ENO: A solution for ACIDity

March 20, 2022

Team Members

Abhinav Gupta - 190050003
Ankit Kumar Jain - 190050019
Tulip Pandey - 190050125
Vibhav Aggarwal - 190050128

Requirements

- **Query Parser:** A command-line interface for CRUD operations on the database. We will support the following types of queries:

    - [WITH table_name1 (col1, col2, ...) AS <select query>,
      table_name2 (col1, col2, ...) AS <select query>,
      ...
      ]
      <select query>

      <select query> syntax:
      SELECT {*| col1, col2, ...} FROM table_name [, table_name2] [WHERE <condition>];
      Note: We support joins for two tables only

      <condition> syntax:
      <relex> <boolop> <relex> *for binary boolop*
      <condition> <boolop> <condition> *for binary boolop*
      <boolop> <relex> *for unary boolop*

<boolop> <condition> *for unary boolop*

<relex> syntax:
<operand> <relop> <operand>

<relop> := {=, <, >, <=, >=, !=}
<boolop> := {AND, OR, NOT}
<operand> := {table_name.column_name, <some literal>}

- INSERT INTO table_name VALUES (value1, value2, …);
- DELETE FROM table_name [WHERE <condition>];
- UPDATE table_name
  SET column1 = value1, column2 = value2, …
  [WHERE <condition>];
- CREATE TABLE table_name (
    column1 <datatype> [RANGE (between <val1> and <val2>)],
    column2 <datatype> [RANGE (between <val1> and <val2>)],
    ....,
    [PRIMARY KEY (col1, col2, …)]
  );
  <datatype> can be: int, float, text
  <val1> can be a number or -INFINITY
  <val2> can be a number or INFINITY
  Note: RANGE is supported only for int and float datatypes
- COMMIT
- ROLLBACK

Limitations:

- SELECT does not support subqueries after the FROM keyword but the same functionality can be achieved by using the WITH clause

- Only two tables can be joined in a single query, so to join multiple tables, the user will need to write multiple queries, joining the tables pairwise
- Set operations like UNION, INTERSECT, etc. are not supported.
- WHERE clause supports only the basic relational and boolean conditions. Other complex conditions like IN, SOME, ALL, etc. are not supported.

- **Atomicity:** By this, we mean that either the entire transaction takes place at once or doesn't happen at all. There is no midway i.e. transactions do not occur partially. Each transaction is considered as one unit and either runs to completion or is not executed at all. Therefore, the following will be supported:
  - The following errors will be dealt with in ensuring atomic transactions:
    - Some error occurs during query execution (like division by zero)
    - The process gets killed in the middle of a transaction
  - **Limitations** - The following errors will NOT be dealt with:
    - Hardware failures
    - Any changes to the database related files by some external process

  For eg:

  - If u1 adds 2 to a row in q(1) and divides the row by 0 in q(2), then, after q(1), u1 sees row as old_row+2 and u2 sees row as old_row; after q(2), the entire transaction rolls back so both u1 and u2 see the row as old_row.

  Transaction details:

  - A transaction starts as soon as any query is executed just after the COMMIT/ROLLBACK keyword, or at the beginning of the database process
  - It commits/rolls back using the COMMIT or the ROLLBACK keyword respectively
  - If the database process is terminated midway through a transaction, then the transaction is rolled back

- **Consistency:** Implementation of the following constraints will be carried out:

  - Primary key

  - Field ranges (only for int and float data types)

- **Limitations -** Foreign key, functions, unique, triggers, sum upto, cascade on delete etc not supported.

For eg:

- CREATE TABLE books (
  name text,
  volume int RANGE (between 0 and 5),
  PRIMARY KEY (name)
  );
  INSERT INTO book VALUES (“Angels and Demons”, 7); // This will give an error due to range violation

  INSERT INTO book VALUES (“Angels and Demons”, 4);
  INSERT INTO book VALUES (“Harry Potter”, 2);
  UPDATE books SET name=”Angels and Demons” WHERE volume=2; // This will give an error due to primary key violation

- If any query causes the violation of any constraint, then the entire transaction will be rolled back

- **Isolation:** This means that the database provides support for multiple independent transactions to be executed simultaneously provided that each is either a read-only transaction or a non-overlapping (table-wise) write. The following will be supported:
  - The user u1, who is writing queries q(1), q(2), … as a part of a transaction would effectively see the states corresponding to updates made till query q(i-1) at the beginning of query q(i).
  - Any other user u2 will see the state of the database as the state present before q(1) (if u2 tries to read the same data as that being updated, but not yet committed by u1) or the state after the entire transaction is committed (after the data is committed by u1).

  Limitations -

- Does not support non-overlapping writes within the same table.

For eg:

- If u1 adds 20 to a row in q(1) and then multiplies the row by 2 in q(2) and then commits the transaction, then after q(1), u1 sees row as old_row+20 and u2 sees row as old_ row; after q(2), u1 sees row as (old_row+20)*2 and u2 sees row as old_row; after commit, both u1 and u2 see the row as (old_row+20)*2.

- **Durability:** This means that the update caused by each successful transaction must reach the disk (must be retained in the database until another transaction makes any changes to the same).
  - All changes are stored in the database itself or some database-related file (which is also eventually written to the database or discarded completely in case the change is no longer relevant). Therefore, all storage is durable unless there is a filesystem failure.