Utilizing U-Semiring Algebra for SQL Query Equivalence Verification

Alejandro Urquieta, Chris Tsongas, Ayooluwa Oladosu, Wasey Mulla, Uthama Kadengodlu

1 Research Issue and Contributions

1.1 Research Issue

A database's main role is in storing large amounts of data and retrieving that data as well. In addition to that, databases also work to find efficiencies in retrieving said data. These efficiencies take the form of query optimization. Query optimization looks to speed up queries usually by reducing the number of rows that need to be accessed. One problem that emerges from this is what steps need to be taken for a query to become more optimal and do these steps leave the semantic meaning of the query unchanged. Achieving optimality and testing equivalence is the main research issue that the project focuses on.

1.2 Contributions

The contribution and additional implementation from the knowledge we gained from the research paper focused on validating the semantic equivalence of SQL queries has been successfully completed, encompassing a meticulous examination of both optimized and unoptimized query pairs. One of our key contributions lies in the meticulous assertion that optimized queries consistently produce equivalent results to their original counterparts[1]. This critical validation process involved an exhaustive analysis, ensuring that the optimized versions maintained identical functionality and delivered the same outcomes as the original queries.

Furthermore, a significant aspect of our contribution involves verifying the potential transformation of unoptimized queries into their optimized forms to ascertain logical equivalence. We implemented and leveraged the u-semiring algorithm, an integral part of our verification process. This sophisticated algorithm allowed us to systematically manipulate unoptimized queries and observe their transformations into optimized versions. Through this meticulous process, we rigorously assessed the logical equivalence between the two query types and meticulously examined the outcomes of this transformation, ensuring that the resulting optimized queries maintained the same logical structures and functional outcomes as their unoptimized counterparts.

Our Project Query Analyzer is a central component of our project, playing a pivotal role in executing these critical verifications. It meticulously scrutinized the transformations and outcomes, ensuring that each step in the process adhered to maintaining the logical equivalence and functionality between optimized and unoptimized queries. This meticulous validation process highlights our dedication to ensuring not only semantic but also logical equivalence between different query versions, showcasing the robustness and uniqueness of our project in the domain of SQL query verification.

2 Survey - related work

Query equivalence validation is a must query optimization. A major part of query optimization comes from rewriting the initial query in a way that is likely to access fewer rows in the database, but still produces the same result. Some of the examples include usage of JOIN operation, subquery optimization, retrieving only needed columns instead of all (avoid Select * operation) wherever possible, which improves the query performance.

Query equivalence from a set perspective for SQL makes it a decidable problem. One such example implementation is the chase algorithm[2]. The algorithm focuses on conjunctive queries with data dependencies. It has successfully been used in databases, but there is a chance it does not terminate.

Following this, an algebraic method of SQL query equivalence emerged. COSSETTE solves equivalence by likening SQL relations to semirings[2]. This formulation is able to tackle more aspects of SQL, but lacks the ability to address foreign key constraints. This leads to the UDP algorithm using U-semirings implemented in this paper.

The algorithm extends standard commutative semiring by adding new axioms. SQL Queries are converted into U Semirings thereafter which equivalence is tested. Overall, there is little progress in SQL query validation of optimization methodologies[8]. Though validation is a stagnant topic in research, there is still a need for improvement[7].

2.1 Advantages:

- Rules consist of SQL queries with features such as subqueries, groupings, Distinct, aggregate and integrity constraints.
- The paper also lists that the UDP algorithm can automatically prove about 39 rules with each rule verification having a runtime of about 30 seconds.
- The paper highlights the validity of 62-real world SQL rewrites using UDP.
- The sqlglot library can also translate any query language into another allowing us to avoid ever changing database schemas and SQL changes[3].

2.2 Disadvantages:

- No support for SQL features such as CASE, UNION, NULL and PARTITION BY.
- Some of the queries require modeling the semantics of integer arithmetic, string concatenation and conversion of strings to dates.
- Rules are expressed using the Expressions class in sqlglot that allows robust definitions for each U-semiring axiom.

The above disadvantages are limited to the current version of UDP described in the paper.

3 Technical, algorithm, analysis, and implementation

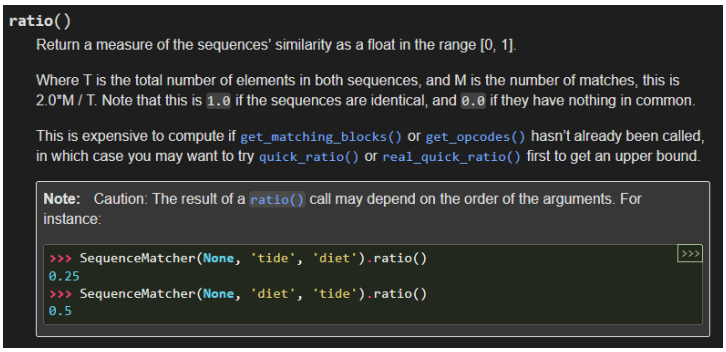
Our implementation was done to replicate their work on a common test database to show the new U-semiring helps create equivalent queries based on their implementation.
Our project achieved all of the following goals:

- Imports a complex DBMS (Sakila[5]) into Python-builtin Sqlite3
- Uses sqlglot[6] a python library that can allow manipulation of sql statements as objects
- Runs a list of queries given to it on the database to simulate real-world use
- Time the query performance by timing the average of 10 runs of each query
- Compares the optimized query results to the unoptimized query result

- Runs and lists required steps needed to turn the SQL unoptimized query into the optimized version.

3.1 Project Strategy

There are several experimental results we will gather from the project. One is a definition of difference between un-optimized and optimized queries. This is done via a Python module called SequenceMatcher.



The SequenceMatcher has the ratio() function which can compare string A to string B and report how 'identical' they are.

This is applied to our SQL queries after some basic formatting/standardizations to ensure the AST representation can easily be manipulated.

The paper's Axioms are essentially what we have already learned as a complete set for SQL manipulation. Unions, True/False, Associativity and Transitivity rules are in the paper and learned in class. In order to speed up the project we have relied on sqlglot to optimize the queries as much as possible before implementing the squash (|| ||), not (not(.)), and unbounded summation (Ɒ) axioms.

Another item we use to test our queries is by timing them using the Python timeit function. This allows us to run the query several times and see what the average timing is in order to get an accurate time difference.

The last item we use to implement the given paper is by having a simple 'game theory' design into the decision of what axiom to apply to the unoptimized query. The way this is done is before any manipulation step is done, the unoptimized query is checked to see how much it matches the optimized via the SequenceMatcher.ratio(). Then a step is made (either Tree manipulation like Select + Project = JOIN, JOIN translations, squash, not, and/or unbounded summation manipulation). Whichever step has increased the SequenceMatcher.ratio() the most is considered the best option for matching the optimized query. Then the next iteration is done until 100% matching is reached.

3.2 Algorithm Steps

- Load the Sakila database into python.
- Run the optimized and unoptimized queries that we created and compare the results to ensure they have equal output.
- We then run our analyzer on the two queries.
- The analyzer starts by creating two trees to represent the queries which will be used when rearranging parts of the query.
- The analyzer then uses sqlglot to find the difference between the unoptimized and optimized query and the steps needed to get from one to the other.
- It then goes through each step and checks if the changes can be applied according to the axioms in the paper.
- For each change it checks if we have come closer to matching the optimized query. If we have come closer to our goal we keep the change and modify our unoptimized query to add in the change. Otherwise, we do not add that change.

4 Evaluation and Key Findings

4.1 Assumptions

The main presumption of the project is that the all optimized queries tested are INITIALLY assumed to be logically equivalent to their unoptimized counterparts. Meaning that, both queries should yield the same set of results when executed by the DBMS.

The implementation of the project is done using Python 3.11 as the base coding language. The project is written in Python and SQL mainly.

The SQL database is a common, widely used database like Sakila which is a movie database. The database is chosen because it is a very interconnected database meaning any JOIN operations will be very computationally intensive.

The current project is able to import any database written in sqlite. The Sakila database is used in this case and also executes any amount of queries given in a text file. After database importing and sql query execution, it can analyze, manipulate, and make decisions on optimization of the unoptimized query. The project can take the optimized and unoptimized queries and output the AST steps required to take the unoptimized query and turn it into an optimized query. It also can time the optimized to unoptimized queries.

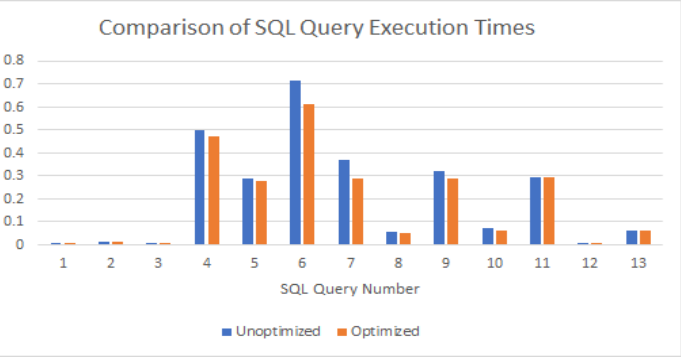
4.2 Evaluation

SQL Query Comparison: The evaluation process began with taking the Sakila SQLite database as the main testing schema. Then, SQL queries are provided from text files to be used in database importing and execution. Once completed, the project analyzes, manipulates, and makes decisions on optimization of the unoptimized query. The project then records the execution times between queries.

Query Manipulation: The project took a set of optimized and unoptimized queries from text files and provided an output AST steps which detail each specific step and the task to convert an unoptimized query into an optimized query.

4.3 Key Findings

SQL Query Comparison: On the graph below, we observed a comparison between the average execution time (in ms) between 13 unoptimized and optimized queries:



#	Unoptimized (ms)	Optimized (ms)	% Improvement
1	0.0019735	0.0019373	2%
2	0.011665	0.0119007	-2%
3	0.0020847	0.0017919	14%
4	0.4988237	0.4726008	5%
5	0.2917168	0.2797925	4%
6	0.7164865	0.6156745	14%
7	0.371002	0.2911933	22%
8	0.0544915	0.0492883	10%
9	0.322163	0.2911784	10%
10	0.0719033	0.0633082	12%
11	0.2954744	0.294301	0%
12	0.0081131	0.0070434	13%
13	0.0626803	0.0611728	2%
Avg	0.208352138	0.187783315	8%

From the timing results, our project is able to give us an average 8% improvement on all SQL queries.

Query Manipulation:

Unoptimized	Optimized
SELECT a.actor_id FROM actor a, film_actor fa WHERE a.actor_id = fa.actor_id;	**SELECT a.actor_id FROM film_actor fa, actor a WHERE a.actor_id = fa.actor_id;**

In the above example query, the only change is the FROM statement where we switch the table accesses. Note this is a simple example for demonstration, not a perfectly optimized query.

The program will run several different axioms on the query but the translation yields the best results:
Query can be optimized by TRANSLATION property:
Original Query Match to goal: 99.08%
Translated Query Match to goal: 100.00%

Finally, the sqlglot library is helpful in that it can output the total AST steps required for Tree manipulation to turn our unoptimized query into the optimized. The above program output is a result of recognizing the below steps as a translation of the database accesses:

```
Query 0
  ORIGINAL TIME: 0.0020468000002438203
  OPTIMIZE TIME: 0.0017217999993590638
  Basic translations and JOIN optimizations done to match
  SQL query optimized! Steps taken
     Steps:
     Step 0 match to goal: %86.87552921253176
     (SELECT expressions: (COL...s: fa, quoted: False)))))
     Step 1 match to goal: %74.76099426386233
     (SELECT expressions: (COL...s: fa, quoted: False)))))
     Step 2 match to goal: %86.29787234042553
     (SELECT expressions: (COL...s: fa, quoted: False)))))
     Step 3 match to goal: %100.0
     (SELECT expressions: (COL...s: fa, quoted: False)))))
```

The above shows 4 steps that will take the fa (film_actor) table and switch it with the a (actor) table. The sqlglot returns the steps in AST-form which is helpful for making tree-structures
For more complex queries, then the steps to optimize are printed out as remove/insert into a AST tree-structure:

```
Query 1
  ORIGINAL TIME: 0.010647799997968832
  OPTIMIZE TIME: 0.010321099998691352
     Step 0 match to goal: %95.53672316384181
     Remove(expression=(JOIN this:
  (TABLE this:
   ...
     Step 1 match to goal: %88.53850818677986
     Remove(expression=(WHERE this:
  (EQ this:
     (...
     Step 2 match to goal: %100.0
     Insert(expression=(JOIN this:
  (TABLE this:
   ...
```

5 Summary and Conclusions

Our research issue was focused on transforming queries into an optimized form and verifying semantic equivalence between optimized and unoptimized queries.

Our implementation was able to run pairs of SQL queries to ensure semantically correct behavior, time the average execution for optimized and unoptimized, manipulate the query to match the optimal statement, and show the steps (based on U-semiring implementation) required to complete the manipulation. Our research shows that the U-semiring described in this paper are good extensions of the original axioms done for SQL query manipulations.

6 Project Members Contributions

Chris Tsongas:

In phase 1, I researched the topics and provided the team with multiple topics to discuss on implementation details. Worked through several papers and captured the potential publications to research in Google Drive.

Discussed the potential hazards and read the chosen topic several times. Phase 2 through 3, I provided a basic task list and general direction of the program implementation. In these phases I was able to make 5 of the 6 items listed in the Research Implementation and Contributions section. In addition I also wrote the Phase 3 Algorithm and Implementation sections. Phase 4 I helped write and reduce the update to match the expectations of the submission size. Phase 5 I handled the paper writing, provided experimental results and contributed the final phase of the implementation into the project.

Ayooluwa Oladosu: Worked on researching current solutions on query optimizations and the limitations they have that our research paper hopes to fill. Also looked into other methods that have been used for query equivalence checking. Worked on documentation for the phases.

Wasey Mulla: Across the project phases, Wasey Mulla made significant contributions.In phase 1 , contributed towards finding and comparing good research papers to settle on a research paper to use for the implementation of the project.. In Phase 2, active collaboration with the team led to the development of a comprehensive project plan and implementation schedule. Phase 3 involved conducting extensive research on various papers, resulting in a well-structured project introduction encompassing its advantages, disadvantages, and enhanced document formatting. During Phase 4, Wasey coordinated with team members, ensuring updates on progress and refining the time management strategy. Lastly, in Phase 5, their focus shifted to completing the group research section, conducting extensive revisions across multiple report segments, and refining the report's structure and wording for improved coherence and presentation. In addition, worked on comparing the research paper to the group's contribution with additional implementations that we completed.

Uthama Kadengodlu: Worked on query un-optimization. Un-optimized queries were checked for equivalence with optimized queries and reported no errors on this goal. Also collaborated with the team on documentation through phases 2 through 5 consisting of surveys, advantages/disadvantages, document formatting.

Alejandro Urquieta: Worked on compiling, formatting, and submitting all project reports and status. Contributed with running and testing the project code in terms of query comparison. And contributed with documentation of Phases 3,4, and 5.

7 Project Information

7.1 Github Link

https://github.com/lctsongas/CS6360

7.2 Github Steps

To replicate the program's functionality, including cloning the repository, opening it in an IDE, and executing it, follow these steps:

Clone the repository to your local computer:

1. Open a terminal or command prompt.
2. Use the command “git clone https://github.com/lctsongas/CS6360” to clone the repository to your local machine.

Open the project in an Integrated Development Environment (IDE):

1. Launch your preferred IDE (e.g., Visual Studio Code, PyCharm, etc.).
2. Choose to open an existing project or folder within the IDE.
3. Navigate to the location where the repository is cloned (the CS6360 directory) and select it to open the project in the IDE.

Install dependencies:

1. Ensure Python3.11 is installed on your system.
2. Install the sqlglot library by executing ‘pip install sqlglot’ in your terminal or

command prompt. ‘sqlglot's path setting has to be taken care of such that the repository can locate it.

Run the program:

1. Once the project is opened in your IDE and the dependencies are installed, locate the src folder within the project directory in the IDE's file explorer or terminal.
2. Find the main.py file within the src folder.
3. Run the program by executing ‘python3 main.py’ in the terminal within the src folder.

Observe the program's output:

1. The program will begin processing queries and display its output in the terminal or console within the IDE.

7.3 Source Information

There are several files and tools used in the project. This section will outline their uses and tools used in/on them:

- **src/main.py** - Main python module that imports a SQL database, runs queries, determines if optimized and unoptimized queries are matching, and runs the paper’s axioms on the unoptimized to prove it is equivalent.
- **src/sqlglot_wrapper.py** - Wrapper script to allow us to easily interface with the sqlglot library to import, analyze, and optimize the queries.
- **src/sql_query_wrapper.py** - Object that takes optimized and unoptimized queries and allows program to manipulate them via the paper’s axioms
- **src/database_wrapper.py** - Simple interface to create database, manipulate, and run SQL queries on the Sakila database
- **src/base_db_builder.py** - wrapper class that is used by database_wrapper.py to allow generic functions to be shared
- **optimized_queries/optimized_query.txt** - list of optimized queries that are run on database to compare to unoptimized
- **optimized_queries/unoptimized_query.txt** - list of unoptimized queries that are run on the database that will be manipulated into the optimized in order to show the paper’s axioms work as expected.

References:

[1] S. Chu, B. Murphy, J. Roesch, A. Cheung, and D. Suciu, “Axiomatic Foundations and Algorithms for Deciding Semantic Equivalences of SQL Queries,” Very Large Data Base Endowment Inc. (VLDB), https://vldb.org/pvldb/vol11/p1482-chu.pdf (accessed Oct. 29, 2023).

[2] Michael Meier Michael Schmidt Georg Lausen, On Chase Termination Beyond Stratification, arXiv preprint arXiv:0906.4228, 2009

[3] S. Faroult, "The Art of SQL," 2006.

[4] A. Doan, A. Halevy, and Z. Ives, "Principles of Data Integration," 2012.

[5] B. Grant, “Sakila Sample Database - SQLite3 Port - Github Repository,” GitHub, https://github.com/bradleygrant/sakila-sqlite3/tree/main/source (accessed Oct. 2, 2023).

[6] T. Mao, Sqlglot API documentation, https://sqlglot.com/sqlglot.html (accessed Oct. 2, 2023).

[7] “Query featuring outer joins behaves differently in Oracle 12c,” Stack Overflow. http://stackoverflow.com/questions/19686262 (accessed Dec. 01, 2023).

[8] R. A. Ganski and H. K. T. Wong, “Optimization of nested SQL queries revisited,” ACM SIGMOD Record, vol. 16, no. 3, pp. 23–33, Dec. 1987, doi: https://doi.org/10.1145/38714.38723.