Abstract
QueryBridge addresses challenges faced by data scientists and database researchers by providing a comprehensive solution to convert between different execution formats. QueryBridge can apply high-level optimisations to DataFrame-API (Pandas) code, giving data scientists a better development experience over SQL. This also provides database researchers with a fair comparison of a databases’ query planning abilities. QueryBridge can also produce execution code in a compiled language which is significantly faster than Pandas, solving data scientists’ performance issues with DataFrame-API languages. Finally, QueryBridge offers universal and language-level optimisations that enable the compiled language to outperform a state of the art database by a substantial margin.

1. Research Problem and Motivation
This work addresses the requirements of two groups of users of database systems: QueryBridge is positioned to meet the needs of data scientists and of database researchers. Data scientists write SQL queries to process data on a daily basis. Unfortunately the SQL development experience, how it feels to write and interact with SQL, is rather poor [2; 10]. SQL does not offer intermediate results, which might help troubleshoot errors and determine correctness, nor does it have debugging support [5]. Therefore, data scientists often resort to using Pandas, a DataFrame-API language in Python syntax that can be used to write data processing code [9].

Pandas is used by millions of data scientists every day [22], but it falls short in some areas, such as performance. Pandas does not perform any high-level optimisations, meaning it does not reorder or rewrite the instructions provided to it. Furthermore, it is also an interpreted language, and in the context of database query processing, this results in poor execution time [16]. These two problems, the usability of SQL and speed of Pandas, are a significant source of challenges for data scientists.

The second set of challenges are those encountered by database researchers. Developing database optimisations is a complex and involved task, not helped by the fact that making comparisons between databases is challenging [3]. The internal components of a database each greatly contribute to their runtime, so a comparison on each component is required to drive progress. Unfortunately, there exist no tools that allow fair comparisons between internal components and commercial alternatives.

I have developed a tool, QueryBridge, that will address these issues by bridging the gap between execution formats. It takes query plans from database systems and generates execution code that adheres to these plans. It uses a universal representation (UPlanIR) to represent operations of the plan and converts between representations using concrete transformation rules. By using QueryBridge, I aim to address the following research questions, which would have been difficult to tackle without its support:

RQ1. Can high-level optimisations be applied to a DataFrame-API language to improve runtime?
RQ2. What is the impact of using a compiled language over a DataFrame-API language for database query processing?
RQ3. To what extent do QueryBridge’s optimisations cause an improvement in execution speed?

2. Background and Related Work
In this section, I provide background information about important concepts in my research. Section 2.1 discusses database query processing, while Section 2.2 examines prior work related to QueryBridge.

2.1 Background on Database Query Processing
A database takes as input an SQL query and returns records from a database that meet the conditions specified in the query. However, this SQL query does not describe how to retrieve the information, just what information is desired. Thus, the database must first plan and then execute the query.

For a given query, multiple entirely equivalent plans are possible due to the equivalences in relational algebra operators. Therefore, the query planner’s job is to find the execution plan with the lowest cost. However, this task is
NP-hard, so query planners use estimation techniques and heuristics to limit the search space [15]. To execute the plan, a job done by the query engine, an interpreted approach has long been considered optimal, as far back as System R in 1979 [1]. However, to mitigate the CPU-bound nature of modern query processing [20; 21], query compilation has become a focus of research interest.

2.2 Related Work
Mutable is a database system with a query engine that offers JIT compilation and optimisation capabilities. This gives it performance akin to both an interpreted engine as well as a compiled one [3]. They use off the shelf components, like WebAssembly as an intermediate representation and Google’s V8 as a backend, in order to avoid long development times and reduce the cost of adoption. Mutable was built as a research project, but offers extensive documentation and testing infrastructure on GitHub. It also comes with an online benchmark visualisation site, that can help researchers detect performance regressions as well as carry out A/B testing of new performance optimisations. This project offers the A/B testing of components that database researchers desire; however, unlike QueryBridge, it cannot compare against existing database systems.

LingoDB is another research database system that uses query compilation; however, in contrast to Mutable it uses MLIR as an intermediate representation [6]. As compiler frameworks like LLVM have grown, the task of defining IRs has grown increasingly complex. However, tools like MLIR simplify dialect construction, bridging the gap between SQL and LLVM. Unlike Umbra [13] and QueryBridge, which have developed their own IR, LingoDB leverages MLIR. Jungmair et al. highlights MLIR’s benefits for query compilers, enabling cross-domain optimisation efforts with existing infrastructure like predefined optimisation passes. They found it reduced implementation effort while maintaining high performance and low compilation latencies.

Owing to its open and versatile nature, the execution backend behind LingoDB has also been used to improve the speed of Pandas code. Imschweiler [4] demonstrated a system for parsing Pandas operations into MLIR, then using the LingoDB backend to execute them. Despite the compilation cost of the MLIR backend versus Pandas’ interpreted nature, they found a significant increase in performance, up to a 14.6x reduction in runtime. However the project does not exist in a very mature state, it only supports enough Pandas operators sufficient to run 3 out of the 22 TPC-H queries.

3. Approach and Uniqueness
QueryBridge uses a compilation pipeline as depicted in Figure 1. Section 3.1 elaborates on the design decisions involved. Subsequently, Section 3.2 outlines the optimisations provided by the QueryBridge framework.
The QueryBridge framework offers 4 different optimisations:

- **Fusion**: leverages information from the database (insights about aims to reduce the cost of populating new data structures.
- **Update to Insert**: materialises results less frequently. Further, it is expected that as records are being kept in the CPU register for longer and are fused together, the pipelining of the code should improve, independently of each other. When non-blocking operators processing performance by maximising the locality of the data and the code. Inspired by work from Neumann [12] and the existing literature on loop fusion [8; 19], my implementation of the Fusion optimisation combines subsequent operators that are non-blocking (i.e. they can processes child records independently of each other). When non-blocking operators are fused together, the pipelining of the code should improve, as records are being kept in the CPU register for longer and the processor is accessing memory to retrieve new records or materialise results less frequently. Further, it is expected that the code locality will increase as the optimised code features small code fragments that are working in tight loops against a large amount of data.

- **Data Structure Specialisation**: this translates database operations into MLIR, employing the ‘relalg’ dialect and leveraging MLIR’s existing passes to generate highly efficient machine code.

However, I already have predefined execution formats I intend to use, Pandas and SDQL.py. Hence, I would derive no advantage from MLIR’s transformations and optimisations. Parsing into and out of MLIR would only result in significant overhead without yielding any tangible benefits. Another argument against using a compiler framework is that the domain of the IR is relatively simple. In my finalised UPlanIR, I have just 9 different edge plan nodes; there would be little utility in having the full expressive power of the dialects offered in compiler frameworks – for just nine nodes. Finally, the back-end optimises the execution IR before converting it into raw execution code.

### 3.2 QueryBridge Optimisations

The QueryBridge framework offers 4 different optimisations: **Fusion**, **Update to Insert**, **Data Structure Specialisation** and **Early Projection**.

**Fusion** is an optimisation that can be applied at the universal level (UPlanIR). This aims to remove unnecessary relation attributes, reducing run-time through shrinking intermediate result sizes.

**Update to Insert** is another optimisation I offer, this one aims to reduce the cost of populating new data structures. Leveraging information from the database (insights about which keys are primary, and therefore unique), I can switch from using aggregation to assignment in specific cases. This will avoid the need to test for existence in a hash map, and simply place the record for a new key – which is expected to contribute to faster code execution.

The **Data Structure Specialisation** optimisation focuses on selecting the most efficient data structures for storing intermediate results. When dealing with highly selective selection conditions (i.e. they are non-restrictive), using an array rather than a hash map can be more advantageous due to fast access times that do not incur the overhead of a hash function. Shahrokhi et al. [18] examined this approach under highly permissive and very restrictive selection conditions. They found that dense arrays created after conditions with low selectivity suffer from significant memory management overhead, reducing efficiency due to the allocation cost of unused array space. In my optimisation, I will introduce a parameter to determine the applicable conditions.

Finally, **Early Projection** is an optimisation that can be applied at the universal level (UPlanIR). This aims to remove unnecessary relation attributes, reducing run-time through shrinking intermediate result sizes.

### 4. Results and Contributions

In this section, I will present the experimental results (Section 4.1) and contributions of QueryBridge (Section 4.2).

#### 4.1 Experimental Results

All experiments were performed on an Intel Core i7 CPU with 16GB of RAM. The TPC-H benchmark [23] was used; for queries involving SDQL.py [18], they were modified to remove ordering and sorting (a limitation of the SDQL.py language). Following prior work [6; 18], each candidate underwent five recorded runs after an initial warm-up run.

**RQ1:** Can high-level optimisations be applied to a DataFrame-API language to improve runtime? Figure 4 shows the effect of using different query planners for the query engine, Pandas. It can be seen that query planners like Hyper DB beat the Human by 2.25 times. This clearly demonstrates that a DataFrame-API language like Pandas can benefit from high-level optimisations. Furthermore, Figure 4 also serves as an apples-to-apples comparison of the databases, since all three databases have their query plans run in a single query engine. This comparison can markedly help database researchers, as they are able to easily compare new optimisations to the state of the art.

**RQ2:** What is the impact of using a compiled language over a DataFrame-API language for database query processing? Figure 5 presents runtime differences of query engines when they follow the same query plan – Hyper DB was chosen, given its leading performance (Figure 4). Switching from a DataFrame-API language (Pandas) to a compiled language (SDQL.py) nets a 29.6% reduction in average runtime. While...
Figure 4: Results for each query planner, all using the same query engine, Pandas, and generated through QueryBridge. Geometric mean presented in red on the right-hand side, lower is better.

Figure 5: Results for each query engine, all using the same query planner, Hyper DB, and generated through QueryBridge. Geometric mean presented in red on the right-hand side, lower is better.

significant, the query engine used in Hyper DB is still 1.62 times faster than our unoptimised (naive) SDQL.py.

RQ3: To what extent do QueryBridge’s optimisations cause an improvement in execution speed? Figure 5 also shows the optimised SDQL.py runtime; this caused a runtime reduction of 38.2% over state of the art (Hyper DB). The optimised SDQL.py runtime includes all optimisations outlined in Section 3.2, but their impact varies. Fusion, the most consequential optimisation, reduces the runtime against naive SDQL.py by 44.0%. Fusion diminishes the number of iterations on the data, with fewer unnecessary iterations the queries run faster. Optimisation Update to Insert improves execution time by reducing the cost of adding data to a data structure. It brings a 1.41% reduction compared to naive SDQL.py, but when combined with Fusion, the reduction becomes more substantial (1.78%). This is because Fusion maximises opportunities for merging selection operators into the same pipeline as the build side of a hash join.

Again, the Data Structure Specialisation optimisation also benefits greatly (6.30% instead of 1.22%) from being combined with Fusion. This is because Fusion creates more candidate locations for the optimisation to be applied. Data Structure Specialisation also has a parameter to control at what selectivity threshold to use an array or hash map. After a hyperparameter search, the best parameter (0.4) was found to offer an 8.78% reduction in runtime (over Fusion SDQL.py). However, given the limited range of selectivities in the TPC-H benchmark, further study is required to achieve a fuller picture. The final optimisation, Early Projection, reduces runtime by 20.7% in SDQL.py and does not have a significant effect in Pandas. This likely stems from Pandas’ columnar memory management approach, suggesting that optimising join orders, as discussed in RQ1, is one of the few effective methods for decreasing Pandas runtime.

4.2 Contributions

For data scientists, a key issue is that SQL has a very poor development experience. QueryBridge converts these SQL queries to Pandas, using the planning decisions of databases; this allows data scientists to use all the development features that Pandas offers. The converted Pandas are also on average 2.25 times faster than the code that a data scientist may write. A second issue I raised, that data scientists often lament is the fact that while Pandas is readable, it fails to address their performance needs – often databases themselves are quicker. Solving this, my framework can also convert SQL queries to SDQL.py (a compiled language). My experiments have shown that SDQL.py is very fast, offering a near 30% reduction in runtime versus Pandas. Further, my framework also offers optimisations, that when applied cause SDQL.py
to run 1.34x times faster than a state of the art database. Through QueryBridge, a data scientist with an existing SQL workload can inspect and understand it in the more readable Pandas, then execute it in the significantly faster SDQL.py.

Finally, my framework also addresses the challenges faced by database researchers. I explained in the Introduction how accurate comparisons of individual components in a database are needed to drive progress, yet no tools exist for this. QueryBridge enables query plans from different databases to be run in a neutral query engine. This means that database researchers can easily compare their modifications in an A/B fashion or to existing database competitors. This comparison of query planners showed a 41.0% difference in runtime between the Hyper DB and PostgreSQL plans. Hence, for a database researcher, QueryBridge offers an attractive mechanism for obtaining real-world comparisons between different components of a database.

5. Conclusion

QueryBridge facilitates the transformation of query plans into various execution formats, incorporating high-level optimisations that surpass an expert human. It supports outputting to compiled languages, resulting in notable performance gains, while also applying both universal and language-specific optimisations that outperform existing state of the art databases.

Future work could extend the existing comparison of query engines. While QueryBridge offers two options at the moment (Pandas and SDQL.py), adding existing databases as query engines would be valuable for database researchers. Projects like Substrait [11], an open-source query plan format, make this goal achievable. Comparing query engines would enable database researchers to get significantly more insight about where they stand versus their contemporaries. For data scientists, this may also lead to discoveries of query engines that are faster than what is current offered in QueryBridge – further enhancing their workflow.

In conclusion, QueryBridge and the work surrounding it aim to improve the usability and efficiency of database systems for both researchers and data scientists, fostering further advancements in the field.

References


