KTH Royal Institute of Technology
Stockholm

School of Electrical Engineering and
Computer Science

Data-Intensive Computing - ID2221

Review Questions 5

Author
Emil Ståhl

Author
Selemawit Fsha

October 10, 2021

Review Questions 5 - ID2221

Emil Ståhl and Selemawit Fsha Nguse

October 10, 2021

1 Assume we have two types of resources in the system, i.e., CPU and Memory. In total we have 28 CPU and 56GB RAM (e.g., 1 CPU = 2 GB). There are two users in the systems. User 1 needs $\langle 2CPU, 2GB\rangle$ per task, and user 2 needs $\langle 1CPU, 4GB\rangle$ per task. How do you share the resources fairly among these two users, considering (i) the asset fairness, and (ii) DRF.

- Total resources:$\langle 28CPU, 56GB\rangle$ (e.g., 1 CPU = 2 GB)
- User 1 has x tasks and wants $\langle 18CPU, 18GB\rangle$ per task
- User 2 has x tasks and wants $\langle 9CPU, 36GB\rangle$ per task

Asset fairness:

In asset fairness we forget the type of resources and equalize all the resources as one type of resource. Using this given (e.g., 1 CPU = 2 GB) we calculate the total unite of resources

user 1: replace 4GB to 2 CPU so user 1 requires 2 + 1 = 3 total unit of resource

user 2: replace 2GB to 1 CPU so user 2 requires 1 + 2 = 3 total unit of resource

First we maximize the location
max(x,y) (maximize allocation)
subject to
$2x + y \leq 28$
$2x + 4y \leq 56$
$x \leq 28/3$
$y \leq 56/3$
User 1: x = 9:
User 2: y = 9:

If resources can not be shared between different frameworks, we see that user 1 should get 18 CPU's and 18 GB, while user 2 should get 9 CPU's and 36 GB. This solution does not violate the share grantee. However, one CPU (and 2 GB of RAM)does not get utilized.

DFS:

By applying max-min fairness to dominant shares: give every user an equal share of her dominant resource.

First we find the dominant resource for each users.
So, Each task from user 1 consumes :

- CPU $\frac{2}{28} = 7.1$ %
- RAM $\frac{2}{56} = 3.57$ %
- Dominant resource of User 1 is CPU (7.1 % $>$ 3.57 %)

Each task from user 2 consumes :

- CPU $\frac{1}{28} = 3.57$ %
- RAM $\frac{4}{56} = 7.1$ %
- Dominant resource of User 2 is RAM (3.57 % $<$ 7.1 %)

According to DRF and above calculation, user 1 has CPU as dominant resource and user 2 has RAM as dominant resource. This gives us:
max(x,y) (maximize allocation)
subject to
$2x + y \leq 28 (CPU constraint)$
$2x + 4y \leq 56 (Memory constraint)$

Now, we see that this equation system is the same as in task 1 when applying asset fairness. Therefore, applying DRF will in this case result in the same resource allocation as when using asset fairness. Which is user 1 should get 18 CPU's and 18 GB, while user 2 should get 9 CPU's and 36 GB.

2 What are the similarities and differences among Mesos, YARN, and Borg?

Mesos

Mesos is a common resource offered-based sharing layer, over which diverse frameworks can run. Its design element consists of fine-grained sharing and resource offers. Fine-grained sharing improves utilization, latency, and data locality. Thanks to resource offers, Mesos is simple since it support future frameworks. It also uses, Max-Min fairness and DRF to divide resources.

Mesos uses a two-level scheduler with separate concerns of resource allocation and task placement. An active resource manager offers compute resources to multiple parallel, independent scheduler frameworks. Mesos determines which resources are available, and it makes offers back to an application scheduler (the application scheduler and its executor is called a "framework"). Those offers can be accepted or rejected by the framework. This model is considered a non-monolithic model because it is a "two-level" scheduler, where scheduling algorithms are pluggable. Mesos allows an infinite number of schedule algorithms to be developed, each with its own strategy for which offers to accept or decline, and can accommodate thousands of these schedulers running multi-tenant on the same cluster.

Yarn

The Yarn architecture consists of:

- Resource Manager (RM)
- Application Master (AM)
- Node Manager (NM)

The fundamental idea of YARN is to split up the functionalities of resource management and job scheduling/monitoring into separate daemons. YARN runs one RM per cluster offering a central global view. Job requests are submitted to the RM. To start a job, RM finds a container to spawn AM. However, the RM only handles an overall resource profile for each job where the local optimization is up to the job. The AM is the head of the job and runs as a container requesting resources from RM. Lastly, the NM is the worker daemon that registers with the RM, there's one NM per node and reports resources to the RM.

Yarn uses a two-level scheduler with separate concerns of resource allocation and task placement. An active resource manager offers compute resources to multiple parallel, independent scheduler frameworks. When a job request comes into the YARN resource manager, YARN evaluates all the resources available, and it places the job. It's the one making the decision where jobs should go; thus, it is modeled in a monolithic way. It is important to reiterate that YARN was created as a necessity for the evolutionary step of the MapReduce framework. YARN took the resource-management model out of the MapReduce 1 JobTracker, generalized it, and moved it into its own separate ResourceManager component, largely motivated by the need to scale Hadoop jobs.

Borg

Borg is cluster management system at Google. The Borg architecture consists of:

- Cell: a set of machines managed by Borg as one unit.
- Job: users submit work in the form of jobs.
- Task: each job contains one or more tasks.
- Alloc: reserved set of resources and a job can run in an alloc set.
- Alloc instance: making each of its tasks run in an alloc instance.
- BorgMaster: The central brain of the system. Holds the cluster state. Replicated for reliability
- Borglet: Manage and monitor tasks and resource

The Borg scheduler works as follows: It applies feasibility checking to find machines for a given job

- Scoring: pick one machines
- User prefs and built-in criteria
- Minimize the number and priority of the preempted tasks
- Picking machines that already have a copy of the task's packages
- Spreading tasks across power and failure domains
- Packing by mixing high and low priority tasks

Borg uses a monolithic scheduler with a single and centralized scheduling algorithm for all jobs.

Similarities

- Yarn and Borg are request-based
- Yarn and Mesos uses two-level scheduler
- Yarn and Mesos are open source, Borg proprietary of Google
- Yarn and Mesos uses heartbeats, Borg polls

- Because YARN 's ResourceManager maintains information such as the status and location of each assigned Container, it is heavier than Mesos Master
- Both YARN and Mesos masters use Zookeeper to solve the high availability problem
- Both Mesos and YARN have good support for batch jobs. The support of such applications is also the simplest

Differences

- Borg slices borglets, neither YARN nor Mesos can do it they have only one master leading to bottlenecks
- Borg has built-in various error retry mechanisms to ensure that the application will not fail in the event of machine failure or network failure
- After Borgmaster and Borglet hang up, the applications and tasks running on it will not be affected. At present, Mesos and YARN have already done so
- Borg supports resource pre-allocation where the application can apply for some resources in advance, and can be used to quickly start some low-latency tasks, dynamic expansion and other aspects. Yarn and Mesos can not.

The primary difference between Mesos and YARN is around their design priorities and how they approach scheduling work. Mesos was built to be a scalable global resource manager for the entire data center. It was designed at UC Berkeley in 2007 and hardened in production at companies like Twitter and Airbnb. YARN was created out of the necessity to scale Hadoop. Prior to YARN, resource management was embedded in Hadoop MapReduce V1, and it had to be removed in order to help MapReduce scale. The MapReduce 1 JobTracker wouldn't practically scale beyond a couple thousand machines. The creation of YARN was essential to the next iteration of Hadoop's lifecycle, primarily around scaling. Borg horizontally slices all the borglets it manages, allowing each Borgmaster to take charge of a part. These borgmasters share cluster meta information. Each Borgmaster can allocate resources for the application, but backup Borgmaster needs to send the allocation information to the active Borgmaster Approval, this place is consistent with the share state in Google Omega. In this respect, neither YARN nor Mesos can do it. They have only one master for resource management and scheduling. In a very large cluster, the master may become a bottleneck. This is the direction that YARN and Mesos need to improve.

Summary

At present, it seems that Mesos/YARN's architecture and design still have a certain gap with Google Borg, but it should be noted that many details are the result of tradeoff. It is difficult to say which mechanism is more suitable for our scenario. For building small and medium-sized clusters and data centers, Mesos/YARN is more than enough.

3 What are the differences between Warehouse and Datalake? What is Lakehouse?

Data warehouse

The concept of Data warehouse goes back to the late 1980s. But it could not support rapidly growing unstructured and semi-structured data: time series, logs, images, documents, etc. It comes with high cost to store large datasets and no support for data science and ML.

Data Lakes

In 2010s, the rapid growing need for large datasets led to the concepts of Data Lakes. It support low-cost storage to hold all raw data, e.g., Amazon S3, and HDFS. ETL jobs then load specific data into warehouses, possibly for further ELT. Directly readable in ML libraries (e.g., TensorFlow and PyTorch) due to open file format. Cheap to store all the data, but system architecture is much more complex. Data reliability suffers: Multiple storage systems with different semantics, SQL dialects, etc. Extra ETL steps that can go wrong. Timeliness suffers and high cost. Extra ETL steps before data is available in data warehouses. Continuous ETL, duplicated storage.

3.1 Data Lake vs. Data Warehouse

Data Lake stores all data irrespective of the source and its structure whereas Data Warehouse stores data in quantitative metrics with their attributes. Data Lake is a storage repository that stores huge structured, semi-structured and unstructured data while Data Warehouse is blending of technologies and component that allows the strategic use of data. Data Lake defines the schema after data is stored whereas Data Warehouse defines the schema before data is stored. Data Lake uses the ELT (Extract, Load, Transform) process while the Data Warehouse uses ETL (Extract, Transform, Load) process. Data Lake is ideal for those who want in-depth analysis whereas Data Warehouse is ideal for operational users.

Functionality	**Data Lake**	**Data Warehouse**
Data Structure	Raw	Processed
Purpose of data	Not yet determined	Currently in use
Users	Data Scientists	Business Professionals
Accessibility	Highly accessible and quick to update	More complicated and costly to make changes

Table 1: Data Lake vs. Data Warehouse

Lakehouse

A data lakehouse is a data solution concept that combines elements of the data warehouse with those of the data lake. Data lakehouses implement data warehouses' data structures and management features for data lakes, which are typically more cost-effective for data storage. Data lakehouses are useful to data scientists as they enable machine learning and business intelligence while simplifying enterprise data architectures. Lakehouse implements Data Warehouse management and performance features on top of directly-accessible data in open formats. Key enabling technologies for Lakehouse include metadata layers for Data Lakes, new query engine designs, and declarative access for data science and ML. The metadata layer add transactions and versioning. It also track which files are part of a table version to offer rich management features like transactions. Clients can then access the underlying files at high speed. Optimistic concurrency.

As a combination of data warehouses and data lakes, data lakehouses feature elements of both data platforms. Namely:

- Concurrent reading and writing of data
- Schema support with mechanisms for data governance
- Direct access to source data
- Separation of storage and compute resources
- Standardized storage formats
- Support for structured and semi-structured data types, including IoT data
- End-to-end streaming

A single data lakehouse has several advantages over a multiple-solution system, including:

- Less time and effort administrating
- Simplified schema and data governance
- Reduced data movement and redundancy

- Direct access to data for analysis tools
- Cost-effective data storage

4 Briefly explain how Deltalake handle concurrent writing on the same file.

Delta Lake is an open source storage layer that brings reliability to Data Lakes. It provides ACID transactions, scalable metadata handling, time travel and versioning, and unifies streaming and batch data processing. Delta Lake maintains information about which objects are part of a Delta table in an ACID manner using a write-ahead log. A Delta Lake table is a directory that holds data objects and a transaction log (DeltaLog) of transaction operations. The DeltaLog is a single source of truth to show users correct views of the data at all times. It works as a central repository that tracks all changes that users make to the table (an ordered record of every transaction). DeltaLake uses the DeltaLog for many features including ACID transactions, scalable metadata handling, time travel, etc. Before applying any operation on a Delta Lake table, Spark checks the table DeltaLog to see what new transactions have posted to the table.

When a user creates a DeltaLake table, that table's DeltaLog is automatically created in the delta log sub directory. Any changes to that table are then recorded as ordered, atomic commits in the DeltaLog. Each commit is written out as a JSON file, starting with 000000.json. Additional changes to the table generate subsequent JSON files in ascending numerical order, e.g., 000001.json, 000002.json, and so on. Each log record object contains a commit, i.e., an array of actions recorded as atomic, ordered units.

- Change metadata: name, schema, partitioning, etc.
- Add/remove file: adds/removes a file
- Protocol evolution: upgrades the version of the transaction protocol
- Set transaction: records an idempotent transaction id
- Commit info: information around commit for auditing

*Concurrent writing on same file DeltaLake utilizes optimistic concurrency control to ensure that the resulting state of the table after multiple concurrent writes is the same as if those writes had occurred serially, in isolation from one another. The process proceeds like this:

- Record the starting table version.
- Record reads/writes.
- Attempt a commit.

- If someone else wins, check whether anything you read has changed.
- Repeat

Example of optimistic concurrency control: Two users read from the same table, then each attempts to add some data to it. Here, we run into a conflict because only one commit can come next and be recorded as 000001.json. User 1 is chosen as the accepted commit. However, Delta Lake does not throw an error for user 2 and handles this conflict optimistically. DeltaLake handles this as follows:

- Delta Lake checks to see whether any new commits have been made to the table, and updates the table silently to reflect those changes
- Then, it retries the commit from user 2 on the newly updated table (without any data processing), successfully committing 000002.json
- In the vast majority of cases, this reconciliation happens silently and successfully

Benefits of DeltaLog

Every table is the result of the sum of all of the commits recorded in the Delta Lake DeltaLog. The DeltaLog provides a step-by-step instruction guide, detailing exactly how to get from the table's original state to its current state. Thus, it's possible to recreate the state of a table at any point in time. This ability is known as time travel or data versioning.