Optimising SQL Queries Using Genetic Improvement

James Callan
University College London, UK
james.callan.19@ucl.ac.uk

Justyna Petke
University College London, UK
j.petke@ucl.ac.uk

Abstract—**Structured Query Language (SQL) queries are ubiquitous in modern software engineering. These queries can be costly when run on large databases with many entries and tables to consider. We propose using Genetic Improvement (GI) to explore patches for these queries, with the aim of optimising their execution time, whilst maintaining the functionality of the program in which they are utilised. Specifically, we propose three ways in which SQL `JOIN` statements can be mutated in order to improve performance. We also discuss the requirements of software being improved in this manner and the potential challenges of our approach.**

Index Terms—**genetic improvement, SQL query optimisation**

I. Introduction

Inefficient SQL queries can be very costly, whether that cost comes from the parsing or execution of the query. However small changes to queries can offer performance benefits. For example, in the commit found at [1], we see a developer has removed a single redundant JOIN statement in order to improve the execution time of this SQL query. This removal may lead to a slightly different result from the query, however, no other code in the project was modified, suggesting that the wider program was ambivalent to the change.

Genetic improvement (GI) has been used to improve the execution time of many different forms of software [2]. Due to the diversity of applications improved by GI in the past, we believe that it can similarly be utilised in order to improve the execution time of SQL queries whilst maintaining their functionality. SQL optimisations would be particularly useful due to the wide array of software which utilise them. Software written in many different languages and used in many different settings could benefit from SQL optimisation.

Previous work optimising SQL queries has mostly concerned optimisations on the SQL server side, including optimising the execution schedule [3], row matching algorithm [4], compilation changes [5], and changes to server implementation [6]. These optimisations aim to be generic to many queries rather than finding optimisations for specific queries.

II. Proposed Approach

Our proposed approach is tailored towards improving queries which read from databases rather than those which write to them. This allows us to avoid changing the structure or contents of the database which could have wide-reaching effects to both other areas of code and even other instances of software, if the database is not local. Therefore, we suggest to only focus on `SELECT` queries.

Different search algorithms, such as genetic programming or local search, may be used to explore the query patch search space. The population should consist of patches containing a list of mutations to apply to the query being optimised [2].

In this work, we propose new mutation operators for modifying `JOIN` statements within `SELECT` queries, in particular.

A. Mutations

We believe that `JOIN` statements are the best targets for mutation as they can result in large operations, comparing the value of many database entries. `JOIN` statements are used to combine tables within the database, the rows of each table are combined based on matching values of particular fields. For example, an `INNER JOIN` presents the combined rows where a match is found, discarding those where no match is found. `OUTER JOIN` functions similarly but also returns rows where no match is found[1].

In order to target these statements, we propose three mutation operators for `JOIN` statements:

- **`JOIN` removal** Delete a `JOIN` statement.
- **`JOIN` reorder** Change the order of `JOIN` statements in a sequence.
- **`JOIN` type** Change the type of `JOIN` operation performed, the list of possible types are: `[INNER,OUTER,LEFT,RIGHT]`.

Sometimes, as seen in [1], unnecessary `JOIN` statements are present which add new fields to results that are not needed by the software, the **`JOIN` removal** mutation aims to remove these statements.

The **`JOIN` reorder** mutation aims to find the optimal order of `JOIN` statements in a sequence. Changing the order of sequential `INNER JOIN` operations will not alter the output, however depending on the data being compared in each operation it may offer performance benefits. `OUTER JOIN` order changes may alter the output of a query, however this may not affect the program making the query. In a series of `JOIN` operations, if the last join discards a significant number of records compared to others it is likely that performing it first will reduce the number of evaluations needed by subsequent

[1]Examples and explanations of each type of `JOIN` statement can be found at http://www.sql-join.com/sql-join-types

joins. Unoptimally ordered `JOIN` sequences can be orders of magnitude more expensive than optimal ones [3]. This mutation operator mimics optimisations used on SQL servers. Servers attempt to find optimal join order when they receive a query [3]. However, this optimisation lengthens the execution time of the query. Cost estimates are used to save time which may introduce errors. Using GI to find these optimisations will allow decisions based on real measurements and remove the cost of optimisation at runtime. The `FORCE ORDER` flag should also be modified to ensure that the server does not override the newly selected order.

The **`JOIN` type** mutation allows changes to the number of rows returned by `JOIN` operations. Improvements would most likely come from reducing the amount of data being transferred. Changing from `OUTER JOIN`s to `INNER JOIN`s or `FULL JOIN`s to `LEFT`/`RIGHT JOIN`s could reduce the number of rows returned by the operation.

Genetic operators which have been used in previous work [2] (e.g. `delete`, `copy`, and `replace`) may also prove useful when applied to the SQL code of a project. The operators could be applied at, e.g, line or statement level.

B. Testing

We believe that variants should be tested in the context in which they are utilised by software, rather than attempting to reproduce the exact values returned by the original query. A mutant query may not return the same fields and format as the original query, yet the program utilising this query may be ambivalent to the differences. It, therefore, may be more useful to run the unit or integration tests of code which call and use the results of queries, rather than directly testing the query. This will allow flexibility but still ensure preservation of functionality.

C. Measurement

The measurement of the execution time of mutant queries will vary depending on the particular database in use. Databases such as SQL-Server[2] automatically measure and report the CPU time on the server, this however may neglect the cost associated with I/O of the program making the request.

Another option would be to simply time the execution of the test suite. This would capture all associated costs of the query, however would be subject to more noise than CPU time due to other processes running on the device.

III. Challenges

One of the main challenges in validating this approach would be finding software with well-tested queries and extracting the areas of the test suite which test these queries. Manual evaluation of existing test suites may be necessary to determine whether a piece of software can be safely improved. Test suites may also need to be expanded to better exercise query results. Automated test suite generation tools, e.g. EvoSuite [7], could be useful for enhancing test suites.

[2]https://www.microsoft.com/en-gb/sql-server/sql-server-downloads

Another concern is that the optimised queries would be biased towards the contents of the database on which they are tested. Factors like the number of records in each table should be representative of the real database in use by the software. If it is not representative, the improvements found may not translate into real-world improvements. A copy of it could be tested against to ensure that the optimisations are effective.

Databases are not static, entries and tables are regularly added, removed, and modified. Therefore, optimisations which are effective at one time may not necessarily generalise throughout the lifespan of the database. Multiple copies of databases from different time periods could be used and average measurements taken to ensure that optimisations generalise and are not biased to the database at a particular time.

This approach is limited to `SELECT` queries and focuses on improving the speed of their `JOIN` operations. However in the future, it could be expanded to improve database writing queries. This would require testing to verify that the structure and contents of the database have been preserved between the original query and improved variants.

Optimisation with respect to execution time may have unintended consequences on other properties of the software being improved. Larger query results may lead to higher memory and bandwidth usage. Multi-objective optimisation, such as the NSGA-II algorithm [8], could be used to find the best trade-offs between these properties.

IV. Conclusions

SQL queries offer a good target for optimisation, a technique to improve their execution time could have far-reaching consequences on many types of software in many environments. We have presented a technique which could find optimisations by targeting often inefficient `JOIN` statements. We intend to verify this approach with empirical analysis.

Acknowledgment

Funded by the EPSRC fellowship EP/P023991/1.

References

[1] "SQL JOIN removing commit." [Online]. Available: https://github.com/erikusaj/fdroidTvClient/commit/620affa239941c764cd5a132bb687857315c744d

[2] J. Petke, S. O. Haraldsson, M. Harman, W. B. Langdon, D. R. White, and J. R. Woodward, "Genetic improvement of software: A comprehensive survey," *IEEE Trans. Evol. Comput.*, vol. 22, no. 3, pp. 415–432, 2018.

[3] V. Leis, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann, "How good are query optimizers, really?" *Proc. VLDB Endow.*, vol. 9, no. 3, pp. 204–215, 2015.

[4] K. Nakabasami, H. Kitagawa, and Y. Nasu, *Optimization of Row Pattern Matching over Sequence Data in Spark SQL*, ser. Lecture Notes in Computer Science. Springer, 2019, vol. 11706, pp. 3–17.

[5] F. Schiavio, D. Bonetta, and W. Binder, "Dynamic speculative optimizations for SQL compilation in apache spark," *Proc. VLDB Endow.*, vol. 13, no. 5, pp. 754–767, 2020.

[6] M. Zhai, A. Song, J. Qiu, X. Ji, and Q. Wu, "Query optimization approach with shuffle intermediate cache layer for spark SQL," in *IPCCC*. IEEE, 2019, pp. 1–6.

[7] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *SIGSOFT FSE*. ACM, 2011, pp. 416–419.

[8] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Trans. Evol. Comput.*, vol. 6, no. 2, pp. 182–197, 2002.