P3 - Techniques for working with cloud-scale datasets

Introduction

In this exercise set you will get hands on experience with some techniques and concepts that often turn up when working with cloud systems and data. You will be using Azure Data Lake, which is a public cloud offering based on previously internal Microsoft technology. These are tools that close to every developer at the Microsoft office here in Tromsø has worked with at some point. World-wide, this technology – together with other systems – power reporting and logging pipelines for most (if not all) large Microsoft services.

You will need to implement some C# code to solve the exercises, and using Visual Studio while doing this is advisable. Visual Studio is a professional software development tool, and like open source products like Eclipse in many ways. In this context, "professional" means that it has many powerful but complex features, so you should expect a small learning curve to become truly efficient while working in it. Do not let this complexity scare you; there is a lot of good documentation available online.

Prerequisites

You will need to learn about U-SQL to be able to solve exercise 2 and 3. We suggest looking at the following tutorials and documentation. Notice that on the lab machines, all dependencies should already be installed. This includes Visual Studio 2017 and the Azure Data Lake SDK. In other words, you should be good to go.

If you want to run in the cloud:
   a) start by getting a free trial and by following this tutorial to verify your setup:
https://docs.microsoft.com/en-us/azure/data-lake-analytics/data-lake-analytics-get-started-portal
   b.   Next, you should follow the tutorial for running U-SQL scripts from Visual Studio:
https://docs.microsoft.com/en-us/azure/data-lake-analytics/data-lake-analytics-data-lake-tools-get-started

Everyone:
At this point you have verified your Azure Subscription is working (or you don't care and only want to run locally), and real development can begin. You should run through the tutorial on how to test and debug U-SQL code in Visual studio:
https://docs.microsoft.com/en-us/azure/data-lake-analytics/data-lake-analytics-data-lake-tools-local-run

To get more comfortable with U-SQL, see:
https://docs.microsoft.com/en-us/azure/data-lake-analytics/data-lake-analytics-u-sql-get-started

You will need to embed C# code in your U-SQL scripts. See:
https://msdn.microsoft.com/en-us/library/azure/mt621316.aspx
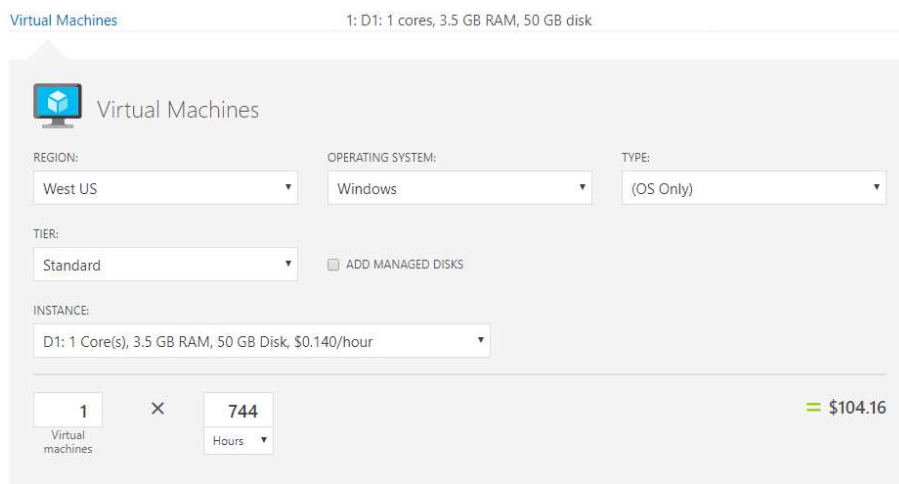
Exercise 1: Schema layouts and performance.

Extensibility and verbosity are often in conflict with performance when handling large amounts of data. One situation in which this manifests itself is choosing how to represent data in a log file or stream. In this exercise you will measure the performance differences between storing data in string and byte array format.

Consider a computer system hosting files. Each file has an associated ACL of variable size. Each user has a set of Claims, corresponding to entries in ACLs. Which users have access to what documents is governed by the ACL's on files and Claims of the users. The ACL of a file and the claims of a user are both modelled as sets of GUID's. For reasons of auditing and compliance, the computer system in this exercise must log the set of claims of a user each time he or she attempts to access a file. The set of claims is potentially very large (thousands).

a) Implement the method `static string GenerateStringClaimss(int numClaims)` in DataGeneration.cs. The method should generate a string representation of set of GUIDs with given size. You can choose how to represent this string in any way you like, for example JSON.

b) Implement the method `static List<Guid> ExtractClaims(string data)` in ClaimParsing.cs, that takes as input a set of GUIDs in string format. This method should work with your solution from a). Run a benchmark and produce a plot showing the cost of this operation for GUID sets of variable size. *Hint:* you can use the `Stopwatch` class to time your code, and create a ConsoleApplication project to implement a small command line program for doing the benchmarking.

c) Implement the method `static byte[] GenerateByteArrayClaims(int numClaims)` in DataGeneration.cs. The method should generate a byte array representation of a set of GUIDs with given size. GUIDs are essentially just 128 bit numbers. C# contains efficient methods for parsing and transforming GUIDs to and from byte array representation.

d) Implement `List<Guid> ExtractClaims(byte[] data)` in ClaimParsing.cs. This method should support variable sized sets of GUIDs from c). Benchmark this method for sets sized similar to what you chose in b), and compare the results.

e) Imagine you have to handle 1000 000 000 file accesses per day, and each file access amounts to approximately 1000 GUIDS being logged. According to the Microsoft Azure Pricing Calculator and the difference in CPU time between the methods of b) and c), what is the estimated cost difference over a year? Assume the following VM setup.

Your Estimate

Virtual Machines	1: D1: 1 cores, 3.5 GB RAM, 50 GB disk

Virtual Machines

REGION:
West US

OPERATING SYSTEM:
Windows

TYPE:
(OS Only)

TIER:
Standard

☐ ADD MANAGED DISKS

INSTANCE:
D1: 1 Core(s), 3.5 GB RAM, 50 GB Disk, $0.140/hour

1 Virtual machines	×	744 Hours	= $104.16

Exercise 2: Workload generation and log aggregates

In this exercise we will look at typical logs from a file storage subsystem, and how these can be aggregated. Such aggregates are useful for reporting reasons, but also take up less space. As such they are often stored much longer than the raw data from which they are created. First, however, you will need to create a tool for generating synthetic data that can be used to simulate this workload.

a) Implement a command line tool, SystemWideFileAccessLog, that can be used to create synthetic data for this exercise. Log entries – described by the class `SystemWideLogEntry` – should have the following format:

`DateTimeOfAccess, Region, TenantName, UserName, FileName, Status, Exception`

You should implement both constructors for `SystemWideLogEntry`, one where you can dictate the status (useful when verifying your solution in b)), and one where the status is randomized to give a reliability likely to be observed in real systems (google around a bit). When the status of a log entry is `Error`, the `Exception` column should be populated. If the status is `OK`, this column should be empty.

The Program itself should take as parameters:
The size of the output file in megabytes
The output file, e.g. C:\myfile.txt, to be created
The fraction of failures, where a default value indicates randomized generation

You can look at the `PerFileAccessLog` project for inspiration. Also look at this project for how to use the `Bogus` package, which can be useful for generating exception messages.

b) For reporting purposes, it is sometimes infeasible to handle the raw (and often verbose) logs. For this reason, it is common to create aggregates summarizing the raw numbers. Create a U-SQL script that takes as input a log file generated with the tool from a), and produces an output that aggregates numbers on a per-day basis. I.e., the output should contain one log line per day present in the input. These output log lines should contain numbers such as:
1. How many files were opened
2. How many distinct users were there
3. Calculate the distribution of calls going to different regions
4. What are the top 3 types of exceptions.
5. What are other interesting metrics you can come up with?

Exercise 3: Compliance, encryption, and probabilistic data structures

For reasons of compliance, all sensitive data (for example data that potentially can identify individuals or organizations) must be stored in encrypted form. Further, new EU regulations - the GDPR - require that users and/or organizations can have their data permanently deleted in a timely manner. Using the program in the `PerFileAccessLog` project, you can generate log files with variable size. The generated output is structured as follows: Each line corresponds to one file and a list of users that has modified the

file (each user is represented by an email-address). The schema of the file is:
<Date, Region, TenantName, FileName, ModifyingUsers, FirstAccess, LastAccess>

In this exercise you will transform the initial schema of the input file to an encrypted equivalent suitable for "cold storage". This will utilize a probabilistic data structure called a bloom filter to tie the users having accessed the file to the encrypted log-entry.

a) Write a U-SQL script to determine how many users access a file per day (average, standard deviation, and a set of percentiles of your choice). This information will come in handy in part b).

b) In the case that a user wants to have his or her data permanently deleted, we will need to delete that user's email address from all log lines containing it. We will leverage the bloom filter to determine whether or not we will have to decrypt a given log entry in order to do this. If we want a false positive rate of max 10%, how many bits will the bloom filter have to be?

c) Write a class implementing the `IBloomfilter` interface in IBloomfilter.cs in the `Util` project.

d) Write a U-SQL script that takes as input a logfile generated using the `PerFileAccessLog` tool, and produces an encrypted version containing a bloom filter and the encrypted data per entry. <Bloomfilter, EncryptedData>. You can choose your own encryption secret and use the code in EncryptionHelper.cs from the Util project.

e) Write a U-SQL script that takes input the augmented per-file access log from d) and an email address. The script should decrypt each line and if the email is located substitute it with "*deleted_user*".

f) Write a U-SQL script that takes input the augmented per-file-accesslog from d) and a user email. The script should query the bloom filter to determine whether a user is likely to be present in the line, and if so decrypt and do the substitution from e).

g) Do a benchmark of the scripts from e) and f).

h) Encryption has cost. What is the storage overhead of doing encryption (in percent)?

i) Extra Credit: Write an alternative implementation of a bloom filter using C# generics, i.e. `Bloomfilter<T>`.