# ADVANCED TOPICS IN DATA ENGINEERING

*Apache Drill & Impala Assignment*

*Supervisor: Mr. Giorgos Alexiou*

ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ
ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

ΣΧΟΛΗ ΔΙΟΙΚΗΣΗΣ ΕΠΙΧΕΙΡΗΣΕΩΝ
SCHOOL OF BUSINESS

ΤΜΗΜΑ ΔΙΟΙΚΗΤΙΚΗΣ ΕΠΙΣΤΗΜΗΣ & ΤΕΧΝΟΛΟΓΙΑΣ
DEPARTMENT OF MANAGEMENT SCIENCE & TECHNOLOGY

Dimitris Matsanganis,
f2822212

Friday August 18th, 2023

## Abstract

This report is an investigation into various aspects of data engineering and management, with a special emphasis on data virtualization and query optimization. It explores the implementation of data virtualization in three different technologies - Apache Hive, Impala, and Drill - identifying the one that most precisely embodies this concept. The report also offers a strategy to simplify data querying from multiple sources and formats for a large bookstore company. In addition, it delves into practical applications, providing SQL commands for creating and querying a database in an Impala environment according to a specific schema. Lastly, the report addresses the issue of query performance in Impala, outlining a systematic approach to diagnose and optimize a slow-running query.

## Contents

## Table of Figures

# Assignment Description

**Task 1 [25 points]**

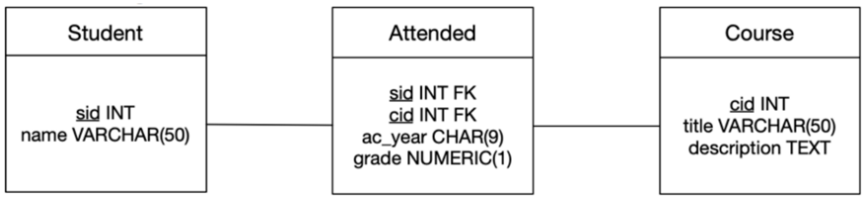
Among Hive, Impala, and Drill, which is the one the implements more precisely the concept of data virtualization? Elaborate.

**Task 2 [25 points]**

You started working for a large bookstore company. Your client has a large data center containing data in various formats. More specifically, all client data (e.g., personal information, orders) are stored in a Mongo DB database, e-books are stored on HDFS, and social media metadata (likes, ratings, reviews) are stored in a Hive database. They would like to simplify the queries used by various User Interface elements. What would you suggest for their case? Elaborate.

**Task 3 [40 points]**

Another client of yours has an Impala database for their needs. They want to have a new database with the following schema:



Please provide detailed answers to the following:

- 3a) Create the Impala database & the required tables.
- 3b) Give an example command that inserts an entry to the Student table (use your own details for that entry).
- 3c) Write a statement that retrieves all the names of the students that have attended the course having title “Artificial Intelligence” during the academic year “2021-2022”.
- 3d) Write a statement that retrieves the titles and the average grades of all the courses for which the average grade of the students that attended them is lower than 6.

**Task 4 [10 points]**

A particular query in the previous Impala database is too slow. Describe what you are going to do to investigate what is going wrong and what can be done to improve efficiency. Provide any commands that you are going to run.

## Case Study Outline

The case study section of this report is divided into four main tasks, each focusing on a different aspect of data engineering.

In Task 1, we provide a theoretical evaluation of three prominent data querying technologies - Apache Hive, Impala, and Drill - focusing on their ability to implement data virtualization.

Task 2 brings us to a real-world scenario where we propose a solution for a large bookstore company looking to simplify their querying process across diverse data sources and formats.

Task 3 is a hands-on exploration of the Impala database system. This task guides us through the creation of a new database and tables, demonstrates how to insert data into these tables, and showcases the formulation of complex SQL queries to extract specific information.

Finally, Task 4 addresses the crucial aspect of query optimization. This task offers a systematic approach to diagnose and enhance the efficiency of a slow-running query in the Impala environment, emphasizing the use of the **EXPLAIN** statement and other optimization techniques.

## Installation and Setup of Impala via Cloudera QuickStart VM

For the purpose of performing tasks that require the use of Apache Impala, we opted to set up a virtual environment using **Cloudera's QuickStart VM**. Cloudera's QuickStart VM is a single-node Apache Hadoop cluster that comes with a pre-configured Apache Impala.

The QuickStart VM offers a simple and convenient way of running Impala queries without the need to install and configure each component of the Hadoop ecosystem separately. This virtual environment not only includes Impala, but also incorporates a wide array of big data tools such as Hive, HBase, Spark, and others, allowing for extensive testing and development.

By creating a virtual machine with Cloudera QuickStart, we were able to quickly set up a fully functional Impala environment. This setup allows us to execute the commands necessary for *Task 3 and Task 4*, enabling the creation and manipulation of the database and tables as required. Furthermore, the Cloudera QuickStart VM provides a user-friendly interface via Cloudera Manager and Hue, making the process of running queries and managing the database more intuitive.

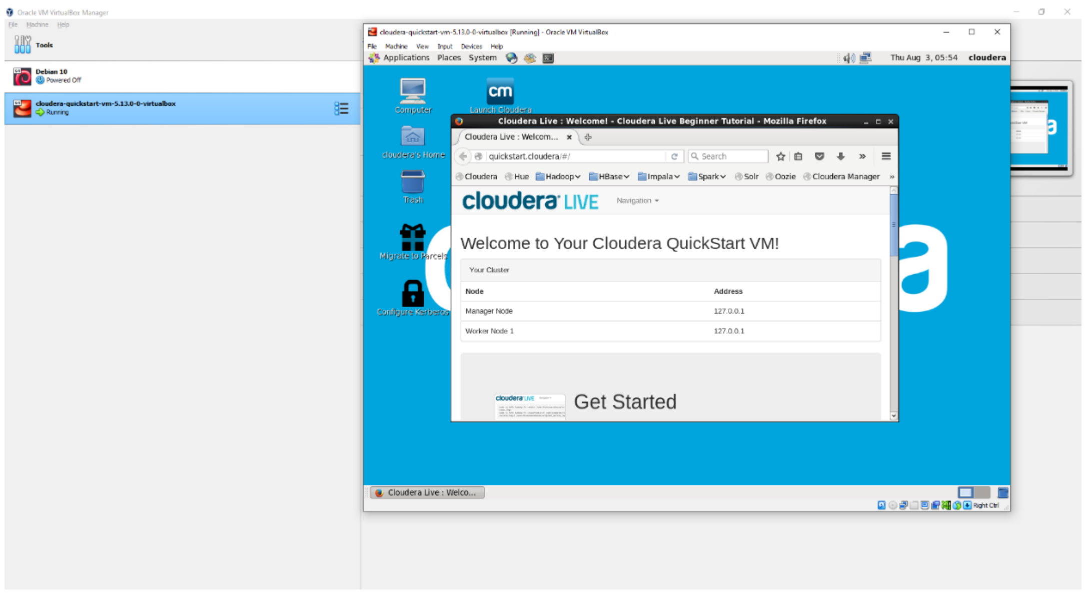


*Figure 1: Installation and Setup of Impala via Cloudera QuickStart VM.*

## Task 1: Evaluation of Data Virtualization in Hive, Impala, and Drill

*" Among Hive, Impala, and Drill, which is the one the implements more precisely the concept of data virtualization? Elaborate."*

In this task, we aim to understand the concept of data virtualization and how it is implemented across different data querying technologies. Among Apache Hive, Impala, and Drill, we will identify the tool that most precisely embodies this concept, analyzing their individual capabilities and how well they conform to the principles of data virtualization.

*Answer of Task 1:*

First of all, Data virtualization is a technique that enables users to retrieve, manipulate, and interact with data from an array of disparate sources, without necessitating an understanding of the specifics of where and how the data is stored. Essentially, it abstracts the underlying details pertaining to the data's physical location and format, thereby providing a unified view of the data.

When we draw comparisons between *Apache Drill, Impala, and Hive*, it becomes unequivocally clear that *Apache Drill emerges as the system best equipped to implement data virtualization*.

To be more precise, *Apache Drill*, a potent and schema-free SQL query engine, is capable of working with a diverse array of data sources and file formats. These encompass NoSQL databases, cloud storage, and nested file formats such as JSON and Parquet. Moreover, Drill can execute SQL queries across the full breadth of the Hadoop Distributed File System (HDFS) and has the capability to dynamically combine data from multiple sources. A key differentiator between Drill and its counterparts, Hive and Impala, is that Drill does not require any predefined schema or centralized metadata. In fact, it can discover the schema of the records as it processes them, an attribute that harmonizes well with the underlying concept of data virtualization.

On the other hand, *Impala*, despite being a SQL-on-Hadoop query engine, is somewhat limited in its capabilities compared to Drill. Specifically, it can only operate on top of the Hive metastore and necessitates at least one table/view to be declared prior to querying. Furthermore, it lacks the capability to query nested file formats like JSON and Parquet, nor can it dynamically handle data from multiple sources.

Meanwhile, *Hive* is a data warehouse software that provides a SQL-like interface to manage and query data stored in Hadoop. However, it is reliant on a predefined schema and is built atop the MapReduce framework, which often results in slower query performance compared to both Drill and Impala.

Taking these comparisons into account, it is apparent that ***Apache Drill is the system that most closely aligns with the principles of data virtualization***. This is primarily due to its schema-free model, its proficiency in querying a wide variety of data sources and file formats, and its ability to dynamically merge data from various sources.

## Task 2: Data Query Simplification for a Bookstore Company

*" You started working for a large bookstore company. Your client has a large data center containing data in various formats. More specifically, all client data (e.g., personal information, orders) are stored in a Mongo DB database, e-books are stored on HDFS, and social media metadata (likes, ratings, reviews) are stored in a Hive database. They would like to simplify the queries used by various User Interface elements. What would you suggest for their case? Elaborate."*

In this task, we are presented with a practical scenario of a large bookstore company. The client's data is distributed across various formats and stored in different databases. The objective is to propose a strategy that can simplify their data querying process across these disparate data sources and formats.

*Answer of Task 2:*

To begin with, given the diversity of our client's data ecosystem, which includes MongoDB for client data, Hadoop Distributed File System (HDFS) for e-book storage, and a Hive database for social media metadata, it is apparent that we are dealing with a high degree of data complexity. Managing and extracting valuable insights from such a multifaceted environment can be a daunting task. Therefore, we suggest the adoption of ***Apache Drill***, a data virtualization tool, to consolidate and streamline the querying process.

More specifically, Apache Drill sets itself apart with its flexibility and ability to query across a variety of storage systems. These systems include NoSQL databases like MongoDB, distributed file systems such as HDFS, and SQL-based data warehouses like Hive. Furthermore, unlike Hive and Impala, which are predominantly optimized for Hadoop, Apache Drill offers a wide range of connectors for diverse data sources. This distinctive capability allows us to conduct queries across all these different data stores as if they were housed within a single database. On top of that, the setup needed for Apache Drill to access the data stored on HDFS and MongoDB can be considered quite fast and easy to be done.

Therefore, a key advantage of Apache Drill is its schema-free SQL Query Engine for Hadoop and NoSQL. This feature negates the need for advanced schema creation and management, a task that can be both time-consuming and complex when dealing with large and diverse datasets. Furthermore, Drill's ability to join data across different data stores simplifies operations that are traditionally challenging with tools like Hive and Impala.

Thus for our case, by leveraging Apache Drill, we can greatly simplify the querying process for various User Interface elements. The tool's ability to access and query data as if it were consolidated in a single location substantially reduces the need for complex ETL processes and extensive data migration.

To be more practical it is essential to note that the deployment of Apache Drill should be supplemented with effective data management and security measures. To be more precise, it is paramount to ensure data consistency, accuracy, and reliability across all systems, which necessitates robust data governance and data quality practices. Given the presence of personal data within our client's MongoDB database, data security remains a high priority. Thus, correctly configuring Apache Drill to maintain data privacy is non-negotiable.

In summary, the incorporation of ***Apache Drill***, combined with a comprehensive data management strategy, provides a streamlined solution for managing and extracting valuable insights from this case's complex data environment. ***Therefore, we suggest for this case the option of Apache Drill.***

## Task 3: Database and Table Creation in Impala

*" Another client of yours has an Impala database for their needs. They want to have a new database with the following schema:*

*Please provide detailed answers to the following:*

- *3a) Create the Impala database & the required tables.*
- *3b) Give an example command that inserts an entry to the Student table (use your own details for that entry).*
- *3c) Write a statement that retrieves all the names of the students that have attended the course having title "Artificial Intelligence" during the academic year "2021-2022".*
- *3d) Write a statement that retrieves the titles and the average grades of all the courses for which the average grade of the students that attended them is lower than 6."*

This task delves into the practical application of commands in Impala environment. We are to create a new database and tables according to a specific schema provided by a client. Further, we will demonstrate data insertion into these tables and formulate complex queries to extract specific information from them. Last but not least, a variety of coding snippets and screenshots will follow to verify the above.

*Answer of Task 3:*

### 3a. Create the Impala database & the required tables.

In question 3a, we are asked to design a simple database for a university. The database will contain information about students, courses, and course attendance records, including grades. To accomplish this, we will utilize SQL, a standard language for managing data held in a relational database management system. Let's break down the SQL code:

```sql
-- If a database named "university" exists. If it does not exist,
-- it creates a new database with that name.
CREATE DATABASE IF NOT EXISTS university;

-- This statement is used to select the "university"
-- database for use in subsequent operations.
USE university;
```

The first two lines check if a database named "university" exists. If it does not, the code creates the database. Then, it selects "university" as the current database for subsequent operations.

```sql
CREATE TABLE Student(
sid INT,
name VARCHAR(50)
);
```

This code creates a table named "**Student**" within the "university" database. This table will hold records for all students. It has two fields: "**sid**" (an integer value representing the student ID) and "**name**" (a variable character field that can hold up to 50 characters, representing the name of the student).

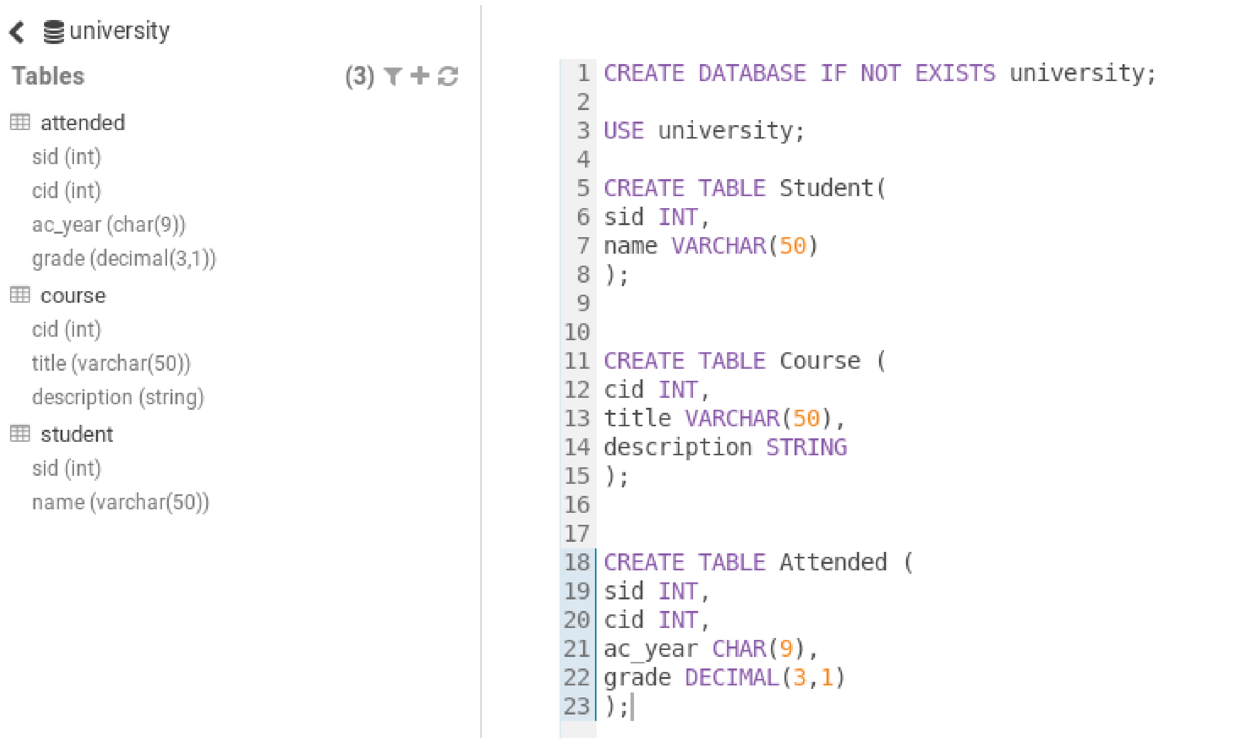
```
CREATE TABLE Course (
cid INT,
title VARCHAR(50),
description STRING
);
```

This block of code creates a table named "**Course**". This table will store information about all the courses offered at the university. It contains three fields: "**cid**" (an integer representing the course ID), "**title**" (a variable character field that can hold up to 50 characters, representing the course title), and "**description**" (a string field for the course description).

```
CREATE TABLE Attended (
sid INT,
cid INT,
ac_year CHAR(9),
grade DECIMAL(3,1)
);
```

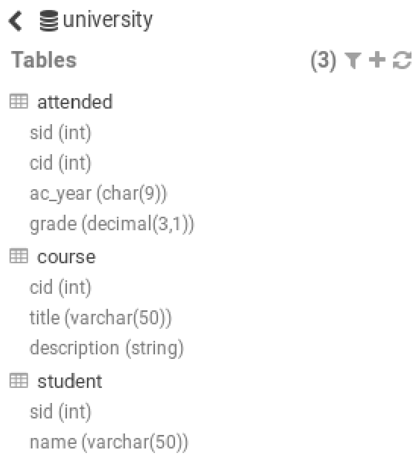
The final block creates a table named "**Attended**". This table acts as a junction table between the "Student" and "Course" tables, representing the many-to-many relationship between students and courses. It includes the following fields: "**sid**" (the student ID), "**cid**" (the course ID), "**ac_year**" (a character field of length 9, representing the academic year), and "**grade**" (a decimal field that can hold grades with one digit after the decimal point - like the university's one instead of the numeric(1) in the original scheme since is not representable).

Below you can see the above documented code after a successful execution and its results.



```
CREATE DATABASE IF NOT EXISTS university;

USE university;

CREATE TABLE Student(
sid INT,
name VARCHAR(50)
);


CREATE TABLE Course (
cid INT,
title VARCHAR(50),
description STRING
);


CREATE TABLE Attended (
sid INT,
cid INT,
ac_year CHAR(9),
grade DECIMAL(3,1)
);
```

*Figure 2: Create the Impala database & the required tables.*

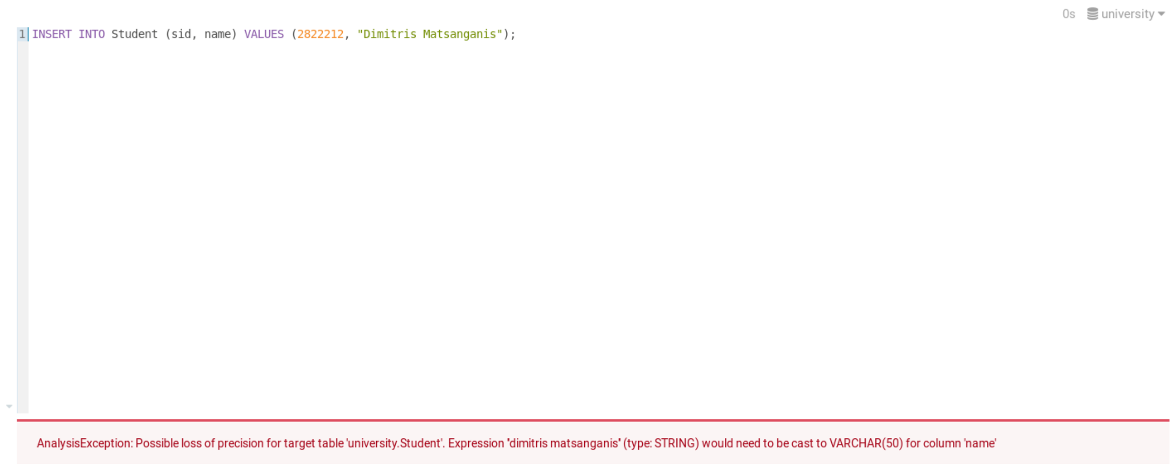


```
university
Tables                (3)
attended
  sid (int)
  cid (int)
  ac_year (char(9))
  grade (decimal(3,1))
course
  cid (int)
  title (varchar(50))
  description (string)
student
  sid (int)
  name (varchar(50))
```

*Figure 3: Provided schema.*

However, upon attempting to insert values into our tables as per task 3b, we encountered an issue due to Impala's handling of the VARCHAR data type. In the version of Impala we're using (Cloudera 5.13 through VM) and after an extensive research, there appears not to support for VARCHAR.

Our initial table creation process involved defining certain fields as VARCHAR, such as the **name** field in the **Student table** and the **title** field in the **Course table**. However, when we tried to insert data into these tables, Impala raised an error indicating potential loss of precision (task 3b).



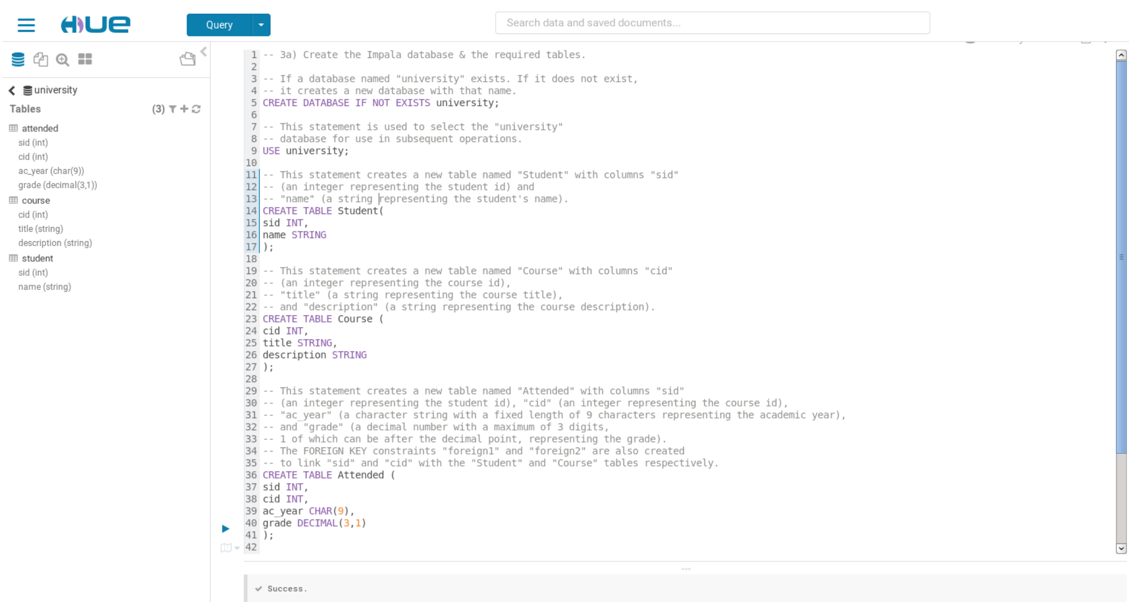
```
INSERT INTO Student (sid, name) VALUES (2822212, "Dimitris Matsanganis");
```

```
AnalysisException: Possible loss of precision for target table 'university.Student'. Expression 'dimitris matsanganis' (type: STRING) would need to be cast to VARCHAR(50) for column 'name'
```

*Figure 4: Executing the Task B's command variable type error.*

After analyzing the issue, we decided to revise our table creation process, replacing the VARCHAR data type with the STRING data type for those fields. The STRING data type in Impala is used to represent variable-length character data, which makes it a suitable replacement for VARCHAR. Another solution could be to cast the specific variables to STRING.

As such, we plan to drop the current tables and recreate them using the STRING data type for the necessary fields. This change will ensure compatibility with our current version of Impala and enable successful data insertion as required by task 3b. Below you can see the updated code that creates the same schema with this minor change on these two fields in order to be able to import data for task 3b.



```sql
-- 3a) Create the Impala database & the required tables.

-- If a database named "university" exists. If it does not exist,
-- it creates a new database with that name.
CREATE DATABASE IF NOT EXISTS university;

-- This statement is used to select the "university"
-- database for use in subsequent operations.
USE university;

-- This statement creates a new table named "Student" with columns "sid"
-- (an integer representing the student id) and
-- "name" (a string representing the student's name).
CREATE TABLE Student(
sid INT,
name STRING
);

-- This statement creates a new table named "Course" with columns "cid"
-- (an integer representing the course id),
-- "title" (a string representing the course title),
-- and "description" (a string representing the course description).
CREATE TABLE Course (
cid INT,
title STRING,
description STRING
);

-- This statement creates a new table named "Attended" with columns "sid"
-- (an integer representing the student id), "cid" (an integer representing the course id),
-- "ac_year" (a character string with a fixed length of 9 characters representing the academic year),
-- and "grade" (a decimal number with a maximum of 3 digits,
-- 1 of which can be after the decimal point, representing the grade).
-- The FOREIGN KEY constraints "foreign1" and "foreign2" are also created
-- to link "sid" and "cid" with the "Student" and "Course" tables respectively.
CREATE TABLE Attended (
sid INT,
cid INT,
ac_year CHAR(9),
grade DECIMAL(3,1)
);

```

```
✓ Success.
```

*Figure 5: Create the Impala database & the required tables (revisited version).*

The final documented and revisited code for 3a can be found below:

```sql
-- If a database named "university" exists. If it does not exist,
-- it creates a new database with that name.
CREATE DATABASE IF NOT EXISTS university;

-- This statement is used to select the "university"
-- database for use in subsequent operations.
USE university;

-- This statement creates a new table named "Student" with columns "sid"
-- (an integer representing the student id) and
-- "name" (a string representing the student's name).
CREATE TABLE Student(
sid INT,
name STRING
);

-- This statement creates a new table named "Course" with columns "cid"
-- (an integer representing the course id),
-- "title" (a string representing the course title),
-- and "description" (a string representing the course description).
CREATE TABLE Course (
cid INT,
title STRING,
description STRING
);

-- This statement creates a new table named "Attended" with columns "sid"
-- (an integer representing the student id), "cid" (an integer representing the course id),
-- "ac_year" (a character string with a fixed length of 9 characters representing the academic year),
-- and "grade" (a decimal number with a maximum of 3 digits,
-- 1 of which can be after the decimal point, representing the grade).
-- The FOREIGN KEY constraints "foreign1" and "foreign2" are also created
-- to link "sid" and "cid" with the "Student" and "Course" tables respectively.
CREATE TABLE Attended (
sid INT,
cid INT,
ac_year CHAR(9),
grade DECIMAL(3,1)
);

-- Drop the database.
-- DROP DATABASE university CASCADE;
```

Now, prior to moving to task 3b and in order to validate the correct execution of task 3a, which entailed the creation of a database and the required tables in Apache Impala, we utilized a series of commands within our Impala environment.

Firstly, to confirm the successful creation of the **university** database, we issued the ***SHOW DATABASES*** command. This command generates a list of all currently existing databases within our Impala environment.

```
SHOW DATABASES;
```

Upon successful creation of the **university** database, it would be displayed within the output list of this command.

Subsequently, we aimed to verify the successful creation of the **Student**, **Course**, and **Attended** tables. To accomplish this, we selected the **university** database for use and executed the ***SHOW TABLES*** command. This command lists all the tables present within the currently active database.

```
USE university;
SHOW TABLES;
```

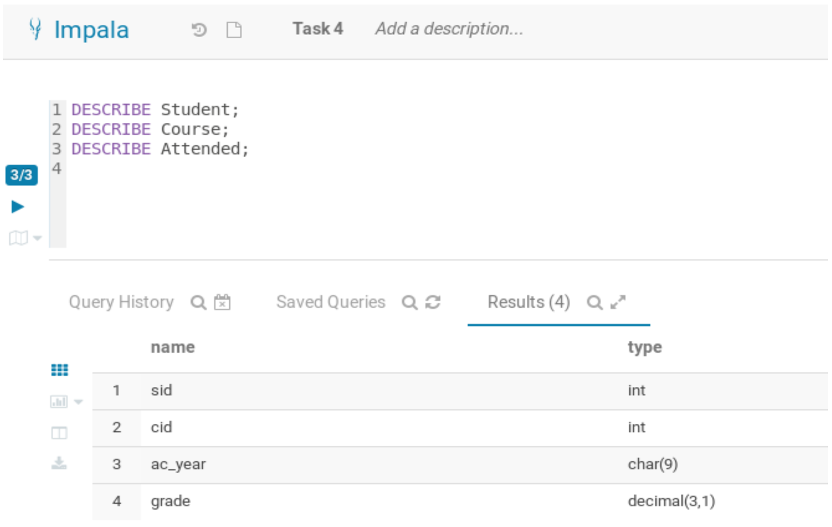
If the **Student**, **Course**, and **Attended** tables were successfully created, they would be included in the output list generated by this command - which was our case.

Finally, to ensure the correct structure of each table, we issued the ***DESCRIBE*** command for each table. This command displays the structure of a specified table, including column names and their associated data types.

```
DESCRIBE Student;
DESCRIBE Course;
DESCRIBE Attended;
```

If each table was created with the correct schema, the output from each ***DESCRIBE*** command would correspond to the schema outlined during the creation of the tables. By following these steps, we were able to systematically validate each component of Task 3a, ensuring the correct creation of the university database and the **Student**, **Course**, and **Attended** tables within our Impala environment.

To not extend the current task even more - we decide to include a screenshot of the ***DESCRIBE*** command execution and the table **Attended**, as a proof of work and to demonstrate our validation procedure.



```
DESCRIBE Student;
DESCRIBE Course;
DESCRIBE Attended;
```

|  | name | type |
|---|---|---|
| 1 | sid | int |
| 2 | cid | int |
| 3 | ac_year | char(9) |
| 4 | grade | decimal(3,1) |

*Figure 6: Validation procedure of task 3a.*

## 3b. Give an example command that inserts an entry to the Student table (use your own details for that entry).

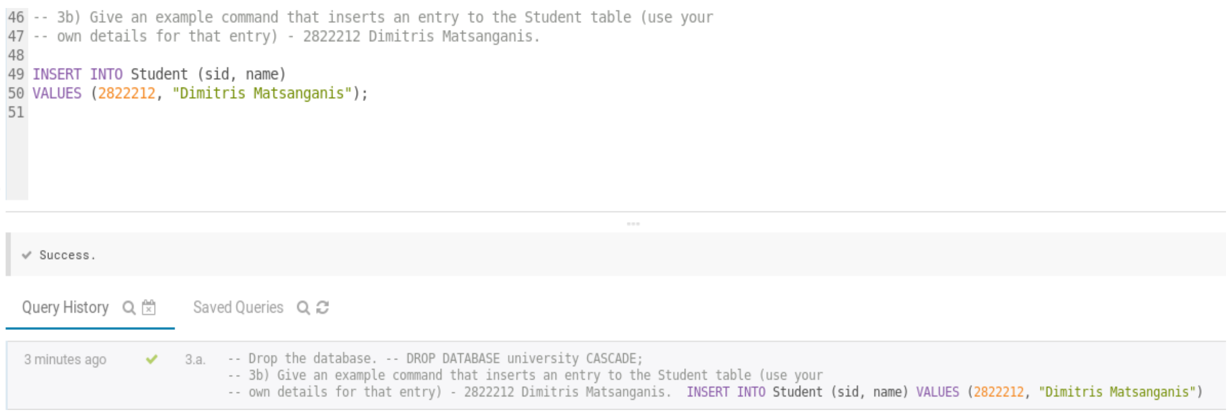
In the current task, Task 3b, we are tasked with demonstrating our capability to insert data into the previously created Student table within our Impala database. Specifically, we are to provide an illustrative example by inserting a fictitious student record using our own details. Below is provided the task's command:

```sql
-- 3b) Give an example command that inserts an entry to the Student table (use your
-- own details for that entry) - 2822212 Dimitris Matsanganis.

INSERT INTO Student (sid, name)
VALUES (2822212, "Dimitris Matsanganis");
```

This command is designed to insert a new record into the Student table. Let's break down the components of this command:

- **INSERT INTO Student (sid, name)**: This segment of the command specifies the target table (Student) and the columns (sid and name) where the data will be inserted.

- **VALUES (2822212, "Dimitris Matsanganis")**: This portion denotes the actual data values that will be inserted into the specified columns. The integer 2822212 will be inserted into the sid column, representing the student ID (f2822212 - but without the letter in an integer format), while the string "Dimitris Matsanganis" will be inserted into the name column, representing the student's name.



```sql
-- 3b) Give an example command that inserts an entry to the Student table (use your
-- own details for that entry) - 2822212 Dimitris Matsanganis.

INSERT INTO Student (sid, name)
VALUES (2822212, "Dimitris Matsanganis");
```

```
✓ Success.
```

```
3 minutes ago  ✓  3.a.  -- Drop the database. -- DROP DATABASE university CASCADE;
                        -- 3b) Give an example command that inserts an entry to the Student table (use your
                        -- own details for that entry) - 2822212 Dimitris Matsanganis.  INSERT INTO Student (sid, name) VALUES (2822212, "Dimitris Matsanganis")
```

*Figure 7: Executed command successfully adds student record in Impala.*

Upon executing this command (*see Figure 7 above*), the Student table will have a new record with the student ID 2822212 and the name "Dimitris Matsanganis". Through this insertion operation, we illustrate the basic data manipulation capabilities within Impala, showcasing how to add new records to existing tables.

## 3c. Write a statement that retrieves all the names of the students that have attended the course having title "Artificial Intelligence" during the academic year "2021-2022".

In the Task 3c, we are tasked with extracting specific data from our Impala database, pinpointing the names of students who enrolled in the "Artificial Intelligence" course during the academic year "2021-2022." To achieve this, we must traverse multiple tables, as the data we seek is dispersed across them.

The ***Student*** table holds the names of the students, while the ***Course*** table contains the details of various courses, and the ***Attended*** table keeps track of the attendance records, including the academic years. Therefore, our first step involves joining these tables to amalgamate the relevant information.

Once the tables are interconnected, our focus will shift to filtering the records. We need to hone in on the course titled "Artificial Intelligence." Furthermore, since we are only interested in entries from the academic year "2021-2022," an additional filter will be applied to narrow down the results to this specific timeframe.

Through the following command that encapsulates these considerations, we aim to produce a concise list of student names that align with the given criteria.

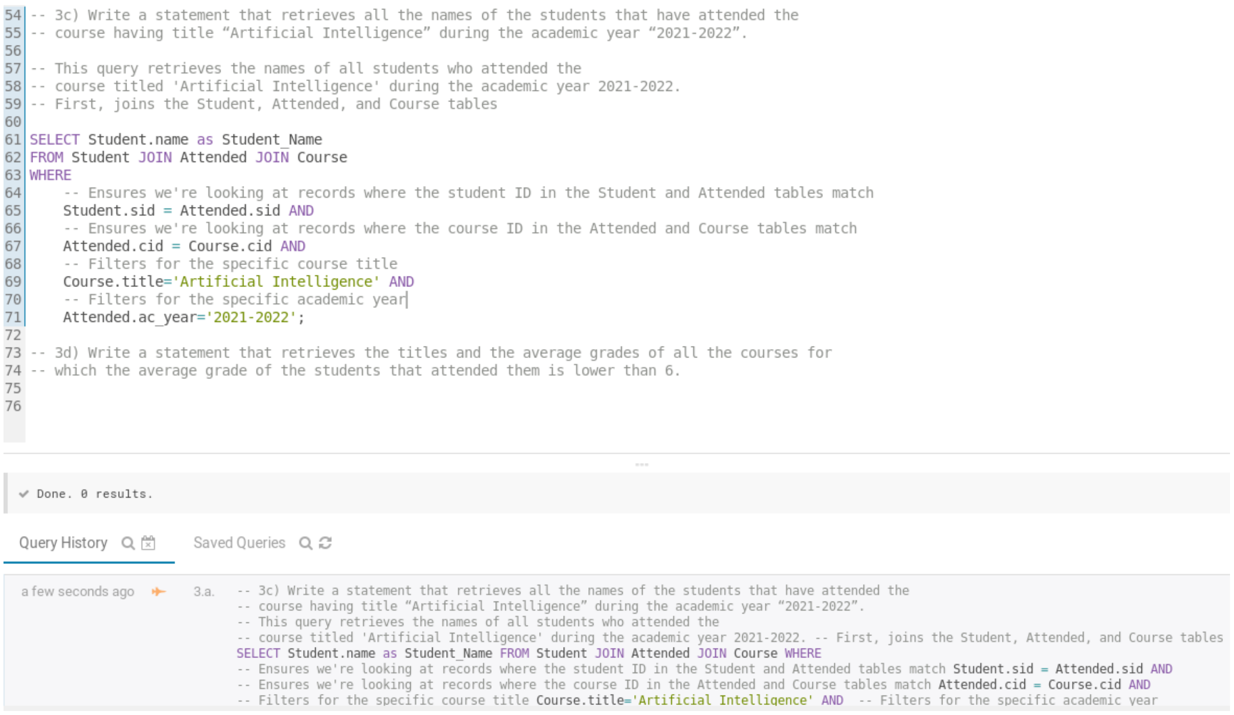
```
--
 3c) Write a statement that retrieves all the names of the students that have attended the
-- course having title “Artificial Intelligence” during the academic year “2021-2022”.

-- This query retrieves the names of all students who attended the
-- course titled 'Artificial Intelligence' during the academic year 2021-2022.
-- First, joins the Student, Attended, and Course tables

SELECT Student.name as Student_Name
FROM Student JOIN Attended JOIN Course
WHERE
    --
 Ensures we're looking at records where the student ID in the Student and Attended tables mat
ch
    Student.sid = Attended.sid AND
    --
 Ensures we're looking at records where the course ID in the Attended and Course tables match

    Attended.cid = Course.cid AND
    -- Filters for the specific course title
    Course.title='Artificial Intelligence' AND
    -- Filters for the specific academic year
    Attended.ac_year='2021-2022';
```

The Task 3c's query methodically filters out the required data. However, it's essential to note that upon execution, the results yielded zero entries. This outcome is due to the current state of our database, which is empty at this stage.

To be more precise codewise, this command is primarily designed to extract the names of students who have enrolled in the "Artificial Intelligence" course during the academic year "2021-2022". Beginning with the ***SELECT*** statement, it aims to retrieve the name column from the Student table, presenting it with the alias "Student_Name" for clarity in the output. The command achieves this by engaging with three tables: Student, Attended, and Course. The ***FROM*** clause highlights Student as the starting table, while the subsequent ***JOIN*** clauses incorporate the Attended and Course tables into the query. The ***WHERE*** clause plays a crucial role, filtering records based on several conditions. Firstly, it ensures a match between the student IDs (sid) in both the Student and Attended tables. This alignment is vital for the accuracy of the join operation.

Secondly, it confirms a correspondence between the course IDs (cid) in the Attended and Course tables. Further refining the selection, the command then focuses on records specifically related to the "Artificial Intelligence" course. Lastly, the condition zeroes in on the desired academic year, "2021-2022", ensuring only relevant records from that period are considered. In essence, this comprehensive SQL command meticulously filters data across three tables to pinpoint students who met the specified criteria.

Following this detailed breakdown, please refer to *Figure 8: Execution of Task 3c's Query in Impala with Result*. This image showcases the successful execution of the query, underlining the aforementioned zero results.



```
-- 3c) Write a statement that retrieves all the names of the students that have attended the
-- course having title “Artificial Intelligence” during the academic year “2021-2022”.

-- This query retrieves the names of all students who attended the
-- course titled 'Artificial Intelligence' during the academic year 2021-2022.
-- First, joins the Student, Attended, and Course tables

SELECT Student.name as Student_Name
FROM Student JOIN Attended JOIN Course
WHERE
    -- Ensures we're looking at records where the student ID in the Student and Attended tables match
    Student.sid = Attended.sid AND
    -- Ensures we're looking at records where the course ID in the Attended and Course tables match
    Attended.cid = Course.cid AND
    -- Filters for the specific course title
    Course.title='Artificial Intelligence' AND
    -- Filters for the specific academic year
    Attended.ac_year='2021-2022';

-- 3d) Write a statement that retrieves the titles and the average grades of all the courses for
-- which the average grade of the students that attended them is lower than 6.
```

```
✔ Done. 0 results.
```

```
-- 3c) Write a statement that retrieves all the names of the students that have attended the
-- course having title “Artificial Intelligence” during the academic year “2021-2022”.
-- This query retrieves the names of all students who attended the
-- course titled 'Artificial Intelligence' during the academic year 2021-2022. -- First, joins the Student, Attended, and Course tables
SELECT Student.name as Student_Name FROM Student JOIN Attended JOIN Course WHERE
-- Ensures we're looking at records where the student ID in the Student and Attended tables match Student.sid = Attended.sid AND
-- Ensures we're looking at records where the course ID in the Attended and Course tables match Attended.cid = Course.cid AND
-- Filters for the specific course title Course.title='Artificial Intelligence' AND  -- Filters for the specific academic year
```

*Figure 8: Execution of Task 3c's query in Impala with Results.*

## 3d. Write a statement that retrieves the titles and the average grades of all the courses for which the average grade of the students that attended them is lower than 6.

For the last task, Task 3d, our objective is to pinpoint courses where the average grade attained by attending students falls *below the threshold of 6*. This necessitates not only an extraction of the course titles but also a computation of the average grades from the database. To achieve this, we'll be engaging with the ***Course*** and ***Attended*** tables, as they collectively house the necessary course details and respective student grades. Through a judiciously crafted SQL query, we'll amalgamate this data, applying filters and calculations to derive courses with an average grade below 6. This task involves utilizing aggregate functions and grouping mechanisms to ensure precise and accurate results, encapsulated by specific conditions to match our criteria.

The Task 3d query's code follows below:

```
-- 3d) Write a statement that retrieves the titles and the average grades of all the courses for
-- which the average grade of the students that attended them is lower than 6.

-- This query retrieves the titles and average grades of all courses for which
```

```sql
-- the average grade of the attending students is lower than 6.

SELECT
    -- Gets the course title
    Course.title as Course_Title,
    -- Calculates the average grade
    avg(Attended.grade) as Average_Grade
FROM
    -- Joins the Course and Attended tables
    Course JOIN Attended
WHERE
    -- Ensures we're looking at records where the course ID in the Course and Attended tables match
    Course.cid = Attended.cid
WHERE
    -- Ensures we're looking at records where the course ID in the Course and Attended tables match
    Course.cid = Attended.cid
GROUP BY
    -- Groups the results by course title
    Course.title
HAVING
    -- Filters for courses where the average grade is less than 6
    avg(Attended.grade)<6;
```

In Task 3d, we embark on a quest to derive course titles and their corresponding average grades, specifically spotlighting those courses where the average grade, as achieved by the students, is notably less than 6. To unravel this, our command is meticulously structured to delve deep into the ***Course*** and ***Attended*** tables. These tables collectively enshrine the pivotal details about each course and the grades students have procured therein.

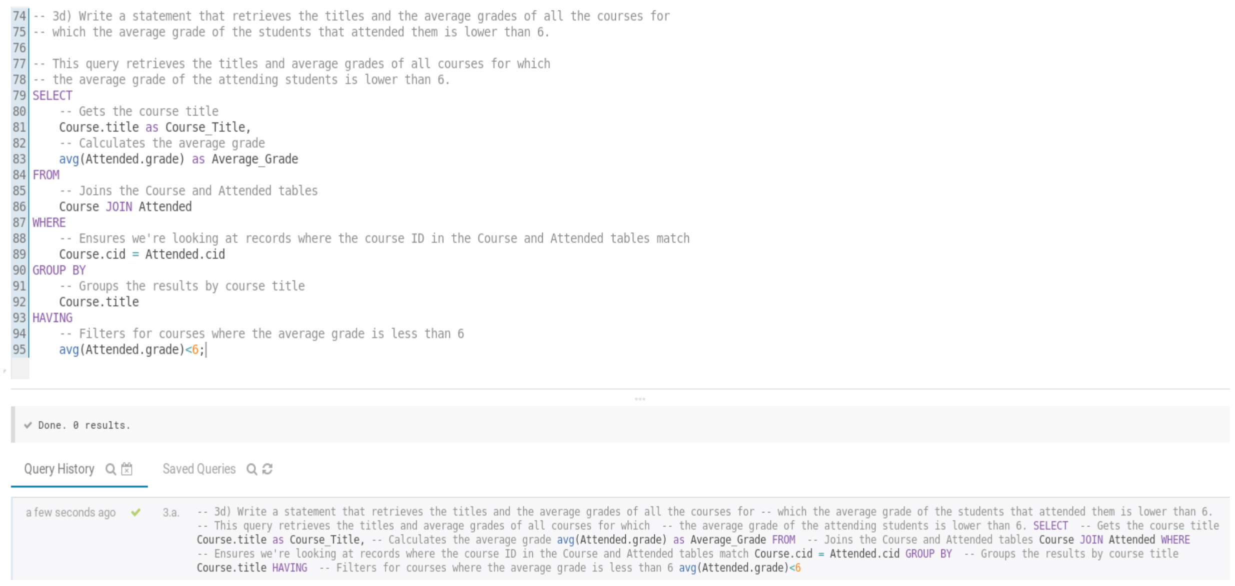
Initiating our exploration with the ***SELECT*** statement, we explicitly call for the title from the ***Course*** table, simultaneously invoking the SQL aggregate function ***AVG()*** to compute the average of the ***grade*** column from the ***Attended*** table. For clarity in our output, this average is christened as *"Average_Grade".*

As we weave our way through the query, the ***FROM*** clause designates the ***Course*** table as our primary point of departure. This is seamlessly integrated with the Attended table using the ***JOIN*** clause, ensuring a harmonious merger based on matching course IDs (**cid**). This joining operation is pivotal, as it facilitates a holistic view of each course alongside its corresponding student grades.

However, our expedition doesn't halt here. The ***WHERE*** clause instills precision, ensuring that we're solely inspecting records where the course IDs in both the ***Course*** and ***Attended*** tables resonate in harmony. Venturing further, the ***GROUP BY*** clause clusters our results by course title, ensuring each course is distinctly represented, preventing any overlap of data.

Then, unlike the traditional ***WHERE*** clause which filters rows, the ***HAVING*** clause filters groups. In our scenario, it meticulously sieves through the aggregated average grades, spotlighting only those groups where this average dips below the 6 mark.

In culmination, through this sophisticated command, we have adeptly navigated the database's depths, emerging with a list of courses and their average grades, with a special emphasis on those averaging less than 6.



```
-- 3d) Write a statement that retrieves the titles and the average grades of all the courses for
-- which the average grade of the students that attended them is lower than 6.

-- This query retrieves the titles and average grades of all courses for which
-- the average grade of the attending students is lower than 6.
SELECT
    -- Gets the course title
    Course.title as Course_Title,
    -- Calculates the average grade
    avg(Attended.grade) as Average_Grade
FROM
    -- Joins the Course and Attended tables
    Course JOIN Attended
WHERE
    -- Ensures we're looking at records where the course ID in the Course and Attended tables match
    Course.cid = Attended.cid
GROUP BY
    -- Groups the results by course title
    Course.title
HAVING
    -- Filters for courses where the average grade is less than 6
    avg(Attended.grade)<6;
```

```
✔ Done. 0 results.
```

```
a few seconds ago ✔ 3.a. -- 3d) Write a statement that retrieves the titles and the average grades of all the courses for -- which the average grade of the students that attended them is lower than 6.
-- This query retrieves the titles and average grades of all courses for which  -- the average grade of the attending students is lower than 6. SELECT  -- Gets the course title
Course.title as Course_Title, -- Calculates the average grade avg(Attended.grade) as Average_Grade FROM  -- Joins the Course and Attended tables Course JOIN Attended WHERE
-- Ensures we're looking at records where the course ID in the Course and Attended tables match Course.cid = Attended.cid GROUP BY  -- Groups the results by course title
Course.title HAVING  -- Filters for courses where the average grade is less than 6 avg(Attended.grade)<6
```

*Figure 9: Execution of Task 3d's query in Impala with Results.*

For a comprehensive examination of the query associated with ***Task 3***, we kindly direct you to the *Appendix* section of the current document.

## Task 4: Query Optimization in Impala

*" A particular query in the previous Impala database is too slow. Describe what you are going to do to investigate what is going wrong and what can be done to improve efficiency. Provide any commands that you are going to run."*

The final task is concerned with improving the performance of a slow-running query in an Impala database. We will describe a systematic approach to diagnose the cause of the inefficiency and propose solutions that can enhance the query's performance. This task emphasizes the importance of efficient data querying in maintaining optimal system performance.

*Answer of Task 4:*

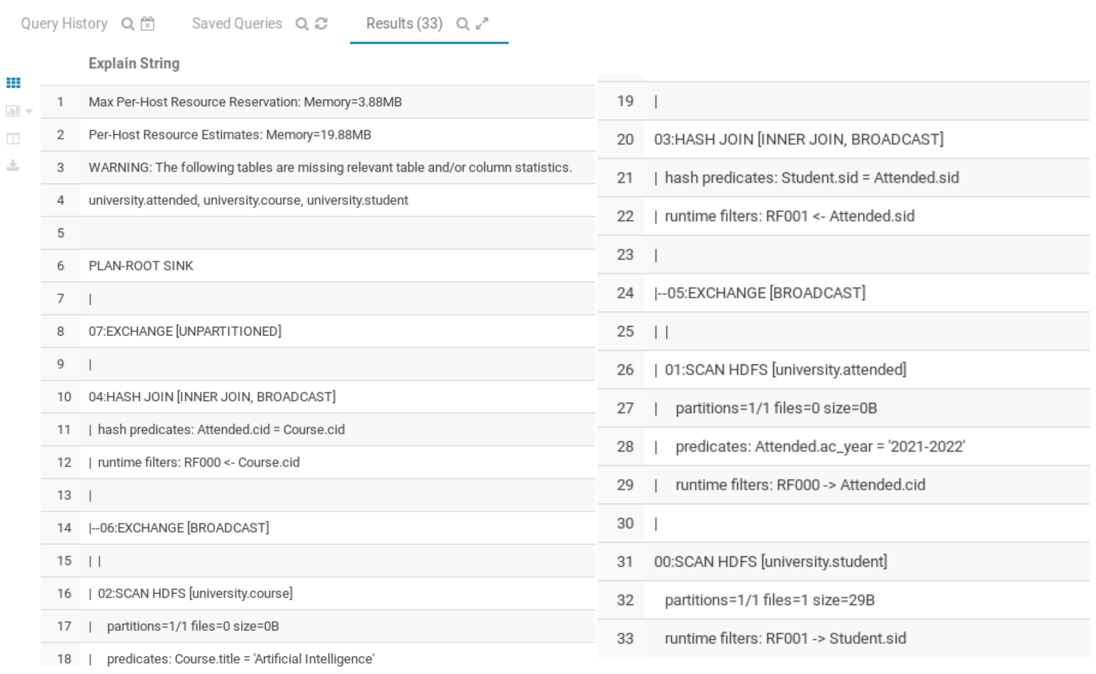
To be more precise in addressing task 4, our primary goal is to diagnose the inefficiencies within a particular query that is exhibiting slow performance in our Impala database. To achieve this, we adopt a multi-pronged approach, applying diagnostic commands and then implementing potential solutions. ***This procedure has been systematically applied to all queries up to this point, in order to achieve our goal.***

Firstly, we utilize the ***EXPLAIN*** command:

```
EXPLAIN SELECT Student.name as Student_Name
FROM Student
JOIN Attended
JOIN Course
WHERE (Student.sid = Attended.sid AND Attended.cid = Course.cid AND
Course.title='Artificial Intelligence' AND Attended.ac_year='2021-2022');
```

This command provides a detailed overview of how Impala plans to execute the given query. *By analyzing the execution plan, we can identify potential bottlenecks such as suboptimal join orders or full table scans that could be contributing to the query's slow performance.*

A demonstrative figure from the results of an execution of the ***Explain*** command follows below (*due to space and time constraints we will stick only to this results of only one query - but in the procedure we execute them for all the queries*):



```
Max Per-Host Resource Reservation: Memory=3.88MB
Per-Host Resource Estimates: Memory=19.88MB
WARNING: The following tables are missing relevant table and/or column statistics.
university.attended, university.course, university.student

PLAN-ROOT SINK
|
07:EXCHANGE [UNPARTITIONED]
|
04:HASH JOIN [INNER JOIN, BROADCAST]
| hash predicates: Attended.cid = Course.cid
| runtime filters: RF000 <- Course.cid
|
|--06:EXCHANGE [BROADCAST]
| |
| 02:SCAN HDFS [university.course]
|    partitions=1/1 files=0 size=0B
|    predicates: Course.title = 'Artificial Intelligence'
|
03:HASH JOIN [INNER JOIN, BROADCAST]
| hash predicates: Student.sid = Attended.sid
| runtime filters: RF001 <- Attended.sid
|
|--05:EXCHANGE [BROADCAST]
| |
| 01:SCAN HDFS [university.attended]
|    partitions=1/1 files=0 size=0B
|    predicates: Attended.ac_year = '2021-2022'
|    runtime filters: RF000 -> Attended.cid
|
00:SCAN HDFS [university.student]
   partitions=1/1 files=1 size=29B
   runtime filters: RF001 -> Student.sid
```

*Figure 10: Results of explain command for demonstrative purposes.*

Next, to gather more granular insights into the runtime behavior of the query, we employ the ***PROFILE*** command:

```sql
SELECT Student.name as Student_Name
FROM Student
JOIN Attended
JOIN Course
WHERE (Student.sid = Attended.sid AND Attended.cid = Course.cid AND
Course.title='Artificial Intelligence' AND Attended.ac_year='2021-2022');
PROFILE;
```

The ***PROFILE*** command offers a comprehensive breakdown of the query's execution, *revealing how much time is spent in each phase. Such insights are invaluable in pinpointing exact stages that might be causing the slowdown.*

For a more concise representation of the query's execution phases, and for more complete research we turn to the ***SUMMARY*** command for each query of Task 3:

```sql
SELECT Student.name as Student_Name
FROM Student
JOIN Attended
JOIN Course
WHERE (Student.sid = Attended.sid AND Attended.cid = Course.cid AND
Course.title='Artificial Intelligence' AND Attended.ac_year='2021-2022');
SUMMARY;
```

The ***SUMMARY*** command provides *a succinct overview of the timings for the different phases of query execution. By contrasting this summary with the detailed profile, we can further narrow down areas requiring optimization.*

Then, since in Impala, the process of query planning relies on statistics to generate efficient execution plans. Without accurate statistics, Impala might make suboptimal decisions, such as choosing an inefficient join order or misallocating memory resources. This can lead to slower query execution times. Therefore, we utilize the ***COMPUTE STATS*** command.

To be more precise the ***COMPUTE STATS*** command gathers statistics on tables, including row counts, data size, and distinct values for columns. *This information allows Impala to better estimate the costs associated with different execution plans and choose the most efficient one.*

```
-- To further optimize, statistics can be collected for the tables involved,
-- aiding Impala in making more informed decisions during query planning.
-- This can potentially lead to more efficient execution plans.
COMPUTE STATS Student;
COMPUTE STATS Attended;
COMPUTE STATS Course;
```

By running the above commands, we are essentially providing Impala *with a deeper understanding of the data distribution and characteristics in the **Student**, **Attended**, and **Course** tables. Once these statistics are computed, subsequent queries on these tables are likely to benefit from more efficient execution plans.*

Furthermore, as data evolves (new data gets inserted, old data gets deleted), it's a good practice to periodically recompute statistics to ensure that they remain accurate and reflective of the current state of the data - *a complete theoretical scenario but we treat this Task as a real world problem.*

Moreover, in the context of Task 4, where we aim to optimize a slow-running query, gathering statistics is a key step. It should be one of the first actions taken when addressing performance issues, especially if statistics haven't been computed recently or if the data has undergone significant changes.

In conclusion, incorporating the ***COMPUTE STATS*** command is not only useful but crucial for ensuring that Impala has the necessary information to optimize query performance efficiently. To sum up, these commands are very useful for Task 4 and should be included as part of the optimization process.

From our observations, *we identified that a significant delay was attributed to the multiple joins in the query*. A proposed solution to mitigate this inefficiency is to restructure our database. By adding a column in the ***Attended*** table to store student names, we can potentially eliminate the need for additional joins:

```
ALTER TABLE Attended ADD COLUMNS (sname STRING);
```

To further streamline our data retrieval, we consider removing the Student table - if it is allowed by our client:

```
drop table Student;
```

Post these alterations, our modified query looks like:

```sql
SELECT Attended.sname as Student_Name
FROM Attended JOIN Course
WHERE (Attended.cid = Course.cid AND
Course.title='Artificial Intelligence' AND Attended.ac_year='2021-2022');
```

In scenarios where **<u>table removal is not feasible from the client's policy</u>** for example, we can resort to the ***STRAIGHT_JOIN*** command to control the order of join operations, ensuring that *<u>Impala does not reorder the join clauses</u>*.

To be more precise, the ***STRAIGHT_JOIN*** command can be invoked to prevent Impala from internally reordering the join clauses. Instead, it depends on the join clauses being sequentially ordered in the query. An illustration of this command is:

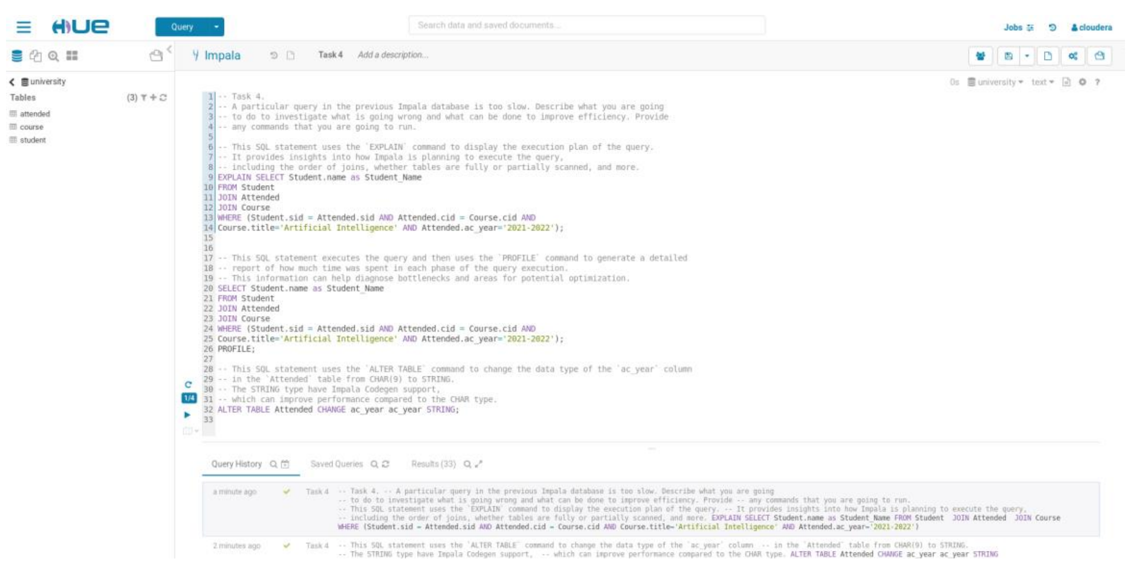
```sql
SELECT STRAIGHT_JOIN Student.name as Student_Name
FROM Student JOIN Attended JOIN Course
WHERE (Student.sid = Attended.sid AND Attended.cid = Course.cid AND
Course.title='Artificial Intelligence' AND Attended.ac_year='2021-2022');
```

In furtherance of our optimization efforts, we advocate for a change in the data type of the ***ac_year*** column in the ***Attended*** table. Transitioning from CHAR(9) to either VARCHAR(9) or STRING is prudent. The rationale behind this recommendation stems from the fact that the CHAR type, in *<u>its current version, lacks the support of Impala Codegen</u>*. *Both VARCHAR and STRING types, on the other hand, enjoy this support, and the performance enhancements from Codegen unequivocally surpass the advantages of a fixed-width CHAR*. The command to execute this modification is:

```sql
-- Modifying the data type of the `ac_year` column in the "Attended" table from
CHAR(9) to VARCHAR(9) or STRING for optimal performance.
ALTER TABLE Attended CHANGE ac_year ac_year STRING;
```

We choose the ***STRING*** variable type, due to the observed performance benefits of ***STRING*** over CHAR in *<u>our version of Impala</u>*.

In conclusion, by adopting this systematic approach of diagnostic commands followed by potential solutions, we aim to ensure the efficient execution of our queries, thereby optimizing the overall performance of our Impala database. A sample image of a part of the above-described procedure follows below:



```sql
-- Task 4.
-- A particular query in the previous Impala database is too slow. Describe what you are going
-- to do to investigate what is going wrong and what can be done to improve efficiency. Provide
-- any commands that you are going to run.

-- This SQL statement uses the `EXPLAIN` command to display the execution plan of the query.
-- It provides insights into how Impala is planning to execute the query,
-- including the order of joins, whether tables are fully or partially scanned, and more.
EXPLAIN SELECT Student.name as Student_Name
FROM Student
JOIN Attended
JOIN Course
WHERE (Student.sid = Attended.sid AND Attended.cid = Course.cid AND
Course.title='Artificial Intelligence' AND Attended.ac_year='2021-2022');


-- This SQL statement executes the query and then uses the `PROFILE` command to generate a detailed
-- report of how much time was spent in each phase of the query execution.
-- This information can help diagnose bottlenecks and areas for potential optimization.
SELECT Student.name as Student_Name
FROM Student
JOIN Attended
JOIN Course
WHERE (Student.sid = Attended.sid AND Attended.cid = Course.cid AND
Course.title='Artificial Intelligence' AND Attended.ac_year='2021-2022');
PROFILE;

-- This SQL statement uses the `ALTER TABLE` command to change the data type of the `ac_year` column
-- in the `Attended` table from CHAR(9) to STRING.
-- The STRING type have Impala Codegen support,
-- which can improve performance compared to the CHAR type.
ALTER TABLE Attended CHANGE ac_year ac_year STRING;
```

```
a minute ago   ✓   Task 4   -- Task 4. -- A particular query in the previous Impala database is too slow. Describe what you are going
                            -- to do to investigate what is going wrong and what can be done to improve efficiency. Provide -- any commands that you are going to run.
                            -- This SQL statement uses the `EXPLAIN` command to display the execution plan of the query. -- It provides insights into how Impala is planning to execute the query,
                            -- including the order of joins, whether tables are fully or partially scanned, and more. EXPLAIN SELECT Student.name as Student_Name FROM Student  JOIN Attended  JOIN Course
                            WHERE (Student.sid = Attended.sid AND Attended.cid = Course.cid AND Course.title='Artificial Intelligence' AND Attended.ac_year='2021-2022')
2 minutes ago  ✓   Task 4   -- This SQL statement uses the `ALTER TABLE` command to change the data type of the `ac_year` column  -- in the `Attended` table from CHAR(9) to STRING.
                            -- The STRING type have Impala Codegen support,  -- which can improve performance compared to the CHAR type. ALTER TABLE Attended CHANGE ac_year ac_year STRING
```

Figure 11: A sample picture demonstrating the procedure followed for the Task 4.

## Conclusion - Task 4

To sum up, in Task 4, our central objective was to pinpoint and remedy inefficiencies within a specific query in our Impala database that exhibited subpar performance. To address this concern, we meticulously formulated a systematic methodology, harmoniously melding diagnostic techniques with actionable solutions. ***It's worth noting that this meticulous approach was not exclusive to a singular query; we consistently applied it across all the queries we tackled in Task 3, ensuring a holistic optimization strategy.***

At the outset, we harnessed the diagnostic power of the ***EXPLAIN*** command. This command offered an intricate glimpse into the intended execution strategy for the query, enabling us to discern potential bottlenecks such as unoptimized join sequences or exhaustive table scans. Such revelations were invaluable in our endeavor to streamline the query's performance.

To delve deeper into the actual runtime dynamics, we turned to the ***PROFILE*** command. This tool, with its granular breakdown, illuminated the exact duration each phase of the query took, furnishing us with a clear picture of where inefficiencies might lurk.

For a more synthesized perspective, we also leveraged the ***SUMMARY*** command. By providing a condensed overview of the various execution phases, it allowed us to juxtapose and contrast this summary against the in-depth profile, focusing our optimization efforts even further.

Through our analysis, it became evident that a primary source of delay was the multiple joins in our query. To mitigate this, we contemplated a structural adjustment to our database. By introducing a new column to store student names directly in the ***Attended*** table, we envisioned a reduction in the need for extraneous joins, potentially hastening our query processes.

Moreover, when table alterations weren't a viable option, we proposed the deployment of the ***STRAIGHT_JOIN command***. This command would ensure the sequence of join operations remained as scripted in the query, preventing any inadvertent internal reordering by Impala.

In addition, we discerned an opportunity to optimize the data type selection for certain columns. Notably, the transition from CHAR(9) to STRING for the ac_year column in the Attended table was considered, based on the benefits the latter type offered in our version of Impala.

In conclusion, through Task 4, we have embarked on a holistic journey of query optimization, blending diagnostic insights with strategic solutions. By adhering to this comprehensive strategy **across all queries from Task 3**, we are poised to elevate the performance of our Impala database, ensuring it operates at its optimal potential.

For a comprehensive examination of the query associated with ***Task 4***, we kindly direct you to the *Appendix* section of the current document.

***Note: For this Task all the evaluation metrics were applied to every query of the Task 3 in order to identify the worst/slowest query and optimize it. We showcase only one query for clarity and proof of work, but we did it for each query in order to find the slowest (the one we showcase) during our research phase.***

## Appendix

In the current Appendix section, we present a concise collection of all the SQL code utilized in ***Tasks 3 and 4*** of our assignment, demonstrating our hands-on interaction with the Apache Impala database system. This section serves as a reference point for the actual code implemented, stripped of extensive contextual or explanatory information. Here, you'll find the commands used to create the database and tables, insert data, retrieve specific information, and diagnose and optimize query performance. While a brief description accompanies each code snippet, detailed discussions and explanations are located in the main body of the document. The Appendix, thus, serves as a 'code snapshot', providing a quick overview of the command structure used in our Impala interactions.

```sql
-- Task 3:
-- 3a) Create the Impala database & the required tables.

-- To replicate the desired scheme - will create issues
-- with 3b and varchar needs to be casted to String.
-- CREATE DATABASE IF NOT EXISTS university;
-- USE university;

-- CREATE TABLE Student(
-- sid INT,
-- name VARCHAR(50)
-- );

-- CREATE TABLE Course (
-- cid INT,
-- title VARCHAR(50),
-- description STRING
-- );

-- CREATE TABLE Attended (
-- sid INT,
-- cid INT,
-- ac_year CHAR(9),
-- grade DECIMAL(3,1)
-- );

-- If a database named "university" exists. If it does not exist,
-- it creates a new database with that name.
CREATE DATABASE IF NOT EXISTS university;

-- This statement is used to select the "university"
-- database for use in subsequent operations.
USE university;

-- This statement creates a new table named "Student" with columns "sid"
-- (an integer representing the student id) and
-- "name" (a string representing the student's name).
CREATE TABLE Student(
sid INT,
name STRING
);

-- This statement creates a new table named "Course" with columns "cid"
-- (an integer representing the course id),
-- "title" (a string representing the course title),
-- and "description" (a string representing the course description).
CREATE TABLE Course (
cid INT,
title STRING,
description STRING
);

-- This statement creates a new table named "Attended" with columns "sid"
-- (an integer representing the student id), "cid" (an integer representing the course id),
--
 "ac_year" (a character string with a fixed length of 9 characters representing the academic
year),
-- and "grade" (a decimal number with a maximum of 3 digits,
-- 1 of which can be after the decimal point, representing the grade).
-- The FOREIGN KEY constraints "foreign1" and "foreign2" are also created
-- to link "sid" and "cid" with the "Student" and "Course" tables respectively.
CREATE TABLE Attended (
sid INT,
cid INT,
ac_year CHAR(9),
grade DECIMAL(3,1)
);

-- Drop the database.
-- DROP DATABASE university CASCADE;
-- 3b) Give an example command that inserts an entry to the Student table (use your
-- own details for that entry) - 2822212 Dimitris Matsanganis.

INSERT INTO Student (sid, name)
VALUES (2822212, "Dimitris Matsanganis");


   --
 3c) Write a statement that retrieves all the names of the students that have attended the
   -- course having title “Artificial Intelligence” during the academic year “2021-2022”.

   -- This query retrieves the names of all students who attended the
   -- course titled 'Artificial Intelligence' during the academic year 2021-2022.
   -- First, joins the Student, Attended, and Course tables

   SELECT Student.name as Student_Name
   FROM Student JOIN Attended JOIN Course
   WHERE
       --
 Ensures we're looking at records where the student ID in the Student and Attended tables mat
ch
       Student.sid = Attended.sid AND
       --
 Ensures we're looking at records where the course ID in the Attended and Course tables match

       Attended.cid = Course.cid AND
       -- Filters for the specific course title
       Course.title='Artificial Intelligence' AND
       -- Filters for the specific academic year
       Attended.ac_year='2021-2022';
   --
 3d) Write a statement that retrieves the titles and the average grades of all the courses fo
r
   -- which the average grade of the students that attended them is lower than 6.

   -- This query retrieves the titles and average grades of all courses for which
   -- the average grade of the attending students is lower than 6.

   SELECT
       -- Gets the course title
       Course.title as Course_Title,
       -- Calculates the average grade
       avg(Attended.grade) as Average_Grade
   FROM
       -- Joins the Course and Attended tables
       Course JOIN Attended
   WHERE
       --
 Ensures we're looking at records where the course ID in the Course and Attended tables match

       Course.cid = Attended.cid
   WHERE
```

```
      --
 Ensures we're looking at records where the course ID in the Course and Attended tables match

      Course.cid = Attended.cid
  GROUP BY
      -- Groups the results by course title
      Course.title
  HAVING
      -- Filters for courses where the average grade is less than 6
      avg(Attended.grade)<6;
```

```
-- Task 4:
--
 A particular query in the previous Impala database is too slow. Describe what you are going

--
 to do to investigate what is going wrong and what can be done to improve efficiency. Provide

-- any commands that you are going to run.

--
 This SQL statement uses the `EXPLAIN` command to display the execution plan of the query.
-- It provides insights into how Impala is planning to execute the query,
-- including the order of joins, whether tables are fully or partially scanned, and more.
EXPLAIN SELECT Student.name as Student_Name
FROM Student
JOIN Attended
JOIN Course
WHERE (Student.sid = Attended.sid AND Attended.cid = Course.cid AND
Course.title='Artificial Intelligence' AND Attended.ac_year='2021-2022');

--
 This SQL statement executes the query and then uses the `PROFILE` command to generate a deta
iled
-- report of how much time was spent in each phase of the query execution.
-- This information can help diagnose bottlenecks and areas for potential optimization.
SELECT Student.name as Student_Name
FROM Student
JOIN Attended
JOIN Course
WHERE (Student.sid = Attended.sid AND Attended.cid = Course.cid AND
Course.title='Artificial Intelligence' AND Attended.ac_year='2021-2022');
PROFILE;

-- This SQL command provides a summarized view of the execution phases for the query,
-- making it easier to pinpoint areas that might be causing delays.
SELECT Student.name as Student_Name
FROM Student
JOIN Attended
JOIN Course
WHERE (Student.sid = Attended.sid AND Attended.cid = Course.cid AND
Course.title='Artificial Intelligence' AND Attended.ac_year='2021-2022');
SUMMARY;

--
 This SQL statement uses the `ALTER TABLE` command to change the data type of the `ac_year` c
olumn
-- in the `Attended` table from CHAR(9) to STRING.
-- The STRING type have Impala Codegen support,
-- which can improve performance compared to the CHAR type.
ALTER TABLE Attended CHANGE ac_year ac_year STRING;

-- To further optimize, statistics can be collected for the tables involved,
-- aiding Impala in making more informed decisions during query planning.
-- This can potentially lead to more efficient execution plans.
COMPUTE STATS Student;
COMPUTE STATS Attended;
COMPUTE STATS Course;

```

```
-- Proposing structural changes to enhance the performance of the query.
-- Introducing a student name (sname) column to the Attended table.
ALTER TABLE Attended ADD COLUMNS (sname STRING);

-- Deleting the Student table to streamline data retrieval.
drop table Student;

-- Modified query post the structural alterations.
SELECT Attended.sname as Student_Name
FROM Attended JOIN Course
WHERE (Attended.cid = Course.cid AND
Course.title='Artificial Intelligence' AND Attended.ac_year='2021-2022');

--
 Deploying the `STRAIGHT_JOIN` command to prevent Impala from reordering the join clauses.
SELECT STRAIGHT_JOIN Student.name as Student_Name
FROM Student JOIN Attended JOIN Course
WHERE (Student.sid = Attended.sid AND Attended.cid = Course.cid AND
Course.title='Artificial Intelligence' AND Attended.ac_year='2021-2022');

-- Modifying the data type of the `ac_year` column in the "Attended" table from CHAR(9)
to VARCHAR(9) or STRING for optimal performance.
ALTER TABLE Attended CHANGE ac_year ac_year STRING;
```