = [&v, &n,
      &helper](int index, int Acc) {
      if (index >= n) return Acc;
      return helper(index + 1, Acc + v[index]);
   };
   return helper(0, 0);
}
int main(int, char**) {
   const int a[] = {1, 2, 3, 4, 5, 6};
   // a[1] = 3; not allowed
   int total = sum(a, 6); // 21
   someFunction(a);
   otherFunction(a);
   if (total == sum(a,6))
      std::cout<<"It should be equal same function same
         argument?!";
   return 0;
}
```

generally speaking "$someFunction$" and "$otherFunction$" could've done al sorts of things with a (changing an element value, adding more elements, removing some elements, delete the pointer entirely ...) but if they were pure functions or like in this case

using a some sort of a language guarantee (here it's $const$) a will not be modified and in turns $total == sum(a, 6)$ and overall our program would be easier to reason about.

4.5 *Laziness*

sometimes referred to as call-by-need, it's the notion that "if data won't change and functions won't neither so would results" meaning that we wouldn't perform any operations unless they're absolutely necessary

```
def from(n: Int): LazyList[Int] =
   n #:: from(n+1)
   // here from(n+1) won't be evaluated
def sieve (s: LazyList[Int]): LazyList[Int] =
   s.head #:: sieve(s.tail.filter(_ % s.head != 0))
// nor would sieve(s.tail.filter(_ % s.head != 0))
val primes = sieve(from(2))

primes
   .take(10)
   .toList // now it's necessary
```

in this example only 10 elements of the infinite "LazyList" are evaluated while the other elements are immutable they are yet to be evaluated and materialized.

4.6 *Structural Sharing*

While imperative programmers are very careful with passing references around and often times rely on cloning. on the other hand persistent data structures allows for more flexibility and safety in sharing and reusing data with ease.

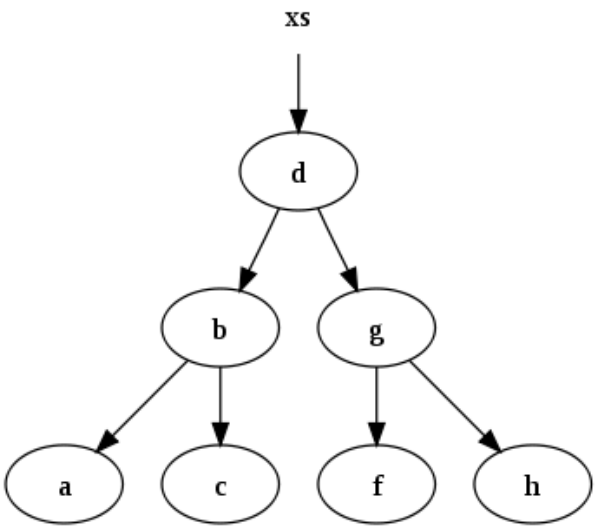


Figure 1: Original BST for the corresponding values [a,b,c,d,f,g,h]

now what if you wanted to add Node(e) to our Binary Search Tree. now if we were to naively implement an algorithm that would insert the node without mutation is to clone the tree and afterwards do the desired mutations. on the other hand if we consider immutability as and advantage we'll take into account that unaffected node (nodes that we don't need to modify e.g. nodes b,a,c,h) would never change and we can safely reuse them.

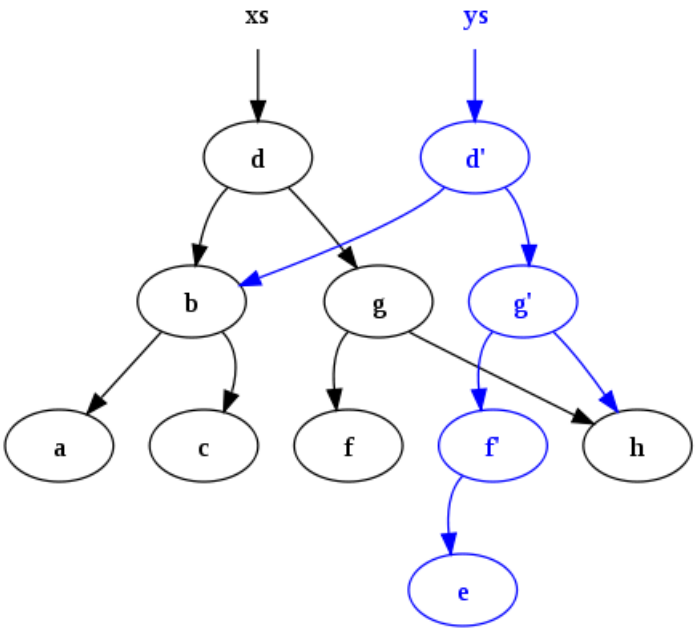


Figure 2: Resulting BST ys which shares most of the elements in xs

Structural Sharing is a very powerful technique and is the core concept for many of optimization techniques such as Copy-on-write paging strategies for address space inheritance have been shown to be effective in reducing the real time required to perform UNIX fork() operations. This reduction in copying also reduces the amount of swap space required, reduces the amount of time spent swapping, increases the number of processes which can be run without paging, and decreases the cost of context switches.[6]

5. Solutions

The trend to leverage immutability as a key architectural concept in new designs is so pervasive we see it in a number of new technologies and open source systems, we'll go through the same four areas of state management and see flagship technologies that leverage immutability in some of their features:

5.1 *Copy-on-Write File Systems*

COW generally follows a simple principle. As long as multiple programs need read only access to a data structure, providing them only pointers[8] which point to the same data structure, instead of copying the whole structure to a new one, is enough. If at least one

of the programs needs at some point write access to the data structure, create a private copy for it. The same holds for each one of the programs which will need write access to the data structure. A new private copy will be created for them. Note that the unchanged data can still be shared between both programs. COW file systems also provides many other powerful features, But our main interest for this paper is only one of them which is snapshots.

A snapshot is a consistent image of the data at a specific point in time. By data I mean the contents of a Fle system, the contents of a database, etc. Snapshots can be read only, or read/write. Writable snapshots are also called clones. Snapshots are extremely useful and have many applications: data recovery, online data protection, undo Fle system operations, testing new installations and configurations, backup solutions, data mining, and more.

One of the uses of the snapshot is for enabling of consistent backups. A backup can get inconsistent if during the backup the file system is used. Another use of the snapshot is that the snapshot is kind of easy backup that allows for the retrieving of removed or changed data. Taking of snapshots can be set up in this way that file system could do multiple level undoes. Especially if taking of snapshots does not affect performance of the system.

When a snapshot is created, its disk space is initially shared between the snapshot and the file system, and possibly with previous snapshots. As the file system changes, disk space that was previously shared becomes unique to the snapshot, and thus is counted in the snapshot's used property. Additionally, deleting snapshots can increase the amount of disk space unique to (and thus used by) other snapshots.

A snapshot's space referenced property value is the same as the file system's was when the snapshot was created.

You can identify additional information about how the values of the used property are consumed. New read-only file system properties describe disk space usage for clones, file systems, and volumes.

References

[1] Berman, J. J. (2018). Immutability and Immortality. In Principles and practice of big data: Preparing, sharing, and analyzing complex information (pp. 80–85). essay, Academic Press.

[2] Trcka, Nikola & Aalst, Wil & Sidorova, Natalia. (2009). Data-Flow Anti-patterns: Discovering Data-Flow Errors in Workflows. Proceedings of the International Conference on Advanced Information Systems Engineering (CAiSE). 5565. 425-439. 10.1007/978-3-642-02144-$2_3$4.

[3] Solarwinds: Top 4 Causes Of Storage I/O Bottlenecks & How To Mitigate Them

[4] Norvig, P. (n.d.). Teach yourself programming in ten years. Retrieved March 5, 2022, from `http://norvig.com/21-days.html#answers`

[5] Axelsson, L. (2017). Immutability : An Empirical Study in Scala LUDVIG AXELSSON.

[6] Smith, Jonathan & Jr, Gerald. (1988). Effects of Copy-on-Write Memory Management on the Response Time of UNIX Fork Operations.. Computing Systems. 1. 255-278.

[7] Levrinc, R. (2008). LLFS : a copy-on-write file system for Linux [Diploma The-

sis]. reposiTUm. https://resolver.obvsg.at/urn:nbn:at:at-ubtuw:1-28018 [8] Kernighan, B. W., and Ritchie, D. M. The C Programming Lan- guage, Second Edition. Prentice Hall, 1988.

Acknowledgements

Thank you for your time