Verified SQL Query Engine in Dafny

Yan Liang (925551485) Muyang Zheng (925564003)

ECS 261 — Project Proposal

1 Introduction and Problem Selection

1.1 Problem Domain and Motivation

SQL is the most widely used language for querying and manipulating structured data. Database engines such as MySQL, PostgreSQL, and SQLite implement SQL query operations including filtering (`WHERE`), projection (`SELECT`), sorting (`ORDER BY`), and aggregation (`GROUP BY`). These operations form the backbone of virtually all data-driven applications, from business analytics to scientific computing.

Despite their ubiquity, bugs in query processing can lead to silent data corruption, sometimes returning incorrect results without any error message. Such bugs are especially dangerous because users typically trust query results without manual verification. In practice, subtle errors in filtering logic, sorting stability, or aggregation computation can go undetected for long periods, leading to flawed business decisions or corrupted data pipelines [1].

Formal verification offers a solution: rather than relying on testing which can only check finitely many inputs, we can use tools like Dafny to *mathematically prove* that query operations produce correct results for *all possible inputs* [2]. Amazon Web Services has successfully applied Dafny to formally verify their cloud-scale authorization engine, demonstrating the practical viability of this approach [3].

1.2 Program

We will build a minimal in-memory SQL query engine entirely in Dafny. The engine operates on a table represented as a sequence of records (rows), where each record contains integer-typed fields. The engine supports four core query operations:

1. **Select** — project specific columns from each row
2. **Where** — filter rows by a condition
3. **OrderBy** — sort rows by a specified column
4. **GroupBySum** — group rows by a column and compute the sum of another column

The input is a table (sequence of records) and a query operation. The output is a new table that is the result of applying the operation. A good solution means that Dafny's verifier accepts all postconditions, i.e., the correctness properties are mathematically proven at compile time.

1.3 Scope

Each of these operations is well-understood and can be implemented straightforwardly in any general-purpose language such as Python. We will be taking known algorithms and formally verifying their correctness using Dafny.

1.4 Specifications of Interest

We are primarily interested in **functional correctness** of each query operation. Specifically, we aim to verify the following properties:

Where (filter): Given a table and a condition, the output must satisfy:

1. Every row in the output satisfies the condition.
2. Every row in the output exists in the input.
3. Every row in the input that satisfies the condition appears in the output.

As a concrete example:

```
method Where(data: seq<Record>, minAge: int)
    returns (result: seq<Record>)
  ensures forall r :: r in result ==> r in data
  ensures forall r :: r in result ==> r.age >= minAge
  ensures forall r :: r in data && r.age >= minAge
              ==> r in result
```

Select (projection): The output has the same number of rows, and only the selected fields are retained.
OrderBy (sorting): The output is sorted in non-decreasing order by the specified column, and the output is a permutation of the input (no rows lost or added).
GroupBySum (aggregation): Every distinct group key from the input appears in the result, and the sum associated with each group equals the actual sum of that column for all rows belonging to that group.

All of the above are functional correctness properties verified at compile time.

2 Implementation Plan

2.1 Tool and Language Selection

Our project will be implemented entirely in **Dafny**. We will use the Dafny VSCode extension for development and the Dafny CLI for final verification.

- It is the primary verification tool covered in this course.
- It supports built-in sequence and map types that are well-suited for representing tables and aggregation results.
- Its auto-active verification approach (preconditions, postconditions, loop invariants) aligns directly with our goal of proving functional correctness.

2.2 Architecture of Code

Our codebase will be organized into the modules in Table 1. Each module is independent: it imports only `DataTypes.dfy` and exports a single verified method. This modular design allows us to work in parallel without merge conflicts.

2.3 Verification Effort

All four query operations will be verified at compile time using Dafny's built-in verifier. The key verification artifacts are:

- **Postconditions** (`ensures`) on each method specifying the correctness properties listed in Section 1.4.

File	Responsibility
`DataTypes.dfy`	Record datatype, validity predicates, helper functions
`Select.dfy`	Select (projection) implementation and verification
`Where.dfy`	Where (filtering) implementation and verification
`OrderBy.dfy`	OrderBy (sorting) implementation and verification
`GroupBySum.dfy`	GroupBySum (aggregation) implementation and verification
`Main.dfy`	Example queries demonstrating all operations

Table 1: Planned code architecture of our project.

- **Loop invariants** (`invariant`) for each iterative implementation to guide Dafny's automated reasoning.
- **Helper lemmas** where needed, particularly for the permutation property in `OrderBy` and the summation correctness in `GroupBySum`.

`Main.dfy` will *not* be verified. It serves only to run example queries and demonstrate the output. We assume nothing about external behavior, as our entire codebase is self-contained in Dafny.

2.4 Challenges

We anticipate the following challenges:

1. **Permutation proof for OrderBy.** Proving that the sorted output is a permutation of the input (same elements, same multiplicities) is relatively difficult in Dafny. We plan to use a multiset-based approach: showing that `multiset(result) == multiset(data)`. If this proves too difficult, we will fall back to proving only that the output is sorted and has the same length, and document the gap.
2. **Summation correctness for GroupBySum.** Proving that the computed sum for each group equals the true sum requires an inductive argument over the sequence. We plan to write a recursive helper lemma. If we cannot complete this proof, we will use an `assume` statement and clearly document it in the final report.
3. **Dafny's sequence reasoning.** Operations like membership testing (`r in seq`) and quantifiers (`forall`) over sequences can require careful invariant design. We will go through the Dafny reference manual and the Leino textbook [2] for established patterns.

3 Development and Timeline

3.1 Team Roles

- **Yan Liang**: Responsible for `DataTypes.dfy` and `Select.dfy`. Lead the implementation of `OrderBy.dfy` and the multiset-based permutation proof.
- **Muyang Zheng**: Responsible for `Where.dfy` and `GroupBySum.dfy`. Lead the integration and demonstration in `Main.dfy`.

3.2 Expected Timeline

Our expected timeline is demonstrated in Table 2.

Week	Dates	Milestones
Week 1	Feb 16 – Feb 22	Define `DataTypes.dfy`; complete and verify `Select.dfy` and `Where.dfy`.
Week 2	Feb 23 – Mar 1	Implement core logic for `OrderBy.dfy` and `GroupBySum.dfy`; establish initial loop invariants.
Week 3	Mar 2 – Mar 8	**Verification Intensive**: Complete the multiset permutation proof and recursive summation lemmas. Complete report.
Week 4	Mar 9 – Mar 12	Finalize `Main.dfy` demonstrations; conduct code cleanup and submit the final report.

Table 2: Weekly project timeline.

3.3 Commitment Statement

I, Yan Liang, commit to work and contribute equally to this project.
I, Muyang Zheng, commit to work and contribute equally to this project.

4 Conclusion

Upon completion, our project will demonstrate that core SQL query operations can be formally verified to produce correct results for all possible inputs.

We believe this project is both practically relevant and educationally valuable. SQL is the dominant language for data manipulation, and data correctness is a critical concern in any data-driven organization. By building a verified query engine, we gain hands-on experience with the challenges of formal verification applied to a domain we are already familiar with.

One open question we have is about the best strategy for the permutation proof in `OrderBy` — we would appreciate any guidance on whether the multiset approach is the recommended pattern in Dafny, or if there is a simpler alternative.

References

[1] Donald R. Slutz. Massive stochastic testing of SQL. In *Proceedings of the 24th International Conference on Very Large Data Bases (VLDB)*, pages 618–622, 1998.

[2] K. Rustan M. Leino. *Program Proofs*. MIT Press, 2023.

[3] Aleks Chakarov, et al. Formally Verified Cloud-Scale Authorization. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, 2025.