Data compression is a unique tool for optimizing resources and is widely used in computing systems to reduce memory footprints. However, Database Managefornoptimisings (DBMSs) use compression techniques not only for memory reduction but also to optimize the runtime performance of the system by performing on compressed data. DBMS algorithms, such as Vertipaq, efficiently store compressed columnar data and enable data processing on the compressed data by vastly enhancing the runtime performance on data with a high compression ratio. Such optimizations do not exist in typical compilers, processing data in an uncompressed format even if they could benefit from similar algorithms in some cases. Identifying Intermediate Representation (IR) patterns that operate on columnar data is crucial to enable such compiler optimizations. Compiler optimizations transform the IR by recognizing small IR patterns; for instance, inlining optimization matches a function call and decides if the optimization will be applied based on the function body.

We propose a semi-declarative pattern-matching framework that could recognize large IR patterns, such as columnar operations. Compression optimization transformations create compressed data structures to enhance the runtime performance. Data compression for every query adds significant overhead, so we leverage data memoization to store the compressed data, mapping them to their uncompressed structures. We implement our pattern-matching framework on the Graal compiler, leveraging Graal's Direct Acyclic Graph (DAG) IR to navigate the graph and apply our transformations. Our framework allows efficient compression and memoization for query execution without decompression, as we allow operations on the compressed data. Our system leverages lightweight compression algorithms that do not modify the nature of the data when compressing them, and the system could still operate on them. More specifically, for our experiments, we implement Run Length Encoding, a lightweight compression algorithm that uses two values to represent big runs of data containing a single character.

Applying our optimizations on JAVA programs, we achieve up to 11x optimization in execution time for queries that involve filter and aggregation operators, such as the TPC-H 6 query. We only apply optimization to columns with high compression ratios for such an experiment. Thus, our framework allows the system to operate on compressed and uncompressed columns for the same query. We also tested our framework on Apache Druid, a data analytics system, where it achieved a 2x performance speedup on aggregation queries. We achieve this speedup by reducing both the aggregation runtime and the memory copy overheads of the system. However, we cannot achieve the same speedup with the Java programs due to system overheads and data structures. Nevertheless, these results demonstrate a significant performance improvement achieved by compiler compression optimization. The demonstrated approach shows opportunities to apply compression in less traditional areas than the one that benefitted from such optimizations.