Incremental Fusion: Unifying Compiled and Vectorized Query Execution

Benjamin Wagner, André Kohn, Peter Boncz*, Viktor Leis

Technical University of Munich {firstname.lastname}@tum.de

*CWI** boncz@cwi.nl*

***Abstract*—Modern high-performance analytical query engines follow one of two execution paradigms. Vectorized engines implement an interpreter for relational algebra operators that operates on batches of tuples to maximize performance. Compiling engines, on the other hand, generate optimized and specialized code for every query. This paper unifies these two approaches. We present Incremental Fusion, a novel execution paradigm for modern, high-performance query engines. An Incremental Fusion engine performs operator-fusing code generation – with a twist: The compiling engine generates its own vectorized interpreter. The engine uses a finite set of building blocks below relational algebra for code generation. It can enumerate each building block and generate a vectorized primitive for it. The vectorized interpreter becomes a free byproduct of carefully choosing the right abstraction for code generation. This allows an Incremental Fusion engine to dynamically switch between vectorized interpretation and operator-fusing code generation. We demonstrate Incremental Fusion in our open-source prototype engine InkFuse. We measure InkFuse against the state-of-the-art vectorized and compiling engines DuckDB and Umbra. InkFuse is able to achieve competitive performance both for low-latency processing, and compute-intensive long-running queries.**

I. Introduction

Analytical query engines have evolved significantly over the past years. Larger main-memory sizes and increased storage bandwidth led to query engines becoming increasingly compute bound. To maximize performance on modern hardware, engines implement one of two query execution paradigms.

Vectorized engines implement an interpreter for relational algebra that operates on tuple batches using columnar primitives [1]. For example, when interpreting the build side of a hash join, the engine invokes primitives for hashing, materializing rows, and inserting pointers to the materialized rows into a hash table. The primitives amortize interpretation overhead and have predictable data access patterns that result in high performance on modern hardware [2].

Code-generating engines just-in-time (JIT) compile each query into machine code. This allows the engine to generate specialized code that fuses pipelined relational operators into the same loop [3]. Tuples can reside in CPU registers across operator boundaries, minimizing memory and cache accesses.

Virtually all commercial state-of-the-art OLAP engines are based on either vectorization [4], [5] or code generation [6], [7]. Kersten et al. [2] experimentally compare the two models and find that while both models can be efficient, neither dominates the other in all use cases. The intellectual battle between the two models has been ongoing for one decade, and neither of the two approaches has superseded the other.

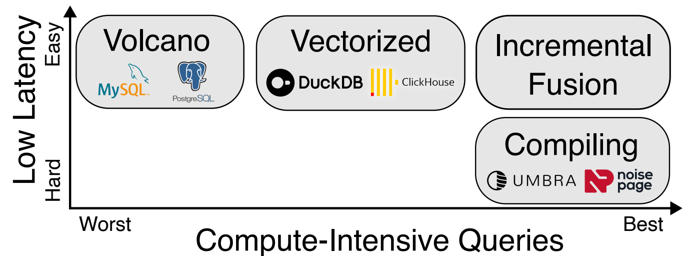

Fig. 1. Combining the best of vectorized and compiled query execution.

Recent advances [8], [9] blur the line between both models. Based on compilation, they make it possible to generate code that imitates vectorized execution. For long-running queries, this consistently outperforms basic vectorization and compilation. However, we argue that this does not solve the core disadvantage of compiling engines: *compilation takes time while a vectorized interpreter can execute queries instantly.* Compiling engines that imitate vectorized execution still need to generate code whenever a new query enters the system. While it is possible to build a compiling engine for low-latency data processing, it requires a sophisticated compilation stack and deep expertise in code generation [10]–[13].

State-of-the art compiling engines translate queries into a custom intermediate representation (IR). This IR is then consumed by different execution backends. In the compiling engine Umbra, a backend optimized for compute-intensive queries takes the custom IR and compiles it to low-level LLVM IR [14]. It then uses LLVM to generate an efficient executable for the query. Since generating machine code with LLVM can take hundreds of milliseconds, Umbra implements multiple backends optimized for low-latency query processing on different CPU architectures [11], [13]. These backends directly translate the custom IR to x86 or ARM Assembly. They implements novel variations of compiler optimizations such as register allocation and liveness checking that minimize compilation time. While the machine code generated by these backends is less performant, they can start processing tuples almost instantaneously. Since a compiling engine cannot know in advance which backend performs best, it needs additional infrastructure to dynamically switch between the backends [10]. This allows competitive performance for long- and short-running queries. Overall, building a low-latency code-generating engine requires deep compiler expertise. This expertise is not needed to build a fast vectorized engine,

significantly reducing the barrier of entrance.

So far, no engine has managed to combine all strengths of code generation and vectorized interpretation. There is no compiling engine that makes low-latency query execution easy. There is no vectorized engine that can match the computational efficiency of a compiling engine. This has a tangible impact on commercial database systems. Companies need to choose between the two execution paradigms, carefully evaluating which strengths matter most for them. Recently, vectorization has come out ahead more frequently [15]–[19]. Databricks, for example, recently switched from compilation to vectorization, arguing that it is "easier to build, profile, debug, and operate at scale" [19].

This paper presents Incremental Fusion, *the first execution paradigm unifying compilation and vectorized interpretation.* Incremental Fusion subsumes compilation and vectorized interpretation, combining both of their strengths. Incremental Fusion can explore the wide space of execution strategies between these two extremes, making it possible to mix interpretation and code generation in almost arbitrary ways.

Incremental Fusion is powered by a novel suboperator abstaction below relational algebra. There are a finite number of suboperators that can represent the execution plan for arbitrary SQL queries. The engine performs operator-fusing code generation on these suboperators.

Using these finite building blocks for code generation, *the query engine generates its own vectorized interpreter*. It implements a special source and sink that read from and write to columnar chunks of data, respectively. By wrapping each suboperator between this source and sink, the engine generates a vectorized primitive for the suboperator. *The vectorized interpreter becomes a free byproduct of having chosen the right abstraction for code generation*.

This allows Incremental Fusion to combine the best of vectorized interpretation and compilation. Since the engine can enumerate all suboperators, a complete vectorized interpreter can be generated ahead of time, allowing for low-latency processing. At the same time, the engine can perform traditional operator-fusing code generation, enabling high peak performance. The engine can dynamically switch between these strategies at runtime, and combine them in novel ways. For example, it is possible to split an executable pipeline into three steps, performing compilation for the first and last step, while interpreting the middle one.

All of this is possible without over-complicating the query engine. An Incremental Fusion engine does not require a sophisticated compilation stack. By generating its own interpreter, the engine can effectively hide compilation latencies. Our open-source, proof-of-concept Incremental Fusion engine InkFuse generates C [20]. InkFuse is able to compete with the state-of-the-art compiling and vectorized systems Umbra and DuckDB across queries and data sizes.

II. Background: Query Execution Paradigms

In this section, we first discuss interpreted and compiled query execution individually, and then compare them.

A. Interpreting Query Engines

Interpreting relational algebra is the most common approach to evaluating queries. Engines of this class first translate queries into a plan of relational algebra operators. Afterwards, they use generic implementations for these operators to interpret the plan directly. Implementing these generic operators traditionally follows the Volcano iterator model [21]. With Volcano, each relational operator provides a simple `nextTuple()` function that emits a single result row. This function is then called repeatedly until the entire result set is produced. Query engines can evaluate complex relational algebra plans by composing multiple of these operators.

However, Volcano engines are not tailored to modern CPUs. They exhibit bad code locality, have complex data access patterns, and require multiple virtual function calls to produce even a single tuple. Yet, this model is still commonly used in OLTP engines where execution is usually not bottlenecked by CPU-intensive work on large data sets [22], [23].

To enable high-performance analytical processing, MonetDB moved away from the Volcano iterator model [24], [25]. Instead of operating on a single row, the interpreter operates on the full, materialized input and produces all output rows at once. This full-materialization model allows for high code locality with simple data-access patterns, resulting in efficient code for modern CPUs. Even today, MonetDB is still one of the faster open-source analytical systems.

MonetDB/X100 went beyond MonetDB by implementing a hybrid between the Volcano model and full materialization [1]. Rather than evaluating queries tuple-at-a-time, X100 implements a vectorized interpreter that operates on tuple batches. Vectorized engines implement operators and expressions by composing fine-granular primitives. These primitives are precompiled and specialized for all type combinations, eliminating the virtual function calls of Volcano engines and unlocking the super-scalar capabilities of modern CPUs. A standard hash join, for example, is implemented using primitives for performing column-level hashing, combining of these hashes, and inserting rows into a hash table. Vectorized interpretation is the most widely-used approach to high-performance analytical query processing today [4], [5], [15]–[17], [19], [26].

Fig. 2a presents a query being evaluated in a vectorized engine. We show four of the primitives invoked during query execution. Most primitives consist of a tight for-loop around a set of input and output columns (1, 3, 4). Only hash-join probing (2) is an exception. As a single input tuple can have multiple matches, the code contains a nested loop. Some primitives such as comparison with a constant (4) require extra arguments. As a user can send arbitrary SQL queries, having a comparison primitive for each integer is not feasible. The same primitive is used for all constants. The constant is provided by the runtime system. Objects such as hash tables are passed to the primitives in a similar way (1, 2).

B. Compiling Query Engines

Another approach to high-performance query processing is runtime code generation. Rather than implementing an inter-

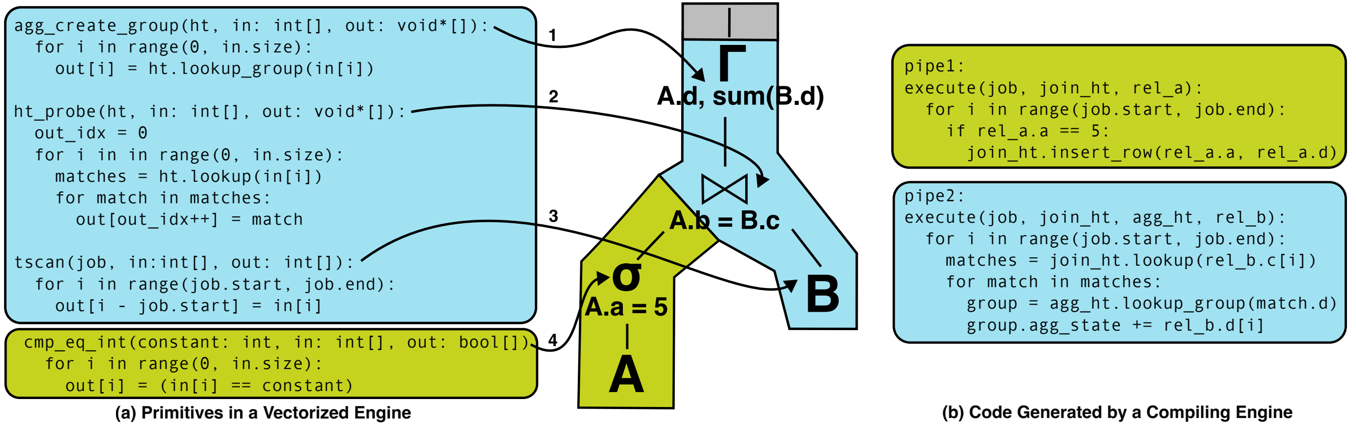

Fig. 2. Execution of the query `SELECT a.d, sum(b.d) FROM A a, B b WHERE a.a = 5 AND a.b = b.c GROUP BY a.d.`

preter for relational algebra, the engine generates optimized code for every query. Every relational algebra operator implements a `produce()`, and `consume()` interface. Calling these functions generates code for the operator in the given query. To evaluate the query, this code gets compiled and executed [3]. Runtime code generation is used in academic and industrial systems [6], [14], [27], [28].

The code generated by these engines is highly specialized. As Fig. 2b shows, all operators of a pipeline are fused into a single loop, blurring operator boundaries. In the second pipeline, the outer loop scans table B. This is followed by a hash join probe. The inner loop iterates over the matching keys in the join's hash table. Finally, each match gets added to the state of the final aggregation. This is fundamentally different from vectorization, where each operator is broken up into simple primitives and operator boundaries stay intact.

A compiling query engine cannot process any tuples before it has generated specialized code for the query. Significant research has focused on bringing low-latency query processing to compiling engines [10]–[13]. Different CPU architectures require custom compilation stacks to quickly generate executable code. This allows compiling systems like Umbra to compete with vectorized engines like DuckDB for low-latency query processing.

C. Interpretation vs. Compilation

Cutting-edge analytical query engines employ either vectorization or code generation. Recent research shows that in terms of raw system performance, there is no clear-cut winner [2]. For compute-intensive queries, code generation offers the best performance. The highly specialized code of compiling engines keeps tuples in CPU registers beyond operator boundaries. Meanwhile, vectorized query engines win in other cases. The simplicity of the vectorized primitives allows compilers to generate optimized code for modern CPUs. The primitives are often branch-free, and make it easy for prefetchers and out-of-order execution to hide cache miss latencies. This is especially useful when accessing hash tables that exceed the cache size. Compared to tuple-at-a-time code generation, a vectorized engine is able to generate more independent loads that saturate the memory bandwidth.

As different execution paradigms perform best in different settings, Relaxed Operator Fusion (ROF) attempts to bring the performance benefits of vectorized query engines into compiling systems [8]. Traditional compiling query engines fuse a pipeline into compact nested loops to keep tuples in CPU registers across operator boundaries. With ROF, a query engine may introduce intermediate staging points that partially materialize tuples. This yields opportunities for using aggressive prefetching and SIMD-instructions after the staging points. ROF brings many of the benefits of batching primitives in vectorized engines to compiling query engines. However, ROF is still a code-generating execution paradigm. It suffers from upfront compilation latencies before any tuples can be processed. As a result, low-latency processing in an ROF engine necessitates a complex compilation stack similar to the one found in Umbra.

Given that there is no clear-cut winner in terms of performance, a common reason for choosing the vectorized model in industry is system complexity [15], [19]. Compiling query engines require a complex code generation stack to hide compilation latencies [10], [11]. This makes it harder for developers to ramp up quickly and have high velocity, which is exasperated by code-generating engines being hard to profile and debug [29], [30].

III. Incremental Fusion for Expressions

This section outlines the basic concepts of Incremental Fusion. We show how to build a code-generating expression evaluator that can also generate a complete vectorized expression interpreter. The ideas presented in this section are the building blocks needed to generalize Incremental Fusion to arbitrary queries in Section IV.

In a traditional compiling query engine, each pipeline becomes an executable fragment of code. A pipeline has a source operator that generates code for reading data, and a sink operator that generates code for materializing the pipeline result. The most common source is a table scan, which directly accesses the underlying storage engine. Pipelines can also read from aggregation hash tables, or sorted buffers after executing an `ORDER BY` clause. Sink operators can materialize data into hash tables for aggregations or the build side of joins. The

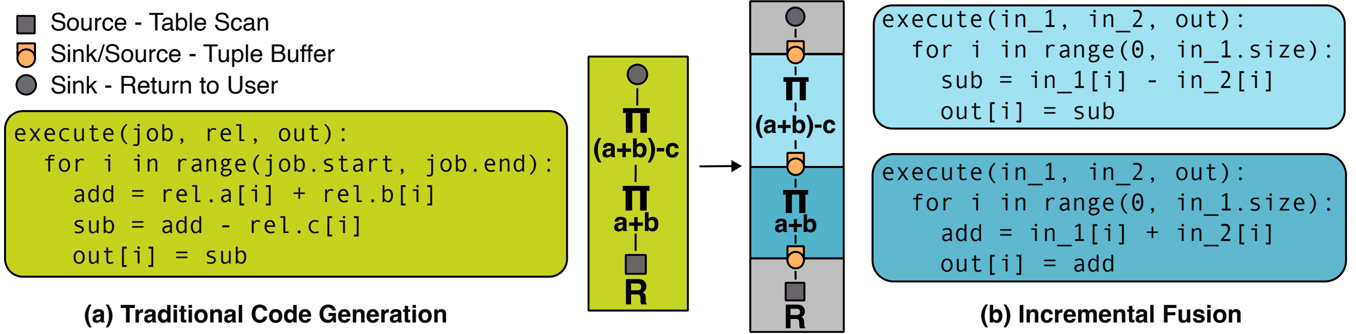

Fig. 3. Executing the query `SELECT (a + b) - c FROM R`. To generate the vectorized primitive for an operator, an Incremental Fusion engine wraps it in a tuple buffer source/sink and performs regular operator-fusing code generation.

sink operator of a query's final pipeline materializes the query result and sends it to the user.

Viewed in this light, the Relaxed Operator Fusion (ROF) model discussed in Section II-C adds a novel type of source and sink to the engine. At an intermediate staging point, the pipeline is split into two steps. A new tuple buffer sink is attached to the end of the first step. The second step starts with a new tuple buffer source. The sink of the first step generates code that materializes the step's result data into a shared buffer. The source of the second step generates code that reads the materialized data from this shared buffer. The two steps are run in lockstep. When the first step fills up the tuple buffer, the engine invokes the second step, draining the buffer again. This allows the materialized tuples to stay in the CPU cache.

Breaking a pipeline into steps does not add complexity to the engine's code generation stack. The compilation stack does not differentiate between a pipeline and the more fine-grained steps. Both are DAGs of operators, starting with a source and ending with a sink. Only the engine's scheduler needs to be aware of the distinction. Pipelines are run one after the other, while the steps in a pipeline are run in lockstep.

The modularity of the tuple buffer source and sink operators is what powers Incremental Fusion. ROF cuts a pipeline in half and strategically places a sink and a source to improve performance. This, however, is just a special case of how a pipeline can be split into steps. Incremental Fusion takes the ideas behind ROF to the extreme by allowing to place tuple buffers between *every* operator in a pipeline. This allows Incremental Fusion to support both execution models in the same engine. If no buffer is introduced, the code is identical to the one of compiling query engines. If buffers are introduced after every operator, the execution is virtually the same as the one from a vectorized engine and allows the use of pre-compiled primitives.

Fig. 3 shows this for the query `SELECT (a + b) - c FROM R`. The projection is broken into two physical algebra operators, one for each arithmetic operation in the query. The left-hand side of the figure shows the code generated by a traditional compiling query engine. The complete expression is fused into the main loop of the table scan. As a code-generating execution paradigm, Incremental Fusion can create exactly the same, specialized code for the query.

At the same time, Incremental Fusion can also generate all primitives used by a vectorized engine to evaluate this query. To do this, it slices the original pipeline into one step per physical operator. Each step starts with a tuple buffer source, followed by the original physical operator, and ends in a tuple buffer sink. This is illustrated on the right-hand side of the figure. The resulting pipeline is split into four steps. The first step reads rows from the base table into the first intermediate buffer. The second and third steps perform the addition and subtraction, respectively. Finally, the fourth step reads from the subtraction's intermediate buffer and returns the query result.

Both the record batches of a traditional vectorized engine, and the intermediate materialization buffers of an Incremental Fusion engine are columnar, contiguous memory regions containing tuples getting pushed through the pipeline. As a result, the code generated by an Incremental Fusion engine performs the same computation as the expression's primitive in a vectorized query engine. In the example, *the Incremental Fusion engine generates the vectorized primitives for adding and subtracting two columns*.

An Incremental Fusion engine uses this property to generate a vectorized expression interpreter. It (1) exhaustively enumerates all the possible primitive expressions a SQL query can contain, such as the addition of two integers, or evaluating the sine function on a float. For each of these, it (2) creates a step that represents this expression. The step starts with a tuple buffer source, followed by a physical operator projecting the primitive expression. It ends in a tuple buffer sink. Afterwards, it (3) uses the regular, operator-fusing code generator on this step to create the corresponding vectorized primitive. When an (arbitrarily-complex) expression enters the system for evaluation, the engine breaks it into a DAG of these finite primitives, and uses the previously generated code to interpret the expression tree. However, the engine is not just limited to vectorized interpretation. It can also perform just-in-time compilation to generate fused code for the expression tree.

Compared to ROF, an Incremental Fusion engine does not need to generate any code when a query enters the system. It can directly initiate high-performance query processing using the complete vectorized interpreter that was generated ahead of time. The next section describes how these ideas can be extended to every relational operator.

IV. Incremental Fusion

This section gives an in-depth overview of Incremental Fusion. It extends and formalizes the ideas outlined in Section III. We show how carefully choosing an intermediate representation (IR) below relational algebra allows the engine to generate a complete vectorized query interpreter. This enables the engine to perform both operator-fusing code generation and vectorized interpretation, without increasing system complexity.

A. Breaking Up Complex Operators

Section III presented Incremental Fusion for expressions. The engine can generate a complete vectorized expression interpreter because there is only *a finite number of expression signatures*. This section extends this idea to arbitrary operators.

We can model any operator P that generates code in a compiling engine. The operator P is parametrized over $p_1, p_2, \ldots, p_n$. The parameters of an operator impact the code generated in calls to `produce()` and `consume()`. When the operator is used to generate code for a query it is instantiated with query-dependent, concrete values for $p_1, \ldots, p_n$. Two instantiations of P, $\tilde{P}$ and $\hat{P}$ generate the same code if and only if $\forall k \in \{1, \ldots, n\} : \tilde{p}_k = \hat{p}_k$.

In this model, the expression operator E is parametrized over $e_1 \in E_O$ modelling the opcode, and $e_2 \in E_T$, $e_3 \in E_T$ modelling the input types. Since the set of opcodes E_O and types E_T is finite, there only is a finite set of possible instantiations of the operator: $(e_1, e_2, e_3) \in E_O \times E_T \times E_T$.

We can now define the core invariant that every operator in an Incremental Fusion engine needs to satisfy.

Enumeration Invariant. *A code-generating operator P that is parametrized over $p_1, \ldots, p_n$ is capable of Incremental Fusion if and only if there exists a finite set of parameters A, such that for every possible instantiation of the operator $\tilde{P}$, it holds that $(\tilde{p}_1, \ldots, \tilde{p}_n) \in A$.*

For an operator P that satisfies the invariant, the engine can enumerate every possible instantiation $\tilde{P}$, wrap it in a tuple buffer source and sink and use its regular compilation stack to generate a vectorized primitive for $\tilde{P}$. Since the generated code is completely specified by $\tilde{p}_1, \ldots, \tilde{p}_n$, we can be sure that no matter what SQL query the user sends, any instantiation of P in the query will have a corresponding pre-compiled primitive. *The enumeration invariant allows the engine to generate a complete vectorized interpreter ahead of time.*

The expression suboperators in Section III respect the enumeration invariant. We can exhaustively enumerate the finite set of supported function signatures. As a counterexample, a holistic aggregation operator cannot satisfy the enumeration invariant. There are infinite possible combinations of aggregate keys and functions, breaking the invariant. This makes it impossible to enumerate all possible instantiations of an aggregation operator ahead of time.

To build a complete Incremental Fusion engine, we need to break up the relational algebra operators into a set of more fine-grained building blocks that respect the enumeration invariant. Incremental Fusion achieves this through a custom suboperator IR. Custom suboperator IRs have recently been adopted by different systems to increase the modularity of query engines and simplify query execution on heterogeneous hardware [31]–[34]. Designing a suboperator IR where every suboperator satisfies the enumeration invariant is the core challenge when building an Incremental Fusion engine.

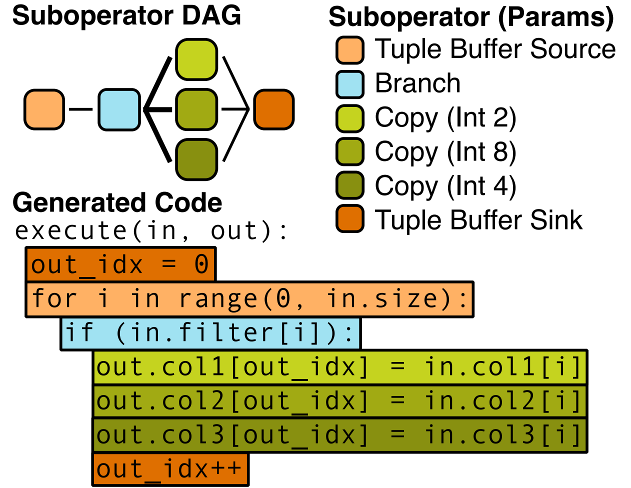

Fig. 4. Incremental Fusion filter representation.

Vectorized query engines face similar problems. They also need to map complex relational algebra operators to a finite set of pre-existing primitives. MonetDB created the concept of Binary Association Tables (BAT) and the Monet Interprer Language (MIL) to tackle this problem [1], [35]. Similar abstractions are found in all modern vectorized query engines [16], [17]. Incremental Fusion brings the granularity of the vectorized primitives into code-generating systems.

B. Filters

Many vectorized query engines use selection vectors to indicate which rows in a record batch are active. Selection vectors can be represented in different ways [36].

Other vectorized query engines such as ClickHouse do not use selection vectors [17]. Instead, the batches of rows passed between primitives are always dense. The filter evaluates the predicate and then removes all rows that do not satisfy the predicate from the columns in the current batch. While this makes the filter primitives slightly more expensive, the non-filter primitives do not have to worry about selection vectors anymore. Their code can be more efficient on modern hardware due to fewer branches and simple memory access patterns. Our open-source Incremental Fusion engine InkFuse represents filters this way [20].

An Incremental Fusion engine cannot have a holistic filter suboperator. Since we can filter any number of columns with arbitrary types, such an operator would violate the enumeration invariant. The engine would not be able to generate all vectorized filter primitives ahead of time. To respect the invariant, the filter needs to be broken into suitable suboperators.

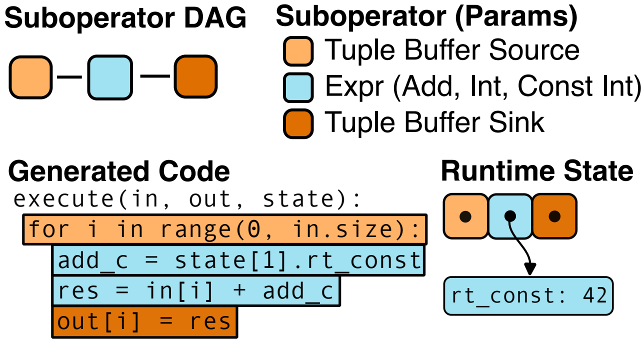

Fig. 5. Evaluating `SELECT x + 42 FROM t` in an Incremental Fusion engine. The constant 42 is provided by the runtime system and gets resolved by the generated code.

In InkFuse, a relational filter operator filters n columns based on a boolean column. The filter operator gets broken into $n+1$ suboperators. This is shown in Fig. 4. The first suboperator generates the required branch based on the boolean input column. It does not have any parameters, since we always filter on a boolean column. Afterwards, one suboperator per filtered column generates the code performing data copying. Every copy suboperator has an input dependency on the branching suboperator. This way, the copying is performed in the filtered scope. The copy suboperator is parametrized over the underlying type being copied. *The unlimited degree of freedom of the holistic filter operator in a traditional engine is now encoded in the topology of the suboperator DAG.*

Since the above suboperators respect the enumeration invariant, the engine can generate all filter primitives for the vectorized interpreter. The engine only has as many filter primitives as it has data types. Primitives for parameterized data types, such as decimals, are defined based on the underlying storage type, which keeps the number of primitives small.

C. Expressions with Constants

Section III outlined how a compiling query engine can generate a vectorized expression interpreter. This is possible because the expression suboperators satisfy the enumeration invariant. Expressions are parametrized by an operation such as subtraction or the cosine function, as well as input types. This set is finite and can be exhaustively enumerated. However, this interpreter cannot yet evaluate SQL queries such as `SELECT x + 42 FROM t`.

The problem is that a user can send queries with arbitrary constants. We cannot create a primitive that adds 42 to an integer column, as this necessitates a primitive for every possible constant. This, in turn, breaks the enumeration invariant. Therefore, *non-discrete parameters get resolved at runtime* in an Incremental Fusion engine and are stripped from all primitives. For each expression primitive shown in Section III, there is a single additional variant of the primitive that accepts a constant argument. This is similar to expression primitives in traditional vectorized engines. Once the required constants are known during query execution, they are provided by the runtime system. This is shown in Fig. 5.

Every suboperator can have custom runtime state. This state can contain objects like hash tables, or constants needed to evaluate the primitive. The runtime state gets set up once the query is known. Within the generated code, the expression suboperator resolves the constant by accessing the runtime state. The value read from the runtime state is used to perform the addition during query execution. This allows the engine to execute expressions with arbitrary constants while respecting the enumeration invariant, completing the Incremental Fusion expression evaluator.

D. Aggregations

In a similar way to a relational filter, a relational aggregation needs to be broken up into a DAG of suboperators that respect the enumeration invariant. By doing this, it is possible to generate a vectorized interpreter for arbitrary aggregations based on these suboperators.

The goal of Incremental Fusion is to choose suboperators that yield a vectorized interpreter *without making the system more complex*. Only the execution backend may be aware of whether it executes a query using the vectorized interpreter or operator-fusing JIT compilation. All other components such as the suboperator implementation and the compilation stack need to be unaware of the different execution modes.

Engines performing operator-fusing code generation treat aggregations differently from vectorized engines, making it harder to create suboperators that work for both execution paradigms. A vectorized engine performing an aggregation usually takes an entire chunk of rows through a multi-step process. First, a set of primitives (1) computes the key hashes. Afterwards, a primitive (2) looks up an initial slot in the hash table based on the hashes. Then, set of primitives (3) performs key comparisons, returning a new boolean column that is true if the key in the current slot matches the row being aggregated. The set of primitives that gets invoked for key checking is query-dependent. Different primitives are used for different types. Compound keys require multiple primitives to be invoked. Once the engine knows which keys in the current chunk match the keys in the hash table slot, another primitive (4) advances the slots for the rows where the key does not match due to hash collisions. The interpreter iteratively invokes primitives (3) and (4) until all collisions are resolved. Only then another set of primitives (5) updates the aggregation state. Some extra care is needed for rows that do not have a match yet and need to create a new group in the hash table.

Meanwhile, a code-generating engine goes through the above flow one tuple at a time. This means that a single row gets hashed (1), leading to an initial hash table lookup (2). After, a nested loop takes the current tuple through the entire collision chain (3-4). Custom key-checking logic based on the used aggregation key is performed in the body of that loop. This is possible because the aggregate key layout is known at compile time. Finally, the aggregation state gets updated (5).

We see that resolving collision chains is done differently. A code-generating engine resolves collisions through a specialized loop on every row. This loop advances the iterator for the

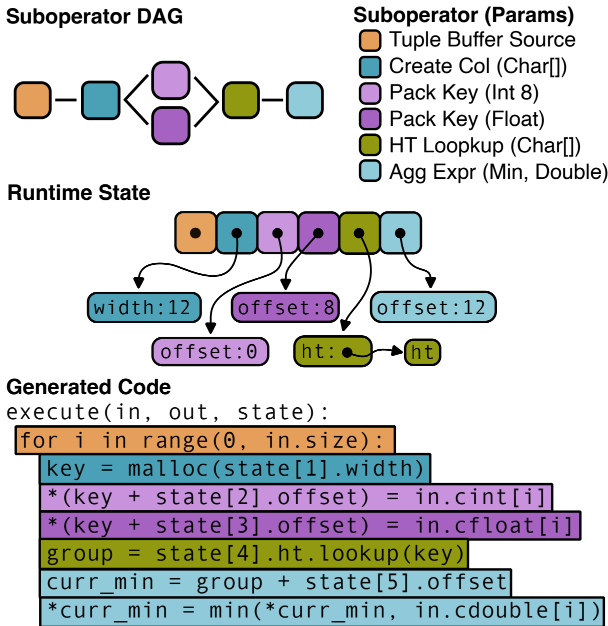

Fig. 6. Evaluating `SELECT cint, cfloat, min(cdouble) FROM t GROUP BY cint, cfloat`. The above pipeline adds the input rows to an aggregate hash table.

current row until the keys match. The vectorized engine unrolls this loop explicitly within the interpreter. Due to the block-based nature of a vectorized engine, the interpreter invokes the primitives for key checking and advancing iterators iteratively until the longest chain is resolved.

Incremental Fusion unifies the two approaches by moving collision resolution into the hash tables themselves. The hash tables are part of the runtime system and do not participate in code generation. This way, the engine knows that the pointers returned by a hash table lookup already point to the correct key. This solves the problem, as it moves the collision resolution loops that are different in the two types of engines outside of the generated code. We no longer need manual loop unrolling in the interpreter.

In order to move key checking into the runtime system, our Incremental Fusion engine InkFuse implements a row layout for tuples. The key columns participating in an aggregation are packed densely into a contiguous memory region. Fig. 6 shows the suboperator layout for the query `SELECT cint, cfloat, min(cdouble) FROM t GROUP BY cint, cfloat`. The query performs an aggregation on an eight byte integer and a four byte floating-point number. In InkFuse, a first primitive allocates a new twelve byte compound key. Another primitive then packs the eight byte integer into the compound key at offset zero. Finally, a third primitive packs the four byte float into the compound key at offset eight. Before query execution starts, the runtime system allocates an empty hash table that expects contiguous twelve byte keys. A special primitive now performs lookups in that hash table with the packed compound key. The hash table is responsible for hashing and key comparison. Key comparison happens through a simple and very efficient `memcmp`. The hash table lookup returns a pointer to the correctly resolved slot. A final primitive updates the `min` aggregate state. In the successive pipeline producing the aggregation result, primitives for key unpacking reconstruct the original keys from the row layout.

These primitives reuse a lot of the infrastructure developed for expressions with constants in Section IV-C. As we can aggregate by any number of columns, the offset for key packing can be arbitrary. Since the engine cannot generate primitives for every possible offset, the offset is resolved as a runtime parameter by accessing the operator state. This allows the key packing suboperators to respect the enumeration invariant.

We again use the enumeration invariant to generate a full vectorized interpreter. As before, this is done by exhaustively enumerating the suboperators that can participate in an aggregation, wrapping them in a tuple-buffer source and sink, and then using the regular operator-fusing code generation stack of the engine to generate the vectorized primitives. To generate all primitives, the engine generates key packing and unpacking suboperators for all supported types, and aggregation suboperators for writing and reading for each supported aggregate function and type combination.

Some care has to be taken when performing aggregations on variable-size keys like strings. Here, a fixed-width row layout with a `memcmp` does not work. For keys like this, the engine has a specialized row layout which contains a variable number of eight byte slots for pointers to string values before the packed, fixed-size key. When aggregating variable-size keys like this, a specialized hash table is used. Key equality comparisons now happen through a `strcmp` on each variable-size slot, followed by a single `memcmp` on the final fixed-size part. This also works for collations by mapping the strings to a representative of the equivalence class. When using a case-insensitive collation for example, every key is turned to lower-case. The keys `ABCD` and `aBcD` are both mapped to `abcd` and compare equal. The normalized representation is only used for key comparison. The original string needs to be stored in the payload and passed on to the parent suboperator. The open-source collation library ICU natively supports this transformation.

Performing the explicit key packing and unpacking for the row layout leads to some extra work compared to directly operating on non-packed rows. In practice, we found that the packing does not cause noticeable performance degradation. Packing happens on data that is in the L1 cache already, providing extremely high bandwidth and low access latency. As the hash tables used in an aggregation grow, this cost is dominated by cache misses to load the hash table from lower-level caches or even main memory. A benefit of our approach is that hashing and key comparison are done on contiguous memory regions. If we only aggregate by a single column, the engine performs no packing but just uses the raw column directly. Similar row layouts are used by modern vectorized query engines such as DuckDB [37].

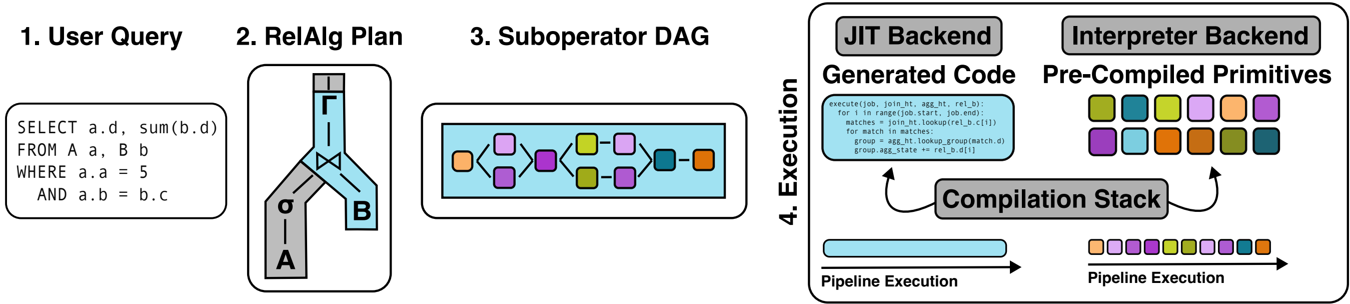

Fig. 7. After query parsing and optimization (1, 2), each pipeline gets transformed into a DAG of suboperators (3). This DAG is then passed to an execution backend (4). All backends rely on the same compilation stack, minimizing system complexity.

E. Joins

Joins build on the concepts of aggregations outlined in Section IV-D. For the build side of a join, key and payload get packed into a contiguous row layout. The tuple's row layout is then inserted into a hash table. Compared to aggregations, the hash table for a join can store duplicate keys.

The probe side of a join is more interesting. The challenge is that probing can have multiple matches in the hash table. The engine needs to produce a result tuple for each match. For this, it again packs key and payload into a contiguous row layout. A hash join probing primitive is responsible for accepting a packed input row and producing a set of packed output rows. Since the probing primitive operates on abstract packed rows rather than arbitrary type combinations it respects the enumeration invariant. The probing suboperator returns two values in row layout. The first one is the packed key and payload from the build side. The second value is the packed row from the probe side. After probing, suboperators unpack the columns from the packed row layouts.

In compiling engines, operators that increase output cardinality generate nested loops. In vectorized engines, individual operators such as the hash join explicitly deal with changing output cardinalities. Some engines have complex iterator logic in these operators to limit the size of output chunks [16]. Other engines implement variable-size output chunks [17]. InkFuse takes the latter approach and handles changing output cardinalities through an additional tuple buffer sink. For pipelines containing a suboperator that can explode chunk sizes, this sink can exponentially grow the internal column representation. This allows InkFuse to completely hide the complexity of the growing chunks of a vectorized engine in a single sink suboperator. The suboperator for join probing is unaware of growing chunks in the vectorized interpreter, it simply receives an input stream and transforms it into an output stream. This also means that when using Incremental Fusion for more traditional operator fusion, the engine does not have to pay any bookkeeping overhead for potentially exploding chunks in the interpreter. The tuple buffer sinks are only used to generate vectorized primitives and are never passed to the JIT backend for traditional code generation.

V. Engineering Incremental Fusion

This section describes how to build an Incremental Fusion engine, focusing on the engineering challenges encountered when assembling an end-to-end query pipeline. We also show how an Incremental Fusion engine can dynamically switch between vectorization and compilation at query runtime.

A. Incremental Fusion Query Life Cycle

The life cycle of a query in an Incremental Fusion engine is visualized in Fig. 7. We have implemented this pipeline in our multithreaded engine InkFuse [20]. By choosing the right abstractions, an Incremental Fusion engine is highly modular and easy to reason about.

The Compilation Stack: As outlined in Section IV, an Incremental Fusion engine generates code at the granularity of suboperators. The compilation stack of an Incremental Fusion engine turns a DAG of suboperators into executable code. Suboperators implement the same `produce` and `consume` interface found in traditional operator-fusing query engines [3]. Since suboperators have a much more narrow scope than holistic relational algebra operators, their code generation logic tends to be far simpler than in a traditional operator-fusing query engine. InkFuse implements twenty suboperators in about two thousand lines of C++ code. InkFuse generates C. However, other languages that compile to efficient machine code such as C++ or LLVM IR can also be used.

Despite supporting JIT compilation and vectorized interpretation, an Incremental Fusion engine only implements a single compilation stack. It is used for operator-fusing JIT compilation, as well as for generating vectorized primitives.

The Vectorized Interpreter: An Incremental Fusion engine uses the compilation stack to generate a vectorized interpreter. Since all suboperators respect the enumeration invariant, the engine can generate all the required primitives for vectorized query execution. These primitives are generated at engine compile time and are loaded once when starting the database.

The vectorized interpreter is an easy to generate artifact of having chosen the right abstraction for code generation. Since generating the interpreter uses the same compilation stack as traditional JIT code generation, the only additional code required to obtain the vectorized interpreter is the one that enumerates the suboperators.

The twenty suboperators implemented in InkFuse generate more than eight hundred primitives. In total, the generated interpreter consists of twenty thousand lines of C code.

The code generated for the interpreter is similar to hand-written primitives in modern vectorized query engines. The generated primitives perform well on modern CPUs. They are cache efficient, have easy to predict memory access patterns, and contain many independent instructions. Furthermore, the compiler is able to auto-vectorize many primitives and generate efficient SIMD instructions.

Query Planning: When a query enters the system, it undergoes traditional query parsing and planning. These are Steps (1) and (2) in Fig. 7. Once an optimized relational algebra tree is generated, each pipeline gets transformed into a suboperator DAG. This is done in one additional pass over the optimized algebra tree. Edges in the DAG represent code generation dependencies. The code for a node in the DAG cannot be generated before *all* input dependencies have generated their code. In Fig. 7, Step (3) breaks the blue pipeline into a suboperator DAG.

Query Execution: The query engine can choose how to execute the suboperator DAG of a pipeline. Different execution backends facilitate operator-fusing code generation and vectorized interpretation.

The JIT backend shown in Step (4) takes the suboperator DAG from Step (3) and uses the system's compilation stack to generate code. This yields a single executable that is invoked to push a batch of tuples through the pipeline.

The vectorized backend uses the primitives loaded on startup to interpret the suboperator DAG. Each suboperator in the pipeline gets mapped to a pre-compiled vectorized primitive. Since the engine exhaustively enumerated all possible suboperator instantiations when generating the vectorized interpreter, it can be sure that a suitable primitive was generated ahead of time. *This is why the enumeration invariant is essential for Incremental Fusion. By making sure the suboperators can be enumerated, the engine can match the entire query to pre-compiled primitives.* If the engine were unable to enumerate all possible suboperator instantiations, it could not generate a vectorized interpreter for arbitrary queries. This is shown in Fig. 7. All suboperators in the DAG generated in Step (3) have a corresponding pre-compiled primitive loaded by the interpreter backend. To push a set of tuples through the pipeline, the interpreter backend iterates over the suboperator DAG and invokes the pre-compiled vectorized primitives in some topological order.

Fig. 7 shows that the vectorized primitives are generated by the regular, operator-fusing code generation stack of the engine. As such, they require almost no special treatment in the runtime system. In our Incremental Fusion engine InkFuse, the vectorized backend has less than 50 lines of C++ code. These are mainly concerned with mapping a suboperator to a function pointer in the primitive cache. This also enables a hybrid execution backend that dynamically switches between vectorized interpretation and operator-fusing code generation. We present this backend in Section V-B.

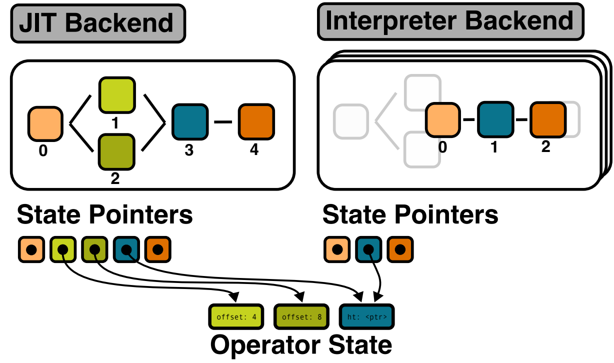

Fig. 8. Each suboperator can allocate a custom state object. The state can be shared across execution backends.

B. Adaptive Query Execution

It is highly query-dependent and hard to predict whether vectorization or compilation performs best for a query [2]. This section shows how an Incremental Fusion engine can dynamically switch execution modes at query runtime.

Injecting Operator State: Each suboperator can define a custom state object containing constant arguments for expressions, or pointers to runtime objects such as hash tables. Fig. 6 shows the runtime state for multiple suboperators. Suboperators access their state through an array of pointers passed to the generated code. During code generation, each suboperator gets a unique ID. The runtime state is resolved by accessing the array at the index of the ID. This is shown in Fig. 8. As the vectorized primitives are generated by the same JIT backend, they resolve their state in the same way. This is shown to the right of Fig. 8, where we interpret the blue suboperator (1).

A Hybrid Execution Backend: Using these abstractions, it is easy to build a hybrid adaptive execution backend similar to [10]. The hybrid backend can dynamically switch between compilation and interpretation during query execution.

When a query enters the system, the hybrid backend sets up the runtime state for every suboperator. It then wires the state pointer arrays of the interpreted and JIT backend to the shared suboperator state. This is shown in Fig. 8. Since all persistent query state is tied to these shared objects, the engine can safely switch between the backends.

The main challenge when building the hybrid backend is ensuring that the engine can pick the best performing execution mode with low overhead. InkFuse uses morsel-driven parallelism to push chunks of data through a pipeline in a multi-threaded way [38]. The hybrid backend chooses whether to use the JIT compiled code or the vectorized interpreter at morsel granularity.

When a new query enters the hybrid backend, no JIT compiled code is available for execution. The engine kicks off background code generation. Currently, InkFuse uses one thread per pipeline for background compilation. For queries with many pipelines, compilation overhead can be bounded by limiting the number of concurrent compilation jobs. InkFuse then starts as many worker threads as there are CPU

cores. As the vectorized interpreter is available right away, all worker threads begin executing the first pipeline using the interpreter. Once the generated code becomes ready, the hybrid backend can dynamically switch between interpretation and compilation. If query execution finishes before background compilation is done, compilation is interrupted.

To facilitate dynamic switching, the engine computes an exponentially decaying average of the tuple throughput for every backend being considered. To retain up-to-date statistics for the different backends, 5% of morsels are run using the interpreter and JIT backend, respectively. The remaining 90% of morsels always choose the backend that currently has the highest measured tuple throughput.

The bookkeeping overhead of the hybrid backend is minimal. Computing the morsel throughput requires measuring the elapsed time for every morsel, which roughly corresponds to taking two timestamps every millisecond. The decision is made in a completely thread-local fashion. In principle, different threads can use different backends at the same time.

If vectorized interpretation is the fastest paradigm for a query, the hybrid backend introduces the overhead of background compilation. For short-running queries this overhead can be significant. It can be mitigated by either delaying compilation, or reducing the resources allocated to compilation. In that case however, the compiled code will be ready later, which can again negatively impact performance. If operator-fusing compilation is the fastest paradigm for a query, the hybrid backend consistently outperforms the pure JIT backend. It can hide compilation latency by performing vectorized interpretation on the idle CPU cores.

VI. Benefits of Incremental Fusion

This section compares Incremental Fusion to other query execution paradigms.

Comparison to Vectorization: An Incremental Fusion engine can generate a complete vectorized interpreter. Rather than implementing the interpreter by hand, an Incremental Fusion engine assembles the pipelines representing the primitives and then compiles them.

Comparison to Compilation: Incremental Fusion is a code-generating execution paradigm. The compilation stack uses the same concepts found in traditional code-generating engines [3]. The engine uses a custom suboperator IR below relational algebra to generate code. Traditional code-generating engines are moving in a similar direction, with multi-level IRs becoming more popular. Multi-level IRs reduce system complexity and allow for novel optimizations [31]–[34], [39]–[42].

Compared to traditional low-latency code-generating engines, we believe that it is easier to build, maintain, and operate an Incremental Fusion query engine. Incremental Fusion also eliminates some of the most complex parts of modern code-generating query engines. To hide compilation latencies of e.g. LLVM, these engines usually have sophisticated compilation stacks [10]–[13]. These implement a fast path to generate machine code from relational algebra. Since LLVM compilation can take hundreds of milliseconds, this is the only way to compete with vectorized engines for low-latency analytics.

Incremental Fusion makes this fast path superfluous. For a new query, the vectorized interpreter is used while operator-fusing code generation happens in the background. This makes it possible to build a low-latency compiling engine with a simple compilation stack. *Our engine InkFuse hides compilation latencies so well that we are able to achieve competitive performance while always generating C.*

For traditional code-generating engines, runtime adaptivity is difficult [43], [44]. In vectorized engines it is simpler to reorder primitives, or choose optimized primitives depending on the properties of the data [19], [45]. We believe that Incremental Fusion enables easier runtime adaptivity in compiling systems. We plan to study runtime adaptivity in an Incremental Fusion engine in future work.

Comparison to ROF: Incremental Fusion is an extension of Relaxed Operator Fusion (ROF) [8]. Our engine InkFuse also implements an ROF backend. In the same way as ROF, Incremental Fusion uses tuple buffers to divide a pipeline into steps. However, an engine using ROF still always needs to JIT compile code. Incremental Fusion extends ROF by matching steps in a pipeline to pre-compiled primitives.

Building an ROF engine for low-latency query execution necessitates a complex compilation stack similar to the one of Umbra [10], [11], [13]. Incremental Fusion's custom sub-operator IR makes low-latency query execution easy. The pre-compiled vectorized interpreter eliminates the dependency between low-latency compilation and low-latency execution.

Research at the Intersection of Compilation and Interpretation: We now compare Incremental Fusion to recent work at the intersection of compiling and interpreting query engines.

LB2 uses Futamura projections to create a query compiler using the abstractions of a query interpreter, promising easier system development [46]–[48]. However, building a compiling system using Futamura projections does not yield a high-performance interpreter. Incremental Fusion goes the opposite route, using its compiler to generate a vectorized interpreter.

VOILA explores the design space of query engines [49], [50]. VOILA synthesizes different flavors of query engines by expressing the control flow of relational algebra operators in a high-level language. It can synthesize interpreting and compiling engines, as well hybrid versions similar to ROF. Using a dynamic execution mode, VOILA can switch between operator-fusing code generation and vectorization [9]. VOILA is similar to Incremental Fusion in its goal to bridge the gap between vectorization and compilation. However, VOILA is primarily focused on finding the most efficient execution strategy for a query. While VOILA can generate vectorized primitives, it does so the first time it encounters them in a query. VOILA uses a compilation cache to hide this latency for repetitive query patterns. Incremental Fusion meanwhile focuses on making low-latency query processing in a compiling engine easy. The IR is specifically designed to be able to generate a complete vectorized interpreter ahead of time.

To the best of our knowledge, Incremental Fusion is the first execution paradigm that is able to combine the instant availability of a vectorized interpreter with the performance of operator-fusing code generation.

VII. Evaluation

In this section, we experimentally evaluate our open-source Incremental Fusion engine InkFuse [20]. We first compare the execution backends of InkFuse in isolation. We then measure InkFuse against the state-of-the-art vectorized and compiling systems DuckDB and Umbra. InkFuse is able to achieve competitive query performance for TPC-H queries across data sizes and queries. Our experiments measure data sizes between 100 MB and 100 GB, corresponding to TPC-H scale factors 0.1 to 100.

Hardware: All experiments are run on an Intel Core i9-10900 CPU. The CPU has ten cores (20 threads) with a 2.8 GHz clock rate and up to 5.2 GHz boost frequency. The system has 20 MB shared, last-level cache and is equipped with 128 GB of main memory. It runs the Linux kernel version 6.1.0.

InkFuse: InkFuse is our open-source prototype Incremental Fusion runtime. Currently, InkFuse has no SQL frontend, requiring the manual creation of physical execution plans. InkFuse natively supports multi-threaded execution. It implements morsel-driven parallelism [38]. We evaluate four execution backends of InkFuse:

Compiling Backend: Performs code generation where all pipelines are fully fused. It takes the physical query plan and generates efficient C code. The C code is then compiled to machine code using `clang-14`.

Vectorized Backend: Exclusively uses the pre-compiled primitives loaded at engine startup.

Hybrid Backend: Starts execution using the vectorized interpreter, and generates machine code in the background. Once the machine code becomes ready, it dynamically chooses the backend that provides the highest tuple throughput.

ROF Backend: Implements relaxed operator fusion. We break up a pipeline before every hash table probe and insert a dedicated prefetching step in the same way as the original ROF paper [8]. The prefetching stage allows issuing many independent loads and fetching data into CPU caches before performing the tuple-at-a-time lookup. In a similar way to vectorized engines, this improves performance when accessing hash tables that exceed the cache size.

Queries: We execute eight different TPC-H queries at different scale factors. Scale factor 1 corresponds to about 1 GB of data, and the data size scales linearly with the scale factor. We implement the same TPC-H queries as the original ROF paper [8]. The TPC-H queries are chosen to cover all TPC-H choke-points [51]. Q1 contains a low-cardinality aggregation. Q3 and Q4 are join queries with a more than 20x size difference between the build and probe sides. Q5 contains a large join tree with five joins. Q6 executes multiple selective filters. Q13 performs an outer join with many unmarked tuples. Q14 contains a join with a more moderate 4x size difference between the build and probe side. Q19 contains complex expression trees.

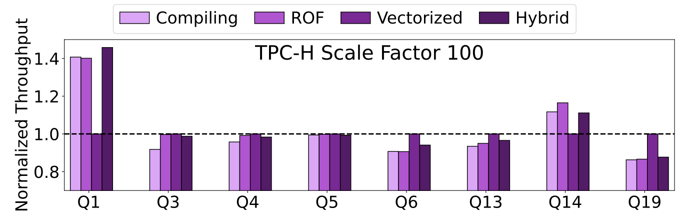

Fig. 9. Relative throughput of the different InkFuse execution backends at SF100. The throughput is normalized by the vectorized throughput.

TABLE I
CPU counters for TPC-H queries (SF100). Counters are normalized by the number of tuples processed in the query.

Query	Backend	Cycles	Instr.	IPC	LLC Miss	Branch Miss
Q1	Compiled	**7.61**	**19.40**	**2.55**	0.09	**0.01**
Q1	Vectorized	13.23	25.41	1.92	**0.08**	0.02
Q4	Compiled	10.21	**5.12**	0.50	0.24	**0.07**
Q4	Vectorized	**8.20**	5.50	**0.67**	**0.14**	0.08

A. Performance of InkFuse Backends

We first focus on the relative computational efficiency of the different InkFuse backends. For this, we measure the performance of the TPC-H queries at SF100. Absolute end-to-end performance numbers are presented in the next section.

Fig. 9 shows the relative throughput of the different execution backends compared to the vectorized backend. Table I shows low-level performance counters for Q1 and Q4. We can see that it is highly query-dependent whether vectorization or compilation offers the best performance.

Compilation outperforms vectorization for Q1 and Q14. The computational efficiency of the generated code leads to higher throughput. For Q1, the compiling backend is able to achieve 40% higher throughput than its vectorized counterpart. Table I shows that the fused code issues fewer instructions *and* is able to achieve higher IPC.

Vectorization meanwhile outperforms compilation for other queries. Q3, Q4, and Q13 build large hash tables. As vectorization can generate more independent memory loads, the probing becomes more efficient. Table I shows that this allows Q4 to hide LLC misses more effectively. Q6 and Q19 contain selective predicates that favor the vectorized backend. The generated primitives are auto-vectorized by the compiler and contain SIMD instructions, while the specialized code for the query does not.

We can nicely see that compared to traditional compilation, ROF always performs the same or better. For Q3, Q4, and Q13 it is able to achieve almost the same performance as the vectorized backend. This is because ROF can also generate independent loads by prefetching hash table buckets. For Q6 and Q19 the ROF prefetching of hash tables does not help, as vectorization performs better due to the selective filters.

For all queries except for Q6 and Q19, the hybrid backend manages to achieve similar performance as the fastest static backend. Once the compiled code is ready, the hybrid backend can dynamically choose the backend that provides the highest

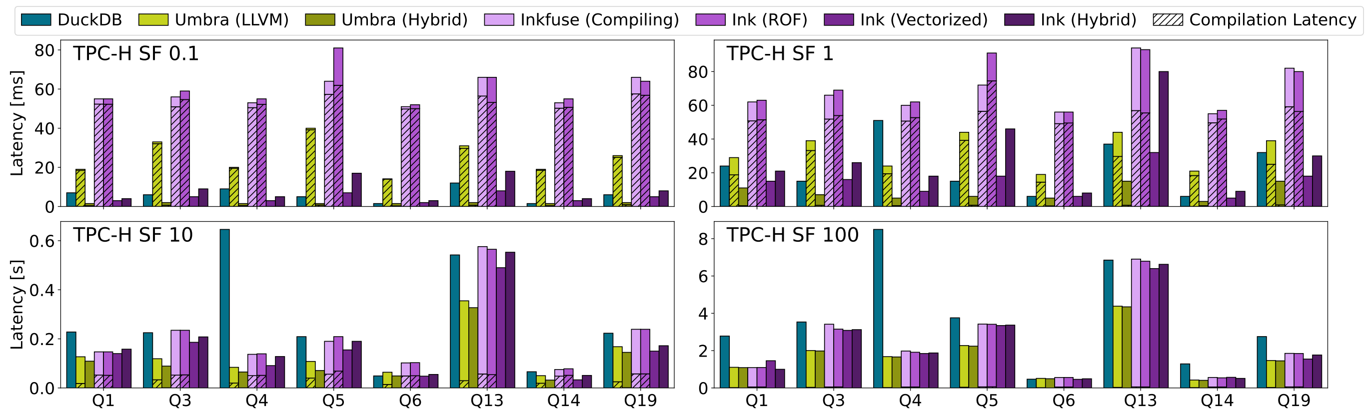

Fig. 10. Compared to DuckDB and Umbra, InkFuse provides competitive performance across queries and data sizes.

throughput. For Q6 and Q19, compilation overhead leads to the hybrid backend performing worse than the vectorized one.

B. Comparison to Other Systems

We compare InkFuse to the open-source vectorized database system DuckDB (version 0.9.1) [16], as well as the closed-source compiling engine Umbra developed at TU Munich [14]. For Umbra, we evaluate two execution backends. The LLVM backend generates LLVM IR that needs to be compiled before any tuples can be processed. This backend is similar to the compiling backend of InkFuse. Umbra's hybrid backend contains a fast path generating efficient x86 assembly while compiling LLVM IR in the background [11]. This backend allows Umbra to effectively hide compilation latencies.

The physical plans used by InkFuse are the same as the optimized plans of Umbra. We only deviate when Umbra uses advanced operators not supported by InkFuse such as index joins or group joins [52]. Results are shown in Fig. 10.

Small Data Sizes: For 100 MB of data (SF 0.1), the vectorized and hybrid backends of InkFuse are able to provide the same performance as the vectorized engine DuckDB. InkFuse is able to execute all queries in less than twenty milliseconds. The hybrid backend of InkFuse never uses the JIT compiled code. It always chooses the vectorized interpreter that was generated ahead of time.

This experiment pinpoints one problem of compiling query engines. The time spent waiting for compiled code to be ready is represented as the dashed area of each bar. With compilation, InkFuse needs more than 40 milliseconds to execute all queries and spends almost the entire duration generating code. The LLVM backend of Umbra is faster as it emits a more low-level representation than C, but still needs around 20 milliseconds. Meanwhile, the interpreter of vectorized engines is instantly available and outperforms traditional compilation by up to 10x.

The hybrid backend of Umbra is able to outperform both InkFuse and DuckDB. However, the impressive performance necessitates a complicated compilation stack [11], [13].

For 1 GB of data (SF 1), the performance is similar. However, we can see that the gap between the interpreted and compiling engines is becoming smaller. This is because code generation makes up less of the total execution time. The hybrid backend manages to consistently outperform the JIT backends. However, it does not achieve the same performance as the interpreted backend. Background compilation slows down queries. This is especially significant for Q13.

Large Data Sizes: With 10 GB of data (SF 10), the performance of the JIT compiled queries becomes similar to DuckDB and the vectorized backend of InkFuse. The time spent waiting for compiled code to become ready (represented by the dashed area of each bar) is amortized by the larger data size that needs to be processed. For Q1, Q4, Q14, and Q19, Umbra and the hybrid backend of InkFuse are able to significantly outperform DuckDB.

For 100 GB of data (SF 100), it is highly query dependent whether interpretation or compilation performs best. The dashed area of the bars is hardly visible anymore, showing that the compilation latencies are amortized by the time spent processing tuples. Hiding compilation latencies is not important anymore. Instead, it matters most whether compilation or vectorization benefits the query.

Overall, InkFuse provides competitive performance to the fastest analytical query engines. By dynamically switching between vectorization and compilation, InkFuse can choose the best execution paradigm across queries and data sizes.

VIII. Conclusion

This paper presents Incremental Fusion, a query execution paradigm unifying compiled and vectorized query execution. Incremental Fusion uses a novel suboperator IR to represent a query plan. This IR allows the engine to factor an arbitrary SQL query into a set of finite building blocks. The engine can generate a complete vectorized interpreter for these finite blocks ahead of time. This enables low-latency query processing without necessitating the complex compilation stacks found in modern compiling engines. Our open-source prototype engine InkFuse is able to compete with the state-of-the art vectorized and compiled engines DuckDB and Umbra. It provides competitive performance both for very short-running queries, as well as compute-intensive long-running ones.

REFERENCES

[1] P. A. Boncz, M. Zukowski, and N. Nes, "Monetdb/x100: hyper-pipelining query execution," in *CIDR*, vol. 5, 2005, pp. 225–237.

[2] T. Kersten, V. Leis, A. Kemper, T. Neumann, A. Pavlo, and P. Boncz, "Everything you always wanted to know about compiled and vectorized queries but were afraid to ask," *PVLDB*, vol. 11, no. 13, pp. 2209–2222, 2018.

[3] T. Neumann, "Efficiently compiling efficient query plans for modern hardware," *PVLDB*, vol. 4, no. 9, pp. 539–550, 2011.

[4] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, T. Vassilakis, H. Ahmadi, D. Delorey, S. Min *et al.*, "Dremel: A decade of interactive sql analysis at web scale," *PVLDB*, vol. 13, no. 12, pp. 3461–3472, 2020.

[5] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang *et al.*, "The snowflake elastic data warehouse," in *SIGMOD*, 2016, pp. 215–226.

[6] N. Armenatzoglou, S. Basu, N. Bhanoori, M. Cai, N. Chainani, K. Chinta, V. Govindaraju, T. J. Green, M. Gupta, S. Hillig *et al.*, "Amazon redshift re-invented," 2022.

[7] SAP, "Hex: Sap's new hana execution engine," 2023. [Online]. Available: https://www.cidrdb.org/cidr2023/slides/sponsor-talk-sap-slides.pdf

[8] P. Menon, T. C. Mowry, and A. Pavlo, "Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last," *PVLDB*, vol. 11, no. 1, pp. 1–13, 2017.

[9] T. Gubner and P. Boncz, "Excalibur: A virtual machine for adaptive fine-grained jit-compiled query execution based on voila," *PVLDB*, vol. 16, no. 4, 2022.

[10] A. Kohn, V. Leis, and T. Neumann, "Adaptive execution of compiled queries," in *ICDE*. IEEE, 2018, pp. 197–208.

[11] T. Kersten, V. Leis, and T. Neumann, "Tidy tuples and flying start: fast compilation and fast execution of relational queries in umbra," *The VLDB Journal*, vol. 30, no. 5, pp. 883–905, 2021.

[12] H. Funke, J. Mühlig, and J. Teubner, "Low-latency query compilation," *The VLDB Journal*, pp. 1–14, 2022.

[13] F. Gruber, M. Bandle, A. Engelke, T. Neumann, and J. Giceva, "Bringing compiling databases to risc architectures," *Proceedings of the VLDB Endowment*, vol. 16, no. 6, pp. 1222–1234, 2023.

[14] T. Neumann and M. J. Freitag, "Umbra: A disk-based system with in-memory performance," in *CIDR*, 2020.

[15] M. Pasumansky and B. Wagner, "Assembling a query engine from spare parts," *International Workshop on Composable Data Management Systems (CDMS)*, 2022.

[16] M. Raasveldt and H. Mühleisen, "Duckdb: an embeddable analytical database," in *SIGMOD*, 2019, pp. 1981–1984.

[17] ClickHouse, "Clickhouse," 2022. [Online]. Available: https://github.com/ClickHouse/ClickHouse

[18] J. M. Patel, H. Deshmukh, J. Zhu, N. Potti, Z. Zhang, M. Spehlmann, H. Memisoglu, and S. Saurabh, "Quickstep: A data platform based on the scaling-up approach," *PVLDB*, vol. 11, no. 6, pp. 663–676, 2018.

[19] A. Behm, S. Palkar, U. Agarwal, T. Armstrong, D. Cashman, A. Dave, T. Greenstein, S. Hovsepian, R. Johnson, A. Sai Krishnan *et al.*, "Photon: A fast query engine for lakehouse systems," in *SIGMOD*, 2022, pp. 2326–2339.

[20] B. Wagner, "Inkfuse," 2023. [Online]. Available: https://github.com/wagjamin/inkfuse

[21] G. Graefe, "Volcano - an extensible and parallel query evaluation system," *IEEE Transactions on Knowledge and Data Engineering*, vol. 6, no. 1, pp. 120–135, 1994.

[22] PostgreSQL, "Postgresql executor," 2022. [Online]. Available: https://www.postgresql.org/docs/14/executor.html

[23] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss *et al.*, "Cockroachdb: The resilient geo-distributed sql database," in *SIGMOD*, 2020, pp. 1493–1509.

[24] S. Idreos, F. Groffen, N. Nes, S. Manegold, S. Mullender, and M. Kersten, "Monetdb: Two decades of research in column-oriented database," *IEEE Data Engineering Bulletin*, vol. 35, no. 1, pp. 40–45, 2012.

[25] P. A. Boncz *et al.*, "Monet: A next-generation dbms kernel for query-intensive applications," 2002.

[26] B. Chattopadhyay, P. Dutta, W. Liu, O. Tinn, A. Mccormick, A. M. P. Harvey, H. Gonzalez, D. Lomax, S. M. R. Ebenstein, N. Mikhaylin *et al.*, "Procella: Unifying serving and analytical data at youtube," *PVLDB*, vol. 12, no. 12, 2019.

[27] A. Kemper and T. Neumann, "Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots," in *ICDE*. IEEE, 2011, pp. 195–206.

[28] A. Pavlo, "Noisepage," 2022. [Online]. Available: https://github.com/cmu-db/noisepage

[29] T. Kersten and T. Neumann, "On another level: how to debug compiling query engines," in *Proceedings of the workshop on Testing Database Systems*, 2020, pp. 1–6.

[30] A. Beischl, T. Kersten, M. Bandle, J. Giceva, and T. Neumann, "Profiling dataflow systems on multiple abstraction levels," in *EuroSys*, 2021, pp. 474–489.

[31] D. Koutsoukos, I. Müller, R. Marroquín, A. Klimovic, and G. Alonso, "Modularis: modular relational analytics over heterogeneous distributed platforms," *PVLDB*, vol. 14, no. 13, pp. 3308–3321, 2021.

[32] M. Bandle and J. Giceva, "Database technology for the masses: sub-operators as first-class entities," *PVLDB*, vol. 14, no. 11, pp. 2483–2490, 2021.

[33] H. Pirk, O. Moll, M. Zaharia, and S. Madden, "Voodoo-a vector algebra for portable database performance on modern hardware," *PVLDB*, vol. 9, no. 14, pp. 1707–1718, 2016.

[34] S. Palkar, J. J. Thomas, A. Shanbhag, D. Narayanan, H. Pirk, M. Schwarzkopf, S. P. Amarasinghe, M. A. Zaharia, and S. InfoLab, "Weld : A common runtime for high performance data analytics," 2016.

[35] P. A. Boncz and M. L. Kersten, "Mil primitives for querying a fragmented world," *The VLDB Journal*, vol. 8, no. 2, pp. 101–119, 1999.

[36] A. Ngom, P. Menon, M. Butrovich, L. Ma, W. S. Lim, T. C. Mowry, and A. Pavlo, "Filter representation in vectorized query execution," in *DaMoN*, 2021, pp. 1–7.

[37] L. Kuiper and H. Mühleisen, "These rows are made for sorting and that's just what we'll do," in *ICDE*. IEEE, 2023.

[38] V. Leis, P. Boncz, A. Kemper, and T. Neumann, "Morsel-driven parallelism: A numa-aware query evaluation framework for the many-core age," in *SIGMOD*, 2014, pp. 743–754.

[39] T. Neumann, "Evolution of a compiling query engine," *PVLDB*, vol. 14, no. 12, pp. 3207–3210, 2021.

[40] M. Jungmair, A. Kohn, and J. Giceva, "Designing an open framework for query optimization and compilation," *PVLDB*, vol. 15, no. 11, pp. 2389–2401, 2022.

[41] A. Shaikhha, Y. Klonatos, L. Parreaux, L. Brown, M. Dashti, and C. Koch, "How to architect a query compiler," in *SIGMOD*, 2016, pp. 1907–1922.

[42] A. Shaikhha, Y. Klonatos, and C. Koch, "Building efficient query engines in a high-level language," *ACM Transactions on Database Systems (TODS)*, vol. 43, no. 1, pp. 1–45, 2018.

[43] P. Menon, A. Ngom, L. Ma, T. C. Mowry, and A. Pavlo, "Permutable compiled queries: dynamically adapting compiled queries without re-compiling," *PVLDB*, vol. 14, no. 2, 2020.

[44] T. Schmidt, P. Fent, and T. Neumann, "Efficiently compiling dynamic code for adaptive query processing," *Workshop on Accelerating Analytics and Data Management (ADMS)*, 2022.

[45] B. Răducanu, P. Boncz, and M. Zukowski, "Micro adaptivity in vector-wise," in *SIGMOD*, 2013, pp. 1231–1242.

[46] N. D. Jones, P. Sestoft, and H. Søndergaard, "Mix: a self-applicable partial evaluator for experiments in compiler generation," *Lisp and Symbolic computation*, vol. 2, no. 1, pp. 9–50, 1989.

[47] Y. Futamura, "Partial evaluation of computation process-an approach to a compiler-compiler," *Systems, computers, controls*, vol. 25, pp. 45–50, 1971.

[48] R. Y. Tahboub, G. M. Essertel, and T. Rompf, "How to architect a query compiler, revisited," in *SIGMOD*, 2018, pp. 307–322.

[49] T. Gubner and P. Boncz, "Charting the design space of query execution using voila," *PVLDB*, vol. 14, no. 6, pp. 1067–1079, 2021.

[50] T. Gubner and P. A. Boncz, "Highlighting the performance diversity of analytical queries using voila." in *ADMS@VLDB*, 2021, pp. 47–54.

[51] P. Boncz, T. Neumann, and O. Erling, "Tpc-h analyzed: Hidden messages and lessons learned from an influential benchmark," in *Technology Conference on Performance Evaluation and Benchmarking*. Springer, 2013, pp. 61–76.

[52] P. Fent and T. Neumann, "A practical approach to groupjoin and nested aggregates," *PVLDB*, vol. 14, no. 11, pp. 2383–2396, 2021.