Garbage Collection in Go and AWS Services

Reference:

https://blog.golang.org/go15gc

https://making.pusher.com/golangs-real-time-gc-in-theory-and-practice/

http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Streams.Lambda.html

1. Go Garbage collection

- **Overview**

Go's new garbage collector is a concurrent, tri-color, mark-sweep collector, an idea first proposed by Dijkstra in 1978. This is a deliberate divergence from most "enterprise" grade garbage collectors of today, and one that we believe is well suited to the properties of modern hardware and the latency requirements of modern software.

In a tri-color collector, every object is either white, grey, or black and we view the heap as a graph of connected objects. At the start of a GC cycle all objects are white. The GC visits all roots, which are objects directly accessible by the application such as globals and things on the stack, and colors these grey. The GC then chooses a grey object, blackens it, and then scans it for pointers to other objects. When this scan finds a pointer to a white object, it turns that object grey. This process repeats until there are no more grey objects. At this point, white objects are known to be unreachable and can be reused.

This all happens concurrently with the application, known as the mutator, changing pointers while the collector is running. Hence, the mutator must maintain the invariant that no black object points to a white object, lest the garbage collector lose track of an object installed in a part of the heap it has already visited. Maintaining this invariant is the job of the write barrier, which is a small function run by the mutator whenever a pointer in the heap is modified. Go's write barrier colors the now-reachable object grey if it is currently white, ensuring that the garbage collector will eventually scan it for pointers.

Deciding when the job of finding all grey objects is done is subtle and can be expensive and complicated if we want to avoid blocking the mutators. To keep things simple Go 1.5 does as much work as it can concurrently and then briefly stops the world to inspect all potential sources of grey objects. Finding the sweet spot between the time needed for this final stop-the-world and the total amount of work that this GC does is a major deliverable for Go 1.6.

At a higher level, one approach to solving performance problems is to add GC knobs, one for each performance issue. The programmer can then turn the knobs in search of appropriate settings for their application. The downside is that after a decade with one or two new knobs each year we end up with the GC Knobs Turner Employment

Act. Go is not going down that path. Instead we provide a single knob, called GOGC. This value controls the total size of the heap relative to the size of reachable objects. The default value of 100 means that total heap size is now 100% bigger than (i.e., twice) the size of the reachable objects after the last collection. 200 means total heap size is 200% bigger than (i.e., three times) the size of the reachable objects. If we want to lower the total time spent in GC, increase GOGC. If we want to trade more GC time for less memory, lower GOGC.

- **Latency vs. throughput**

If a concurrent GC can yield much lower latencies for large heap sizes, why would we ever use a stop-the-world collector? Isn't Go's concurrent garbage collector just better than GHC's stop-the-world collector?

Not necessarily. Low latency has costs. The most important cost is reduced throughput. Concurrency requires extra work for synchronization and duplication, which eats into the time the program can be doing useful work. GHC's garbage collector is optimized for throughput, but Go's is optimized for latency. At Pusher, we care about latency, so this is an excellent tradeoff for us.

A second cost of concurrent garbage collection is unpredictable heap growth. The program can allocate arbitrary amounts of memory while the GC is running. This means the GC must be run before the heap reaches the target maximum size. But if the GC is run too soon, then more collections will be performed than necessary.

2. DynamoDB Streams and AWS Lambda Triggers

Amazon DynamoDB is a fully managed NoSQL database service that provides fast and predictable performance with seamless scalability. DynamoDB lets we offload the administrative burdens of operating and scaling a distributed database, so that we don't have to worry about hardware provisioning, setup and configuration, replication, software patching, or cluster scaling.

Amazon DynamoDB is integrated with AWS Lambda so that we can create triggers—pieces of code that automatically respond to events in DynamoDB Streams. With triggers, we can build applications that react to data modifications in DynamoDB tables.

If we enable DynamoDB Streams on a table, we can associate the stream ARN with a Lambda function that we write. Immediately after an item in the table is modified, a new record appears in the table's stream. AWS Lambda polls the stream and invokes our Lambda function synchronously when it detects new stream records.

The Lambda function can perform any actions we specify, such as sending a notification or initiating a workflow. For example, we can write a Lambda function to simply copy each stream record to persistent storage, such as Amazon Simple Storage

Service (Amazon S3), to create a permanent audit trail of write activity in our table.