Final Project Report:  
Simple Query Optimizer

By: Christen Ford

CSUID: 2741896

Contents

[Introduction 3](#_Toc26574999)

[Background and Motivation 3](#_Toc26575000)

[Anticipated Risks 4](#_Toc26575001)

[Project Goals 6](#_Toc26575002)

[Design and Implementation 6](#_Toc26575003)

[Assumptions 6](#_Toc26575004)

[Data Dictionary and Schema 7](#_Toc26575005)

[Platform Setup 11](#_Toc26575006)

[System Architecture 11](#_Toc26575007)

[Experimental Results 13](#_Toc26575008)

[Conclusion 14](#_Toc26575009)

[Appendix 15](#_Toc26575010)

[Sample Input 15](#_Toc26575011)

[Source Code 18](#_Toc26575012)

# Introduction

This document describes an attempt, by me, to implement a simple query optimizer. Within this document, I cover the background and motivation for doing so, anticipated risks, and project goals. I then move on to discuss the design and implementation of the optimizer including any assumptions, the data dictionary and schema, platform setup, and system architecture. I then discuss my experimental results from implementing the optimizer and conclude the report. An appendix is provided that contains ancillary information readers may find useful.

## Background and Motivation

Query optimization is extremely important in data management, particularly within the realm of relational databases. Optimization is necessary because Structured Query Language (SQL) is designed to be used not only by database experts but also by users who simply want to retrieve information from a database without needing to be an expert to do so. That said, such users often write queries that are vastly more wasteful than queries written by the experts. These queries take more computation time, utilize more disk space, and require more disk operations to execute. Rather than require that users of a database be experts in SQL, the far better solution is simply to design middle‑man software that optimizes a user queries before passing them to the databases file system handler.

It turns out that quantitative analysis can be employed on queries to perform more intelligent optimization. In this regard, many databases cost‑based optimization is highly speculative and very sensitive to several factors including whether data is sorted and/or indexed, the index type, whether the index is clustered, table size, block size, tuple size, correlation, and many other factors. Optimizers found in databases such as Microsoft SQL Server, Postgres SQL, and Oracle’s MySQL consider a vast number of factors and more importantly they do so in real‑time. The complexity of writing such a program is best left to organizations with the capacity, faculties, and competencies to so. This document seeks to define a simple query optimizer that performs the same functions as an optimizer found in the aforementioned database engines, albeit, with consideration to far less query factors.

## Anticipated Risks

This section details the risks I anticipated from the onset of this project. It additionally deals with mitigation strategies and contingency plans in case stated risks occur. The following table is the risk register for this project:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| RiskID | Description | Probability | Impact | Ranking | Trigger | Contingency Plan |
| R00 | Program abnormally terminates because the user enters a bad path for a query file. | 10% | Medium | Moderate | User enters path for a non‑existent or incorrect query file. | In this case, the optimizer will print an error message and gracefully exit rather than forcefully terminate. |
| R01 | Program abnormally terminates because the input query file contains bad syntax. | 30% | Medium | Moderate | User inputs a bad query file. | In this case, the optimizer will print an error message and gracefully exit rather than forcefully terminate. |
| R02 | Program abnormally terminates because the user enters in a query file containing more subqueries than the recursion limit allows. | 10% | Low | Benign | User inputs a bad query file containing an abnormally large number of subqueries. | In this case, the optimizer will have no choice but to forcefully terminate. Realistically, the query file would have to contain many nested queries to trigger this risk. |
| R03 | Estimated query cost is incorrect because query cost models are incorrect. | 30% | High | Severe | Designer implements incorrect cost models for one or more query operators. | This risk is likely not to happen given that I already have all of the cost models, however, refinements to the cost models that consider sorting and indexing. |
| R04 | Query estimation fails because the optimizer does not consider all possible input queries. | 25% | High | Severe | User enters a query containing formatting or syntax that the optimizer does not understand. | The optimizer is not equipped to handle all possible input queries. Input queries have been restricted based on descriptions given in the Design and Implementation section of this document. |
| R05 | The optimizer does not run because the user does not have all the required libraries installed. | 75% | High | Critical | User runs the optimizer with missing libraries. | I have documented how to install all pre‑requisite libraries needed for the optimizer in the Platform Setup section of this document. |

Risks are ranked based on probability and impact. Probability is self‑explanatory while impact is the risks impact on the success of the project or successful execution of the optimizer. Risks are ranked into one of four categories: Benign, Moderate, Severe, Critical.

## Project Goals

The primary goal of the project is to build a simple query optimizer that takes in one or more query files in a specific JSON syntax alongside optional optimizer parameters and determines the optimal execution plan for each file. The project also lists the following sub‑goals that must be achieved in order to satisfy the primary goal of the project.

1. Design an architecture for the optimizer that makes sense given the scope of the project.
2. Implement complete and correct cost models for each relational algebra and query operator.
3. Implement a permutation algorithm for generating join orders for the join operation.
4. Implement a method for iterating over all possible execution plans and selecting the best per file.
5. Fully test the optimizer with a large variety of query files.

# Design and Implementation

## Assumptions

This sub‑section lists any assumptions regarding this project. This includes assumptions relating to both the design and the associated reference implementation. Design assumptions include:

* Assume that input queries only contain a subset of SQL operators including SELECT, PROJECT, JOIN, and GROUP BY.
* Assume that inputs queries are already rewritten ahead of time. For instance, if a query contains a correlated sub‑query, assume that a rewritten query was already generated from applying a decorrelation operation.
* Assume that the optimizer does not have to optimize queries by rewriting them. The optimizer only must generate all execution plans for a query and select the appropriate one.

While reference implementation assumptions include:

* Assume that disk I/O cost is calculated in milliseconds.
* Assume that input queries are in the correct JSON format as specified in this document.
* Assume that the GROUP BY operator only operates on a single field.

These assumptions are meant to make the implementation for the query optimizer easier. Some of these assumptions were given in class by Dr. Chung while others were assumptions that I made on my own. I did my best on this project with the information available and the reasonable assumptions that I made.

## Data Dictionary and Schema

This sub‑section details the data format and schema required by data input into the optimizer. The optimizer utilizes the JSON file format to represent queries on disk. JSON was chosen as it can be used to naturally represent a tree in a compact and easy to work with syntax. The top‑level objects in the JSON query format are as follows:

There are a few clarifications worth mentioning regarding the diagram above. First, the ‘stats’ and ‘query’ objects above are always the top‑level objects in the queries JSON file. Each underlined keyword is a key string literal mapping the key to the indicated type. The ‘steps’ keyword in the ‘query’ object contains an list of query steps that map to the various ‘join#’, ‘project#’, ‘select#’, ‘aggregate#’, ‘subquery#’ keys. Likewise, each object within both the ‘stats’ and ‘query’ objects are defined here:

In the ‘project\_rates’ and ‘select\_rates’ objects, the ‘default’ key is required. It acts as a fallback key for the optimizer and if it is not found, the optimizer will fail. Each table in the above ‘tables’ object defines the following members:

I have provided examples of the above JSON syntax above in the appendix section of this document under App. 1a and App. 2a. A more formal definition of this syntax is planned for the future, but it will not be included in this document.

## Platform Setup

This sub‑section clarifies the software necessary to run the implementation contained at the end of this document. The reference implementation is written in Python 3.7 and requires the following modules installable via Pythons ‘pip’ script. The reference implementation may always be found at this link: <https://github.com/gollum18/query-optimizer>. The reference implementation requires the following Python modules:

* click
* simplejson

‘click’ is used to implement the command line interface for the reference implementation while ‘simplejson’ is used to provide a more reliable and flexible JSON parsing engine for reading in query files. These packages may be installed separately through pip or may be installed all at once through pip using the included ‘requirements.txt’ file:

pip install -r requirements.txt

python -m pip install -r requirements.txt

If the first command does not work (because you are on a machine where pip is not in the path), use the second command instead as unless pip is not installed at all, it is guaranteed to work.

## System Architecture

The system architecture for the simple query optimizer is well… simple. I intended for the system to be easy to implement and maintain and I believe I achieved that. Now, the optimizer is divided into three primary components:

The first component, the Input Validator deals with parsing query files and determining if they are in the correct format. Design wise, this component was designed to contain verification and validation sub‑components that dealt with ensuring the input format was correct. In actuality, these sub‑components are integrated into the various methods and modules of other the other components.

The second component, the Cost-Based Plan Generator deals with creating all possible execution plans for a given input query it receives from the Input Validator. This component implements cost‑based heuristics for each relational algebra operator. These costs are utilized in conjunction with a permutation algorithm to generate all possible execution plans for a query. The plan with the lowest overall cost is selected as the optimal plan and is passed off to the Output Formatter. During plan generation, the optimizer generates all possible join scenarios on all permutations of all join tables per join operation. Likewise, sub‑query handling is implemented recursively and can handle multiple sub‑queries and correlations per query.

The final component, the Output Formatter is responsible for generating human readable output from execution plans it receives from the Cost‑Based Plan Generator. The Output Formatter is the simplest component and is implemented on a per‑module basis within the source code. The Output Formatter is also responsible for outputting the formatted execution plan to standard output for the user to see. It would be trivial to extend the optimizers implementation to include a flag that enables writing the formatted execution plan to disk as well.

These three components were designed to be modular but are implemented within a single module in the attached source code. If I had more time for the project, I would have properly separated each module as I intended. I was so focused on making sure that the cost estimation was correct that I lost track of ensuring I was adhering to my original design. The optimizer is written in a manner that I assume to be correct, however, I cannot be certain. To note, there are a few issues with the attached implementation:

1. Not every join method considers indexing. I was unsure where it would fit in and did not completely implement it. Currently, only the tuple‑nested join cost method fully supports sorted and indexed data sets. The other join methods consider sorting when necessary such as the sort‑merge join cost method.
2. I am unsure what units the resultant I/O cost metric is in. I know that the formula for converting from the I/O cost metric is simply “time = cost \* (avg latency + avg seek time)”, but I do not know what units “time” should have. The implementation assumes that since the time elements in the prior formula are in milliseconds that the cost metric is in milliseconds. In the implementation, I divide both the avg latency and avg seek time by 1000 to make the resultant unit be seconds.
3. If you look at the outputs for the example query files in the Experimental Results section, you will see that there is a huge discrepancy between the execution time of the first query and the execution time of the second query. I am aware that this should be the case, but I am still skeptical of the results I received as they seem too far apart. I am unsure why this would be as I implemented the cost metrics directly from the slides presented in class.

# Experimental Results

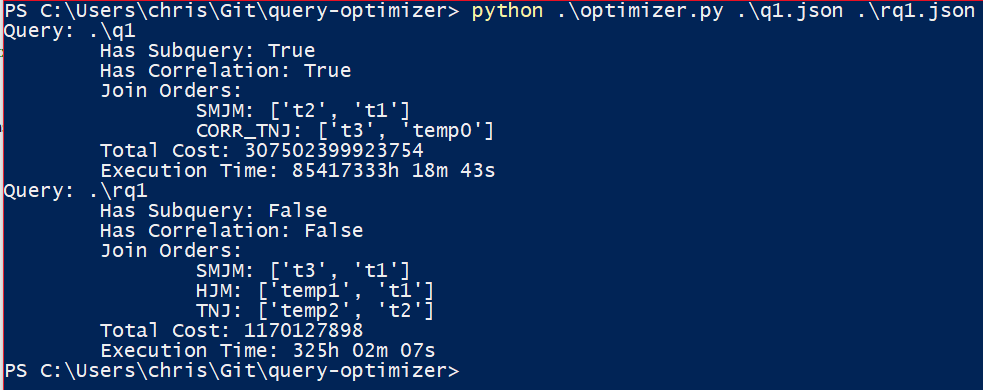
This section details experimental results generated from running the two queries specified by App. 1a/1b and App. 2a/2b against the optimizer. The results presented here were roughly what I expected.

Running the Optimizer

Once the prerequisites are installed, you can run the optimizer from the command line by invoking the following command:

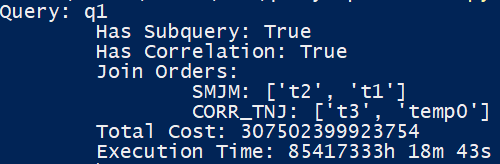
python optimizer.py [query file 1] … [query file n]

The optimizer will attempt to evaluate each query. If it fails it will print an error message and move onto the next.



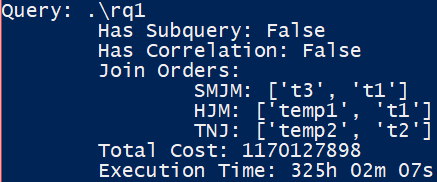
Optionally, you can specify the block size and page size using the ‑‑bs|‑blocksize and ‑‑ps|‑pagesize switches.

Output for Query 1



As can be seen, this query generates an enormous runtime. This is because the query is correlated, as indicated by the second join operation of a correlated tuple‑nested join. Total cost is internally calculated in terms of block I/O but is represented in milliseconds starting from 0.

Output for Rewritten Query 1



The output from this query is much smaller than the output observed from the prior query. This query is semantically the same as the prior query, but it has been decorrelated so that it no longer possesses a correlated sub‑query. The resultant runtime is vastly improved upon with regards to the prior query.

# Conclusion

In this report I detailed the background and motivation needed to implement a cost‑based query optimizer. I then defined the data scheme and format required of input fed into the optimizer. Afterwards, I detailed the implementation of the optimizer and explained results from I received based on two sample queries. Finally, I included relevant documents in the appendix such as query source code and optimizer source code. I greatly enjoyed this semester and learned a great deal regarding advanced database management and implementation. Thanks Dr. Chung, have a great break!

# Appendix

The goal of this appendix is to provide supplementary information to the reader. This includes information such as sample input queries and data files as well as source code. Information contained in the appendix is prefixed in order of appearance under the following scheme:

App. #:

Where # is the corresponding appendix entry in order of appearance. This scheme is used in the above documentation to directly refer to appendix entries.

## Sample Input

This section contains samples both in the form of SQL queries and their corresponding JSON format as described in the Design → Data Dictionary and Design → Data Schema sections found earlier in this document.

App. 1a: JSON File for Query 1

{  
 **"stats"**: {  
 **"avg\_seek\_time"**: 8,  
 **"avg\_latency"**: 4,  
 **"select\_rates"**: {  
 **"default"**: 0.15,  
 **"T2.X2=T2.X3"**: 0.10,  
 **"T1.X1=T3.X3"**: 0.20  
 },  
 **"project\_rates"**: {  
 **"default"**: 0.70  
 },  
 **"tables"**: {  
 **"t1"**: {  
 **"num\_pages"**: 1000,  
 **"tuple\_size"**: 20,  
 **"index\_type"**: **"none"**,  
 **"clustered"**: **false**,  
 **"clustering\_factor"**: 0,  
 **"primary\_index"**: **false**,  
 **"sorted"**: **false** },  
 **"t2"**: {  
 **"num\_pages"**: 500,  
 **"tuple\_size"**: 40,  
 **"index\_type"**: **"none"**,  
 **"clustered"**: **false**,  
 **"clustering\_factor"**: 0,  
 **"primary\_index"**: **false**,  
 **"sorted"**: **false** },  
 **"t3"**: {  
 **"num\_pages"**: 2000,  
 **"tuple\_size"**: 100,  
 **"index\_type"**: **"none"**,  
 **"clustered"**: **false**,  
 **"clustering\_factor"**: 0,  
 **"primary\_index"**: **false**,  
 **"sorted"**: **false** }  
 }  
 },  
 **"query"**: {  
 **"has\_subquery"**: **true**,  
 **"is\_correlated"**: **true**,  
 **"steps"**: [**"join0"**, **"subquery0"**, **"aggregate0"**],  
 **"join0"**: {  
 **"tables"**: [**"t1"**, **"t2"**],  
 **"select\_rates"**: [**"default"**],  
 **"result"**: **"temp0"** },  
 **"subquery0"**: {  
 **"is\_correlated"**: **true**,  
 **"table"**: **"temp3"**,  
 **"steps"**: [**"join0"**, **"project0"**, **"aggregate0"**],  
 **"join0"**: {  
 **"tables"**: [**"temp0"**, **"t3"**],  
 **"select\_rates"**: [**"T2.X2=T2.X3"**, **"T1.X1=T3.X3"**],  
 **"result"**: **"temp1"** },  
 **"project0"**: {  
 **"table"**: **"temp1"**,  
 **"fields"**: [**"\*"**],  
 **"project\_rates"**: [**"default"**],  
 **"result"**: **"temp2"** },  
 **"aggregate0"**: {  
 **"table"**: **"temp2"**,  
 **"groupby"**: **true**,  
 **"result"**: **"temp3"** },  
 **"result"**: **"temp4"** },  
 **"aggregate0"**: {  
 **"table"**: **"temp4"**,  
 **"groupby"**: **false**,  
 **"result"**: **"temp5"** }  
 }  
}

App. 2a: JSON File for Rewritten Query 1

{  
 **"stats"**: {  
 **"avg\_seek\_time"**: 8,  
 **"avg\_latency"**: 4,  
 **"select\_rates"**: {  
 **"default"**: 0.15,  
 **"T2.X2=T2.X3"**: 0.10,  
 **"T1.X1=T3.X3"**: 0.20  
 },  
 **"project\_rates"**: {  
 **"default"**: 0.70  
 },  
 **"tables"**: {  
 **"t1"**: {  
 **"num\_pages"**: 1000,  
 **"tuple\_size"**: 20,  
 **"index\_type"**: **"none"**,  
 **"clustered"**: **false**,  
 **"clustering\_factor"**: 0,  
 **"primary\_index"**: **false**,  
 **"sorted"**: **false** },  
 **"t2"**: {  
 **"num\_pages"**: 500,  
 **"tuple\_size"**: 40,  
 **"index\_type"**: **"none"**,  
 **"clustered"**: **false**,  
 **"clustering\_factor"**: 0,  
 **"primary\_index"**: **false**,  
 **"sorted"**: **false** },  
 **"t3"**: {  
 **"num\_pages"**: 2000,  
 **"tuple\_size"**: 100,  
 **"index\_type"**: **""**,  
 **"clustered"**: **false**,  
 **"clustering\_factor"**: 0,  
 **"primary\_index"**: **false**,  
 **"sorted"**: **false** }  
 }  
 },  
 **"query"**: {  
 **"has\_subquery"**: **false**,  
 **"is\_correlated"**: **false**,  
 **"steps"**: [**"join0"**, **"aggregate0"**, **"join1"**, **"join2"**, **"project0"**, **"aggregate1"**],  
 **"join0"**: {  
 **"tables"**: [**"t1"**, **"t3"**],  
 **"select\_rates"**: [**"T1.X1=T3.X3"**],  
 **"result"**: **"temp0"** },  
 **"aggregate0"**: {  
 **"table"**: **"temp0"**,  
 **"groupby"**: **true**,  
 **"result"**: **"temp1"** },  
 **"join1"**: {  
 **"tables"**: [**"t1"**, **"temp1"**],  
 **"select\_rates"**: [**"default"**],  
 **"result"**: **"temp2"** },  
 **"join2"**: {  
 **"tables"**: [**"t2"**, **"temp2"**],  
 **"select\_rates"**: [**"T2.X2=T2.X3"**],  
 **"result"**: **"temp3"** },  
 **"project0"**: {  
 **"table"**: **"temp3"**,  
 **"fields"**: [**"\*"**],  
 **"project\_rates"**: [**"default"**],  
 **"result"**: **"temp4"** },  
 **"aggregate1"**: {  
 **"table"**: **"temp3"**,  
 **"groupby"**: **true**,  
 **"result"**: **"temp5"** }  
 }  
}

## Source Code

This section contains the full and final source code for the query optimizer as described throughout this document. Each method contained in the optimizer is documented via the Python docstring syntax. Assumptions regarding this project are listed under a comment block near the top of the source code.

App. 3: Source Code for ‘optimizer.py’

*# Name: optimizer.py  
# Since: 11/12/2019  
# Author: Christen Ford  
# Purpose: Implements a simple query optimizer that evaluates SQL-like queries  
# passed in via a conformant JSON format.  
  
# Assumptions:  
# 1.) Disk I/O is converted to milliseconds that is then converted to a  
# timestamp. There was no unit on the result of converting disk I/O  
# using the formula (disk I/O \* (avg\_seek\_time + avg\_latency)).  
# 2.) As long as the JSON schema is adhered to, this implementation allows for  
# evaluating arbitrarily nested queries. My initial implementation only handled  
# a single nested query.  
# 3.) The result of an aggregate operation are currently not calculated correctly.  
# Currently aggregate operations do not correctly determine the block size and  
# tuple size of the resultant table. This should not matter in the long run as the  
# most costly operation in relational algebra is the join operation which tends  
# to dominate the other relational algebra operations.***import** click  
**import** math, os, sys  
**import** simplejson  
  
  
**def** cost\_to\_time(cost, avg\_seek\_time=8, avg\_latency=4):  
 *"""  
 Converts a disk I/O metric to a milliseconds.* **:param** *cost: The disk I/O metric to convert.* **:param** *avg\_seek\_time: The average seek time in milliseconds.* **:param** *avg\_latency: The average latency in milliseconds.* **:return***: A disk I/O in milliseconds.  
 """* **return** int(  
 math.ceil(  
 cost \*  
 ((avg\_seek\_time/1000) +  
 (avg\_latency/1000))  
 )  
 )  
  
  
**def** create\_table(  
 name, num\_pages, tuple\_size, index\_type=**"none"**, is\_sorted=**False**,  
 clustered=**False**, clustering\_factor=0, primary\_index=**False**):  
 *"""  
 Creates a table (as a dict) and returns it as a pair where the first  
 element in the pair is the table name and the second element is the  
 table itself.* **:param** *name: The name of the table.* **:param** *num\_pages: The number of pages in the table.* **:param** *tuple\_size: The tuple size of each record in the table.* **:param** *index\_type: The index type, one of {'bpt', 'hash', 'none'}.* **:param** *is\_sorted: Whether the table is sorted.* **:param** *clustered: Whether the table index is clustered.* **:param** *clustering\_factor: The average number of records per cluster.* **:param** *primary\_index: Whether the index is built on a primary key or  
 not.* **:return***: A pair (name, table) where name is the table name and table  
 is a Python dict containing the following mapping:  
 'num\_pages': int  
 'tuple\_size': int  
 'index\_type': str  
 'sorted': boolean  
 'clustered': boolean  
 'clustering\_factor': int  
 'primary\_index': boolean  
 """* **return** (  
 name, {  
 **'num\_pages'**: num\_pages,  
 **'tuple\_size'**: tuple\_size,  
 **'index\_type'**: index\_type,  
 **'sorted'**: is\_sorted,  
 **'clustered'**: clustered,  
 **'clustering\_factor'**: clustering\_factor,  
 **'primary\_index'**: primary\_index  
 }  
 )  
  
  
**def** create\_timestamp(total\_cost):  
 *"""  
 Creates a timestamp from a cost metric, average seek time and average latency.* **:param** *total\_cost: A I/O cost metric in milliseconds.* **:return***: A formatted string timestamp in the form "##h ##m ##s"  
 """* **if not** total\_cost:  
 **raise** ValueError  
 **if** total\_cost == 0:  
 **return** 0, 0, 0  
 milli = total\_cost  
 *# 3600000 milliseconds in an hour* hours = milli // 3600000  
 milli = milli - (3600000 \* hours)  
 *# 60000 milliseconds in a minute* minutes = milli // 60000  
 milli = milli - (60000 \* minutes)  
 *# 1000 milliseconds in a second* seconds = milli // 1000  
 **return "{:02d}h {:02d}m {:02d}s"**.format(hours, minutes, seconds)  
  
  
**def** permute(items):  
 *"""  
 Permutes a list of items.* **:param** *items: A list of items to permute.* **:return***: A generator that yields permutations of 'items' if  
 'items' is not empty, otherwise items.  
 """* **def** \_permute(\_items, \_k):  
 *"""  
 Internal permutation method that implements Heaps' algorithm for  
 efficiently generating permutations. Permutations are performed in  
 place and yielded as such. Yielded lists should not be modified and  
 should be treated as read-only.* **:param** *\_items: A list of items to permute.* **:param** *\_k: The permutation index used to control Heaps' algorithm.* **:return***: A generator that yields permuted lists of items.  
 """* **if** \_k == 1:  
 **yield** \_items  
 **else**:  
 **for** i **in** range(\_k):  
 **yield from** \_permute(\_items, \_k-1)  
 **if** i < \_k - 1:  
 **if** \_k % 2 == 0:  
 \_items[i], \_items[\_k-1] = \_items[\_k-1], \_items[i]  
 **else**:  
 \_items[0], \_items[\_k-1] = \_items[\_k-1], \_items[0]  
  
 **if not** items:  
 **return** items  
  
 **yield from** \_permute(items, len(items))  
  
  
**def** read\_query\_file(filename):  
 *"""  
 Parses a JSON query file and returns the contained JSON as a Python  
 dictionary.* **:param** *filename: The path to the query file.* **:return***: A Python dictionary containing contents of the query file.  
 Mappings are identical to what is described in the project report  
 under the Design section.  
 """* **try**:  
 **if not** os.path.exists(filename):  
 **raise** OSError  
 **with** open(filename) **as** f:  
 qdict = simplejson.load(f)  
 **if not** isinstance(qdict, dict):  
 **raise** OSError  
 **return** qdict  
 **except** OSError:  
 **raise** OSError  
  
  
**class** QueryPlan(object):  
  
 **def** \_\_init\_\_(self, query\_name, join\_methods, join\_orders,  
 total\_cost, timestamp, has\_subquery=**False**,  
 is\_correlated=**False**):  
 *"""  
 Returns a QueryPlan object.* **:param** *query\_name: The name of the query.* **:param** *join\_methods: A list containing join methods.* **:param** *join\_orders: A list containing lists of join orders.* **:param** *total\_cost: The total cost of the query in estimated I/O.* **:param** *timestamp: The timestamp generated via the create\_timestamp  
 method.* **:param** *has\_subquery: Whether the query has a subquery or not.* **:param** *is\_correlated: Whether the query contains a correlated  
 subquery.  
 """* **if not** (isinstance(join\_methods, list) **and** isinstance(join\_orders, list)):  
 **raise** ValueError  
 **if** len(join\_methods) != len(join\_orders):  
 **raise** ValueError  
 self.query\_name = query\_name  
 self.has\_subquery = has\_subquery  
 self.is\_correlated = is\_correlated  
 self.join\_methods = join\_methods  
 self.join\_orders = join\_orders  
 self.total\_cost = total\_cost  
 self.timestamp = timestamp  
  
 **def** \_\_str\_\_(self):  
 *"""  
 Returns a string representation of the query plan.* **:return***: A string representing the query plan.  
 """* name = **"Query: {}\n\t"**.format(self.query\_name)  
 subquery = **"Has Subquery: {}\n\t"**.format(self.has\_subquery)  
 correlated = **"Has Correlation: {}\n\t"**.format(self.is\_correlated)  
 join\_str = **"Join Orders:\n\t\t"  
 for** i **in** range(len(self.join\_orders)):  
 **if** i == len(self.join\_orders) - 1:  
 join\_str += **"{}: {}\n\t"**.format(self.join\_methods[i], self.join\_orders[i])  
 **else**:  
 join\_str += **"{}: {}\n\t\t"**.format(self.join\_methods[i], self.join\_orders[i])  
 total\_cost = **"Total Cost: {}\n\t"**.format(self.total\_cost)  
 exec\_time = **"Execution Time: {}"**.format(self.timestamp)  
 **return** name + subquery + correlated + join\_str + total\_cost + exec\_time  
  
  
**class** QueryOptimizer(object):  
  
 **def** \_\_init\_\_(self, page\_size=4096, block\_size=100):  
 *"""  
 Returns a new instance of a QueryOptimizer object.* **:param** *page\_size: The page size (in bytes).* **:param** *block\_size: The block size (number of records per block).  
 """* self.page\_size = page\_size  
 self.block\_size = block\_size  
 self.join\_scenarios = dict()  
  
 **def** add\_join\_scenario(self, key, join\_function, args):  
 *"""  
 Adds a join scenario to the optimizer.* **:param** *key: The key for the join scenario -> uniquely identifies scenario.* **:param** *join\_function: The join calculation function to call.* **:param** *args: A list of arguments to pass to join\_function.  
 """* self.join\_scenarios[key] = (join\_function, args)  
  
 **def** clear\_join\_scenarios(self):  
 *"""  
 Removes all join scenarios from the optimizer.  
 """* self.join\_scenarios.clear()  
  
 **def** delete\_join\_scenario(self, key):  
 *"""  
 Removes a join scenario from the optimizer.* **:param** *key: The key of the join scenario to remove.  
 """* **if** key **in** self.join\_scenarios:  
 **del** self.join\_scenarios[key]  
  
 **def** yield\_join\_scenarios(self):  
 *"""  
 Yields join scenarios from the query optimizer.* **:return***: A generator containing join scenarios.  
 """* **yield from** self.join\_scenarios.items()  
  
 @staticmethod  
 **def** get\_matching\_cost(index\_type, primary, clustered, clustering\_factor):  
 *"""  
 Gets the cost of finding a matching keys in the index.* **:param** *index\_type: The index type, one of {"bpt", "hash", "none"}.* **:param** *primary: Whether the index is a primary index or not.* **:param** *clustered: Whether the index is clustered or not.* **:param** *clustering\_factor: The average number of records per cluster.* **:return***: The total cost of finding a matching key in the index.  
 """* **if not** primary:  
 **if** clustered:  
 clustered\_cost = 1  
 **else**:  
 clustered\_cost = clustering\_factor  
 **else**:  
 clustered\_cost = 0  
 **if** index\_type == **'bpt'**:  
 index\_cost = 4  
 **elif** index\_type == **'hash'**:  
 index\_cost = 1.2  
 **elif** index\_type == **"none"**:  
 index\_cost = 0  
 **else**:  
 **raise** ValueError  
 **return** index\_cost + clustered\_cost + 1  
  
 **def** tuples\_per\_page(self, tuple\_size):  
 *"""  
 Determines the total number of tuples per page.* **:param** *tuple\_size: The tuple size.* **:return***: The number of tuples per page.  
 """* **if** tuple\_size == 0:  
 **return** 0  
 **return** int(math.ceil(self.page\_size / tuple\_size))  
  
 @staticmethod  
 **def** calc\_cartesian\_product(stats, left\_table\_name, right\_table\_name):  
 *"""  
 Determines the total I/O required of a cartesian product between two  
 tables. This method simulates a join operation without any conditions.* **:param** *stats: A dictionary containing table statistics.* **:param** *left\_table\_name: The name of the left table.* **:param** *right\_table\_name: The name of the right table.* **:return***: The total I/O required for a cartesian product operation.  
 """* **try**:  
 tables = stats[**'tables'**]  
 left\_table = tables[left\_table\_name]  
 right\_table = tables[right\_table\_name]  
 left\_num\_pages = left\_table[**'num\_pages'**]  
 right\_num\_pages = right\_table[**'num\_pages'**]  
 **except** KeyError:  
 print(**'KeyError: Table \'{}\' or \'{}\' not found! Cannot continue...'**.format(  
 left\_table\_name,  
 right\_table\_name)  
 )  
 sys.exit(-1)  
 **return** cost\_to\_time(left\_num\_pages \* right\_num\_pages)  
  
 **def** calc\_tuple\_nested\_join\_cost(self, stats, left\_table\_name, right\_table\_name):  
 *"""  
 Determines the total cost to perform a tuple-nested join operation.* **:param** *stats: A dictionary containing table statistics.* **:param** *left\_table\_name: The left table name.* **:param** *right\_table\_name: The right table name.* **:return***: The total I/O required to perform a tuple-nested join operation.  
 """* **try**:  
 tables = stats[**'tables'**]  
 left\_table = tables[left\_table\_name]  
 right\_table = tables[right\_table\_name]  
 left\_num\_pages = left\_table[**'num\_pages'**]  
 left\_tuple\_size = left\_table[**'tuple\_size'**]  
 right\_num\_pages = right\_table[**'num\_pages'**]  
 right\_tuple\_size = right\_table[**'tuple\_size'**]  
 **except** KeyError:  
 print(**'KeyError: Table \'{}\' or \'{}\' not found! Cannot continue...'**.format(  
 left\_table\_name,  
 right\_table\_name)  
 )  
 sys.exit(-1)  
 **try**:  
 left\_clustered, right\_clustered = left\_table[**'clustered'**], right\_table[**'clustered'**]  
 left\_clustering\_factor, right\_clustering\_factor = (  
 left\_table[**'clustering\_factor'**],  
 right\_table[**'clustering\_factor'**]  
 )  
 left\_index\_type, right\_index\_type = left\_table[**'index\_type'**], right\_table[**'index\_type'**]  
 left\_indexed, right\_indexed = left\_table[**'indexed'**], right\_table[**'indexed'**]  
 left\_primary, right\_primary = left\_table[**'primary\_index'**], right\_table[**'primary\_index'**]  
 **except** KeyError:  
 left\_clustered, right\_clustered = **False**, **False** left\_clustering\_factor, right\_clustering\_factor = 0, 0  
 left\_index\_type, right\_index\_type = **None**, **None** left\_indexed, right\_indexed = **False**, **False** left\_primary, right\_primary = **False**, **False  
 if** left\_indexed:  
 left\_tuples\_per\_page = self.tuples\_per\_page(left\_tuple\_size)  
 matching\_right\_cost = QueryOptimizer.get\_matching\_cost(  
 right\_index\_type,  
 right\_primary,  
 right\_clustered,  
 right\_clustering\_factor  
 )  
 cost = int(  
 math.ceil(  
 left\_tuple\_size + ((left\_tuple\_size \* left\_tuples\_per\_page) \* matching\_right\_cost)  
 )  
 )  
 **elif** right\_indexed:  
 right\_tuples\_per\_page = self.tuples\_per\_page(right\_tuple\_size)  
 matching\_left\_cost = QueryOptimizer.get\_matching\_cost(  
 left\_index\_type,  
 left\_primary,  
 left\_clustered,  
 left\_clustering\_factor  
 )  
 cost = int(  
 math.ceil(  
 right\_tuple\_size + ((right\_tuple\_size \* right\_tuples\_per\_page) \* matching\_left\_cost)  
 )  
 )  
 **else**:  
 left\_tuples\_per\_page = self.tuples\_per\_page(left\_tuple\_size)  
 cost = int(  
 math.ceil(  
 left\_num\_pages + left\_tuples\_per\_page \* left\_tuple\_size \* right\_num\_pages  
 )  
 )  
 **return** cost  
  
 @staticmethod  
 **def** calc\_page\_nested\_join\_cost(stats, left\_table\_name, right\_table\_name):  
 *"""  
 Determines the total cost to perform a page nested join operation.* **:param** *stats: A dictionary containing table statistics.* **:param** *left\_table\_name: The left table name.* **:param** *right\_table\_name: The right table name.* **:return***: The total I/O cost required to perform a page-nested join operation.  
 """* **try**:  
 tables = stats[**'tables'**]  
 left\_table = tables[left\_table\_name]  
 right\_table = tables[right\_table\_name]  
 left\_num\_pages = left\_table[**'num\_pages'**]  
 right\_num\_pages = right\_table[**'num\_pages'**]  
 **except** KeyError:  
 print(**'KeyError: Table \'{}\' or \'{}\' not found! Cannot continue...'**.format(  
 left\_table\_name,  
 right\_table\_name)  
 )  
 sys.exit(-1)  
 cost = int(  
 math.ceil(  
 left\_num\_pages + left\_num\_pages \* right\_num\_pages  
 )  
 )  
 **return** cost  
  
 @staticmethod  
 **def** calc\_block\_nested\_join\_cost(stats, outer\_table\_name, inner\_table\_name, block\_size):  
 *"""  
 Determines the total cost to perform a block-nested join operation.* **:param** *stats: A dictionary containing table statistics.* **:param** *outer\_table\_name: The name of the outer table.* **:param** *inner\_table\_name: The name of the inner table.* **:param** *block\_size: The block size for block-nested join.* **:return***: The total I/O needed to perform a block-nested join operation.  
 """* **try**:  
 tables = stats[**'tables'**]  
 outer\_table = tables[outer\_table\_name]  
 inner\_table = tables[inner\_table\_name]  
 outer\_num\_pages = outer\_table[**'num\_pages'**]  
 inner\_num\_pages = inner\_table[**'num\_pages'**]  
 **except** KeyError:  
 print(**'KeyError: Table \'{}\' or \'{}\' not found! Cannot continue...'**.format(  
 outer\_table\_name,  
 inner\_table\_name)  
 )  
 sys.exit(-1)  
 outer\_blocks = int(  
 math.ceil(  
 outer\_num\_pages / block\_size  
 )  
 )  
 cost = outer\_num\_pages + (outer\_blocks \* inner\_num\_pages)  
 **return** cost  
  
 @staticmethod  
 **def** calc\_sort\_merge\_join\_cost(stats, outer\_table\_name, inner\_table\_name, block\_size):  
 *"""  
 Calculates the cost in I/O as time for a sort merge join.* **:param** *stats: A dict containing the table statistics.* **:param** *outer\_table\_name: The outer table name.* **:param** *inner\_table\_name: The inner table name.* **:param** *block\_size: The block size.* **:return***: Sort merge disk I/O cost as time.  
 """* **try**:  
 tables = stats[**'tables'**]  
 outer\_table = tables[outer\_table\_name]  
 inner\_table = tables[inner\_table\_name]  
 outer\_num\_pages = outer\_table[**'num\_pages'**]  
 inner\_num\_pages = inner\_table[**'num\_pages'**]  
 outer\_table\_sorted, inner\_table\_sorted = outer\_table[**'sorted'**], inner\_table[**'sorted'**]  
 **except** KeyError:  
 print(**'KeyError: Table \'{}\' or \'{}\' not found! Cannot continue...'**.format(  
 outer\_table\_name,  
 inner\_table\_name)  
 )  
 sys.exit(-1)  
 max\_num\_pages = max(outer\_num\_pages, inner\_num\_pages)  
 **if** outer\_table\_sorted **and** inner\_table\_sorted:  
 cost = outer\_num\_pages + inner\_num\_pages  
 **elif** block\_size > math.sqrt(max\_num\_pages):  
 cost = 3 \* (outer\_num\_pages + inner\_num\_pages)  
 **else**:  
 cost = int(  
 2 \* (outer\_num\_pages + inner\_num\_pages) \* (  
 1 + math.ceil(  
 math.log(  
 outer\_num\_pages/block\_size,  
 block\_size-1  
 )  
 )  
 )  
 )  
 **return** cost  
  
 @staticmethod  
 **def** calc\_hash\_join\_cost(stats, outer\_table\_name, inner\_table\_name, block\_size):  
 *"""  
 Calculates the has join disk I/O cost as time.* **:param** *stats: A dict containing table stats.* **:param** *outer\_table\_name: The outer table name.* **:param** *inner\_table\_name: The inner table name.* **:param** *block\_size: The block size.* **:return***: Hash join disk I/O cost as time.  
 """* **try**:  
 tables = stats[**'tables'**]  
 outer\_table = tables[outer\_table\_name]  
 inner\_table = tables[inner\_table\_name]  
 outer\_num\_pages = outer\_table[**'num\_pages'**]  
 inner\_num\_pages = inner\_table[**'num\_pages'**]  
 **except** KeyError:  
 print(**'KeyError: Table \'{}\' or \'{}\' not found! Cannot continue...'**.format(  
 outer\_table\_name,  
 inner\_table\_name)  
 )  
 sys.exit(-1)  
 **if** block\_size > math.sqrt(outer\_num\_pages):  
 cost = 3 \* (outer\_num\_pages + inner\_num\_pages)  
 **else**:  
 cost = int(  
 2 \* (outer\_num\_pages + inner\_num\_pages) \* (  
 1 + math.ceil(  
 math.log(  
 outer\_num\_pages / (block\_size - 1),  
 block\_size - 1  
 )  
 )  
 ) + (outer\_num\_pages + inner\_num\_pages)  
 )  
 **return** cost  
  
 **def** calc\_join\_cost(self, stats, join):  
 *"""  
 Calculates the cost for a join operation in disk I/O as time.* **:param** *stats: A dict containing table stats.* **:param** *join: The join operation to run.* **:return***: A dict containing the join cost in milliseconds, join order,  
 join method, and join function.  
 """* best\_join\_method = **None** best\_join\_order = **None** best\_join\_function = **None** best\_join\_cost = float(**'inf'**)  
 tables = join[**'tables'**]  
 *# cartesian product is selected if no select rates are specified  
 # no need to perform permutation, the result is the same regardless* **if not** join[**'select\_rates'**]:  
 **return** cost\_to\_time(self.calc\_cartesian\_product(stats, \*tables))  
 **else**:  
 **for** order **in** permute(tables):  
 **for** method, value **in** self.yield\_join\_scenarios():  
 function, args = value[0], value[1]  
 **if** args **is not None**:  
 temp\_cost = function(stats, \*tables, \*args)  
 **else**:  
 temp\_cost = function(stats, \*tables)  
 **if** temp\_cost < best\_join\_cost:  
 best\_join\_cost = temp\_cost  
 best\_join\_method = method  
 best\_join\_order = order  
 best\_join\_function = function  
 **return** {  
 **"total\_cost"**: cost\_to\_time(best\_join\_cost),  
 **"join\_order"**: best\_join\_order,  
 **"join\_method"**: best\_join\_method,  
 **"join\_function"**: best\_join\_function  
 }  
  
 @staticmethod  
 **def** calc\_selection\_cost(stats, select):  
 *"""  
 Determines the total cost of performing a selection operation.* **:param** *stats: A dictionary containing table statistics.* **:param** *select: The select operation to perform.* **:return***: The total I/O cost in milliseconds for the select operation.  
 """* **try**:  
 table\_name = select[**'table'**]  
 table = stats[table\_name]  
 num\_pages = table[**'num\_pages'**]  
 **except** KeyError **as** e:  
 print(**'KeyError: Selection expected key {} but not found! Cannot continue...'**.format(e))  
 sys.exit(-1)  
 **try**:  
 table\_sorted = table[**'sorted'**]  
 **except** KeyError:  
 table\_sorted = **False  
 if** table\_sorted:  
 cost = int(  
 math.ceil(  
 math.log2(  
 num\_pages  
 )  
 )  
 )  
 **else**:  
 cost = num\_pages  
 **return** cost\_to\_time(cost)  
  
 @staticmethod  
 **def** calc\_projection\_cost(stats, project):  
 *"""  
 Determines the total I/O cost of performing a projection operation.* **:param** *stats: A dictionary containing table statistics.* **:param** *project: The project operation to perform.* **:return***: The I/O cost in milliseconds of the project operation.  
 """* **try**:  
 table\_name = project[**'table'**]  
 table = stats[**'tables'**][table\_name]  
 num\_pages = table[**'num\_pages'**]  
 **except** KeyError **as** e:  
 print(**'KeyError: Projection expected key {} but not found! Cannot continue...'**.format(e))  
 sys.exit(-1)  
 **try**:  
 table\_sorted = table[**'sorted'**]  
 **except** KeyError:  
 table\_sorted = **False  
 if** table\_sorted:  
 cost = num\_pages  
 **else**:  
 cost = int(  
 math.ceil(  
 num\_pages \* math.log2(  
 num\_pages  
 )  
 )  
 )  
 **return** cost\_to\_time(cost)  
  
 @staticmethod  
 **def** calc\_aggregation\_cost(stats, aggregate):  
 *"""  
 Determines the I/O cost of running an aggregation operation.* **:param** *stats: A dictionary containing table statistics.* **:param** *aggregate: The aggregate operation to perform.* **:return***: The I/O cost in milliseconds of an aggregation operation.  
 """* **try**:  
 table\_name = aggregate[**'table'**]  
 groupby = aggregate[**'groupby'**]  
 table = stats[**'tables'**][table\_name]  
 num\_pages = table[**'num\_pages'**]  
 **except** KeyError **as** e:  
 print(**'KeyError: Aggregation expected key {} but not found! Cannot continue...'**.format(e))  
 sys.exit(-1)  
 **try**:  
 table\_sorted = table[**'sorted'**]  
 **except** KeyError:  
 table\_sorted = **False  
 if** groupby:  
 **if** table\_sorted:  
 cost = num\_pages  
 **else**:  
 cost = int(  
 math.ceil(  
 math.log2(num\_pages) + num\_pages  
 )  
 )  
 **else**:  
 cost = num\_pages  
 **return** cost\_to\_time(cost)  
  
 **def** calc\_best\_exec\_plan(self, filename):  
 *"""  
 Determines the best execution plan for a given query.* **:param** *filename: The filepath to a supported query file.* **:return***: A QueryPlan object.  
 """* **def** exec\_query(query, is\_subquery=**False**):  
 **try**:  
 query\_steps = query[**'steps'**]  
 **if** is\_subquery:  
 is\_correlated = query[**'is\_correlated'**]  
 **else**:  
 is\_correlated = **False  
 except** KeyError **as** e:  
 print(**'KeyError: Key \'{}\' not found in Query, cannot continue...'**.format(e))  
 sys.exit(-1)  
 query\_cost = 0  
 query\_join\_methods = []  
 query\_join\_orders = []  
 query\_join\_functions = []  
 **for** step **in** query\_steps:  
 **if 'select' in** step:  
 select = query[step]  
 table = select[**'table'**]  
 query\_cost += self.calc\_selection\_cost(stats, select)  
 num\_pages = stats[table][**'num\_pages'**]  
 temp\_name = select[**'result'**]  
 **for** rate **in** select[**'select\_rates'**]:  
 num\_pages \*= stats[**'select\_rates'**][rate]  
 tuple\_size = stats[table][**'tuple\_size'**]  
 is\_sorted = stats[table][**'sorted'**]  
 temp\_table = create\_table(name=temp\_name,  
 num\_pages=num\_pages,  
 tuple\_size=tuple\_size,  
 is\_sorted=is\_sorted)  
 stats[**'tables'**][temp\_table[0]] = temp\_table[1]  
 **elif 'project' in** step:  
 project = query[step]  
 table = project[**'table'**]  
 query\_cost += self.calc\_projection\_cost(stats, project)  
 num\_pages = stats[**'tables'**][table][**'num\_pages'**]  
 temp\_name = project[**'result'**]  
 **for** rate **in** project[**'project\_rates'**]:  
 num\_pages \*= stats[**'project\_rates'**][rate]  
 tuple\_size = stats[**'tables'**][table][**'tuple\_size'**]  
 is\_sorted = stats[**'tables'**][table][**'sorted'**]  
 temp\_table = create\_table(name=temp\_name,  
 num\_pages=num\_pages,  
 tuple\_size=tuple\_size,  
 is\_sorted=is\_sorted)  
 stats[**'tables'**][temp\_table[0]] = temp\_table[1]  
 **elif 'aggregate' in** step:  
 aggregate = query[step]  
 table = aggregate[**'table'**]  
 query\_cost += self.calc\_aggregation\_cost(stats, aggregate)  
 num\_pages = stats[**'tables'**][table][**'num\_pages'**]  
 temp\_name = aggregate[**'result'**]  
 tuple\_size = stats[**'tables'**][table][**'tuple\_size'**]  
 *#* ***TODO: FIGURE OUT WHAT PAGES AND TUPLE SIZE SHOULD BE*** temp\_table = create\_table(name=temp\_name,  
 num\_pages=num\_pages,  
 tuple\_size=tuple\_size)  
 stats[**'tables'**][temp\_table[0]] = temp\_table[1]  
 **elif 'subquery' in** step:  
 subquery = query[step]  
 table = subquery[**'table'**]  
 subquery\_result = exec\_query(subquery, **True**)  
 **if** subquery[**"is\_correlated"**]:  
 corr\_cost = subquery\_result[**'total\_cost'**]  
 join\_order = query\_join\_orders[-1]  
 num\_pages = stats[**'tables'**][join\_order[0]][**'num\_pages'**]  
 query\_cost += num\_pages \* corr\_cost  
 **else**:  
 query\_cost += subquery\_result[**'total\_cost'**]  
 query\_join\_methods.extend(subquery\_result[**'join\_methods'**])  
 query\_join\_orders.extend(subquery\_result[**'join\_orders'**])  
 temp\_name = subquery[**'result'**]  
 stats[**'tables'**][temp\_name] = stats[**'tables'**][table]  
 **elif 'join' in** step:  
 join = query[step]  
 **if** is\_correlated:  
 **try**:  
 tnj\_tables = join[**'tables'**]  
 **except** KeyError **as** e:  
 print(**'KeyError: Correlated join expected {} key but not found, cannot continue!'**.format(e))  
 sys.exit(-1)  
 corr\_join\_cost = float(**'inf'**)  
 corr\_join\_order = **None  
 for** order **in** permute(tnj\_tables):  
 temp\_cost = self.calc\_tuple\_nested\_join\_cost(stats, \*tnj\_tables)  
 corr\_join\_order = order  
 corr\_join\_cost = temp\_cost  
 query\_cost += stats[**'tables'**][corr\_join\_order[0]][**'num\_pages'**] \* corr\_join\_cost  
 query\_join\_orders.append(corr\_join\_order)  
 query\_join\_methods.append(**'CORR\_TNJ'**)  
 query\_join\_functions.append(self.calc\_tuple\_nested\_join\_cost)  
 **else**:  
 join\_result = self.calc\_join\_cost(stats, join)  
 query\_cost += join\_result[**'total\_cost'**]  
 query\_join\_orders.append(join\_result[**'join\_order'**])  
 query\_join\_methods.append(join\_result[**'join\_method'**])  
 query\_join\_functions.append(join\_result[**'join\_function'**])  
 temp\_name = join[**'result'**]  
 join\_order = query\_join\_orders[-1]  
 num\_pages = (  
 stats[**'tables'**][join\_order[0]][**'num\_pages'**] \*  
 stats[**'tables'**][join\_order[1]][**'num\_pages'**]  
 )  
 **for** select\_rate **in** query[step][**'select\_rates'**]:  
 num\_pages = int(  
 math.ceil(  
 num\_pages \*  
 stats[**'select\_rates'**][select\_rate]  
 )  
 )  
 tuple\_size = (  
 stats[**'tables'**][join\_order[0]][**'tuple\_size'**] +  
 stats[**'tables'**][join\_order[1]][**'tuple\_size'**]  
 )  
 join\_function = query\_join\_functions[-1]  
 is\_sorted = **True if** join\_function == self.calc\_sort\_merge\_join\_cost **else False** temp\_table = create\_table(  
 name=temp\_name,  
 num\_pages=num\_pages,  
 tuple\_size=tuple\_size,  
 is\_sorted=is\_sorted  
 )  
 stats[**'tables'**][temp\_table[0]] = temp\_table[1]  
 **return** {  
 **'total\_cost'**: query\_cost,  
 **'join\_orders'**: query\_join\_orders,  
 **'join\_methods'**: query\_join\_methods,  
 **'is\_correlated'**: is\_correlated  
 }  
  
 **try**:  
 query\_file = read\_query\_file(filename)  
 **except** OSError:  
 print(**'IOError: Unable to open query file! Cannot continue...'**)  
 sys.exit(-1)  
 **try**:  
 stats = query\_file[**'stats'**]  
 root\_query = query\_file[**'query'**]  
 **except** KeyError:  
 print(**'Error: Unable to parse required information for building execution plan, cannot continue!'**)  
 sys.exit(-1)  
 query\_name = filename.split(**'/'**)[-1].replace(**'.json'**, **''**)  
 result = exec\_query(root\_query)  
 total\_cost = result[**'total\_cost'**]  
 join\_orders = result[**'join\_orders'**]  
 join\_methods = result[**'join\_methods'**]  
 timestamp = create\_timestamp(total\_cost)  
 **return** QueryPlan(  
 query\_name=query\_name,  
 total\_cost=total\_cost,  
 join\_orders=join\_orders,  
 join\_methods=join\_methods,  
 has\_subquery=root\_query[**'has\_subquery'**],  
 is\_correlated=root\_query[**'is\_correlated'**],  
 timestamp=timestamp  
 )  
  
  
@click.command()  
@click.option(**'-ps'**, **'--page-size'**, type=int, default=4096)  
@click.option(**'-bs'**, **'--block-size'**, type=int, default=100)  
@click.argument(**'filepaths'**, nargs=-1)  
**def** main(page\_size, block\_size, filepaths):  
 *"""  
 Runs the query optimzer with a preconfigured set of join operations.* **:param** *page\_size: The page size in bytes.* **:param** *block\_size: The number of pages per block.* **:param** *filepaths: A list containing paths to queries to evaluate -- variadic.  
 """* qo = QueryOptimizer(page\_size, block\_size)  
 join\_scenarios = [  
 [**"TNJ"**, qo.calc\_tuple\_nested\_join\_cost, **None**],  
 [**"PNJ"**, qo.calc\_page\_nested\_join\_cost, **None**],  
 [**"BNJM"**, qo.calc\_block\_nested\_join\_cost, [50]],  
 [**"BNJL"**, qo.calc\_block\_nested\_join\_cost, [30]],  
 [**"SMJM"**, qo.calc\_sort\_merge\_join\_cost, [50]],  
 [**"SMJL"**, qo.calc\_sort\_merge\_join\_cost, [30]],  
 [**"HJM"**, qo.calc\_hash\_join\_cost, [50]],  
 [**"HJL"**, qo.calc\_hash\_join\_cost, [30]]  
 ]  
 **for** scenario **in** join\_scenarios:  
 qo.add\_join\_scenario(\*scenario)  
 **try**:  
 **for** filepath **in** filepaths:  
 print(qo.calc\_best\_exec\_plan(filepath))  
 **except** OSError:  
 print(**'No plan generated for path: {}, path not valid!'**)  
  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 main()