Reflection

Building the MiniQuery compiler helped our team move from theoretical understanding of compiler phases to actually implementing them in a connected pipeline. Writing each phase ourselves—lexer, parser, semantic analyzer, IR generator, optimizer, and interpreter—made it clear how tightly coupled these stages are and how a small mistake early on can break everything that follows. The project also taught us how important it is to design a language that is simple enough to implement yet expressive enough to demonstrate all required concepts.

What We Learned

- How to design a clean and minimal DSL with well-defined grammar and semantics.
- The practical interaction between AST generation, symbol tables, and IR.
- How basic optimizations like constant folding and dead code elimination actually work on a real program.
- The importance of modular code, since every compiler phase depends on correct output from the previous one.

Challenges Faced

- Keeping the grammar unambiguous while still supporting all required features.
- Ensuring semantic rules were correctly enforced, especially type consistency across queries and aggregations.
- Making the optimizer safe—removing code without accidentally changing program behavior.
- Debugging parser errors, which often required checking multiple earlier phases.

What We Would Improve Next Time

- Add small language features like conditionals or scoped variables to make the DSL more expressive.
- Improve error messages to be more user-friendly and helpful during debugging.
- Expand the optimizer with more analyses (CSE, more folding, better propagation).
- Refactor the runtime so new operations can be added more easily.

Overall, the project gave us a solid, hands-on understanding of how compilers work in practice and showed us how each phase contributes to a working language implementation.