Open Source Research Experience - SkyhookDM

Introduction

Proposal Title

SkyhookDM - Ability to Push back query execution to Client in case of overload OSDs

Problem Definition

Currently, SkyhookDM allows pushing down compute operations such as selection and projection into the Ceph file system, which is the Storage Layer. However, when a large number of clients try to push down computation into OSDs at a time, the CPU and memory pressure of the OSDs may quickly increase, causing run-time side effects such as blocked and slow OSD operations. The goal of this project is to create functionality, such that if there is high CPU and memory pressure in the OSDs, for which they cannot process filters or expressions efficiently, the query execution is pushed back to the client for processing.

Background of the Problem

- <u>Background: Work is offloaded to Ceph to process, rather than the client doing the work</u>
    - The Arrow Dataset API allows the definition of SQL-style expressions to filter data when reading data from files. Normally, expressions are applied after the client receives data from the source/file system. But with the Ceph File system, these expressions are applied at the server-side. This is because the SkyhookDM project includes custom read methods that are added to Ceph, where the storage layer can directly execute the custom read method when reading the data from the local disk. This offloading computation to storage servers should reduce the data returned to the client. In turn, less data is sent over the network, and the CPU work is distributed across many storage servers, rather than being done by the single client.

- <u>Problem: Storage servers in Ceph may be busy</u>
    - The problem that arises with the offloading computation to storage servers is that these storage servers may be too busy to apply an expression after reading their

local data. Then they may want to reject the request to apply the expressions. If so, the storage servers should 'pushback' the expressions to the client, along with the unprocessed data read from their local disk.

Solution

- Apply filters if possible, send the rest back to the Client
  - The client encodes the expression as it is into a custom read request and 'pushes down' the filter to storage. The storage then receives the read request along with the filters, but decides through some as-yet-undetermined mechanism that it cannot apply the filters (in *scan_op* function).
  - Therefore, it reads from the local disk as normal, but does not apply all the filters, although some may be applied (*ScanParquetObject* function).
  - Whatever filters that remain are pushed back to the client along with the data. This can be done by having the storage update the way it returns the data. The old assumption was the data was already filtered, so only the resultant data table was returned.
  - We will need to update the data returned from the server-side by adding a header with a length followed by the serialized filter expressions that remain, followed by the actual data (in *scan_op* function).
  - When the client receives this result, it can unpack the header, deserialize the expression given that there was indeed at least an expression pushed back from the server-side, and then the client can apply the expression to the data (done in *RadosParquetScanTask* method).

  - Links and more information are in the Expected Deliverables section.

Project goals

Project Objectives

- Create functionality in SkyhookDM, such that if expression(s) cannot be applied to a partitioned parquet table at the Ceph server-side (because the storage servers are too busy), the remaining filters are sent back to the client along with the data, so the client can apply the filters.

Expected Deliverables

- The *SerializeScanRequestToBufferlist* function is called on the client side and *scan_op* is called, which is the custom read code executed by the storage nodes that can read the data and then apply the filters that we pushed to storage. Below is a link where both the functions are called on the client-side.
- https://github.com/uccross/arrow/blob/rados-dataset-dev/cpp/src/arrow/dataset/file_rados_parquet.cc#L55-L59
- The storage unpacks the input buffer and deserializes the filter expression to be applied to the data through the *scan_op* function.
- *Scan_op*: https://github.com/uccross/arrow/blob/rados-dataset-dev/cpp/src/arrow/adapters/arrow-rados-cls/cls_arrow.cc#L153
- *Deserialize*: https://github.com/uccross/arrow/blob/rados-dataset-dev/cpp/src/arrow/adapters/arrow-rados-cls/cls_arrow.cc#L162

- In Storage:
    - A check on the storage side needs to be added to represent if a pushback is needed. This could be a boolean value, where True is returned if the CPU and memory usage passes a user-defined threshold or some other metric. This could also be done by comparing the filters that have been applied to the original filters sent to the storage.
    - I believe the check would go after the *ScanParquetObject* Function: https://github.com/uccross/arrow/blob/rados-dataset-dev/cpp/src/arrow/adapters/arrow-rados-cls/cls_arrow.cc#L169-L174

    - A new character buffer layout will be used for the return data of *scan_op*, which will have:

        - a header with a length and the serialized filter expressions that have not been applied yet
        - data table

    - If the check condition is True or *ScanParquetObject* function is not successful in applying all the filters, the header length will not be 0 because of filter expressions that need to be applied.
    - Else if the check is false, the header size will be 0 because there are no more filter expressions to be applied and the serialized resultant table should be returned.
    - As of right now, there is a function called *SerializeTableToBufferlist*, and this was responsible for returning the old resultant table as the output buffer list. Now, we must either create a new method to include the new header along with the table serialized or edit the *SerializeTableToBufferlist* function.
    - *SerializeTableToBufferlist* called: https://github.com/uccross/arrow/blob/rados-dataset-dev/cpp/src/arrow/adapters/arrow-rados-cls/cls_arrow.cc#L178
    - *SerializeTableToBufferlist* defined: https://github.com/uccross/arrow/blob/rados-dataset-dev/cpp/src/arrow/dataset/rados_utils.h#L58-L59

- In Client:
    - An unpacking method must be created to deserialize the new header it receives and apply filters if necessary.
    - https://github.com/uccross/arrow/blob/rados-dataset-dev/cpp/src/arrow/dataset/rados_utils.h
    - Above is a link that contains the header file for the serialize and deserialize methods. The new method to be added on the client side would be defined here.
    - Here is where the new method must be called in order to unpack the output buffer list and apply the filters: https://github.com/uccross/arrow/blob/rados-dataset-dev/cpp/src/arrow/dataset/file_rados_parquet.cc#L63

- Create new tests

Challenges

- As of right now, I am still familiarizing with the code and the different functions in each file, although I have compiled it and run the demo with the Jupyter notebook.
- I believe that finding the check condition may be difficult because I am not sure how Skyhook will determine that it cannot apply the filters or how it will partially apply filters. If it applies specific filters, I think that we may have to keep track of the applied filters and compare that with those that aren't applied, but I am not sure yet.
- Finding tests and optimizing the solution may also be difficult.

Implementation Plan

Project methodology (How will you work to meet your goals?)

- The framework for the expected deliverables already exists, so adding functionality, adding robust tests, and doing benchmarking then optimizations is what remains.

- Since the requirements for the project are already defined, I will implement a feature of the pushback functionality, such as creating a buffer for remaining filters, then test this feature. Once the testing stage for the feature is done, I will move on to implementing the next feature, and so on.

- Once all the requirements for the project are met, I will do thorough testing for the project as a whole, benchmarking, and optimizations.

Describe the project's technical elements

- Programming Languages: C++, Python

- <u>SkyhookDM</u>: SkyhookDM is a client-server architecture, which comprises the Arrow Dataset API and the Ceph Distributed File System. The goal of Skyhook is to transparently grow and shrink storage and processing needs as demands change. It supports offloading computation and other data management techniques in order to reduce the resources used by the client and unnecessary data transfers across a network. Skyhook takes as input a Parquet file, partitions it by rows, and writes each partition to Ceph. These partitions, which contain rows, are read, then filters get applied to them. This is what is stored and processed by the storage.

- <u>Ceph</u>: Ceph is a widely used distributed object store system, whose foundation is the Reliable Autonomic Distributed Object Store (RADOS), which provides applications with object, block, and file system storage in a single unified storage cluster. Data is stored on servers, and those storage servers receive read requests from the client and process them server-side.

- <u>Apache Arrow</u>: SkyhookDM uses Arrow's Dataset API including the Arrow Compute library. All the data movements from the Ceph OSDs to the client happen in Apache Arrow format. Apache Arrow supports the definition of SQL style expressions to filter data when reading from data files/sources such as selection, projection, as well as user-defined functions. The Arrow Dataset API is extended with a "RADOS fragment" for SkyhookDM. This supports applying functions both within storage and on the client.

<u>Project Timeline</u>

Project plan and deliverables schedule

<u>Program Duration:</u> June 14th - Sept 10th

- <u>Step 0</u>: May 16th - June 13th
  - Receive result of proposal, continue practicing C++ and doing C++ projects.
- <u>Step 1</u>: June 14th - June 21st
  - Familiarize with and understand the architecture of SkyhookDM.
  - Review code on the client side and server side of SkyhookDM.

    - Start implementing check functionality to see if the server side needs to pushback query execution to the client.
- Step 2: June 22nd - June 29th
    - Finish implementing the check for the pushback to the client.
    - Test to see if this functionality works.
- Step 3: June 30th - July 7th
    - Create a buffer for filter(s) and data that need to be serialized and sent to the client for processing. Test this functionality.
- Step 4: July 8th - July 16th
    - Add deserialization function at the client to see if there still are filters that need to be applied to the data. Test new deserialization function.
- Step 5: July 9th - July 16th
    - Add and test a method to process the query on the client-side.
- Step 6: July 17th - August 2nd
    - Add robust tests, do benchmarking and optimizations
- Step 7: August 3rd - August 24th
    - Try to add functionality that sends filters and data back to server-side when CPU and memory pressure of OSDs decrease. Test this functionality
- Step 8: August 25th - September 10th
    - Extra time in case steps don't go as planned.

Biographical information

Contact information

- Full name: Eshan Bhargava
- Email address: esbharga@ucsc.edu
- Current academic affiliation: Incoming USC MS CS student
- Github: github.com/4eshanb

Relevant experience / previous work

- Multi-threaded Timer

    - Built a multi-threaded timer using Pintos OS in C. The sleep function suspends execution of the calling thread until time has advanced by the requested number of timer ticks. Unless the system is otherwise idle, the thread does not wake up after the requested number of timer ticks. Rather, the thread is put on the ready queue.
- Priority Scheduler
    - Developed priority-based scheduling for threads in Pintos OS in C using concurrency primitives. New threads created have a default priority of 31 out of 63. They are placed on a ready list and it is re-ordered if needed. Implemented priority donation for single and multiple priorities.
- Linux System Calls
    - Created a program that implemented Linux system calls in Pintos OS in C. Arguments are passed to user processes from the command line. System calls: create, open, close, read, and some of write.
- Tcsh Shell
    - Implemented echo, exit, alias, set, redirect, and pipe commands in C. Used Exec and Fork to create threads and complete other commands. Used Hash tables to store variables and aliases.
- Merge Sort Speedup
    - Constructed a multi-process and multi-threaded merge sort in C. Multi-process merge sort utilized shared memory and inter-process communication. Achieved 1.5 times speedup compared to single process and single threaded merge sort.
- Algorithms and Data Structures
    - Implemented various Data Structures in Java and C, such as Linked lists, Binary Trees, Hash tables, Binary Search, Trees, Stacks, and Queues. Implemented many Algorithms, such as Recursive algorithms, Dynamic programming algorithms, Backtracking algorithms, Divide and conquer algorithms, and Greedy algorithms.
- Distributed Key-Value Store
    - Developed REST API in python with Flask to create Sharded, Fault-tolerant, Key-Value Store that enforces Causal Consistency with Vector Clocks. Used Docker to create containers and a network for nodes to communicate. Each shard

had at least 2 nodes to provide fault tolerance. Distributed keys with consistent hashing. Supports CRUD operations.

- Smart Home Network
  - Built an Object Oriented, Client-Server Application to simulate Smart Home Network controlling multiple appliances in a house. Once a user logs in, they control lights, locks, and the house alarm.
- Simple Router
  - Created Mininet Topology with server, 8 hosts, 1 untrusted host, 6 switches. Used Pox python API to create a controller. Blocked the untrusted host from sending ICMP traffic to other hosts and the server. Blocked an untrusted host from sending IP traffic to the server. Allowed other non-IP traffic to be communicated between hosts and the server.

Relevant education background; relevant coursework

- UCSC BS Computer Science - Sep 2017 to June 2020
  - 3.58 GPA
  - Graduated with Honors in the Major
  - Made Dean's List for 3 quarters
  - Relevant courses: Computer System Design (CSE 130), Distributed Systems (CSE 138), Database Systems (CMPS 180), Algorithm Analysis (CSE 102), Comparative Programming Languages (CMPS 112), Computer Networks (CSE 150)

Programming/development interests and strengths

I completed my Bachelor of Science in Computer Science at the University of California, Santa Cruz, in June 2020. I finished my degree in just three years, with honors in my major while maintaining a competitive GPA. This program provided me with a firm foundation and exposed me to many disciplines within Computer Science. However, I decided I wanted to learn more about Computer Science, and thus applied for Master's in Computer Science degree programs; I got accepted into many programs and will attend USC in the Fall of 2021.

During my time at UC Santa Cruz, I gained three years of experience with C programming through doing projects in Operating Systems, Algorithms, and Systems Programming. I believe that I can pick up C++ quickly because I am familiar with C syntax and have experience with Object-Oriented design in both Java and Python. I have three years of experience with Python and have used this programming language to build projects in Distributed Systems, Computer Networks, Artificial Intelligence, and Natural Language Processing.