INFORMATICS ENGINEERING STUDY PROGRAM
SCHOOL OF ELECTRICAL ENGINEERING DAN INFORMATICS
BANDUNG INSTITUTE OF TECHNOLOGY

MEGA Project Specification

mini Database Management System Development

Prepared By:
Database Systems Graduate Teaching Assistants
Database Laboratory Teaching Assistants

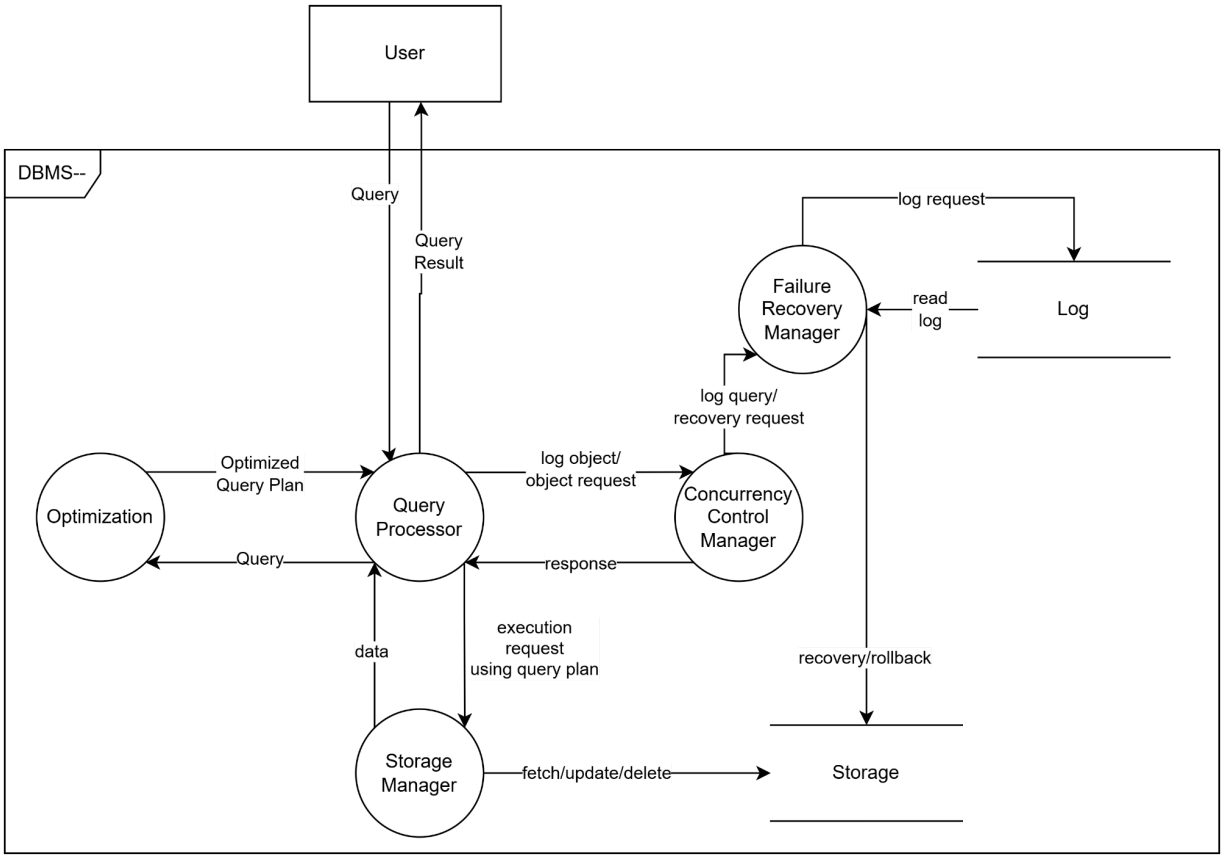
Revisi 21/11/2024

IF3140 - Database Systems

2024

Part I. Expected mDBMS To Develop Overview

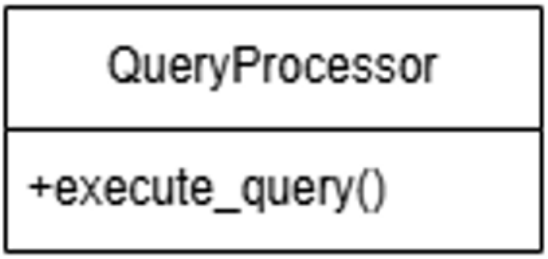
The goal of this project is to develop a mini database management system (mDBMS) for relational databases. Each GROUP will **only** work on **1 (one) component** of mDBMS which will be explained in the next section. Since components of mDBMS have dependency on each other, assistants will provide test cases for each component and component interactions to help every group to make sure that their implementation is correct. Below is the data flow diagram of the mDBMS that will be implemented in this project.



The mDBMS will consist of **5** components:

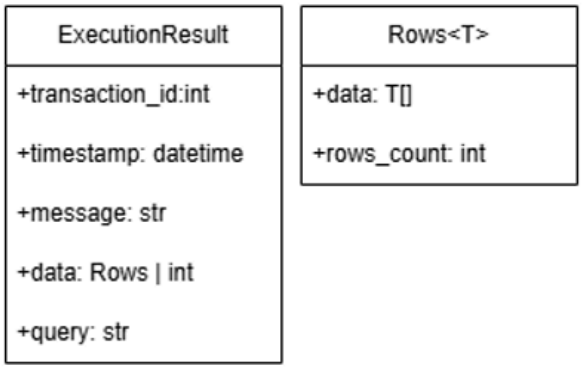
1. Query Processor

This component handles the execution of a query. This component has **one** public method:



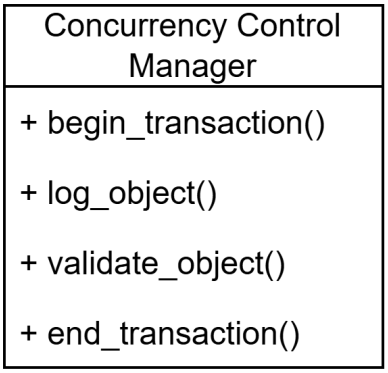
a. **execute_query(query:str)->ExecutionResult**: This method accepts a query, which will then be sent to the query optimizer module to be optimized. After that, it will receive the query plan from the query optimizer and get the appropriate rows from the storage manager, as well as computing the manipulation to the rows retrieved such as the JOIN method (the algorithm to use is based on the query plan given). For every transaction, make sure that the object it is about to read or write is allowed according to the concurrency control schema. If it is allowed it will proceed to execute the query, otherwise, the transaction is to be aborted. A query can be executed if and only if the concurrency control manager gives a response, otherwise it will wait until the given response. For every query executed, it will be logged to the failure recovery module.

For objects used in the component, here's the class interface for those objects:



2. Concurrency Control Manager

This component manages the scheduling of queries and handles transaction concurrency within the mDBMS. The component includes methods for managing locks, tracking timestamps, and preventing deadlocks using wound-wait or wait-die strategy. It communicates with the Query Processor to ensure a safe environment for object read and write operations. This component has **four** public methods.



a. **begin_transaction() -> transaction_id: int**
   On 'BEGIN TRANSACTION' command given or for every statement given, assign a transaction id to the transaction that has just started.
b. **log_object(object: Row, transaction_id: int)**
   Logs an object on transaction x. This log can be used to implement the lock on an object or to have a timestamp on the object. What to log depends on the chosen concurrency control algorithm. You are required to choose **one** concurrency control algorithm, however **if you want to implement more than one (2-3 algorithms), your group will receive a bonus point.** The mechanisms for the concurrency control can be either lock-based, timestamp-based, or multi version-based. Be sure to implement a command to change the mechanisms.
c. **validate_object(object: Row, transaction_id: int, action: Action) -> Response**

Validates a given object whether it is allowed to do a particular action or not. For example, if there is a request to do a write on object x from transaction 1, it will return whether the write action can be implemented or not. The response would either be to allow the transaction or not.

d. **end_transaction(transaction_id: int)**

   Flushes objects belonging to a particular transaction after it has successfully committed/aborted. The method to be implemented depends on the chosen concurrency control algorithm. Also terminates the transaction.

Ideas for the interface of objects used on this module are given as follows,

Action
action: Union['write', 'read']

Response
allowed: boolean transaction_id: int

3. Query Optimizer

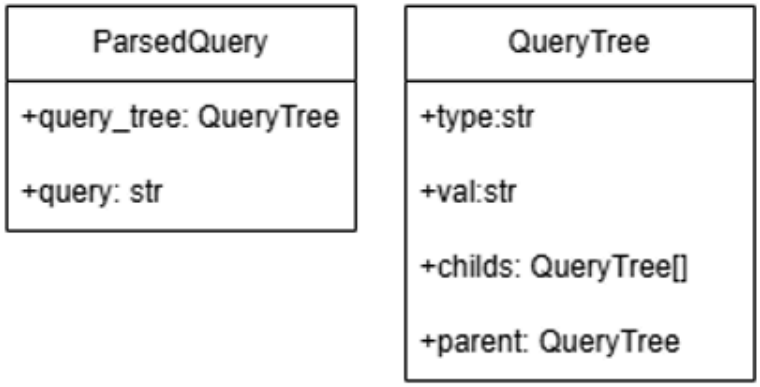
This component accepts a query as input, parses it, and returns an optimized query plan in the form of a parsed query **object**. This component has **two** public method and **one** private method:

OptimizationEngine
+parse_query() +optimize_query() -get_cost()

a. **parse_query(query:str)->ParsedQuery**: This method accepts a query in the form of a string and parses it into an **objec**t that represents a parsed query. The internal implementation of the parsed query object is left to the participant.

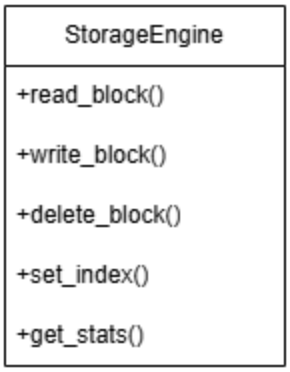
b. **optimize_query(query:ParsedQuery)->ParsedQuery**: This **public** method optimizes the parsed query it receives as input according to optimization rules and returns the optimized query. There's also a **bonus point** if the group implements optimization using **genetic algorithm**.
c. **get_cost(query:ParsedQuery)->int**: This **private** method calculates execution cost for a given query. This method acts as an auxiliary function for **optimize_query**.

For objects used in the method, here's the class interface for those objects



4. Storage Manager

This component handles storage, modification, and retrieval of data in harddisk. It also handles and stores indexes and provides statistical information for query optimization. Data is stored in harddisk in the form of **binary file(s)**. Whether to have one file store all of the tables or a different file for each table is left to the group to decide. In order to help the implementation, this component has **five** public methods:



a. **read_block(data_retrieval:DataRetrieval)->Rows**: This method accepts data retrieval **objects** as input and retrieves data from the harddisk. Data retrieval objects should be parsed from the query plan. Data retrieval **object** contains data to help the storage manager **determine** which data to be retrieved from the hard disk. The internal implementation of the data retrieval object is left for each group to decide. This method returns retrieved data.
b. **write_block(data_write:DataWrite)->int**: This method accepts a data write **object** as input and performs addition and modification of data in the harddisk. Data write **object** contains data to help the storage manager **determine** which existing data to be modified, and also contains the **modified data** for modification operation and **new data** for addition operation. This method returns the number of affected rows.
c. **delete_block(data_deletion:DataDeletion)->int**: This method accepts data deletion **object** as input and performs deletion of data in harddisk. Data deletion **object** contains data to help the storage manager **determine** which existing data to be deleted. This method returns the number of removed rows.
d. **set_index(table:str,column:str,index_type:str)->None**: This method handles creation of index for **column** in a given table. It accepts **three** inputs, **table name**, **column name**, and **index type**. Index type can be **B+ Tree** or **Hash**. For implementation, only **one** index type is mandatory. If **both** are implemented, the group will get **bonus points**.

e. **get_stats()->Statistic**: This method retrieves metrics, such as number of **tuples** in relation, **size** of tuples, **blocking factors**, etc. These metrics will be used by the query optimizer calculating the query plan cost.

   n_r: number of tuples in a relation r.

   b_r: number of blocks containing tuples of r.

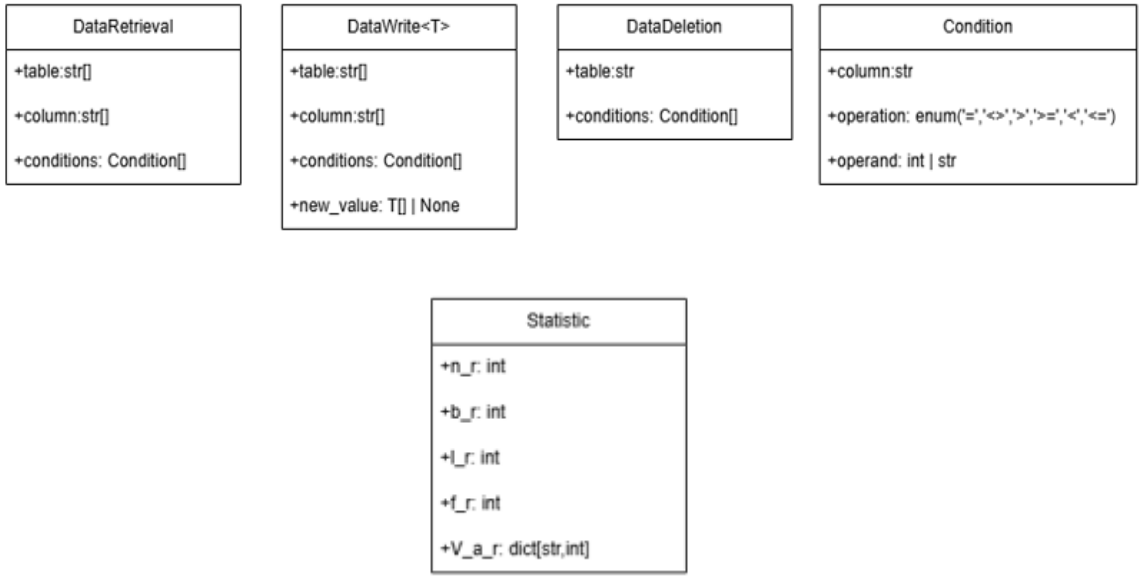
   l_r: size of tuple of r.

   f_r: blocking factor of r – i.e., the number of tuples of r that fit into one block.

   $V(A, r)$: number of distinct values that appear in r for attribute A; same as the size of $\Pi A(r)$.

   If tuples of r are stored together physically in a file, then: $b_r = \lceil n_r / f_r \rceil$

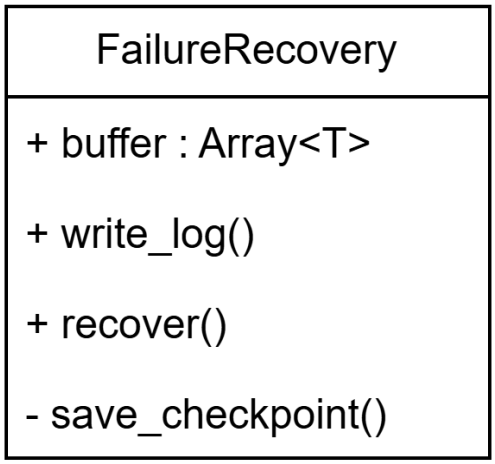
Here's the class diagram for objects used in this component. For data retrieval, student **could** also add attribute to determine **search** type, for example, whether to perform linear search or search using index.



The storage manager would have to communicate with the failure recovery manager to handle 'dirty' written data and to minimize I/O access on the 'disk'

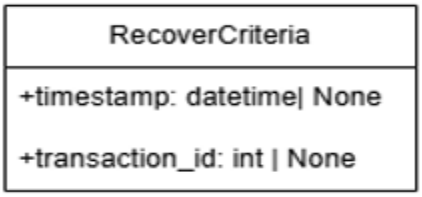
5. Failure Recovery Manager

This component handles logging and recovery of execution and data. This component has **two** public methods, **one** private method and **one** attribute that is the buffer:



a. **write_log(info:ExecutionResult)->None**: This method accepts execution result **object** as input and appends an entry in a **write-ahead log** based on execution info **object**.
b. **save_checkpoint()**: This method is called to save a checkpoint in log. In this method, all entries in the write-ahead log from the last checkpoint are used to update data in physical storage in order to synchronize data. This method can be called after certain time periods (e.g. 5 minutes), and/or when the write-ahead log is almost full.
c. **recover(criteria:RecoverCriteria)->None**: This method accepts a checkpoint **object** that contains the criteria for checkpoint. This criteria can be timestamp or transaction id. The recovery process started backward from the latest log in **write-ahead log** until the criteria is no longer met. For each log entry, this method will interact with the query processor to execute a recovery query, restoring the database to its state prior to the execution of that log entry.

For **RecoverCriteria** object, here's the class diagram:



The buffer's function is to temporarily hold data that has recently been accessed or updated in the database. The buffer has a finite size and has to be emptied on checkpoint operations or when it is almost or already full. The storage manager would have to communicate with the buffer on read or about to write on the data.

Part II. What To Do

Develop a Database Management System that contains five components: query processor, concurrency control manager, optimization engine, storage manager, and failure recovery. All students can use the **Python** programming language with OOP.

1. Query Processor GROUP

For **query processors**, the types of queries supported are as follows.

1. SELECT

   Users can select data from one or more tables based on the **specified attributes** or **all attributes using the '*' sign**.

2. UPDATE

   Users can change the value of records in a table by using the UPDATE and SET statement. For this functionality, please implement **only one condition** after the WHERE statement. For example: **UPDATE employee SET salary=1.05*salary WHERE salary > 1000;**

3. FROM

   Users can select one or more tables to retrieve data from. For selecting data from more than one table, the resulting table is the **cartesian product** of the tables.

4. AS

   By using the AS statement, users can perform aliasing on a table.

   E.g. SELECT * FROM **student AS s, lecturer AS** l WHERE s.lecturer_id=l.id;

5. JOIN

   Users can join tables by using **JOIN ON** or **NATURAL JOIN** statements.

6. WHERE

   Following the WHERE statement is the condition(s) that should be fulfilled in data selection. Comparison operators that should be available are **equal to (=), not equal to (<>), greater than (>), greater than or equal to (>=), less than (<)**, and **less than or equal to (<=**).

7. ORDER BY

   Users can order selected data based on **one** attribute, either **ascending or descending**. Attributes with numerical value should be ordered according to their value. Attributes which value type is string or other special characters should also be ordered.

   Hint: Use the ord() function in Python to implement this.

8. LIMIT

   Users can limit the amount of records they want to select by using the LIMIT statement.

9. BEGIN TRANSACTION

   Starts a transaction

10. COMMIT

    Commits a transaction

11. DELETE **(BONUS)**

    Users can delete a record from a table based on **only one condition**. For example: **DELETE FROM employee WHERE department=”RnD”;**

12. INSERT **(BONUS)**

    Users can insert a record into a table using the INSERT INTO statement. **Only one record can be inserted in a query.** Since we don’t handle special values such as the NULL value, all columns should be specified in the query.

13. CREATE TABLE **(BONUS)**

Users can create tables using the CREATE TABLE statement. Supported attribute types are **integer, float, char and varchar**. Please make sure that you implement the **foreign key** and **primary key constraints**. Do not forget to consult with the storage manager team.

14. DROP TABLE **(BONUS)**

    This statement should delete or drop a table. Make sure that you handle special cases such as **cascading delete** and **restricted delete**.

2. Query Optimizer <u>GROUP</u>

For **optimization**, all **supported** equivalence rules are as follows.

1. Conjunctive selection operations can be deconstructed into a sequence of individual selections

$$\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$$

2. Selection operations are commutative
3. Only the last in a sequence of projection operations is needed, the others can be omitted

$$\Pi_{L_1}(\Pi_{L_2}(\ldots(\Pi_{L_n}(E))\ldots)) = \Pi_{L_1}(E)$$

4. Selections can be combined with Cartesian products and theta joins.
   a. $\sigma_\theta(E_1 \times E_2) = E_1 \bowtie_\theta E_2$
   b. $\sigma_\theta(E_1 \bowtie_\theta E_2) = E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$
5. Theta-join operations (and natural joins) are commutative

$$E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$$

6. a. Natural join operations are associative:

$$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$$

   b. Theta joins are associative in the following manner

$$(E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_1 \wedge \theta_2} E_3 = E_1 \bowtie_{\theta_1 \wedge \theta_2} (E_2 \bowtie_{\theta_2} E_3)$$

   Where θ_2 involves attributes from only E_2 and E_3

7. The selection operation distributes over the theta join operation under the following two conditions:
    a. When all the attributes in θ_0 involve only the attributes of one of the expressions (E_1) being joined

$$\sigma_{\theta_0}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta_0}(E_1)) \bowtie_\theta E_2$$

    b. When θ_1 involves only the attributes of E_1 and θ_2 involves only the attributes of θ_2

$$\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta_1}(E_1)) \bowtie_\theta (\sigma_{\theta_2}(E_2))$$

8. The projection operation distributes over the theta join operation as follows:
    a. If θ involves only attributes from $L_1 \cup L_2$:

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = (\Pi_{L_1}(E_1)) \bowtie_\theta (\Pi_{L_2}(E_2))$$

    b. Consider a join $E_1 \bowtie_\theta E_2$

    Let L_1 and L_2 be sets of attributes from E_1 and E_2 respectively

    Let L_3 be attributes of E_1 that are involved in join condition θ, but are not in $L_1 \cup L_2$ and

    Let L_4 be attributes of E_2 that are involved in join condition θ, but are not in $L_1 \cup L_2$

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = \Pi_{L_1 \cup L_2}((\Pi_{L_1 \cup L_3}(E_1)) \bowtie_\theta (\Pi_{L_2 \cup L_4}(E_2)))$$

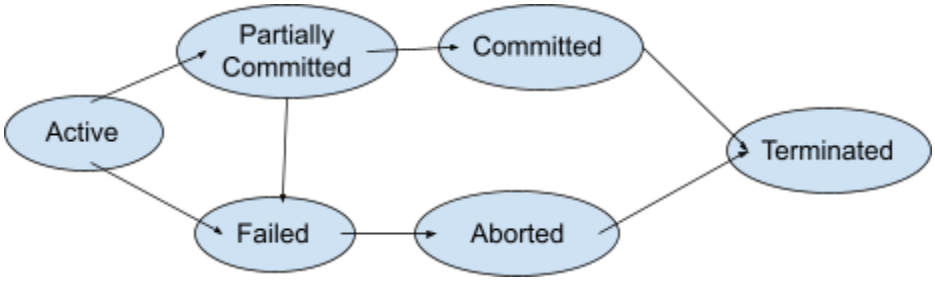
Use a **heuristic** metric to determine the best of some query plans that you have built up upon. Not all plans have to be implemented.

3. Concurrency Control Manager <u>GROUP</u>

The concurrency control manager acts as the ‘moderator’ of transactions in a database. It helps to fulfill the ACID property of a transaction. Implement so that any transaction started will fulfill ACID and also prevents deadlocks from happening.

To enable two or more transactions, a group (not fixed to concurrency control manager group) needs to create a simple client object that users would have to interact with. Therefore, there could be many clients, but for the 5 components of the mDBMS, there is only one at a certain point of time (singleton).

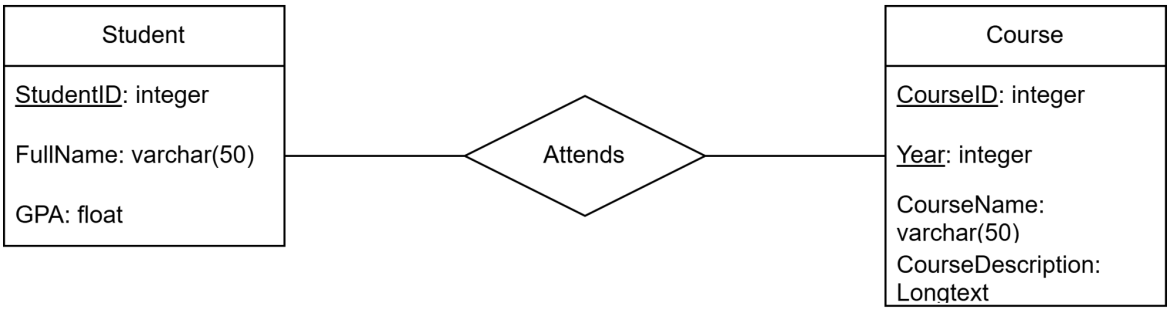
The possible states allowed for transactions can be viewed below. Make sure that the concurrency control manager can reach each state and their following states based on the conditions that you have learned in this course.



4. Storage Manager GROUP

The storage manager is responsible for managing the storage disk. The component has direct access to the storage. **Read and write** procedures are handled by this component. Supported attribute types are **integer, float, char, and varchar**. Handling scheme creation or deletion and data creation or deletion is part of a bonus to be implemented with the query processor team.

This group should also implement a serializer for tables and rows to help with block allocation. The minimum database to build should have at least three tables that can be joined and at least 50 rows on each table. The serializer should be able to serialize the schema to a file and process it using block allocation. A sample **ERD** that could be used is as follows



5. Failure Recovery Manager GROUP

For **the failure recovery**, the two features that must be developed are as follows:

1. Storing log

   All committed transactions should be stored in a separate storage. You may define how you design the log scheme.

2. Recovery

   The database can be recovered by replaying all transactions that have been stored as logs. This could also be used to do a rollback on transactions.

To test failure recovery from multiple transactions, a group (not fixed to failure recovery group) needs to create a simple client object that users would have to interact with. Therefore, there could be many clients, but for the 5 components of the mDBMS, there is only one at a certain point of time (singleton).

Part III. Project Mechanisms

1. This project will be done in one unit called SUPER GROUP. One SUPER GROUP consists of 5 GROUPs. Each GROUP consists of 4-5 people. Member allocation for each GROUP has already been determined **based on your performance on IF3140 Lab Sessions Results**, you can see your GROUP in this spreadsheet on Tab 'Group Member Lists'.
2. There are 5 components that each SUPER GROUP will work on. Components to be created for each of the SUPER GROUP includes,
    a. Query Processor
    b. Query Optimizer
    c. Concurrency Control Manager
    d. Storage Manager
    e. Failure Recovery Manager
3. Implement each of the components using Python with an Object Oriented Programming paradigm. You are **NOT** allowed to use Python library (should be no use) outside **basic library** like **datetime, ABC, UUID, re**, **struct**, etc. If unsure, you can ask in QnA thread whether this library is allowed or not. Besides permissible library, implement all classes, methods, functions, and procedures **FROM SCRATCH.**
4. You are allowed to make any modifications to the class diagrams and methods as needed if it helps you to do your work better. Some diagrams are just ideas so you might need to modify it to suit your needs.
5. Each GROUP in a SUPER GROUP will work on a **different** component of the mDBMS to form a complete mDBMS. Each GROUP in the SUPER GROUP can select their desired components to be developed. Please discuss in your SUPER GROUP to determine which component will be developed for each GROUP. Please fill out the task allocation of each SUPER GROUP by **November 16th 2024, 03.14.00 AM** in this spreadsheet on tab 'Group Task Assignment'

6. Members of each GROUP **ARE NOT ALLOWED** to contribute to other GROUP's component development process even for another GROUP in the same SUPER GROUP.
7. Grading system will be based on 90% of your GROUP's component and 10% of your SUPER GROUP mDBMS system. (It is not worth to contribute to other GROUP's component development process)
8. All SUPER GROUP are obliged to make all components work together. Make sure to design a system that is compatible with each of the other components. Because the Query Processor module will have to combine the other modules, groups that work on this module will have to design the interaction of each module so that they could integrate it too.
9. We highly recommend you to elect one person for each GROUP to integrate your SUPER GROUP's mDBMS components. This person will take charge of component design, coordinate with other GROUPs in the same SUPER GROUP, and integrate your GROUP's component at the end of the project. We are hoping that these persons will only take a minimal portion of the component's development, since they are in charge of a much bigger system. **This work scheme is not mandatory**, you can work on your project at your earliest convenience.
10. Deadline for the whole mDBMS is on **December 8th 2024**. However, there are several checkpoints you need to be aware of,
    a. On **November 22nd 2024**, the Query Processor GROUP must be able to accept queries and send read/write queries such as SELECT, UPDATE and other statements to the Query Optimizer GROUP and transaction related queries such as BEGIN TRANSACTION, ABORT and COMMIT to the Concurrency Control Manager GROUP and Failure Recovery Manager GROUP. The Storage Manager GROUP should also have created a simple database that could be processed
    b. On **November 28th 2024**, all GROUPs should at least finish 50% of their work. Try to integrate the query processor, storage manager, and the

query optimizer components first. Start creating a simple client too (no need for a GUI, just use CLI).

c. On **December 4th 2024**, try to start integrating the Concurrency Control Manager and Failure Recovery Manager to the rest of the mDBMS. Start testing for bugs and test cases
d. On **December 8th 2024**, all components should already be integrated as one fully functional mDBMS. A simple client should be available to use

11. On each checkpoint, every GROUP submit their report in the form of a checkpoint report (template) and a GitHub Release. Checkpoint document must be written in **ENGLISH**. You do not need to write a long report, a simple and compact one will suffice. All checkpoint deadlines are at **03.14.00 AM** on the corresponding day. For each checkpoint, report the progress that you have done so far. Submit the checkpoint report to this form.
12. Each SUPER GROUP is required to make 1 **FINAL REPORT**. Final report must also be written in **ENGLISH**. Different from the checkpoint report, you should write this final report in as detail as possible. We grade your components & system architecture and implementation based on this report. On the final deadline, each SUPER GROUP **MUST** submit their work in the form of a final report (template) and a GitHub Release. Complete deliverables should be submitted by **December 8th 2024 03.14.00 AM**. Submit the report and GitHub repository link to this form.
13. For Git Releases, for each repository that you have (usually per component), make a release in the format of v<x>.<y> with x being the checkpoint and y the release version starting from 1. If you only have one repository, just release one. Please provide the release link on your report appendix. For the final report, provide all repositories that are involved in this mega project.
14. Late submission for each checkpoint will be subject to penalties for corresponding GROUP (not SUPER GROUP). Late submission for the final report will be subject to penalties for corresponding SUPER GROUP.
15. An example of the project folder can be viewed below

```
|- Query Processor
```

```
        |- classes.py
        |- UnitTest.py
|- Query Optimizer
        |- classes.py
        |- UnitTest.py
|- Concurrency Control Manager
        |- classes.py
        |- UnitTest.py
|- Storage Manager
        |- classes.py
        |- UnitTest.py
|- Failure Recovery
        |- classes.py
        |- UnitTest.py
main.py
data.dat
```