Fully Compiled Execution of Conjunctive Graph Queries

van de Wall, A.A.G.

Award date:
2020

Link to publication

Department of Mathematics and Computer Science
Databases Research Group

Fully Compiled Execution of Conjunctive Graph Queries

A.A.G van de Wall

Supervisor:
N. Yakovets

Assessment Committee:
N. Yakovets
G.H.L. Fletcher
R.M. Carvalho

Eindhoven, March 2020

Fully Compiled Execution of Conjunctive Graph Queries

A.A.G VAN DE WALL

As the available amount of main-memory grows, the performance of query evaluation is increasingly determined by the CPU costs. In the classical iterator model, performance is poor due to lack of locality and branch mispredictions. Several techniques such as processing multiple tuples at one in batches, or data-centric compilation have been proposed and shown to be an effective solution in the relational database domain. In this work, we apply these techniques to the graph database domain, and show that both techniques are reasonable solutions. Furthermore, we propose an abstraction model for reasoning about compiled query plans, and show how a query can be compiled in its entirety using this model. Finally, we show how operators such as merge-join, which don't map to data-centric code very well, can still be mapped to this model. We perform an experimental evaluation on a large real-world graph to confirm the effectiveness of our proposed approach.

1 INTRODUCTION

Graph databases have become increasingly popular in many application domains recently. For example, they are used to store and query complex heterogeneous biological data [19], perform social network analysis [18], store and query graphs containing encyclopedic data (knowledge graphs), and other workloads that query the underlying graph structure. Graph databases provide significant performance advantages over traditional, relational, databases in such situations [7].

We are primarily interested in the evaluation of sub-graph queries. Such queries are generally described in a high-level declarative language – such as SPARQL or RPGA, in which graph pattern are defined. The set of sub-graphs on which the pattern of a query matches is its output.

In this work, we are primarily interested in the part of a database management system (DBMS) that executes the queries: the execution engine. Before a query is handed over to it, the query first goes through a parsing and a planning stage.

Traditionally, the execution engine interprets the plan to execute. An operator tree is laid out in memory, and the engine walks over it executing a bit of code associated with every operator type in the tree. Such interpreting engines primarily focus on minimizing the amount of I/O operations, as those have historically been the dominating factor in execution time. Due to limited availability of main-memory, most of the time is spent on moving data from disk up the memory hierarchy.

However, as the attainable I/O-performance has grown with the introduction of SSDs, and the ever increasing amount of main-memory available, I/O operations are no longer necessarily a dominating factor. With sufficient memory available, entire data sets can now fit in memory. This requires different design considerations when building an execution engine, as the dominating factor moves up the memory hierarchy, and memory caches have essentially taken the place of the main-memory bottleneck-wise.

A very common way to implement query execution is to use the Volcano model. In this model, every operator implements an iterator interface that emits a single tuple at a time. This abstraction is however a potential performance bottleneck, as passing this standardized interface has some associated cost, and this needs to be done many times for every single tuple.

A common way to work around this bottleneck is Vectorization. When it is used, operators no longer emit a single tuple at a time, but instead emit large blocks of tuples. To keep the extra materialization costs originating in copying from and to such blocks low, additional tricks such as storing tuples column-wise are needed. However, they come with additional optimization opportunities.

Rather than minimizing the cost of the operator-tree abstraction, we can also completely get rid of it by generating code for it. This effectively translates the operator-model into a lower level code model, that a compiler can optimize more freely. Depending on the optimizations performed, compilation may however be costly, requiring caching or other measures to reduce its impact. Code generation is also fairly complex to implement, maintain and debug.

Research Questions. The application of these different execution-engine strategies has been researched extensively on relational databases. This can't be said for execution engines of graph databases. Therefore, we try to answer the following: 1) What performance impact does vectorization have on executing graph queries? 2) How can compilation be implemented effectively in a graph database? 3) And finally, how does compilation affect the performance of graph query execution?

As we answer these research questions, we furthermore contribute the following:

(1) We implement a fully vectorized execution engine in a high-performance in-memory graph database (Section 3.2),
(2) We propose a model for reasoning about generated code for arbitrarily complex query plans (Section 3.3.2) – allowing the compilation of complete query plans,
(3) Furthermore, we describe how operators such as merge-join can be implemented in the data-centric code generation process (Section 3.4.5),
(4) By using the proposed model, we implement a fully compiled execution engine in a high-performance in-memory graph database (Section 3.3),
(5) We perform an extensive evaluation of the effects of vectorized vs. compiled query executions on a large real-world network (Section 4).

The rest of this work is organized as follows: Section 2 provides an overview of preliminaries and background information. In Section 3, we show how vectorization and compilation were implemented. Section 4 provides an evaluation of the different implemented techniques. We then compare our implementation to related work in section 5. And finally we conclude in Section 6

2 BACKGROUND

2.1 Query Semantics

The goal of a graph database is to answer queries with information from a graph. In this work, we consider queries on property graphs.

Which are graphs where every vertex and edge have are assigned set of labels and key-value pairs.

Such graphs are defined formally by the tuple (V, E, η, λ, v). Where the disjoint sets V and E are the vertices and edges respectively. The function $\eta : E \longrightarrow V \times V$ assigns an ordered vertex pair to every edge – the source and the target vertex. The function $\lambda : V \cup E \longrightarrow \mathcal{P}(\mathcal{L})$ assigns every vertex and edge zero or more labels. And finally the partial function $v : (V \cup E) \times \mathcal{K} \longrightarrow \mathcal{N}$ assigns a value from $\mathcal{N}$ to properties in $\mathcal{K}$. For every vertex or edge in the graph, the amount of assigned $(\mathcal{K})$ is finite.

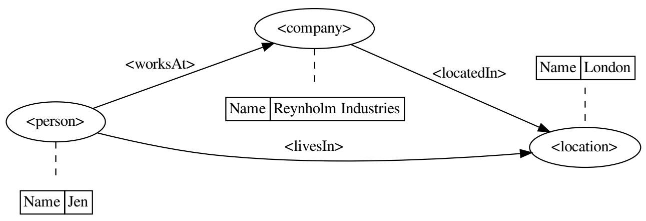


Fig. 1. An example property graph.

An example graph can be seen in Figure 1. Labels are wrapped in angle brackets, and all vertices and edges are assigned one. Properties are shown in the tables, and all vertices have associated properties.

Formally this would be defined as follows:

$$V = \{\ 0,\ 1,\ 2\ \}$$
$$E = \{\ 3,\ 4,\ 5\ \}$$
$$\eta(3) = (0,\ 1)$$
$$\eta(4) = (0,\ 2)$$
$$\eta(5) = (1,\ 2)$$
$$\lambda(0) = \{\ \texttt{<person>}\ \}$$
$$\lambda(1) = \{\ \texttt{<company>}\ \}$$
$$\lambda(2) = \{\ \texttt{<location>}\ \}$$
$$\lambda(3) = \{\ \texttt{<worksAt>}\ \}$$
$$\lambda(4) = \{\ \texttt{<livesIn>}\ \}$$
$$\lambda(5) = \{\ \texttt{<locatedIn>}\ \}$$
$$v(0, \text{“Name”}) = \text{“Jen”}$$
$$v(1, \text{“Name”}) = \text{“Reynholm Industries”}$$
$$v(2, \text{“Name”}) = \text{“London”}$$

On such a graph we perform conjunctive property-graph queries (CQ). Informally, these simply match a set of sub-graphs of the overall graph. More formally, such a query consists out of a set of vertex variables $(\mathcal{V})$, edge predicates of the form (v_1, v_2, p) where $v_1,\ v_2\ \in \mathcal{V}$ define an edge between two different vertex variables, and $p : E \longrightarrow \mathbb{B}$ defines a predicate over the edge, and vertex predicates $q : V \longrightarrow \mathbb{B}$. Such predicates can either require the edge or vertex to be assigned a given label, or can define conditions on the properties assigned to the vertex or edge.

The result of such a query is the set of sub-graphs for which there exists an assignment $a : V(G') \longrightarrow \mathcal{V}$ (where $V(G')$ are vertices in the sub-graph) such that all vertex and edge predicates are met.

For example, we might want to know the set of people that live in the same city as where their job is located. A graph pattern representing this can be seen in Figure 2. If we run this example on the example graph previously shown in Figure 1, we find that the only sub-graph matching the pattern is the entire graph.

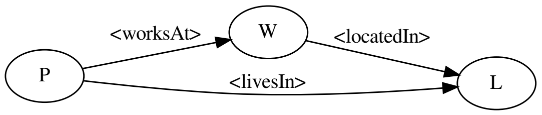


Fig. 2. A graph pattern describing people (P) that work in the same city as where their job (W) is located (L).

2.2 Query Pipeline

In a typical database, a query goes through three different stages during its life-cycle – illustrated in Figure 3. First, it is parsed and translated into some intermediate representation, such as an AST (Abstract Syntax Tree). Then, the best way to execute a query is determined by the planner using statistics and heuristics – yielding a physical plan. This plan represents, unlike the original query and its intermediate representation, how to answer the query, whereas the abstract representation and the query source code merely describes a question to be answered. The physical plan consists out of operator primitives that describe operations that can be performed, such as hash-join and table scan. This plan is then passed on to the execution engine, which executes it.

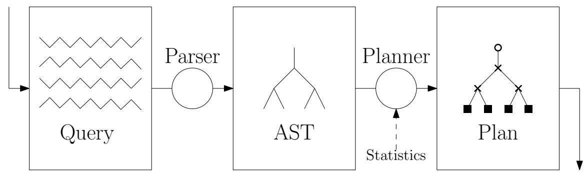


Fig. 3. The different representations the query goes through before it is executed.

For the planner to find a good plan, it needs to predict how different physical operators affect the performance. Only then can the planner compare different equivalent plans. To do so, planners generally make use of statistics about the data set collected ahead of time. For example, the number of edges with a given label, or the distribution of values of a property may be indicative of the size of intermediate results generated by a join, read or filter operation. When combining these values with a cost model that describes the cost of physical operators given some statistics about the input, the planner can find a good plan.

2.3 The Volcano Execution Model

In the traditional, or Volcano, execution model, every operator implements an iterator interface consisting out of a `next` method that returns a single tuple at a time, for as long as there is more output available. To produce one output tuple, the `next` method of the root operator is called, which then calls the appropriate `next` method of its children and so onward. For example, in the physical plan shown in Figure 4, to generate a tuple of output, once the build phase is complete, the `next` operator of both hash joins would be called, as well as the `next` operator of the `<locatedIn>` Read Label operator.

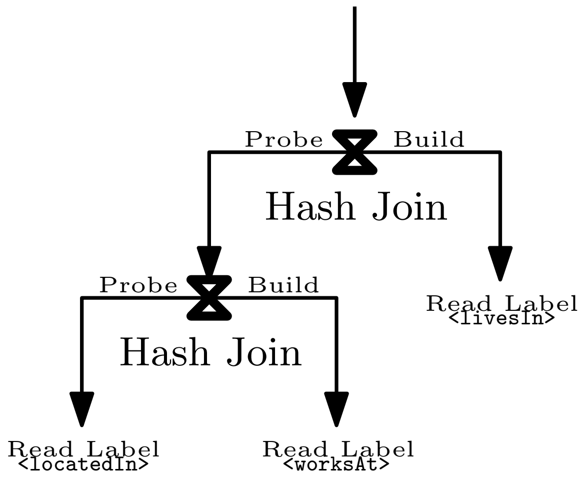


Fig. 4. A physical plan corresponding with the query in Fig. 2

The performance bottleneck of this execution model lies in the amount of `next` calls that need to be made. To produce a single tuple of output, `next` is called many times and control flows through many different operators. Because there is no way to tell at compile time which operator types may appear as a child of an operator type, as the planner can arbitrarily compose operators to create a plan, all infra-operator calls to be virtual. Meaning that to make such a call, the address of the method needs to be looked up from the operator type information. This requires more instructions and memory reads than a regular call, making it more expensive.

Furthermore, a compiler cannot inline across such a virtual method boundary, making many potential optimization opportunities impossible. For example, when multiple operators read the same tuple field, that data would in an ideal case be read once and then passed by a register. But as there is a method boundary, all operators need to read the value individually.

2.4 Vectorized Execution

A relatively simple way to reduce the cost of abstraction that comes with the Volcano execution model, is to reduce the amount of virtual method calls that are made when executing a plan. This can be done by returning multiple tuples for every invocation, spreading out the cost of the call over many tuples. For example, if we choose a block size of 1024, we only perform one `next` call for every 1024 tuples.

This method does however impose additional materialization costs, as tuple values need to be buffered before being returned in a block. To minimize the impact of this, we can store our tuples in a column layout. That way no data needs to be copied when the tuple is extended. As we only need to pass pointers to the first value of every column, and can simply pass an extra pointer and keep existing columns.

To efficiently implement operators that perform filtering or reordering operations on a block, we want to push the cost of updating the materialization up the tree as far as possible. When done properly, we only need to do this only once when multiple subsequent filtering operations are performed.

In order to do this, we maintain an additional column in the tuple block if filtering is applied: the selection vector. In this vector, we specify the indexes of the tuples that were not removed – in the order that they should be outputted. All operators use this selection vector when reading tuples if it is present, moving the cost of updating the materialization to when the data is read.

If an operator does not have any filtering or reordering operators, the selection vector can be omitted and the slots can be accessed directly. This can also be seen as there being a default identity selection vector that is equal to $\mathbb{N}$.

An example block containing a selection vector can be seen in Figure 5. This block contains four tuples, but as some tuples were filtered out by some parent operator, the selection vector skips a few numbers. Arrows are used to indicate the values the rows the selection vector points to.

To look up the first field of the second tuple, we would first look up the index of the second tuple in the block, which in this case gives us the offset 2. We can then read the third field of the first column to get the proper value.

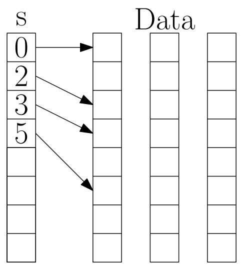


Fig. 5. An example tuple block

When intermediate data is stored column wise, various other techniques also become applicable. For example, we can employ SIMD instructions, which are instructions that operate on multiple values at once, as data is laid out in continuous blocks of memory, which is needed to use such data-parallel instructions effectively [20].

Because of the column layout, we can also make use of type-specific specialized (sub-) operators, generally referred to as primitives. They only operate on the columns that they use, and have a low degree of freedom, operating on independent (restricted) arrays of a fixed shape. This allows the compiler to emit faster code, as aggressive loop pipelining is possible [1].

An example primitive can be seen in 1. This primitive applies a comparison operator on two columns, and writes out a new selection vector. The `__restrict__` annotations on the arguments are used to tell the compiler that the pointers will always point to distinct non-overlapping blocks of memory – giving the compiler much more freedom, as the order of writing and reading to the different arrays is guaranteed to not affect the behavior. The actual body of the primitive is very simple, it simply walks over the arrays and if the predicate matches, an entry is added to the selection vector with the row index of the matching row. When the amount of matching rows is returned, the selection vector will contain that amount of indexes as well.

Listing 1. An example primitive.

```
static int filter_int_lt(
    int n,
    const uint64_t* __restrict__ l,
    const uint64_t* __restrict__ r,
    int* __restrict__ select
) {
  int count = 0;

  for (int i = 0; i < n; i++) {
    if (l[i] < r[i]) {
      select[count++] = i;
    }
  }

  return count;
}
```

As writing and maintaining such primitives for every data type and operator is time-consuming and potentially error-prone, code generation or meta-programming approaches can be applied to reduce the size of the code that needs to be maintained. For example, we could replace the concrete type `uint64_t` in the previous example with a template variable. That way we only have to write this once and can specify the actual type that is as an argument when the primitive is invoked.

2.5 Compiled Query Execution

Rather than reducing the cost of the operator tree abstraction, by amortizing the calls across operator boundaries, we can instead get rid of the abstraction, and the boundaries that come with it. Similar to how early DBMSs compiled all available queries ahead of time [2], we can use the operator tree to generate code. However, unlike those early systems, we can use JIT (just in time) compilation techniques [8] to compile a query right before executing it.

Essentially, when we execute a query plan, we generate a bit of code for every physical operator in the tree. This code is then compiled, optimized and executed to generate the output.

This comes at the cost of having to compile code for every query – which may be rather slow depending on the compiler used and which optimizations are enabled. To circumvent this, compiled code could be cached, and queries could be parameterized by moving constants out to make such caching more effective.

Once the code has been generated for a plan, there are no longer any operator boundaries in place. The compiler can thus much more freely perform optimizations that would be impractical in other models, as the code provides great degree of freedom. The compiler can optimize for a specific combination of data types and operators used. To do such a thing ahead of time, the execution engine would need a specialized method for every combination. The amount of such combinations grows super-linearly (exponential in length), making it unpractical to achieve the same optimizations the compiler's optimizer can find, as all of these need to be compiled and stored ahead of time.

3 APPROACH

3.1 Overview

In this work, we implement both vectorization and query compilation in the preexisting graph database Avantgraph – a single-threaded main-memory-only graph-database developed at the TU/e. We completely replace the execution engine to support both vectorized and compiled execution.

3.2 Vectorization

3.2.1 Overview. We implement vectorized execution by changing the standard iterator interface, we pass a block to write to as an argument and return the amount of written tuples – effectively making the parent operator responsible for the block's memory. This way operators can also easily add or remove columns by creating a new block with the previous block's column pointers, as the memory is managed by the parent, it is guaranteed to remain valid while passing the new block down the line.

For the parent operator to know what block to pass to a child, it needs to be aware of the columns that are present. And whether there is a selection vector present. To do so, the standard operator interface is augmented with an additional method that can be used to query the operator's output layout. This layout maps locations in the tuple to column indexes, and contains the data types of every column.

3.2.2 Primitives. To implement the various operators, we make use of reusable primitives. We define these using C++ templates so that we only need to specify the shape of them once, and can then plug in different data types or other constants to obtain the specialized variants. The C++ compiler will apply expressions on constants passed via template arguments (constant evaluation), so we can, for example, pass in the comparison as and argument and write a case-distinction inside the primitive that gets optimized by removing the case distinction.

3.2.3 Hash Join. In our hash join operator, we make use of group prefetching [3]. Again, we rely on C++ templates and constant evaluation to generate primitives for different group sizes and with or without a selection vector.

3.3 Query Compilation

3.3.1 Overview. To compile a query, every operator emits a bit of code that implements the semantics of that particular operator. This code then gets compiled, optimized and executed. This is depicted in Figure 6, the code generation process translates a plan into some programming language, which the compiler takes as input and translates it into optimized machine code. For the target language, we can pick basically any general purpose programming language. We choose to emit LLVM IR [11] (intermediate representation), both for the reasons described below.

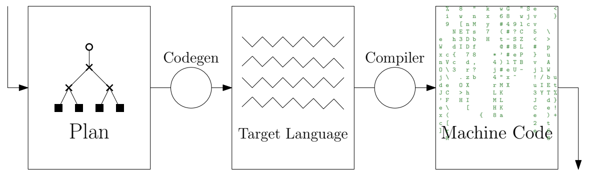


Fig. 6. The process of translating the plan to machine code.

The well-supported C++ API lets us emit code without ever having to fall back to a textual representation, and also supports many low-level operations that may be difficult to express in languages such as C or C++ by mapping more closely to commonly used CPUs [16]. For example, LLVM natively supports SIMD instructions that get lowered to appropriately sized SIMD instructions that are supported by the target CPU.

Apart from having an API, it is also quite a bit faster than using a standalone compiler such as clang, or GCC [16]. To use such a compiler, we need to emit source code as text, which then gets parsed and transformed into some AST by such a compiler. Once the AST is validated, such compilers then transform it into an intermediate representation that they pass to the compiler back-end, which performs optimizations and lowers it into machine code. By using LLVM IR, which is used as the intermediate language by the clang compiler, we can most of this work, and directly invoke the compiler back-end to optimize and lower the code.

Once we have generated the code in LLVM IR, there are no longer any strict operator boundaries. The optimizer can freely optimize across operators and may for example, combine predicates of different filter operators into a single predicate and operator. To perform such arbitrary contractions in non-compiling engines, there needs to exist an optimized operator variant for all combinations of predicates ahead of time. But as the number of variants is generally unbounded, and the amount of variants grows exponentially with the amount of predicates, this greatly increases the size of the code, making it not very practical [9].

One downside of compilation, apart from the complexity of implementing it, is of course the start up time. Depending on the optimizations performed during the compilation process, the time it takes to compile a query may very well outweigh the speedup gained by executing it, or even take longer than actually executing it. So, care should be taken to cache compiled plans and in execution engines that can partially compile a plan, to only compile the parts of a plan where it matters the most.

3.3.2 Generated Code. To translate a plan into LLVM IR, we walk over the operator tree and generate a bit of code for every operator. Rather than emitting code that would correspond with the Volcano model, in which every operator *pulls* data from its children by calling their `next` operator, we emit data-centric code in which every child *pushes* the data up the operator tree. This way data can be kept in registers as long as possible, avoiding breaking the pipeline and spilling data to the much slower main-memory [15].

To better understand the structure of the generated code, we propose a model that lets us subdivide the generated code in smaller, easier to understand, sections. In this model, the code consists out of pipelines that always read from fully materialized data, and write to fully materialized data. For example, a pipeline might read all edges with a given label and build a hash table, or reads some index and build a plan's final output. Within a pipeline data may be momentarily partially materialized so that processing happens in tight loops – better utilizing CPU pipelines [1], or to perform instructions that require data to be in a certain shape, such as prefetching or SIMD instructions. The areas of code between such materialization boundaries we call stages. Because of that, these boundaries are referred to as stage boundaries [14]. Finally, every operator may modify the tuple as it passes from a child to the parent operator. This final level is referred to as a step.

Buffers at stage boundaries are of a fixed-size, so special care needs to be taken to jump to the next stage once it is full, and to correctly resume the previous stage once it is done. It may be useful to consider transferring control to the next stage as a function call, the state of the registers is pushed to the stack and control is transferred to the function. Once the function is done, the registers are restored and the processing continues. As such resumption is quite hard to implement when it passes through multiple operators without actual function calls, where it is needed we keep the state on the stack. When done using LLVM's `alloca` instruction, LLVM can translate these stack allocations to virtual registers automatically.

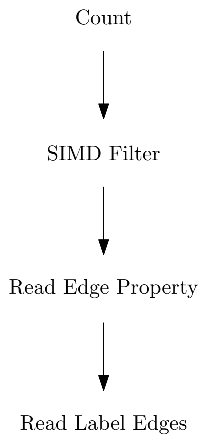


Fig. 7. An example plan

We will illustrate the proposed model with an example. In Figure 7, a simple plan is shown. It consists out of an operator that reads all edges with a given label, an operator that reads a property of such an edge, an operator that filters the tuples, and finally an operator that counts the tuples.

In Figure 8, the generated code is depicted in our model. The operators that read from (fully) materialized data (shown in green),

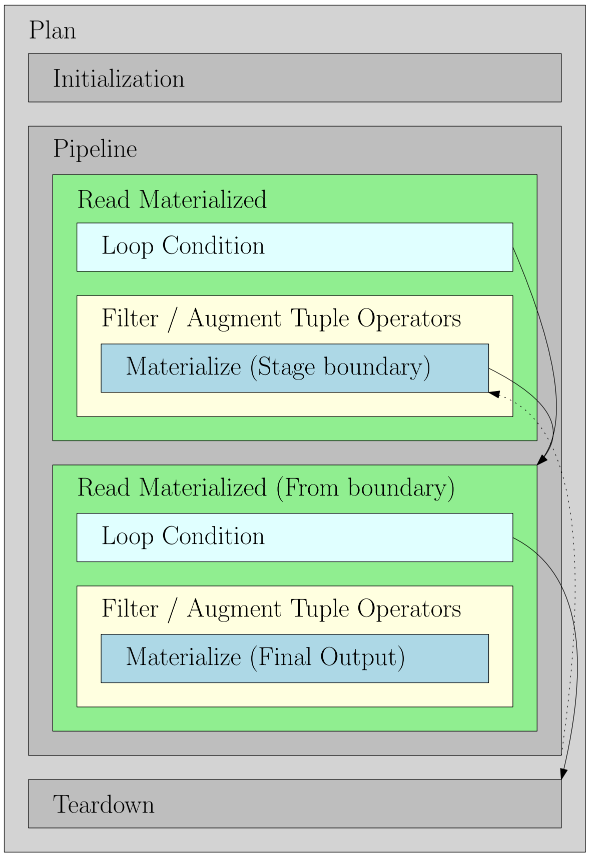

Plan
Initialization
Pipeline
Read Materialized
Loop Condition
Filter / Augment Tuple Operators
Materialize (Stage boundary)
Read Materialized (From boundary)
Loop Condition
Filter / Augment Tuple Operators
Materialize (Final Output)
Teardown

Fig. 8. Code Generated for the example plan

are always the first operator in either the pipeline or the stage. Operator steps that only map and/or filter are depicted yellow. The last, or innermost, operator of a stage always writes the data in some materialized form. Here it is depicted in blue. Such an operator in the last stage always writes data in some fully materialized form. In this case, it is used as the final output.

The arrows in the diagram depict noteworthy jumps between code blocks. In the loop conditions, generated by the first operator in every stage, control can jump to the end of the stage if no more tuples are remaining. The innermost operator of the first stage, writes the data to the materialization boundary, and may thus temporarily jump to the next stage if the buffer is full. When the stage was entered from this operator, it will jump back to the previous stage once it is complete, as indicated by the dotted arrow.

In some plans, there may be multiple pipelines, each with a final materialization stage. For example, when hash join is used, the build operator will be placed in a separate pipeline that materializes to a hash table. However, this does not contribute much additional complexity, as there is no jumping back to a previous pipeline.

3.4 Code Generation For Operators

In the following section, we will go over some individual and highlight what code would be generated for each of them. For this to be an indicative sample, we will show in-depth samples for one operator that reads from a fully materialized state, one operator that is an intermediate step, and one operator that materializes to some stage boundary. For every operator, we describe both its semantics, and a corresponding implementation.

3.4.1 Read Label Edges.

Semantics. This operator emits a tuple containing an edge ID for every edge that has a given label.

Implementation. This operator accesses the graph, so it must be aware of the format in which the graph's data is laid out in memory. In our case, every label has a continuous block of memory in which edge IDs with that label are stored. To emit a tuple for an edge, we only need a pointer to the first ID and the amount of IDs to loop over them.

To get this information we make use of a small helper function written in regular C++ shown in listing 2. It takes the address of the graph, the label ID and returns the location and size of the edge ID table. By making use of LLVM's built in support for calling C functions, all we have to do is simply take the address of the function and define it in our LLVM module.

Listing 2. A helper function to access the edge id table.

```
static void labelEdges(
  Graph* g,
  Label* l,
  uint64_t* len,
  uint64_t** edges,
) {
  std::vector<uint64_t>& edges;
  edges = g->edgesWithLabel[l->id];
  *len = edges.size();
  *edges = &edges[0];
}
```

The actual code that is generated consists out of some initialization code – looking up the address and size of the Edge-ID table, allocating a stack address for a counter – and a loop. The loop walks over every address in the table and reads the stored Edge-ID.

The initialization code can be seen in listing 3. We allocate some space on the stack for the return values of the helper functions before calling it, and then reading the values that were written. Finally, we allocate a counter variable and initialize it with zero.

Listing 3. The initialization code

```
%lenPtr = alloca i64
%edgesPtr = alloca i64*
tail call @labelEdges(
  i8* %G, i8* %L,
  i64* %lenPtr, i64** %edgesPtr)
%len = load %lenPtr
%edges = load i64** %edgesPtr

%i = alloca i64
store i64 0, i64* %i
```

The body section can be seen in listing 4. It starts by entering the condition block. If there are no more edges to be read, it will jump to the end of the operator (`%tail`). If there is at least one more edge, control jumps to the `%body` block. In the body we increment the loop counter and read the edge. After this the body of the consumer is inserted (provided by the parent operator). Finally, we jump back to the loop condition.

Listing 4. The body code

```
  jmp %cond

%cond:
  %offset = load %i
  %end = cmd i64 %offset, i64 %len
  br i1 %end, %tail %body

%body:
  %next = add %offset, 1
  store i64 %next, i64* %i

  %edge = getelementptr
  i64* %edges, i64 %offset
  %id = load i64* %edge

  ; <consumer>

  br %cond

%tail:
```

If the consumer contains a stage boundary, it may momentarily jump to a next stage before returning where it left off. This is generally implemented by inserting an additional basic block at the end of `<consumer>`, that lets the next stage return to this operator.

3.4.2 *Filter Tuple.*

Semantics. This operator emits all tuples matching a predicate that its child operator emits.

Implementation. This operator is essentially equivalent to a single if statement with the predicate condition. So we only have to emit code for the predicate, which gives us a boolean, and a conditional jump that will jump over the consumer if the predicate does not hold.

A sample of code that might be generated can be seen in listing 5. The predicate yields a boolean variable `%cond`, that jumps over the `%holds` block if the condition is not met. Inside the `%holds` block, the consumer callback of the parent operator inserts its code.

Listing 5. The body code

```
  %cond = <predicate>
  br i1 %cond, %holds, %tail

  %holds:
    ; <consumer>

    br %tail

  %tail:
```

3.4.3 *Materialize to Column.*

Semantics. This operator writes all incoming tuples to a fixed-size buffer, momentarily continuing with the next stage once it is full.

Implementation. For this operator we need to allocate a buffer to write to, and a counter that keeps track of how many tuples were written. This can be inserted at the start of a pipeline. In Listing 6, we show an example. We allocate space for a single-column buffer, `%b`, and a counter, `%u` on the stack. We also initialize the counter by writing zero to it.

Listing 6. Pipeline Start

```
  %b = alloca i64, i64 64
  %u = alloca i64
  store i64 0, i64* %u
```

Inside of the operator body code, we write the tuple to the buffer, and jump to the next stage if it is full. We also need to add a block so that we can resume operation if the next stage completes. An example of this can be seen in Listing 7. We load in the current counter value into `%o` and use it to write the tuple to the buffer at the correct offset. We then increment it by one and update the counter value. Finally we check if we have reached the buffer capacity and jump to the start of the second stage (`%stage2`) if that is the case. We also add a block (`%resume`) that can be used to resume operation, and is also used in the conditional jump when the buffer is not yet full.

Listing 7. The body code

```
%body0:
  %o = load i64, i64 %u
  %addr = getelementptr i64, i64* %b, i64 %o
  store i64 %tupleSlot, i64* %addr

  %newUsed = add i64 %usedV, i64 1
  store i64 %newUsed, i64* %u

  %f = icmp eq i64 %newUsed, i64 64
  %full i1 %f, label %stage2, label %resume

%resume:
```

At the end of the first stage, we insert the root operator of the second stage, and wrap it with some code that receives jumps from the previous stage and properly resumes it when this stage is done. An example of this can be seen in Listing 8. We insert a block for the second stage (`%stage1`), and a jump to it when the first stage ends normally. Inside this block we use a `phi` instruction to keep track of how the block was entered. This instruction assigns `0` when entered from `%stageTail0`, and `1` when entered via `%body0`. Once the stage completes, the `switch` instruction is used to jump back to `%resume` or `%stageTail1`.

Listing 8. The stage tail

```
%stageTail0:
  br %stage1

%stage1:
  %src = phi i64 [ 0, %stageTail0 ],
                 [ 1, %body0 ]

  ; <code using the stage>

  switch i64 %src, label %stageTail1
                 [ i64 1, label %resume ]

%stageTail1:
```

In these examples we rely on LLVM to translate our counter variable from the stack to a register. Even though it is technically possible to do this ourselves by inserting more phi nodes and passing the state around, this results in much more complex code. However, LLVM has a built-in optimization pass `mem2reg` that takes care of this for us. It translates instances of `alloca` that only have loads and stores to phi nodes.

3.4.4 *Hash Join.*

Semantics. This operator is a join operator. As most join operators, it combines two streams of tuples by taking the cross-product, and then applying a filtering predicate on it.

Implementation. Rather than actually performing a cross-product, this operator uses an equality predicate to build a hash table of one of the tuple streams, and generates output by probing the hash table using the other stream.

To implement this in code, we distinguish two distinct phases: the build phase, in which the hash table is constructed, and the probe phase, in which the hash table is queried. The first phase happens in a separate pipeline that must be completed before probing can occur.

For this operator, we need a hash table. It must at least support insertion and querying all results for a given key. We chose to implement this hash table as a part of the code generation process – avoiding calls to C++ code where possible to avoid boundaries for the optimizer.

The first step in implementing a hash table, is to design it. We used an open-addressing, linear-probing for key collisions, and an external data section for duplicate values, as this works well for hash join [3, 14].

The exact memory layout can be seen in Figure 9 – which is designed top be robust and cache-friendly [3]. Buckets consist out of the following. A 64-bit status flag, the tuple values that make up the hash key, all tuple values that are not part of the hash key, and finally the 32-bit hash. In the case of duplicate values, the status flag acts as a pointer to external storage. This external storage, of which the layout is shown in Figure 10, consists out of two 64-bit numbers that store respectively the usage and, the capacity. This header is directly followed by tuple values that are not part of the hash key.

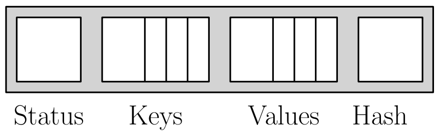


Fig. 9. The layout in memory of a hash table bucket.

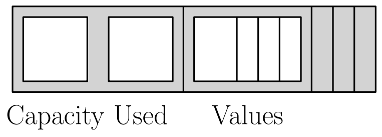


Fig. 10. The layout of the external data.

To implement this with LLVM, struct types are created at runtime for the part of the tuple that is in the key, the part of the tuple that is not in the key, the external data header (size and capacity), and finally the hash table bucket. This lets LLVM know what is going on, and in return it provides us with type-safety.

To rehash the table, or to (re-) allocate external storage, we make use of C++ callbacks. We pass sizes and offsets of the structs and its fields, as they depend on the size of the tuple.

To generate the code for the pipeline that builds the hash table, we use the "begin pipeline" callback in our plan visitor to insert the pipeline code before the pipeline that needs the probe is executed. The probe side does not need special attention, it simply queries the hash table and walks over all of its results and runs the code of the consume callback for every result.

Since the code for hash join is rather complex and unwieldy, we will not show an actual code sample. Instead, we abstract away and look at the code blocks that make up a plan containing a hash join – in accordance with the proposed abstraction model. In figure 11, we can see a schematic visualization of the compiled code corresponding to a plan containing a hash join operator. It consists out of two pipelines, one for building the hash table, and one for generating the plan's output.

In the building phase, for every tuple received from the build operator, the key is extracted and a hash is calculated. This is used to begin the probing process that will look for the correct slot. There are two possible outcomes of probing here, either we find an empty slot and can directly insert to it, or we find an existing slot, and we need to append to some external data section. If we append to a slot that was previously used, we need to check if it is inline or not before using it. When it was inline, we allocate an external data section and copy the inline tuple to it. If not, we check if we need to resize the external buffer before appending to it. Not shown here is rehashing when the table gets full, or the bookkeeping needed for it.

During the probing phase, we also start by finding an initial probe location using the hash, and then linearly scanning over the buckets. However, when we find and empty slot, we know that the key does not appear in the table. And when we find a slot matching the key, we loop over all results and pass them to the consume callback of the parent operator. Some care needs to be taken here to ensure the value that is stored inline is also emitted if it was available.

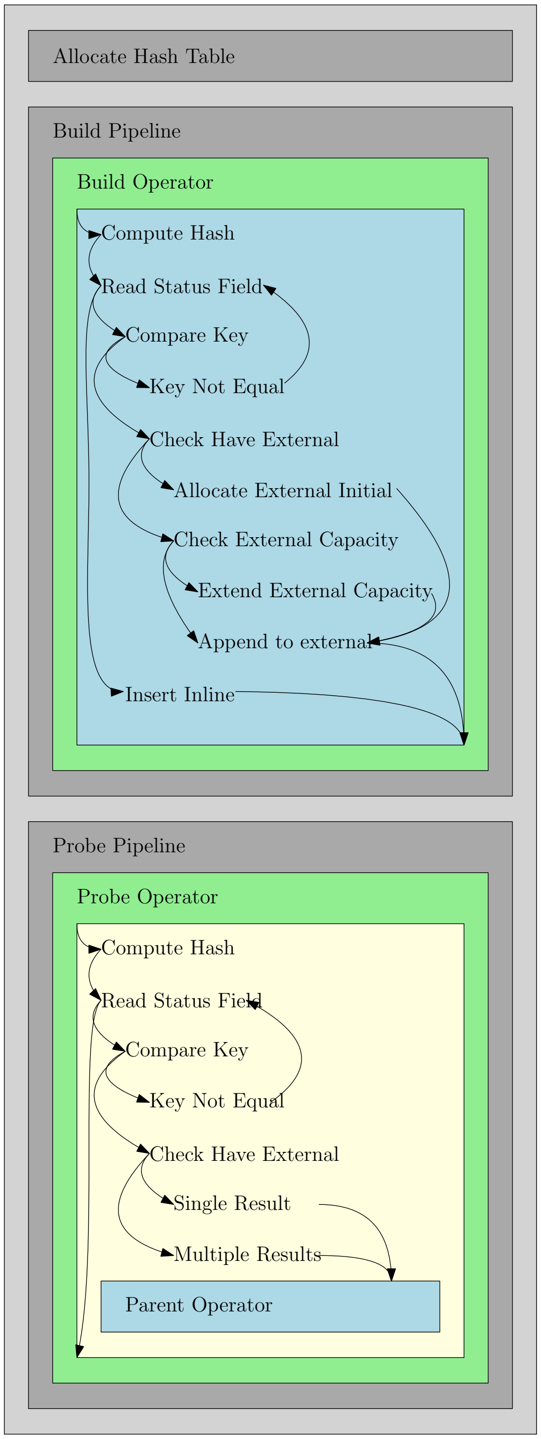


Fig. 11. Schematic description of a plan containing a Hash-Join operator.

3.4.5 Iterating Operators. Some operators, most notably merge-join, don't really fit in the consumer-based model. For example, merge-join proceeds with one of its inputs depending on the outcome of a comparison. To make such operators work, we cannot use the consumer model as described before, as the consumer does not have any control over the operator the tuple is received from.

One way to circumvent this that doesn't require code changes in other operators to explicitly support this, is by using co-routines [4]. Co-routines are functions that return (yield) multiple times – saving the state of the execution and allowing it to be resumed. By placing each child operator in such a function, we can easily iterate over its output.

The saving and resuming of state, as well as allocating and freeing of the memory in which this is done of course adds overhead. But because LLVM has low-level support for co-routines, they can be inlined, and the allocations moved to the stack – effectively completely getting rid of them. This lets LLVM optimize across the iterator boundary which might not be possible with an explicitly implemented iterator.

3.5 Implementing Code Generation

In the previous section, we have shown the code we wish to emit. In this section, we describe how to actually do so.

3.5.1 The Plan as the Input. As input, our compiler front-end receives a plan. It consists out of an operator tree. Each node has an associated operator type, and information about the shape of the tuples that are emitted for a particular node. We refer to this as the data layout.

Every slot has a globally unique identifier as it passes through the operator tree – stored in the data layout. This allows us to uniformly address tuples, and allows us to easily abstract away the source of a tuple slot by hiding it in a tuple view. This also lets us pass a view through a parent operator without having to remap it at every operator. Though, it is important to note that such views are purely to make code generation easy, as no actual code is generated for them.

We also define reusable tuple formats, consisting out of a struct type and some code generation to access it, to have a reusable way to store materialized data. For example, we have a "dense" tuple format that stores the values right after each other as they would be stored in a hash table, and a "column" tuple format that decomposes tuples into columns. They facilitate reuse between operators, and provide code generation to write or read tuple views.

3.5.2 Generation Pass. We generate the code by first building a plan visitor by walking over the operator tree in a depth-first manner – at every level passing down an operator visitor. In operators that have multiple children, each child typically receives a specialized visitor.

The operator visitor interface contains various methods that let it insert code at the beginning or the end of the plan, pipeline or stage. Depending on the visitor, it may also contain a method that will generate code that consumes a single tuple or intermediately materialized data from a stage boundary.

The plan visitor, which is created for a subtree every time a plan visitor is passed down, has "begin plan", "run plan" and "end plan" methods that generate setup code, the pipelines, and tear-down code respectively.

Every visitor is responsible for calling its parent where appropriate. For example, an operator may call the "begin stage" code of its parent operator in the "end stage" code of the current operator to implement a stage boundary, or a hash join operator may insert a pipeline by calling the "run plan" method of the plan visitor returned by the operator of the build-side before calling "begin plan" of the parent.

By first constructing visitors, we generate code that is more or less in the order it is executed, which makes local values a lot easier to deal with, as they need to be declared before they are used. Using the visitor pattern we can also very flexibly insert code where it is needed, without having to break out of the abstraction.

To generate code for an operator that needs external iteration, such as merge join, we simply create a new function for the subtree to iterate on and pass it as the generation target as we run the visitors.

4 EVALUATION

4.1 Dataset

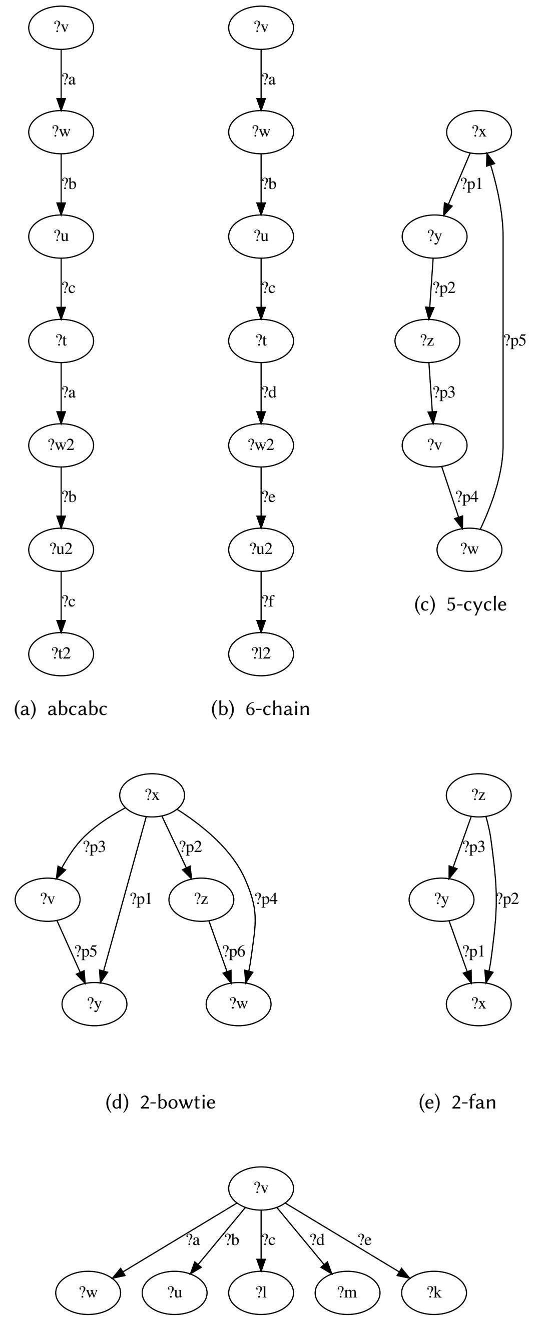
All experiments were performed on the YAGO2 dataset – a knowledge-graph mined from Wikipedia. On this dataset, we use queries provided by queryminer. This project provides graph queries, and their respective expected outputs, that were generated by defining a pattern and then finding all label assignments that match the pattern.

For example, one might define a chain pattern of a given length, then find all of such chains in the dataset, and store the labels of the chain along with the amount of occurrences of that label assignment.

Every pattern thus corresponds with a class of queries of some shape. The following patterns were available, and are used in our queries:

(a) **abcabc:** A subset of the set 6-chain in which labels are repeated along the chain in the pattern indicated by the name.
(b) **n-chain:** n nodes, each with an outgoing edge, and a single node with no outgoing edges so that a chain through all nodes is formed.
(c) **n-cycle:** n nodes, each with one outgoing edge so that the edges form cycle through all the nodes.
(d) **n-star:** A single node with n outgoing edges to n nodes.
(e) **n-bowtie:** A single node that has outgoing edges to all nodes of n node pairs that have a single edge between them. This is essentially a modification of n-star with added edges between disjunct node pairs.
(f) **n-fan:** A single node that has outgoing connections to n nodes. Additionally, there is a path through all of those n nodes. Essentially, this is a modification of n-star with a path added through all non-center vertices.

For each of these classes an example is available in Figure 12. Vertices and edges are annotated with variables prefixed with ?. For a sub-graph to match the pattern, every vertex in the sub-graph maps to one or more vertex-variables, all edge-variables are assigned a label, and the connections with the given labels exist in the graph.


?v ?a ?w ?b ?u ?c ?t ?a ?w2 ?b ?u2 ?c ?t2
?v ?a ?w ?b ?u ?c ?t ?d ?w2 ?e ?u2 ?f ?l2
?x ?p1 ?y ?p2 ?z ?p3 ?v ?p4 ?w ?p5
?x ?p3 ?v ?p5 ?y ?p1 ?p2 ?z ?p6 ?w ?p4
?z ?p3 ?y ?p1 ?x ?p2
?v ?a ?b ?c ?d ?e ?w ?u ?l ?m ?k

(a) abcabc (b) 6-chain (c) 5-cycle (d) 2-bowtie (e) 2-fan (f) 5-star

Fig. 12. Various query shapes.

4.1.1 *Plan Generation.*

In order to use the queries from queryminer, we first transform these into queries the planner and the execution engine supports. As queryminer only provides a pattern and a set of outputs, we translate the outputs, which contain the label assignments, into queries using the pattern.

We can then pass these queries to the planner and generate plans for them. We compute all plans ahead of time and persist them to disk, as we are only interested in the execution engine, and we use the same plan for all execution engines.

Out of all the generated plans, we randomly sample a fixed set of plans for every query class, as the amount of available plans is very large.

4.2 Experiment

To compare the performance of compiled execution, we execute every plan both in vectorized and in compiled mode, while measuring the duration. We separately measure the time spent compiling the plan in compiled execution mode.

Additionally, to benchmark the performance of our vectorized execution engine, we will also run plans with different block sizes of the vectorized engine.

Furthermore, we measure the performance impact of group prefetching, which is implemented in our vectorized execution engine, by differing the group size.

All experiments were performed using the same binary, compiled with clang and inter procedural optimization enabled. We compile in release mode with optimization flags `-O3 -march=native`, where `-O3` tells the compiler to enable optimizations, and `-march=native` tells the compiler that it can use all features the CPU supports, and that the machine code it generates does not need to work on other machines.

4.3 Results

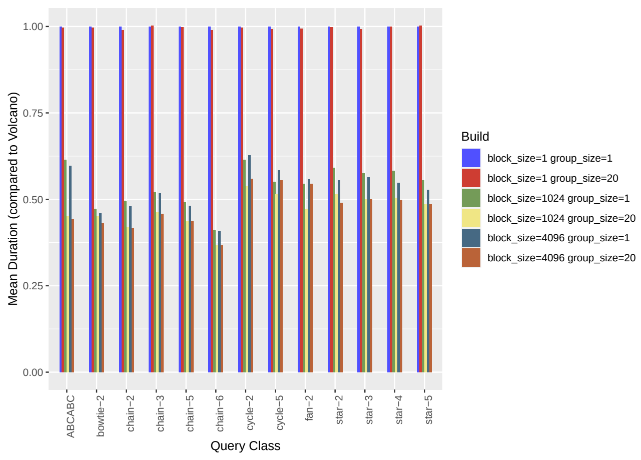


Fig. 13. The average query duration, for every query class, for the different configurations, expressed in the average duration of queries of that class in a configuration with block size of 1 and a hash group size of 1.

4.3.1 Performance of Vectorization. To analyze the impact of vectorization, we calculate the average duration for all queries in a class, for all block/group size variants. The result of this can be seen in Figure 13. Here, and in all following figures, `block_size` refers to the size of the vector, and `group_size` refers to the amount of tuples to process at the same time in a hash table. So, a configuration with a `block_size` of one is equivalent to the Volcano model, and a configuration `group_size` of one does not use group prefetching.

In all cases, vectorization provides a significant performance boost – halving the duration in most classes. When group prefetching is used, an additional speedup is obtained in all classes. Having larger blocks may slightly increase performance (such as in `ABCABC` or `star-2`), but this is not always the case (`cycle-2` and `fan-2` are examples of this).

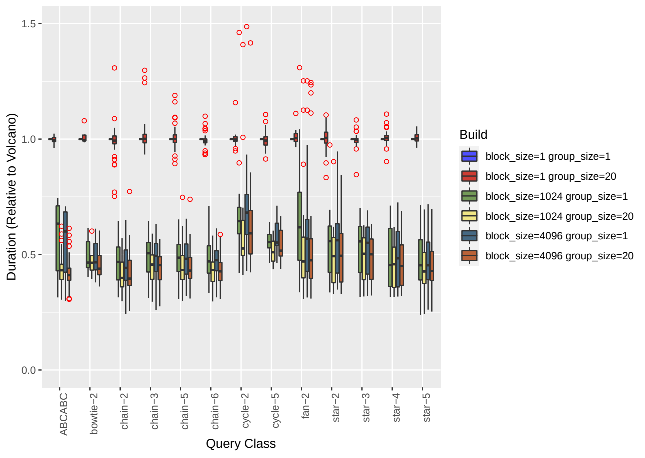


Fig. 14. The average query duration, for every query, for the different configurations, expressed in the duration of the same query in a configuration a block size of 1 and a hash group size of 1.

We further analyze these results by inspecting the speedups on a per-query level. This can be seen in Figure 14. Here, we plot the distribution of individual relative durations for every query, rather than the average per class. We see that when group prefetching is used, not only the average, but also the worst-case performance is improved.

4.3.2 Performance of Compilation. We also analyze the performance impact of compilation – again, by comparing it to a Volcano baseline. The average execution times can be seen in Figure 15, and the overall distribution in 16. We see a significant speedup on average, and an interesting pattern surrounding the parameterized query classes. For larger parameter values, the relative duration decreases, and thus the performance gained increases. This is not universally true however, as `chain-3` is higher than `chain-2`.

To get a clear picture of the effect of compilation on the execution times, compared to vectorization, we express, for every query, the execution of compiled execution, in the execution time of vectorized execution. The results of this can be seen in Figure 17. We then group these per class in Figure 18. Here we see that in the vectorized configurations (`block_size` $\neq 1$), compiled execution is better on average for some classes. The differences we see between different configurations are a consequence of the differences in duration when running queries with the vectorized engine.

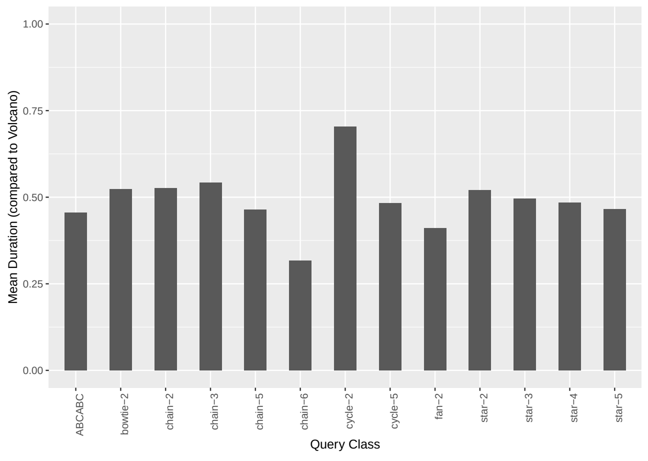


Fig. 15. The average query duration when compiled, expressed in the average duration in the Volcano configuration.

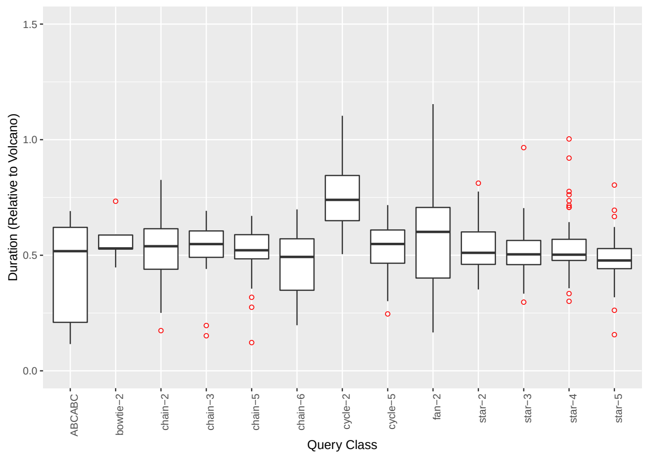


Fig. 16. The average duration, for every query, when executed using the compiling engine, expressed in the duration of the same query in the Volcano configuration.

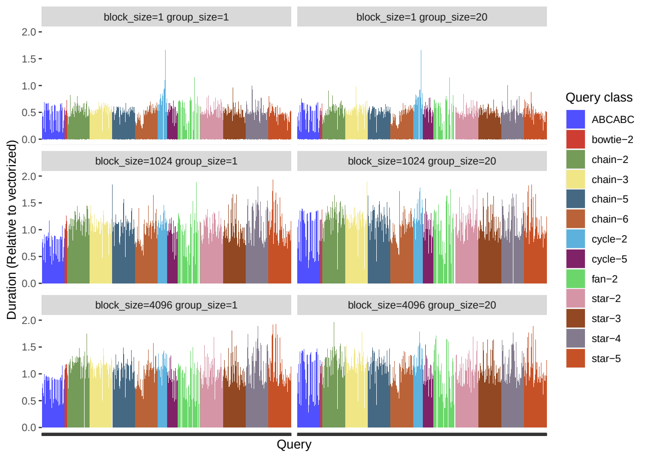


Fig. 17. The duration of a query when executed with the compiled engine, expressed in the duration of the same query executed with the vectorized engine of that configuration.

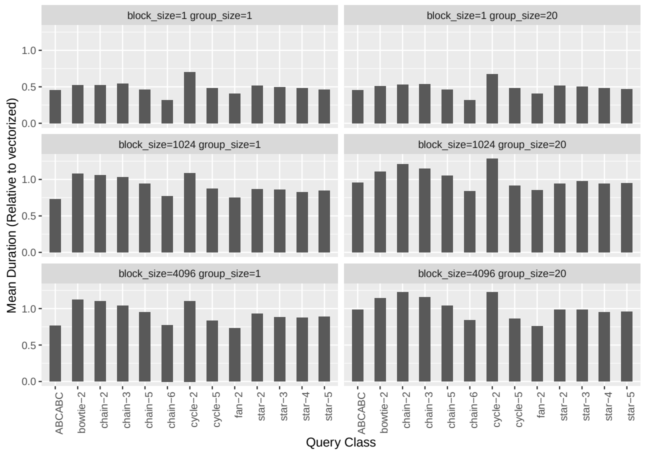


Fig. 18. The total relative speedup per class.

To better understand where this difference originates from, we collect the amount of intermediate tuples that are processed – which we will refer to as the total cardinality. As we use the same plan for both engines, this will stay consistent. We define this amount as the amount of tuples that were either materialized or filtered out by some operator. If a tuple is simply passed through an operator, or extended with a field, it is not counted multiple times.

The distribution of this total cardinality over different classes can be seen in Figure 19. We see that the cardinality for the class abcabc and the class `fan-2` are the highest. These two classes also happen to consistently achieve a speedup when using compiled execution.

To gain further insights we plot the speedup of individual queries against the total cardinality. This is shown in Figure 20. When we look at the variant with a block size of 1024 and a group size of 1 we can see that the low-cardinality queries tend to perform worse with compilation and the high cardinality queries perform better. In the variant next to it, where group-prefetching is enabled, we see that the vectorized engine performs better on the high cardinality queries and worse on the low cardinality ones.

From this previous figure we also learn that there is a wide variety of relative speedup within query classes. So to further investigate that, we plot the relative speedup of all queries in a box plot. We only compare compiled execution with vectorized execution here, so we omit the variant with a block-size of one. This plot can be seen in Figure 21.

Unlike the average execution time shown in Figure 18, the median execution time favors vectorized execution more in most classes. The lower average seems to be caused by better best-case performance caused by compilation optimizations, which is most notable from the ABCABC class. The relative speedup increasing for longer chains and cycles is again visible. As we know from Figure 13 that the

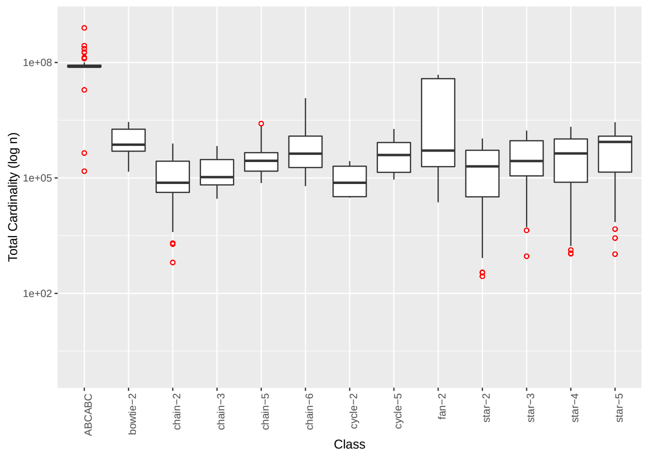


Fig. 19. Total Cardinality Per Class.

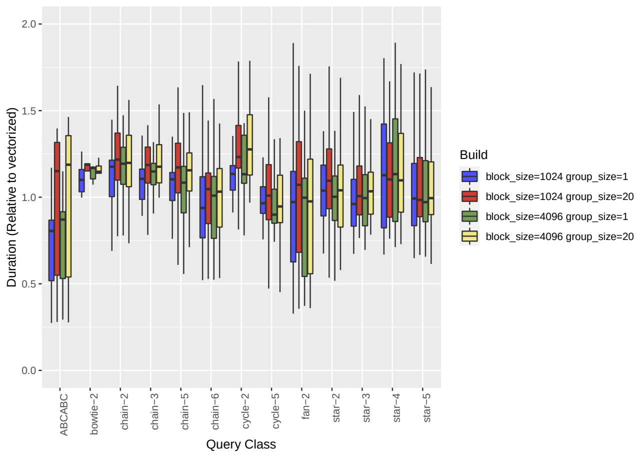


Fig. 21. The distribution of speedup for the query classes.

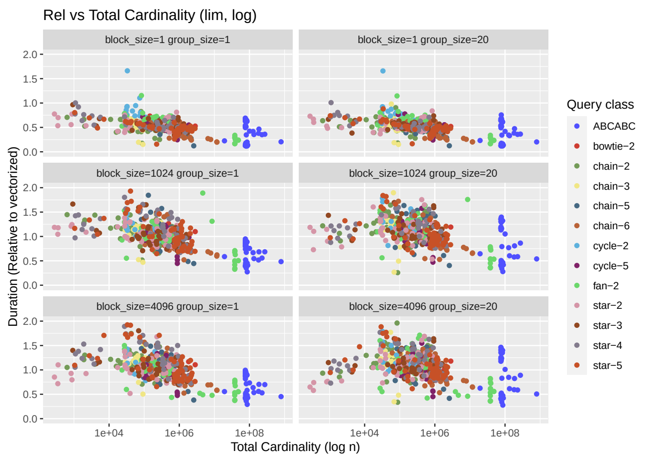


Fig. 20. Speedup vs Cardinality.

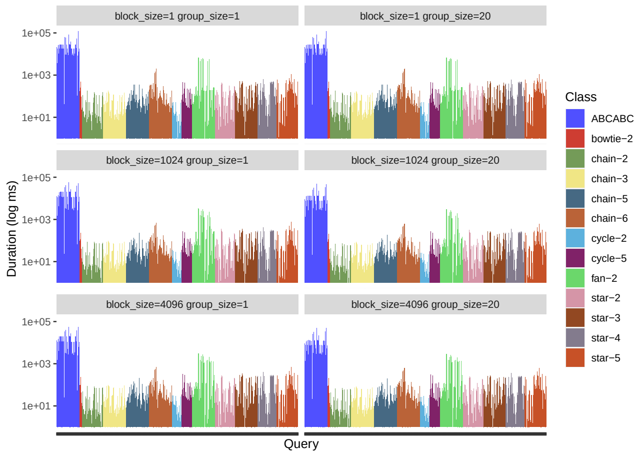


Fig. 22. The absolute amount of time spent executing queries.

performance of vectorized execution also increases for these query classes, the performance of compilation increases more quickly.

4.3.3 Simulation. To further investigate the cause of the performance difference between the vectorized and the compiling execution engine, we run one plan that is consistently slower in the compiling execution engine with Valgrind's callgrind – which is an emulator that can simulate and collect cache and branch-prediction misses.

The results of this can be seen in Table 1, where all simulated values are shown. On the top we see the amount of operations were performed, such as the amount of instructions that were read, or the data that was read or written. Below that we see the amount of cache misses that occurred, here L1 refers to the first layer of caching, and LL to the last level of caching before the main-memory needs to be accessed. Finally, we have the results of Valgrind's built-in branch prediction simulation.

As expected, most values are actually higher for Vectorized execution, including the amount of executed instructions, as well as the amount of data written and read. As vectorization materializes intermediate results. Which means that the tuples are written to memory every time an operator is passed.

Consequentially, in order to gain a performance advantage over compiled execution, it needs to be able to make better use of the hardware. In the simulated values we can see that the amount of Last-Level cache misses is higher in the compiled version. On modern machines accessing the main memory is much slower than accessing values that are present in any of the caches, leading to the CPU having to wait for values before processing can continue. Since the vectorized execution engine uses prefetching instructions, this may partially explain the performance difference. Apart from that, the vectorized engine also has a different access pattern, accessing the same hash table repeatedly, whereas the compiled engine accesses a different table every time as a tuple flows through multiple hash-join operators.

		Compile	Vectorized (1024)
Operations	Instruction Fetch	440 563 624	1 072 339 932
	Data Read Access	80 074 664	289 660 417
	Data Write Access	54 338 013	176 804 733
Cache-Misses	L1 Data Write Miss	6 373 882	9 777 180
	LL Instr. Fetch Miss	1 195	1 775
	LL Data Read Miss	9 405 687	8 459 408
	LL Data Write Miss	5 675 709	6 325 669
	L1 Miss Sum	21 567 284	28 925 264
	Last-level Miss Sum	15 082 591	14 786 852
Branches	Conditional Branch	49 910 017	117 828 035
	Mispredicted Cond. Branch	3 999 782	4 493 491
	Indirect Branch	1 904 350	10 022 228
	Mispredicted Ind. Branch	274	8 002
	Mispredicted Branch	4 000 056	4 501 493

Table 1. The values measured by Valgrind.

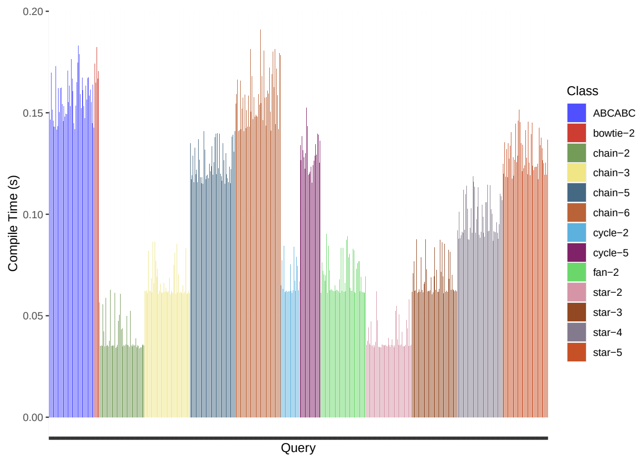


Fig. 23. Compile Time of queries.

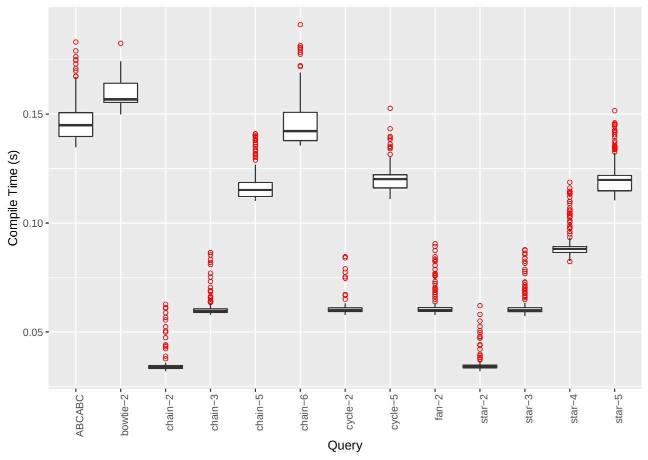


Fig. 24. Compile Time of query classes.

4.3.4 Compilation Time. In the previous section we did not include the compilation time in our measurements. We measured these separately and can be seen in Figure 23. As these do not depend on the build we can much more easily plot them in a box plot as seen in Figure 24.

The compilation time differs between classes, mostly because the plans are wildly different, resulting in different, and a different amount of code, being emitted.

Even though the queries compile rather quickly, for short running queries this will be a significant part of its duration. But for long-running queries or queries that are predictable and/or callable the time spent compiling can be gained back.

For example, the ABCABC queries would have benefited from compilation, even with the additional time compiling.

5 RELATED WORK

The Volcano, or iterative model, that was previously mentioned in the background section, was introduced by [12] and later popularized by the Volcano project [6]. This model was developed with disk-based storage in mind, but as keeping the entire data set in main memory became feasible, the interpretation overhead of this model has become problematic.

For this reason, the MonetDB system was developed, which fully materializes intermediate results[13], which reduces the cost of interpretation by crossing operator bounds only once. Later this system was developed further, only materializing large vectors at a time[13]. This way the cost of crossing the operator boundaries is amortized while still having to ability to pipeline data.

When using vectorization, various techniques that operate on multiple tuples at once become available. One of such techniques is group-prefetching as described in [3]. By processing multiple tuples at a time, the latency caused by cache misses can partially be hidden. Another technique that is possible is the use of SIMD instructions as described by [20]. These can operate on a vector of tuples and perform operations data-parallel.

Alternatively, we can use query-compilation to avoid the interpretation overhead. A primitive form of compiled execution was used by IBM in System R back in 1970, which used assembly code-templates for each operator. This was later abandoned for compatibility and complexity reasons. When compilation was reconsidered much later, other approaches were introduced such as compiling to Java byte-code [17], which is then optimized and executed by the JVM. Or generating C code using code templates before handing it off to a compiler [10]. However, in some of these approaches the operator boundaries remain clearly visible in the generated code. In engines, such as Microsoft's Hekaton [5], which also generates C code as an intermediate language, the entire plan is collapsed into a single function, where the operator boundaries are no longer obvious. These approaches still use an iterator model however, unlike HyPer [15], which instead uses the data-centric consumption model, which maps more closely to the semantics of the operators. In later work by [14], it was shown that vectorization and prefetching can be used in a compiled, data-centric, execution engine by introducing stage boundaries.

In this work, we show how to generate code in LLVM IR that fully implements all operators – barring some rarely called methods such as memory allocation. This is unlike HyPer [15], which falls back to C++ for complex operators such as hash-join. Furthermore, our code generation process emits code that matches our proposed code model. Unlike HyPer, which explicitly generates `produce` and `consume` methods, our generated code matches the structure our proposed model.

In the evaluation section, we compare the performance of both vectorized, and compiled execution with a Volcano engine. Like in [9], we find that vectorized execution is better at hiding the latency of cache misses than data-centric compiled execution, when materialization boundaries are not used – yielding a greater speedup compared to the Volcano model. We additionally show that is also the case when applied to a graph database.

6 CONCLUSION

In this work, we have shown how to effectively compile complete query plans into machine code using LLVM. We have introduced an abstraction model to reason about generated code. And we have shown how to effectively generate code using the proposed model for different operators. We have also shown how materialization barriers can be inserted, and how to integrate operators that don't work well with the consumer-style execution model. Furthermore, we have shown how to generate code for a hash-join operator.

In our evaluation, we have found that both compilation and vectorization provide a significant speedup compared to Volcano-style execution. The compiled approach does outperform the vectorized engine in many query classes on average, especially for the more complex query classes, with larger intermediate cardinalities. However, for individual queries, vectorization is still preferred in the majority of the cases, as it can better make use of inter-tuple parallelism and prefetching. A future iteration of the compiled engine may make use of materialization barriers to implement the same group prefetching the vectorized engine is capable of. Furthermore, compilation adds an up-front cost. A future planner may be able to weigh the expected speedup against the expected cost and decide to not, or partially compile a query.

REFERENCES

[1] Boncz, P., Zukowski, M., and Nes, N. Monetdb/x100: Hyper-pipelining query execution. *2nd Biennial Conference on Innovative Data Systems Research, CIDR 2005* (01 2005).

[2] Chamberlin, D. D., Astrahan, M. M., King, W. F., Lorie, R. A., Mehl, J. W., Price, T. G., Schkolnick, M., Griffiths Selinger, P., Slutz, D. R., Wade, B. W., and et al. Support for repetitive transactions and ad hoc queries in system r. *ACM Trans. Database Syst. 6*, 1 (Mar. 1981), 70–94.

[3] Chen, S., Ailamaki, A., Gibbons, P., and Mowry, T. Improving hash join performance through prefetching. vol. 32, pp. 116– 127.

[4] Conway, M. E. Design of a separable transition-diagram compiler. *Commun. ACM 6*, 7 (jul 1963), 396–408.

[5] Freedman, C., Ismert, E., Larson, P.-Å., et al. Compilation in the microsoft sql server hekaton engine. *IEEE Data Eng. Bull. 37*, 1 (2014), 22–30.

[6] Graefe, G. The volcano optimizer generator: Extensibility and efficient search. In *ICDE* (1993), pp. 209–218.

[7] Have, C. T., and Jensen, L. J. Are graph databases ready for bioinformatics? *Bioinformatics 29*, 24 (Dec 2013), 3107–3108.

[8] Hölzle, U. Adaptive optimization for self: reconciling high performance with exploratory programming, 1994.

[9] Kersten, T., Leis, V., Kemper, A., Neumann, T., Pavlo, A., and Boncz, P. A. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *PVLDB 11*, 13 (2018), 2209–2222.

[10] Krikellas, K., Viglas, S. D., and Cintra, M. Generating code for holistic query evaluation. In *In ICDE* (2010).

[11] LLVM Contributors. LLVM Language reference manual, 2020.

[12] Lorie, R. A. XRM - an extended (n-ary) relational memory. *Research Report / G / IBM / Cambridge Scientific Center G320-2096* (1974).

[13] Manegold, S., Boncz, P. A., and Kersten, M. L. Optimizing database architecture for the new bottleneck: Memory access. *The VLDB Journal 9*, 3 (Dec. 2000), 231–246.

[14] Menon, P., Pavlo, A., and Mowry, T. C. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *PVLDB 11* (2017), 1–13.

[15] Neumann, T. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow. 4*, 9 (June 2011), 539–550.

[16] Neumann, T., and Leis, V. Compiling database queries into machine code.

[17] Rao, J., Pirahesh, H., Mohan, C., and Lohman, G. Compiled query execution engine using jvm. In *Proceedings of the 22nd International Conference on Data Engineering* (USA, 2006), ICDE '06, IEEE Computer Society, p. 23.

[18] Truong, Q. D., Truong, Q. B., Dkaki, Taoufiq", e. P. C., and Barolli, L. Graph methods for social network analysis. In *Nature of Computation and Communication* (Cham, 2016), Springer International Publishing, pp. 276–286.

[19] Yoon, B. H., Kim, S. K., and Kim, S. Y. Use of Graph Database for the Integration of Heterogeneous Biological Data. *Genomics Inform 15*, 1 (Mar 2017), 19–27.

[20] Zhou, J., and Ross, K. A. Implementing database operations using simd instructions. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2002), SIGMOD '02, Association for Computing Machinery, p. 145–156.