Scalability Practice Exam

Thor Olesen

November 2017

Contents

The exam is a 4-hour written exam, at ITU with all aids allowed (books, on-line resources, ...). The exam will have a similar form to this practice exam, with 4 thematic parts each contributing to 25% of the grade.

1 Part 1 - Reflecting on the assignments (25%)

A. Did you use Big Query for your assignments? (i) describe BigQuery, (iiI) its pros and cons and (iii) why it is (or it is not) relevant for your assignments?

1.1 BigQuery

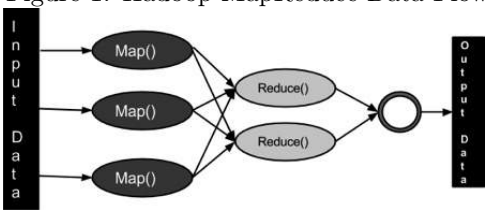
Yes, we used Big Query, as one of the vast range of big data processing tools that may be used to analyze and process large data sets. According to Google, BigQuery is an *Enterprise data warehouse that solves the problem of storing and querying massive datasets by enabling super-fast SQL queries*[1]. Specifically, it enables these super-fast SQL queries by using a distributed query engine, "Dremel" also developed by Google[2]. In our case, BigQuery leverages the power of Dremel for Big Data processing requirements that is available through a public REST API, used in Go to fetch satellite data, including granules and image folder links.

1.2 Pros and cons

In order to evaluate the pros and cons of BigQuery, one may want to consider the database landscape throughout the past. Traditionally, one would previously always choose a traditional RDBMS (relational database management system) for data storage and processing purposes. Namely, this is the de facto standard for database management today. However, it requires data to be well-structured and it has a very limited scope and purpose. Thus, RDBMS is generally not thought of as a scalable solution to meet the needs of 'big' data that usually comes in an unstructured format, from many different sources that are not necessarily well-defined.

Alternatively, Google has previously been using MapReduce for Big Data Processing, which has been popularized through Hadoop, as a programming framework for batch processing of large datasets. Hadoop has been a popular implementation of MapReduce, where data is first split into chunks and processed by map tasks in parallel (Map phase), which is further processed by reduce tasks, that combine (Reduce phase) all of the results to produce an appropriate result. The result is usually stored in a distributed file system like HDFS or a distributed key-value store like the NoSQL database HBase, that can handle a variety of data formats and support horizontal scaling by spreading the data across multiple servers/nodes. All together, a big data processing framework like Hadoop MapReduce and storing it in HDFS enables data to be processed in a distributed manner that scales well across multiple clusters of computers.

Figure 1: Hadoop MapReduce Data Flow



[1] `https://cloud.google.com/bigquery/what-is-bigquery` seen 15/11/17
[2] `https://cloud.google.com/files/BigQueryTechnicalWP.pdf` seen 15/11/17

However, MapReduce does come with a range of limitations, which arguably led to the invention of Dremel and BigQuery. Specifically, MapReduce does address the issue of processing unstructured data in a scalable way compared to a traditional relational database technology. Despite this, MapReduce is designed as a batch processing framework that was not designed for ad hoc, real-time, trial-and-error data analysis. In this regard, Dremel is also a parallel computing framework but it was designed specifically to run queries on structured data in seconds[3].

Thus, one benefit of using the BigQuery implementation of Dremel as a developer is the ability to query large datasets for quick analysis. Further, the BigQuery API integrates well with the rest of the Google Cloud services and the Go programming language. Thus, it has been ideal to use for the project, since it integrates well with these services and the sentinel-2 data is accessible via both the BigQuery and Storage Bucket API. Either way, MapReduce may be a better choice if you want to process unstructured data programmatically but BigQuery allowed us to quickly test the data flow of satellite data and integrated well with the developer tools at our disposal.

1.3 Relevance to project

BigQuery is relevant to the assignments because of the fact that the sentinel-2 satellite data has been published in a Google Cloud Bucket (i.e. data container) that may be accessed with the BigQuery REST API in Go easily. Namely, BigQuery has been used in the project to quickly retrieve images located in specific granules (i.e. 100 km rectangular areas on earth) and links to satellite image folders. The index data is available in BigQuery to easily query in SQL: `https://bigquery.cloud.google.com/table/bigquery-public-data:cloud_storage_geo_index.sentinel_2_index`

B. In the first assignment, you were asked to implement a web service that accepts a GET request and returns a JSON array. Describe your design and implementation.

1.4 Web Service

The web service is located in "service.go" and accepts GET requests with query parameters, based on a given latitude and longitude that specify a location. Further, the "query.go" file contains the functionality used to query image data with SQL through the BigQuery REST API. Asa result, all granules (i.e. 100 km rectangular zones on earth) or images that match a given location may be requested based on the following format:

HTTP GET Request: https://tvao-178408.appspot.com/images?lat=&lng=

In terms of the design and implementation, the service HTTP handler located in "service.go" has the responsibility of handling client HTTP GET requests and returns satellite images that match the location specified by the client in as JSON in a HTTP response. This has been done with a custom HTTP handler that includes error values for better error handling (see section 5.1. Further, the data provided by the client is validated with a regular expression before being submitted to BigQuery in "query.go". The image links are then fetched in the "getLinks" method in "query.go" that uses a simple SQL expression to return all images that match the specified location. Finally, the result is written to JSON in the HTTP response by the service that returns the result in the images handler function (see section 5.2).

[3] `https://cloud.google.com/files/BigQueryTechnicalWP.pdf` p.7-8 seen 15/11/17

2 Part 2 - Server-side programming (25%)

A. Describe process of deploying an app on Google Cloud Platform

2.1 Google Cloud Deployment

The web service was deployed on the Google Cloud App Engine service that abstracts away the hardware and infrastructure used to host the web service, thus allowing the developer to focus on the Go code only. The App Engine will automatically provision the infrastructure, integrates well with the other services in the Google Cloud ecosystem and supports automatic load balancing.

In order to deploy the web service, one has to specify an app.yaml file within the source code directory that specifies the runtime language (i.e. go) and its handlers for each request (see 5.3). After this, one may simply deploy the app to the app engine with the following command: "gcloud app deploy app.yaml".

B. Consider several services that are deployed on the Google Cloud Platform and interact with each other. What are the pros and cons of using RPC to support this interaction? Can RPC scale with the number of services?

As recalled, RPC (i.e. Remote Procedure Call) is a communication model that builds on top of message passing and supports interprocess communication between applications in a distributed environment through function calls. Normally, applications reside on different machines and the RPC mechanism typically addresses this by defining a network protocol, data exchange format and a way of serializing and deserializing data with specific programming language support.

One benefit of enhancements in RPC may be expressed, when addressing the use of microservices. Today, it has become popular to employ a microservices architecture where an application is split into small, loosely coupled services that communicate with each other. Typically, large distributed systems follow this approach, where the application is composed out of several services. For this purpose, one may need enhanced RPC's. Arguably, RPC scales well with the number of services, based on their need for interprocess communication, as seen in today's large distributed web services. A indication of this may be verified by Google that introduced the "gRPC" framework to handle remote procedure calls in a scalable way and helpexpose their public services via a HTTP/2[4]. Besides this, RPC provides a general interoperability support between different service implementation that would otherwise not be possible in a distributed environment.

However, due to the fact that RPC is not a "standard", it can be implemented in many ways. Also, the context switching between processes introduce a performance overhead, since the data (i.e. "state") communicated between services has to be stored and restored to allow multiple services communicate effectively with each other.

[4]The HTTP/2 protocol ensures that only a single connection is used to reduce the number of round trips needed to set up TCP connections and the data is compressed and presented in a binary format instead of text to reduce performance overhead.

3 Part 3 - Topics from the lectures (25%)

A. Google claims that Spanner is a distributed CA store. a. What are C and A in the CAP theorem? What is a CA store? b. Describe Spanner's architecture c. What aspects of the Spanner architecture impact the validity of the claim?

3.1 CAP Theorem and CA Store

The CAP theorem is a proposition that help developers to reason about proposed system traits and understand the tradeoffs involved in DDBS (i.e. distributed database systems). Namely, distributed database systems may either allow reads before updating all nodes to ensure high availability or lock all nodes before allowing reads to ensure high consistency. Thus, the CAP theorem helps reason about the general trade off between consistency (C) and availability (A) in a DDBS. However, it is important to emphasize the fact that THE CAP theorem only applies due to the occurrence of failures, caused by having multiple network partitions that may introduce failures. In short, the replication of data in a DDBS introduces the possibility of failures that force the system developer to decide between reducing either the desirable property of availability or consistency. Further, the availability property is often associated with the latency of the system that is high when the availability is low. That is to say, an unavailable system provides high latency. In regards to the CAP theorem, a CA store is a data store that stores and manages data, while it upholds the desirable property of consistency and availability at the same time. Thus, a CA store does seemingly not compromise between consistency and availability, as otherwise advocated by the CAP theorem.

3.2 Spanner Architecture

Spanner distributes data globally and supports consistent distributed transactions. In terms of achieving high availability, it does this by sharding (i.e. partitioning) data across multiples machines, spread in data centers all over the world. For this purpose, the Paxos consensus algorithm is used to decide in which datacenters data is sharded. The replication helps ensure that data is both globally available and geographically local (i.e. close to where you are). In terms of consistency, Spanner assigns global commit timestamps to transactions that reflect the serialization order (i.e. order of execution). By way of example, if transaction "T1 commits before T2 starts, then the T1 Commit timestamp is smaller than the timestamp of T2, ultimately guaranteeing that the order is maintained. To enable this, a "TrueTime" API is used that bounds clock uncertainty to enforce stronger time semantics in the distributed system. As a result, the system does not depend on loosely synchronized clocks and a weak time API, which allows the distributed datastore to ensure consistent transactions, while also being highly available due to data replication[3].

3.3 Validity of the claim

Google Cloud Spanner claims to be a distributed CA store, which contradicts the CAP theorem. According to Google, they introduce no trade-off between consistency and availability, because the service scales horizontally and serves data with low latency while maintaining transactional consistency with a 99,999% (five 9s) availability). In theory, Spanner is not a distributed CA system, as they did reduce the availability. In practice however, they managed to reduce the availability by such a small factor due to their infrastructure. Thus, the 99,9% availability may justify their claim of Spanner being a CA system.

3.4 RPC and REST

B. Describe RPC and REST. How do they compare?

As already mentioned, RPC is a mechanism that allows you to call a procedure (i.e. function) on another process and facilitate communication between distributed applications/processes through message passing. On the other hand, REST is an architecture, used to represent expose data to clients as ressources through the HTTP protocol, using the right HTTP verb. The endpoint contains the resource you manipulate and the CRUD analogy is often used to explain the request type. In other words, the HTTP verb indicates whether you want to (Create, Read, Update, Delete) with the resource.

On the other hand, the semantics of RPC are somewhat looser and there is no global shared understanding of its meaning. For example, to get an image in the web service, you could have either "GET /item?lat=..." or "GET /image", or maybe even something entirely different. Thus, with RPC you rely on human interpretation of the endpoint's meaning to understand what it does. As opposed to REST, where the semantic relies on the HTTP verb that is globally shared and agreed on. However, due to its non-restrictive nature, RPC may offer more flexibility as compared to REST that is somewhat "limited".

Nontheless, a fundamental problem with RPC is coupling. Due to its nature and the tight coupling to its service implementation, clients are required to know procedure names, parameter orders etc. Arguably, the REST architecture on top of the HTTP protocol makes it easy for clients to interact with the web service. Specifically, they do not need to worry about procedure names and arguments but instead rely on data exchange formats (e.g. text, XML, JSON). Also, it is stateless and therefore easier to design. Finally, the integration with HTTP ensures that data is communicated in a consistent, standardized and globally-agreed-on way.

REST (generic verbs on resources) VS RPC (specific custom actions)
RPC prone to failures and does not necessarily provide guarantees of request processing
REST framework provides universal HTTP verbs to perform consistent and transparent transactions on resources, whereas RPC thinks in terms of custom hand-coded “verbs”
Link: https://sites.google.com/site/wagingguerillasoftware/rest-series/what-is-restful-rest-vs-rpc
REST is a set of constraints (may work on top of RPC): http://etherealbits.com/2012/12/debunking-the-myths-of-rpc-rest/

A stateless protocol does not require the server to retain session information or status about each communications partner for the duration of multiple requests. In contrast, a protocol that requires keeping of the internal state on the server is known as a stateful protocol.

A stateful server maintains client’s state information from one remote procedure call to the next.

Merits of a stateful server: A stateful server provides an easier programming paradigm. It is typically more efficient than stateless servers.

Demerits of stateful server: If the server crashes and restarts, the state information it was holding may be lost and the client may produce inconsistent results. If the client process crashes and restarts, the server will have inconsistent information about the client.

4 Part 4 - Go programming (25%)

A. Simple TCP server: Explain the Go language concepts that allow you to use Fprintln to write to a net.Conn. The reason is because 'Fprintln' writes to an 'io.Writer' and 'net.Conn' is an 'io.Writer'. In other words, 'net.Conn' satisfies the interface of the 'io.Writer'. Namely, the type Writer is an interface that wraps the basic Write method:

```
type Writer interface {
        Write(p []byte) (n int, err error)
}
```

In this regard, 'net.Conn' implements the Write method as well to enable writing data to the connection:

```
type Conn interface {
        Write(b []byte) (n int, err error)
}
```

As a result, interfaces in Go make the code more flexible, scalable and it is a way to achieve to polymorphism in the way that the same Write interface may be shared by two different types. Instead of requiring a specific type, the interface allow you to specify a behavior as a method without enforcing a particular implementation. To implement (i.e. satisfy) the 'Write' interface, all you have to do is define the method with the desired name and signature[6].

4.1 Go Language Concepts

4.2 Modified Version

B. Modified version: Explain what this modified program does. How many users 'N' can it handle concurrently? Can you explain why it can handle only 'N' users concurrently? How can you fix it using go routines? (hint: you just need to change a single line).

The program starts a TCP server running on port 4000 and listens on the localhost IP (i.e. 127.0.0.1) for any incoming connection requests. In case of a connection, it is accepted and the bytes in the request are copied from the connection to itself. In practice, it can only handle one client connection, as it currently shares the 'io.Copy' operation between connections. In other words, it is occupied by the first connection to the server. To fix this, one may simply add a goroutine with the "go" keyword as in **"go io.Copy(c,c)**. As a result, multiple connections may now write to it concurrently.

4.3 Simple Chat Program

C. Using the above code as a starting point, design a simple chat program where multiple users can connect and are assigned a random partner. They then can chat with each other (i.e., everything user A writes is sent to user B and likewise). Your program should scale with the number of pairs that chat concurrently. You should use and discuss Go concurrency concepts (goroutines, channels, select) in your design. You are encouraged to write some code together with your explanation.

The server application running on the TCP socket should be extended with two types, Client and Channel. The server should then create an instance of the Client type for each TCP socket connection. In this regard, a Client will act as an intermediary between the TCP socket connection and an instance of a Channel type. The Channel type will maintain a set of registered clients and broadcast messages to the clients. As shown below, the extended version of the simple TCP listener program now provides chat functionality and handles multiple client requests using goroutines and channels:

```
func main(chan *Channel, w http.ResponseWriter, r *http.Request) {
        l, err := net.Listen("tcp", listenAddr)
    if err != nil {
                log.Println(err)
        }
        for {
                c, err := l.Accept()
                if err != nil {
                        log.Fatal(err)
                }
                client := &Client{channel: chan, conn: c, send: make(chan []byte, 256)}
                client.channel.register <- client
                go client.sendMessages()
                go client.readMessages()
        }
}
```

The application runs a goroutine for the Channel and two goroutines for each Client. In this way, the goroutines may communicate with each other using channels. The Channel has channels for registering clients, unregistering clients and broadcasting messages. A Client has a channel of outgoing messages. One of its goroutines is used to read messages from this channel and write them to the TCP socket. Finally, the other client goroutines should read messages from the TCP socket and send them to the Channel.

Effectively, by using channels and goroutines, the program scales with the number of pairs that may chat concurrently. Notice that the code has been attached in the appendix and illustrates the concepts implemented with a server (a modified version of the one provided) in section 5.6, a client in section 5.5 and a chat channel in section 5.4.

References

[1] J.C. Corbett et al. *Cloud Spanner: TrueTime and External Consistency*. https://cloud.google.com/spanner/docs/true-time-external-consistency. [Online; accessed 15/11/17]. 2012.

[2] Sau Sheong Chang. *Go Web Programming*. Manning Publications, 2016. ISBN: 9781617292569.

[3] James C. Corbett et al. "Spanner: Google's Globally-Distributed Database". In: (2012).

[4] *Debunking the Myths of RPC & REST*. http://etherealbits.com/2012/12/debunking-the-myths-of-rpc-rest/. [Online; accessed 15/11/17. 2012.

[5] Jonathan Furst. "Concurrency in Go". In: Scalability of Web Systems (MSc), IT University of Copenhagen. 2017.

[6] Michal Lowicki. *Interfaces in Go*. https://medium.com/golangspec/interfaces-in-go-part-i-4ae53a97479c. 2017.

5 Appendix

5.1 service.go

```
// Custom HTTP appHandler that includes error value for better error handling
type appHandler func(http.ResponseWriter, *http.Request) *appError
// User friendly error representation with error, message and HTTP status code
type appError struct {
        Error   error
        Message string
        Code    int // Server (500 Internal Error) or Client (400 Bad Request Error)
}
// Implement ServeHTTP to comply with the http.Handler interface
// Go functional feature: fn is a first order function
// that invokes the underlying http request function (e.g. get)
func (fn appHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
        w.Header().Set("Content-Type", "application/json")
        if err := fn(w, r); err != nil {
                http.Error(w, err.Message, err.Code)
        }
}
//Returns JSON array with links to all satellite images based on a location
func images(w http.ResponseWriter, r *http.Request) *appError {
        if err := r.ParseForm(); err != nil {
                return &appError{err, "Cannot parse data", http.StatusInternalServerError}
        }
        address := r.Form.Get("address")
        lat, lng, err := convertAddressToCoords(address, r)
        if err != nil {
                lat, lng = r.Form.Get("lat"), r.Form.Get("lng")
        }
        validLat, validLng := regexp.MustCompile(Latitude).MatchString(lat),
                                    regexp.MustCompile(Longitude).MatchString(lng)
        if !validLat && !validLng {
                return &appError{errors.New("Invalid coordinates"),
                                "Please provide a valid latitude and longitude",
                                http.StatusBadRequest}
        }
        projectID := os.Getenv("GOOGLE_CLOUD_PROJECT_ID")
        links, err := getLinks(lat, lng, projectID, r)
        if err != nil {
                return &appError{err, "Unable to retrieve links",
                                http.StatusInternalServerError}
        }
        if err := json.NewEncoder(w).Encode(links); err != nil {
                return &appError{err, "Unable to map JSON to response",
                                http.StatusInternalServerError}
        } return nil } // Success
```

5.2 query.go

```
// Retrieves links (i.e. granules) of satellite images that match location
func getLinks(lat, lng, proj string, r *http.Request) (Links, error) {

        granuleQuery := strings.TrimSpace(fmt.Sprintf(
                `SELECT granule_id, base_url
                 FROM %[1]sbigquery-public-data.cloud_storage_geo_index.sentinel_2_index%[1]s
                 WHERE %[2]s < north_lat
                 AND south_lat < %[2]s
                 AND %[3]s < east_lon
                 AND west_lon < %[3]s;`, "`", lat, lng))

        ctx := appengine.NewContext(r)

        client, err := bigquery.NewClient(ctx, proj)
        if err != nil {
            return links, err
        }

        query := client.Query(granuleQuery)
        query.QueryConfig.UseStandardSQL = true
        rows, err := query.Read(ctx)

        for {
                var row []bigquery.Value
                err := rows.Next(&row) // No rows left
                if err == iterator.Done {
                        return links, nil
                }

                if err != nil {
                        return links, err
                }

                granuleID := row[baseGranuleColumn].(string)
                links = append(links, granuleID)
        }
}
```

5.3 app.yaml

runtime: go api_version: go1
handlers: - url: /images script: service.images
- url: /.* script: service.index

5.4 chat.go

```
package main
// Channel maintains clients and broadcasts messages to clients.
type Channel struct {
        // Registered clients.
        clients map[*Client]bool

        // Inbound messages from the clients.
        broadcast chan []byte

        // Register requests from the clients.
        register chan *Client

        // Unregister requests from clients.
        unregister chan *Client
}
// Create new empty channel
func newChannel() *Channel {
        return &Channel{
                broadcast:  make(chan []byte),
                register:   make(chan *Client),
                unregister: make(chan *Client),
                clients:    make(map[*Client]bool),
        }
}
// Differentiate between clients that register/unregister and messages using select
func (c *Channel) run() {
        for {
                select {
                case client := <-c.register:
                        c.clients[client] = true
                case client := <-c.unregister:
                        if _, ok := c.clients[client]; ok {
                                delete(c.clients, client)
                                close(client.send)
                        }
                case message := <-c.broadcast:
                        for client := range c.clients {
                                select {
                                case client.send <- message:
                                default:
                                        close(client.send)
                                        delete(c.clients, client)
}}}}}
```

5.5 client.go

```
// Client is a middleman between the TCP socket connection and the channel
type Client struct {
        channel *Channel
        conn *net.Conn
        send chan []byte
}
// read messages from server socket connection to channel in separate goroutine
func (c *Client) readMessage() {
        defer func() {
                c.channel.unregister <- c
                c.conn.Close()
        }()
        for {
                _, message, err := c.conn.ReadMessage()
                if err != nil {
                        log.Printf("error: %v", err)
                        break
                }
                message = bytes.TrimSpace(bytes.Replace(message, newline, space, -1))
                c.channel.broadcast <- message
        }
}
// send messages from channel to server socket connection on separate goroutine
func (c *Client) sendMessage() {
        defer func() {
                c.conn.Close()
        }()
        for {
                select {
                case message, ok := <-c.send:
                        if !ok { // Channel closed
                                c.conn.Write([]byte{"Connection is closed"})
                                return
                        }
                        w, err := c.conn.Write(message) // TODO: pass message
                        if err != nil {
                                return
                        } // Add queued chat messages to current TCP socket message
                        w.Write(message)
                        n := len(c.send)
                        for i := 0; i < n; i++ {
                                w.Write(newline)
                                w.Write(<-c.send)
                        }
                        if err := w.Close(); err != nil {
                                return }}}}
```

5.6 main.go

```
// main handles TCP socket requests from from clients
func main(chan *Channel, w http.ResponseWriter, r *http.Request) {
        l, err := net.Listen("tcp", listenAddr)
    if err != nil {
                log.Println(err)
        }
        for {
                c, err := l.Accept()
                if err != nil {
                        log.Fatal(err)
                }
                client := &Client{channel: chan, conn: c, send: make(chan []byte, 256)}
                client.channel.register <- client
                go client.sendMessages()
                go client.readMessages()
        }
}
```