JFP **28**, e10, 45 pages, 2018. © Cambridge University Press 2018
doi:10.1017/S0956796818000102

Push versus pull-based loop fusion in query engines

AMIR SHAIKHHA, MOHAMMAD DASHTI
and CHRISTOPH KOCH
École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland
(*e-mails:* amir.shaikhha@epfl.ch, mohammad.dashti@epfl.ch, christoph.koch@epfl.ch)

Abstract

Database query engines use pull-based or push-based approaches to avoid the materialization of data across query operators. In this paper, we study these two types of query engines in depth and present the limitations and advantages of each engine. Similarly, the programming languages community has developed loop fusion techniques to remove intermediate collections in the context of collection programming. We draw parallels between databases (DB) and programming language (PL) research by demonstrating the connection between pipelined query engines and loop fusion techniques. Based on this connection, we propose a new type of pull-based engine, inspired by a loop fusion technique, which combines the benefits of both approaches. Then, we experimentally evaluate the various engines, in the context of query compilation, for the first time in a fair environment, eliminating the biasing impact of ancillary optimizations that have traditionally only been used with one of the approaches. We show that for realistic analytical workloads, there is no considerable advantage for either form of pipelined query engine, as opposed to what recent research suggests. Also, by using micro-benchmarks, which demonstrate certain edge cases on which one approach or the other performs better, we show that our proposed engine dominates the existing engines by combining the benefits of both.

1 Introduction

Database query engines successfully leverage the compositionality of relational algebra-style query plan languages. Query plans are compositions of operators that, at least conceptually, can be executed in sequence, one after the other. However, actually evaluating queries in this way leads to grossly suboptimal performance. Computing ("materialising") the result of a first operator before passing it to a second operator can be very expensive, particularly if the intermediate result is large and needs to be pushed down the memory hierarchy. The same observation has been made by the programming languages and compilers community and has led to work on loop fusion and deforestation (the elimination of data structure construction and destruction for intermediate results).

Already relatively early on in the history of relational database systems, a solution to this problem has been proposed in the form of the Volcano Iterator model (Graefe, 1994). In this model, tuples are *pulled* up through a chain of operators that are linked by iterators that advance in lock-step. Intermediate results between operators are not accumulated, but tuples are produced on demand, by request by conceptually "later" operators.

More recently, an operator chaining model has been proposed that shares the advantage of avoiding materialisation of intermediate results but which reverses the control flow; tuples are *pushed* forward from the source relations to the operator producing the final result. Recent papers (Neumann, 2011; Klonatos *et al.*, 2014a) seem to suggest that this push-model consistently leads to better query processing performance than the pull model, even though no direct, fair comparisons are provided.

One of the main contributions of this paper is to debunk this myth. As we show, if compared fairly, push- and pull-based engines have very similar performance, with individual strengths and weaknesses, and neither is a clear winner. Push engines have in essence only been considered in the context of query *compilation*, conflating the potential advantages of the push paradigm with those of code inlining. To compare them fairly, one has to decouple these aspects.

In this paper, we present an in-depth study of the trade-offs of the push versus the pull paradigm. Choosing among push and pull – or any reasonable alternative – is a fundamental decision that drives many decisions throughout the architecture of a query engine. More specifically, the interface exposed for implementing different query operators is dependent on the type of the query engine (Graefe, 1993). Furthermore, the query processing engine needs to interact with the storage manager to benefit from different *access methods* (such as hash-based and B+ tree indexes) for efficiently accessing data in the underlying storage (Hellerstein *et al.*, 2007). Hence, different design choices of query engines result in different design choices for the storage manager component as well. Thus, one must understand the relevant properties and trade-offs deeply, and should not bet on one's ability to overcome the disadvantages of a choice by a hack later.

Furthermore, we illustrate how the same challenge and trade-off has been met and addressed by the PL community, and show a number of results that can be carried over from the lessons learned there. Specifically, we study how the PL community's answer to the problem, *stream fusion* (Coutts *et al.*, 2007), can be adapted to the query processing scenario, and show how it combines the advantages of the pull and push approaches. Furthermore, we demonstrate how we can use ideas from the push approach to solve well-known limitations of stream fusion. As a result, we construct a query engine that combines the benefits of both push and pull approaches. In essence, this engine is a pull-based engine on a coarse level of granularity (i.e., on the level of collections), however, on a finer level of granularity (i.e., on the level of tuples), it pushes the individual data tuples.

In summary, this paper makes the following contributions:

- We discuss pipelined query engines in Section 2. After presenting loop fusion for collection programming in Section 3, we show the connection between these two concepts in Section 4. Furthermore, we demonstrate the limitations associated with each approach.
- Based on this connection with loop fusion, we propose a new pipelined query engine in Section 5 inspired by the stream fusion (Coutts *et al.*, 2007) technique

developed for collection programming in the PL community. Also, we discuss implementation concerns and compiler optimizations required for the proposed pipelined query engine in Section 6.

- We experimentally evaluate the various query engine architectures in Section 7. Using micro-benchmarks, we discuss the weaknesses of the existing engines and how the proposed engine circumvents these weaknesses by combining the benefits of both worlds. Then, we demonstrate using TPC-H queries that good implementations of these three query engines do not show a considerable advantage for either form of query engine.

Throughout this paper, we are using the Scala programming language for all code snippets, interfaces, and examples. None of the concepts and ideas require specifically this language – other impure functional object-oriented programming languages (or object-oriented with functional features) such as OCaml, F#, C++11, C#, or Java 8 could be used instead.

2 Pipelined query engines

Database management systems accept a declarative query (e.g., written in SQL). Such a query is passed to a query optimizer to find a fast physical query plan, which then is either interpreted by the query engine or compiled to low-level code (e.g., C code).

A physical query plan is a data flow graph of query operators that perform calculations and data transformations. Each query operator can be connected to one or more input operators (which we refer to as the *source* operators) and one output operator (which we refer to as the *destination* operator).

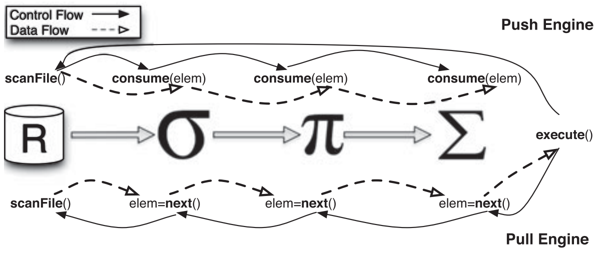
A sequence of query operators can be *pipelined*, which means that the output of one operator is *streamed* into the next operator. Pipelining a query operator removes the need for *materializing* the intermediate data and reading it back again, which can bring a significant performance gain.

There are two approaches for pipelining. The first approach is demand-driven pipelining in which an operator repeatedly *pulls* the next data tuple from its source operator. The second approach is data-driven pipelining in which an operator *pushes* each data tuple to its destination operator. Next, we give more details on the pull-based and push-based query engines.

2.1 Pull engine – a.k.a. the iterator pattern

The iterator model is the most widely used pipelining technique in query engines. This model was initially proposed in XRM (Lorie, 1974). However, the popularity of this model is due to its adoption in the Volcano system (Graefe, 1994), in which this model was enriched with facilities for parallelization.

In a nutshell, in the iterator model, each operator pipelines the data by requesting the next element from its source operator. This way, instead of waiting until the entire intermediate relation is produced, the data is *lazily* generated in each operator. This is achieved by invoking the `next` method of the source operator by the destination



```
SELECT SUM(R.B) FROM R
WHERE R.A < 10
```

```
R.filter(r => r.A < 10)
 .map(r => r.B)
 .fold(0)((s, r) => s + r)
```

Fig. 1. Data flow and control flow for push- and pull-based query engine for the provided SQL query.

operator. The design of pull-based engines directly corresponds to the iterator design pattern in object-oriented programming (Vlissides *et al.*, 1995).

Figure 1 shows an example query and the control flow of query processing for this query. Each query operator performs the role of a destination operator and *requests* data from its source operator (the predecessor operator along the flow direction of data). In a pull engine, this is achieved by invoking the `next` function of the source operator, and is shown as control flow edges. In addition, each operator serves as source operator and *generates* result data for its destination operator (the successor operator along the flow direction of data). The generated data is the return value of the `next` function, and is represented by the data flow edges in Figure 1. Note the opposing directions of control-flow and data-flow edges for the pull engine in Figure 1.

From a different point of view, each operator can be considered as a `while` loop in which the `next` function of the source operator is invoked per iteration. The loop is terminated when the `next` function returns a special value (e.g., a `null` value). In other words, whenever this special value is observed, a `break` statement is executed to terminate the loop execution.

There are two main issues with a pull-based query engine. First, the `next` function invocations are implemented as virtual functions – operators with different implementations of `next` have to be chained together. There are many invocations of these functions; each invocation requires looking up a virtual table, which leads to suboptimal instruction locality. Query compilation solves this performance issue by inlining these virtual function calls, which is explained in Section 2.3.

Second, in practice, selection operators are problematic. When the `next` method of a selection operator is invoked, the destination operator has to wait until the selection operator returns the next data tuple satisfying its predicate. This makes the control flow of the query engine more complicated by introducing more loops and branches, which is demonstrated in Figure 2(c). Push engines solve the problem of complicated control flow graphs (CFGs).

2.2 Push engine – a.k.a. the visitor pattern

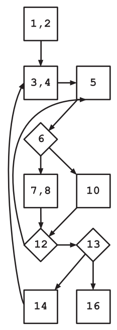
Push-based engines are widely used in streaming systems (Hirzel *et al.*, 2014). The Volcano system uses data-driven pipelining (which is a push-based approach) for

```
 1 var sum = 0.0
 2 var index = 0
 3 while(true) {
 4   var rec = null
 5   do {
 6     if(index < R.length) {
 7       rec = R(index)
 8       index += 1
 9     } else {
10       rec = null
11     }
12   } while(rec != null && !(rec.A < 10))
13   if(rec == null) break
14   else sum += rec.B
15 }
16 return sum
```

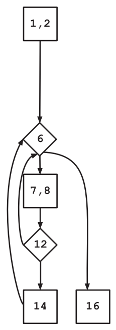
(a)

```
var sum = 0.0
var index = 0


while(index < R.length) {
  val rec = R(index)
  index += 1



  if(rec.A < 10)

    sum += rec.B
}
return sum
```

(b)



(c)



(d)

Fig. 2. Specialized version of the example query in pull and push engines and the corresponding control-flow graphs (CFG). (a) Inlined query in pull engine. (b) Inlined query in push engine. (c) The CFG of the inlined query in pull engine. (d) The CFG of the inlined query in push engine.

implementing inter-operator parallelism in query engines. In the context of query compilation, stream processing engines such as StreamBase (Tibbetts *et al.*, 2011) and Spade (Gedik *et al.*, 2008), as well as HyPer (Neumann, 2011) and LegoBase (Klonatos *et al.*, 2014a; Shaikhha *et al.*, 2018) use a push-based query engine approach.

In push-based query engines, the control flow is reversed compared to that of pull-based engines. More concretely, instead of destination operators requesting data from their source operators, data is pushed from the source operators toward the destination operators. This is achieved by the source operator passing the data as an argument to the `consume` method of the destination operator. This results in *eagerly* transferring the data tuple-by-tuple instead of requesting it *lazily* as in pull-engines.

A push engine can be implemented using the *Visitor* design pattern (Vlissides *et al.*, 1995) from object-oriented programming. This design pattern allows separating an algorithm from a particular type of data. In the case of query engines, the visitor pattern allows us to separate the query operators (data processing algorithms) from a relation of elements. To do so, each operator should be defined as a visitor class, in which the `consume` method, which is responsible for pushing the elements down the pipeline, has the functionality of the `visit` method, whereas the `produce` method, which is responsible for initializing the chain of operators, has the functionality of the `accept` method of the visitor pattern.

Figure 1 shows the query processing workflow for the given example query. Query processing in each operator consists of two main phases. In the first phase, operators prepare themselves for producing their data. This is performed only once in the initialization. In the second phase, they consume the data provided by the source operator and produce data for the destination operator. This is the main processing phase, which consists of invoking the `consume` method of the destination operator and passing the produced data through it. This results in the same direction for both control-flow and data-flow edges, as shown in Figure 1.

Push engines solve the problem pull engines have with selection operators. A selection operator ignores the produced data if it does not satisfy the given predicate by not passing the data to the destination operator. This is in contrast with pull engines in which the destination operator should have waited for the selection operator to serve the request.

However, push engines experience difficulties with *limit* and *merge join* operators. For limit operators, push engines do not allow terminating the iteration by nature. This is because, in push engines, the operators cannot control when the data should no longer be produced by their source operator. This lack of control over when to stop producing more elements causes the production of elements that will never be used.

The merge join operator suffers from a similar problem. There is no way for the merge join operator to guide which one of its two source operators (which are both sorted and there is a 1-to-n relationship between them) should produce the next data item. Hence, it is not possible to pipeline the data coming fro\m both source operators in a merge join. As a result, at least for one of the source operators, the pipeline needs to be broken. Hence, the incoming data coming from one of the source operators can be pipelined (assuming it is correctly sorted, of course), but the input data coming from the other source operator must be materialized.

The mentioned limitation is not specific to operators such as merge joins. A similar situation can arise in the case of more sophisticated analytical tasks where one has to use collection programming APIs (such as Spark RDDs (Zaharia *et al.*, 2012)). In collection programming many different methods (for example, element-wise operations on two numeric vectors) are implemented using the `zip` method. The situation is similar for operations which are variants of the merge join operator such as Leapfrog Triejoin (Veldhuizen, 2014). These methods require parallel traversal on two (or more) collections similar to the merge join operator, which cannot be easily pipelined in push-based engines.

Note that these limitations can be resolved by providing special cases for these two operators in the push engine. In the case of limit operator, one can avoid producing unnecessary elements by manually fusing the limit operator with its source operator, which is an ordering operator in most cases. This is because in most cases limit-queries have an order-by clause.[1] For the merge join operator, one can implement a variant of this operator which uses different threads for its source operators and uses synchronization constructs in order to control the production of data by its two inputs, which can be costly. However, such engines can be considered as hybrid engines, and in this paper, by push engine, we mean a *purely standard* push engine without such augmentations.

2.3 Compiled engines

In general, the runtime cost of a given query is dependent on two factors. The first factor is the time it takes to transfer the data across storage and computing components. The second factor is the time taken for performing the actual computation (i.e., running the instructions of the query). In disk-based DBMSes, the dominating cost is usually the data transfer from/to the secondary storage. Hence, as long as the pipelining algorithm does not break the pipeline, there is no difference between pull engines and push engines. As a result, the practical problem with selections in pull engines (c.f. Section 2.1) is obscured by data transfer costs.

With the advent of in-memory DBMSes, the code layout of the instructions becomes an ever more important factor. More specifically, the virtual function calls (or function calls via function to pointers) appearing in the iterator model start becoming a bottleneck in the performance. One solution is block-oriented processing of elements instead of processing elements one-by-one, which hides the cost of virtual calls behind the cost of processing a large number of elements (Padmanabhan *et al.*, 2001; Zukowski *et al.*, 2005). The alternative approach, which is the main focus of this paper, is query compilation. This approach uses code generation and compilation techniques in order to inline virtual functions and further specialize the code to improve cache locality (Grust *et al.*, 2009; Ahmad & Koch, 2009; Koch, 2010; Krikellas *et al.*, 2010; Neumann, 2011; Nagel *et al.*, 2014; Koch, 2014; Koch *et al.*, 2014; Klonatos *et al.*, 2014a; Viglas *et al.*, 2014; Armbrust *et al.*, 2015; Crotty *et al.*, 2015; Karpathiotakis *et al.*, 2015; Shaikhha *et al.*, 2016; Karpathiotakis *et al.*, 2016). As a result of that, the code pattern used in each pipelining algorithm really matters. Hence, it is important to investigate the performance of each pipelining algorithm for different workloads.

Figure 2(a) shows the inlined pull-engine code for the example SQL query given in Figure 1. Note that for the selection operator, we need an additional `while` loop. This additional loop creates more branches in the generated code, which makes the CFG more complicated. Figure 2(c) demonstrates the CFG of the inlined pull-engine code. Each rectangle in this figure corresponds to a block of statements, whereas diamonds represent conditionals. The edges between these nodes represent the execution flow. The backward edges represent the jumps inside a loop. This complicated CFG

[1] Even without manually fusing the ordering and limit operators, the cost of sorting dominates the cost of final scan. This reduces the impact of pipelining the limit operator in realistic workloads, as we observe in Section 7.2.

makes the code harder to understand and optimize for the optimizing compiler. As a result, during the runtime execution, performance degrades.

Figure 2(b) shows the specialized query for a push engine of the previous example SQL query. The selection operator here is summarized in a single `if` statement. As a result, the CFG for the inlined push-engine code is simpler than the one for the pull engine, as shown in Figure 2(d). This simpler CFG results in fewer branching machine instructions generated by the underlying optimizing compiler, leading to better run time performance.

Up to now, there is no separation of the concept of pipelining from the associated specializations. For example, HyPer (Neumann, 2011) is in essence a push engine that uses compiler optimizations by default, without identifying the individual contributions to performance by these two factors. As another example, LegoBase (Klonatos *et al.*, 2014a) assumes that a push engine is followed by operator inlining, whereas the pull engine does not use operator inlining (Klonatos *et al.*, 2014b). On the other hand, there is no comparison between an inlined pull engine – we suspect Hekaton (Diaconu *et al.*, 2013) to be of that class – with a push-based inlined engine in the same environment. Hence, there is no comparison between pull and push engines which is under completely fair experimental conditions, sharing environment and code base to the maximum degree possible. In Section 7, we attempt such a fair comparison.

Furthermore, naïvely compiling the pull engine does not lead to good performance. This is because a naïve implementation of the iterator model does not take into account the number of `next` function calls. This can lead to in-efficient code, due to the code explosion resulting from inlining too many `next` calls. For example, the naïve implementation of the selection operator invokes the `next` method of its source operator twice, as it is demonstrated below:

```
1 class SelectOp[R] (p: R => Boolean) {
2  def next(): R = {
3   var elem: R = source.next()
4   while(elem != null && !p(elem)) {
5    elem = source.next()
6   }
7   elem
8  }
9 }
```

The first invocation is happening before the loop for the initialization (line 3), and the second invocation is inside the loop (line 5). Inlining can cause an explosion of the code size, which can lead to worse instruction cache behavior. Hence, it is important to take into account these concerns while implementing query engines. For example, our implementation of the selection operator in a pull-based query engine invokes the `next` method of its source operator only once by changing the shape of the `while` loop (c.f. Figure 5(a)). Section 7.2 shows the impact of this *inline-friendly* implementation of pull engines.

3 Loop fusion in collection programming

Collection programming APIs are getting more and more popular. Ferry (Grust *et al.*, 2009; Grust *et al.*, 2010) and LINQ (Meijer *et al.*, 2006) use such an API to seemlessly integrate applications with database back-ends. Spark *RDDs* (Zaharia *et al.*, 2012) use the same operations as collection programming APIs. Also, functional collection programming abstractions exist in mainstream programming languages such as Scala, Haskell, and recently Java 8. The theoretical foundation of such APIs is based on Monad Calculus and Monoid Comprehensions (Wadler, 1990; Breazu-Tannen & Subrahmanyam, 1991; Breazu-Tannen *et al.*, 1992; Trinder, 1992; Grust & Scholl, 1999; Fegaras & Maier, 2000).

Similar to query engines, the declarative nature of collection programming comes with a price. Each collection operation performs a computation on a collection and produces a transformed collection. A chain of these invocations results in creating unnecessary intermediate collections.

Loop fusion or Deforestation (Wadler, 1988) removes the intermediate collections in collection programs. This is a nonlocal and brittle transformation which is difficult to apply to impure functional programs (i.e., in languages which include imperative features) and is thus absent from mainstream compilers for such languages. In order to provide a practical implementation, one can restrict the language to a pure functional domain-specific language (DSL) for which the fusion rules can be applied locally. Here, intermediate collections are removed using local transformations instead of global transformations. This approach is known as *short-cut* deforestation. It is more realistic to integrate this approach into real compilers; short-cut deforestation has been successfully implemented in the context of Haskell (Gill *et al.*, 1993; Svenningsson, 2002; Coutts *et al.*, 2007) and Scala-based DSLs (Jonnalagedda & Stucki, 2015; Shaikhha *et al.*, 2016).

Next, we present two approaches for short-cut deforestation, *fold fusion*, and *unfold fusion*, in the order they were discovered. They employ two kinds of "collection" micro-instructions each, to which a large number of collection operations can be mapped. This allows to implement fusion using very few rewrite rules.

3.1 *Fold fusion*

In this approach, every collection operation is implemented using two constructs: (1) the `build` method for *producing* a collection, and (2) the `foldr` method for *consuming* a collection. Some collection-transforming methods such as `map` use both of these constructs for consuming the given collection and producing a new collection. However, some methods such as `sum`, that produces an aggregated result from a collection, require only the `foldr` method for consuming the given collection.

We consider an imperative variant of this algorithm, in which the `foldr` method is substituted by `foreach`. The main difference is that the `foldr` method explicitly handles the state, whereas in the case of `foreach`, the state is handled internally and is not exposed to the interface.

Using Scala syntax, the signature of the `foreach` method on lists is as follows:

```
class List[T] {
  def foreach(f: T => Unit): Unit
}
```

The `foreach` method consumes a collection by iterating over the elements of that collection and applying the given function to each element. The `build` function is the corresponding producer for the `foreach` method. This function produces a collection for which the `foreach` method applies the higher order function `consumer` to the function `f`. The signature of the `build` function is as follows:

```
def build[T](consumer: (T => Unit) => Unit): List[T]
```

We illustrate the meanings of these two methods by an example. Consider the `map` method of a collection, which transforms a collection by applying a given function to each element. This method is expressed in the following way using the `build` and `foreach` functions:

```
class List[T] {
  def map[S](f: T => S): List[S] = build { k =>
    this.foreach(e => k(f(e)))
  }
}
```

The implementation of several other collection operators using these two methods is given in Figure 4(b).

After rewriting the collection operations using the `build` and `foreach` constructs, a pipeline of collection operators involves constructing intermediate collections. These intermediate collections can be removed using the following rewrite rule:

Fold-fusion rule:

```
build(f1).foreach(f2)   ⇝   f1(f2)
```

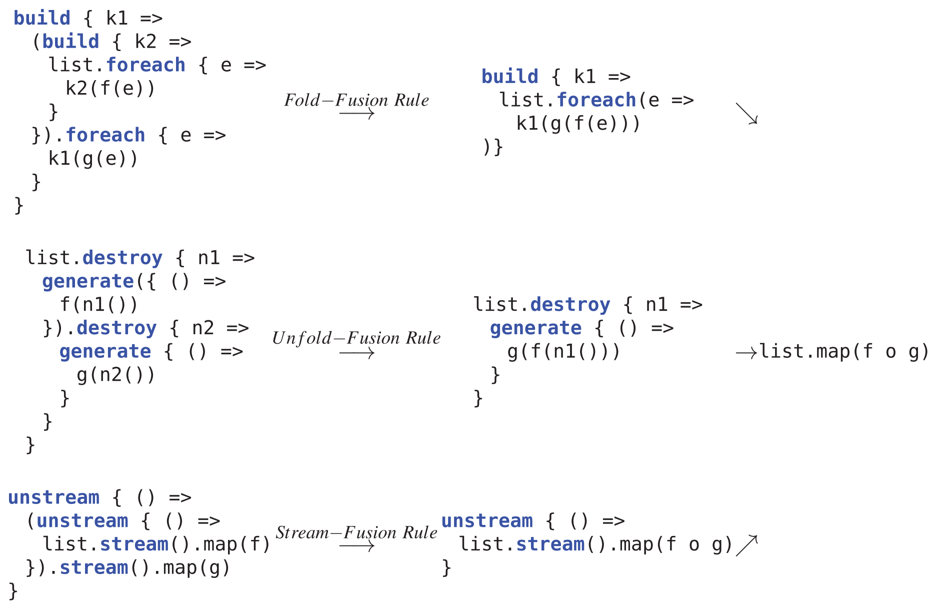
For example, there is a loop fusion rule for the `map` function, which fuses two consecutive `map` operations into one. More concretely, the expression `list.map(f).map(g)` is converted into `list.map(f o g)`. Figure 3 demonstrates how the fold-fusion technique can derive this conversion by expressing the `map` operator in terms of `foreach` and `build`, following by application of the fold-fusion rule.

One of the key advantages of this approach is that instead of writing fusion rewrite rules for every combination of collection operations, it is sufficient to only express these operations in terms of the `build` and `foreach` methods. This way, instead of writing $O(n^2)$ rewrite rules for n collection operations, it is sufficient to express these operations in terms of `build` and `foreach`, which is only $O(n)$ rewrite rules. Hence, this approach greatly simplifies the maintenance of the underlying compiler transformations (Shaikhha *et al.*, 2016).

This approach successfully deforests most collection operators very well. However, it is not successful in the case of `zip` and `take` operations. The `zip` method involves iterating over two collections, which cannot be expressed using the `foreach` construct that iterates only over one collection. Hence, this approach can deforest only one of the collections. For the other one, an intermediate collection must be created. Also, for the `take` method, there is no way to stop the iteration of the `foreach` method halfway to finish. Hence, the fold fusion technique does not perform well in these two cases. The next fusion technique solves the problem with these two methods.

3.2 Unfold fusion

This is considered a dual approach to fold fusion. Every collection operation is expressed in terms of the two constructs `unfold` and `destroy`. We use an imperative


```
build { k1 =>
  (build { k2 =>
    list.foreach { e =>
      k2(f(e))
    }
  }).foreach { e =>
    k1(g(e))
  }
}
```

```
build { k1 =>
  list.foreach(e =>
    k1(g(f(e)))
)}
```
```
list.destroy { n1 =>
  generate({ () =>
    f(n1())
  }).destroy { n2 =>
    generate { () =>
      g(n2())
    }
  }
}
```

```
list.destroy { n1 =>
  generate { () =>
    g(f(n1()))
  }
}
```

```
unstream { () =>
  (unstream { () =>
    list.stream().map(f)
  }).stream().map(g)
}
```

```
unstream { () =>
  list.stream().map(f o g)
}
```

Fig. 3. Different fusion techniques on `list.map(f).map(g)`.

version of unfold fusion here, which uses the `generate` function instead of `unfold`. The prototype of `generate` and `destroy` are as follows:

```
class List[T] {
  def destroy[S](f: (() => T) => S): S
}
def generate[T](gen: () => T): List[T]
```

The `destroy` method consumes the given list. Each element of this collection is accessible by invoking the `next` function available by the `destroy` method. The `generate` function generates a collection whose elements are specified by the input function passed to this method. In the case of `map` operator, the elements of the result collection are the images of the elements of the input collection under the function `f`.

The `map` method of collections is expressed in the following way using the `generate` and `destroy` methods:

```
class List[T] {
  def map[S](f: T => S): List[S] = this.destroy { n =>
    generate { () =>
      val elem = n()
      if(elem == null) null
      else f(elem)
    }
  }
}
```

The implementation of some other collection operators using these two methods is given in Figure 5(b).

In order to remove the intermediate collections, the chain of intermediate `generate` and `destroy` can be removed. This fact is shown in the following transformation rule:

Unfold-fusion rule:

```
generate(f1).destroy(f2)   ⇝   f2(f1)
```

Table 1. *Correspondence between query operators and collection operators*

Operator category	Query operator	Collection operator
Producer	Scan	`fromArray`
Transformer	Selection	`filter`
	Projection	`map`
	OrderBy	`sortBy`
	Limit	`take`
	Join*	`flatMap`*
	Merge Join[†]	`zip`[†]
Consumer	Agg[‡]	`fold`[‡]

*Nested loop join can be expressed using two nested flatMaps, but there is no equivalent for hash-based joins. Also, flatMaps can express nested collections, whereas in relational query engines every relation is considered to be flat.
[†]Both merge join and `zip` perform parallel traversal on two collections, even though they are otherwise quite different.
[‡]An Agg operator representing a GROUP BY is a transformer, whereas the one folds into only a single result is a consumer.

Figure 3 demonstrates how this rule fuses the previous example, `list.map(f).map(g)` into `list.map(f o g)`. Note that the null checking statements, which are for checking the end of a list, are removed for brevity.

This approach introduces a recursive iteration for the `filter` operation. In practice, such a recursive iteration, which is for finding the next satisfying element, can cause performance issues, even though the deforestation is applied successfully (Hinze *et al.*, 2011). Also, this approach does not fuse operations on nested collections, which is beyond the scope of this paper.

4 Loop fusion is operator pipelining

By chaining query operators, one can express a given (say, SQL) query. Similarly, a given collection program can be expressed using a pipeline of collection operators. The relationship between relational queries and collection programs has been well studied. In particular, one can establish a precise correspondence between relational query plans and a class of collection programs (Paredaens & Gucht, 1988).

Operators can be divided into three categories: (1) The operators responsible for *producing* a collection from a given source (e.g., a file or an array), (2) the operators which *transform* the given collection to another collection, and (3) the *consumer* operators which aggregate the given collection into a single result.

The mapping between query operators and collection operators is summarized in Table 1. Most join operators do not have a directly corresponding collection operator, with two exceptions: Nested loop joins can be expressed using nested `flatMaps` and the `zip` collection operator is very similar to the merge join query operator. Both operators need to traverse two input sequences in parallel. For the

Table 2. *Correspondence among pipelined query engines, object-oriented design patterns, and collection programming loop fusion*

Pipelined Query ngines	Object-oriented Design pattern	Collection Loop fusion
Pull Engine	Iterator	Unfold fusion (Svenningsson, 2002)
		Stream fusion (Coutts *et al.*, 2007)
Push Engine	Visitor	Fold fusion (Gill *et al.*, 1993)

rest of join operators, we extend collection programming with join operators (e.g., `hashJoin`, `semiHashJoin`). A similar mapping between the LINQ (Meijer *et al.*, 2006) operators and Haskell lists is shown in Steno (Murray *et al.*, 2011). Note that we do not consider nested collections here, although straightforward to support in collection programming, in order to emphasize similarity with relational query engines.

Pipelining in query engines is analogous to loop fusion in collection programming. Both concepts remove the intermediate relations and collections, which break the stream pipeline. Also, pipelining in query engines matches well-known design patterns in object-oriented programming (Vlissides *et al.*, 1995). The correspondence among pipelining in query engines, design patterns in object-oriented languages, and loop fusion in collection programming is summarized in Table 2.

Push engine = Fold fusion. There is a similarity between the Visitor pattern and fold fusion. On one hand, it has been proven that the Visitor design pattern corresponds to the Church-encoding (Böhm & Berarducci, 1985) of data types (Buchlovsky & Thielecke, 2006). On the other hand, the `foldr` function on a list corresponds to the Church-encoding of lists in λ-calculus (Pierce, 2002; Shivers & Might, 2006). Hence, both approaches eliminate intermediate results by converting the underlying data structure into its Church-encoding. In the former case, specialization consists of inlining, which results in removing (virtual) function calls. In the latter case, the fold-fusion rule and β-reduction are performed to remove the materialization points and inline the λ expressions. The correspondence between these two approaches is shown in Figure 4 (compare (a) versus (b)). The invocations of the `consume` method of the destination operators in the push engine correspond to the invocations of the `consume` function, which is passed to the `build` operator, in fold fusion.

Pull engine = Unfold fusion. In a similar sense, the Iterator pattern is similar to unfold fusion. Although the category-theoretic essence of the iterator model was studied before (Gibbons & Oliveira, 2009), there is no literature on the direct correspondence between the `unfold` function and the Iterator pattern. However, Figure 5 shows how a pull engine is similar to unfold fusion (compare Figure 5(a) versus (b)), to the best of our knowledge for the first time. Note the correspondence between the invocation of the `next` function of the source operator in pull engines and the invocation of the `next` function which is passed to the `destroy` operator in unfold fusion, which is highlighted in the figure.

```
class ScanOp[R](arr: Array[R]) {
  def init(): Unit = {
    var i = 0
    while(i < arr.length) {
      dest.consume(arr(i))
} } }
class ProjectOp[R, P](f: R => P) {
  def consume(e: R): Unit =
      dest.consume(f(e))
}
class SelectOp[R](p: R => Boolean) {
  def consume(e: R): Unit =
    if(p(e))
       dest.consume(e)
}
class AggOp[R, S](f: (R, S) => S) {
  var result = zero[S]
  def consume(e: R): Unit = {
      result = f(e, result)
  }
  def getResult: S = result
}
class HashJoinOp[R, R2]
  (leftHash: R => Int)
  (rightHash: R2 => Int)
  (cond: (R, R2) => Boolean) {
  val hm = new MultiMap[Int, R]()
  def consumeLeft(e: R): Unit = {
      hm.addBinding(leftHash(e) -> e)
  }
  def consumeRight(e: R2): Unit = {
    hm.get(rightHash(e)) match {
      case Some(list) =>
        for(l <- list) {
          if(cond(l, e)) {
              dest.consume(l.concat(e))
          }
        }
      case None =>
} } }
```

(a)

```
class QueryMonad[R] {
  def fromArray[R](arr: Array[R]) =
    build { k =>
      var i = 0
      while(i < arr.length) {
        k(arr(i))
    } }
  def map[S](f: R => S) = build { k =>
    for(e <- this)
        k(f(e))
  }
  def filter(p: R => Boolean) = build { k =>
    for(e <- this)
      if(p(e))
        k(e)
  }
  def fold[S](zero: S)(f: (R, S) => S): S = {
    var result = zero
    for(e <- this) {
      result = f(e, result)
    }
    result
  }
  def hashJoin[R2](rightList: QueryMonad[R2])
    (leftHash: R => Int)
    (rightHash: R2 => Int)
    (cond: (R, R2) => Boolean) = build { k =>
    val hm = new MultiMap[Int, R1]()
    for(e <- this) {
        hm.addBinding(leftHash(e) -> e)
    }
    for(e <- rightList) {
      hm.get(rightHash(e)) match {
        case Some(list) =>
          for(l <- list) {
            if(cond(l, e)) {
                k(l.concat(e))
            }
          }
        case None =>
} } } }
```

(b)

Fig. 4. Correspondence between push-based query engines and fold fusion of collections. (a) Push-based query engine. (b) Fold fusion of collections.

5 An improved pull-based engine

In this section, we first present yet another loop-fusion technique for collection programs. Then, we suggest a new pull-based query engine inspired by this fusion technique based on the correspondence between queries and collection programming.

5.1 *Stream fusion*

In functional languages, loops are expressed as recursive functions. Reasoning about recursive functions is very hard for optimizing compilers. Stream fusion tries to solve this issue by converting all recursive collection operations to non-recursive stream

```
class ScanOp[R](arr: Array[R]) {
  var i = 0
  def next(): R = {
    if(i < arr.length) {
      val elem = arr(i)
      i += 1
      elem
    } else
      null
} }
class SelectOp[R](p: R => Boolean) {
  def next(): R = {
    var elem: R = null
    do {
       elem = source.next()
    } while (elem != null && !p(elem))
    elem
} }
class ProjectOp[R, P](f: R => P) {
  def next(): P = {
    val elem = source.next()
    if(elem == null) null
    else f(elem)
} }
class AggOp[R, S]
  (f: (R, S) => S) {
  def next(): S = {
    var result = zero[S]
    while(true){
      val elem = source.next()
      if(elem = null)
        break
      else
        result = f(elem, result)
    }
    result
} }
class LimitOp[R](n: Int) {
  var count = 0
  def next(): R = {
    if(count < n) {
      count += 1
      source.next()
    } else {
      null
} } }
```

(a)

```
class QueryMonad[R] {
  def fromArray[R](arr: Array[R]) = {
    var i = 0
    generate { () =>
      if(i < arr.length) {
        val elem = arr(i)
        i += 1
        elem
      } else
        null
  } }
  def filter(p: R=>Boolean) = destroy { n =>
    generate { () =>
      var elem: R = null
      do {
        elem = n()
      } while(elem != null && !p(elem))
      elem
  } }
  def map[P](p: R => P) = destroy { n =>
    generate { () =>
      val elem = n()
      if(elem == null) null
      else f(elem)
  } }
  def fold[S](zero: S)
    (f: (R, S) => S): S =
    destroy { n =>
      var result = zero
      while(true){
        val elem = n()
        if(elem = null)
          break
        else
          result = f(elem, result)
      }
      result
    }
  def take(n: Int) = {
    var count = 0
    destroy { n =>
      if(count < limit) {
        count += 1
        n()
      } else {
        null
} } } }
```

(b)

Fig. 5. Correspondence between pull-based query engines and unfold fusion of collections. (a) Pull-based query engine. (b) Unfold fusion of collections.

operations. To do so, first all collections are converted to streams using the `stream` method. Then, the corresponding method on the stream is invoked that results in a transformed stream. Finally, the transformed stream is converted back to a collection by invoking the `unstream` method.

The signature of the `unstream` and `stream` methods is as follows:

```
def unstream[T](gen: () => Step[T]): List[T]
class List[T] {
  def stream(): Step[T]
}
```

```
trait Step[T] {
 def filter(p: T => Boolean): Step[T]
 def map[S](f: T => S): Step[S]
 def fold[S](yld: T => S, skip: () => S, done: () => S): S
}

case class Yield[T](e: T) extends Step[T] {
 def filter(p: T => Boolean) =
   if(p(e)) Yield(e) else Skip
 def map[S](f: T => S) =
   Yield(f(e))
 def fold[S](yld: T => S, skip: () => S, done: () => S): S =
   yld(e)
}

case object Skip extends Step[Nothing] {
 def filter(p: Nothing => Boolean) =
   Skip
 def map[S](f: Nothing => S) =
   Skip
 def fold[S](yld: Nothing => S, skip: () => S, done: () => S): S =
   skip()
}

case object Done extends Step[Nothing] {
 def filter(p: Nothing => Boolean) =
   Done
 def map[S](f: Nothing => S) =
   Done
 def fold[S](yld: Nothing => S, skip: () => S, done: () => S): S =
   done()
}
```

Fig. 6. The operations of the Step data type.

For example, the `map` method is expressed in using these two methods as:

```
class List[T] {
 def map[S](f: T => S): List[S] = unstream { () =>
   this.stream().map(f)
 }
}
```

The `stream` method converts the input collection to an intermediate stream, which is specified by the `Step` data type. The function `f` is applied to this intermediate stream using the `map` function of the `Step` data type. Afterwards, the result stream is converted back to a collection by the `unstream` method.

As discussed before, one of the main advantages of the intermediate stream, the `Step` data structure, is that its operations are mainly non-recursive. This simplifies the task of the optimizing compiler to further specialize the program. The implementation of several methods of the `Step` data structure is given in Figure 6.

Such transformations do not result in direct performance gain – they may even degrade performance. This is because of the intermediate conversions between streams and collections. However, these intermediate conversions can be removed using the following rewrite rule:

Stream-fusion rule:

```
unstream(() => e).stream()   ⇝   e
```

Figure 3 demonstrates how the stream fusion technique transforms `list.map(f).map(g)` into `list.map(f o g)`. Note that for the `Step` data type, the `step.map(f).map(g)` expression is equivalent to `step.map(f o g)`.

The idea behind stream fusion is very similar to unfold fusion. The main difference is the `filter` operator. Stream fusion uses a specific value, called `Skip`, to implement the `filter` operator. This is in contrast with the unfold fusion approach for which the `filter` operator is implemented using an additional nested `while` loop for skipping the unnecessary elements. Hence, stream fusion solves the practical problem of unfold fusion associated with the `filter` operator.

Next, we define a new pipelined query engine based on the ideas of stream fusion.

5.2 Stream-fusion engine

The proposed query engine follows the same design as the iterator model. Hence, this engine is also a pull engine. However, instead of invoking the `next` method, this engine invokes the `stream` method, which returns a wrapper object of type `Step`. We refer to our proposed engine as the *stream-fusion engine*.

As we mentioned in Section 2.1, one of the main practical problems with a pull engine is the case of the selection operator. In this case, an operator waits until the selection operator returns the next satisfying element. The proposed engine solves this issue by using the `Skip` object that specifies that the current element should be ignored. Hence, selection operators are no longer a blocker for their destination operator.

The correspondence between the stream fusion algorithm and the stream-fusion engine is shown in Figure 7. Every query operator provides an appropriate implementation for the `stream` method, which invokes the `stream` method of the source operator to request the next element. Similarly, stream fusion uses the `stream` method to fetch the next element. Then, by invoking the `unstream` method, the generated stream is converted back to a collection.

From a different point of view, a push engine can be expressed using a `while` loop and a construct for skipping to the next iteration (e.g., `continue`). By nature, it is impossible for a push-based engine to finish the iteration before the producer's `while` loop finishes its job. In other words, the generated C code using a push-based engine never uses the `break` construct.

In contrast, a pull engine is generally expressible using a `while` loop and a construct for terminating the execution of the `while` loop (i.e., the `break` construct). This is because of the demand-driven nature of pull engines. However, in a pull-based engine there is no way to skip an iteration. As a result, skipping an iteration should be expressed using a nested `while` loop that results in performance issues (c.f. Section 2).

The stream-fusion engine combines the benefits of both engines by providing the following two constructs for early termination of loops and skipping an iteration. First, the `Done` construct denotes the termination of loops, and in essence has the same effect as the `break` construct in an imperative language like C. Second, the `Skip` construct results in skipping to the next iteration, and has an equivalent effect

```
class ScanOp[R](arr: Array[R]) {
  var i = 0
  def stream(): Step[R] = {
    if(i < arr.length)
      Yield(arr(i))
    else
      Done
} }
class SelectOp[R](p: R => Boolean) {
  def stream(): Step[R] = {
    source.stream().filter(p)
} }
class ProjectOp[R, P](f: R => P) {
  def stream(): Step[P] = {
    source.stream().map(f)
} }
class AggOp[R, S](f: (R, S) => S) {
  def stream(): Step[S] = {
    var result = zero[S]
    var done = false
    while(!done){
      source.stream().fold(
        e => { result = f(e, result) },
        () => ,
        () => { done = true }
      )
    }
    result
} }
class LimitOp[R](n: Int) {
  var count = 0
  def stream(): Step[R] = {
    if(count < n) {
      source.stream().map(e => {
        count += 1
        e
      })
    } else {
      Done
} } }
```

(a)

```
class QueryMonad[R] {
  def fromArray[R](arr: Array[R]) = {
    var i = 0
    unstream { () =>
      if(i < arr.length)
        Yield(arr(i))
      else
        Done
  } }
  def filter(p: R => Boolean) = {
    unstream { () =>
      stream().filter(p)
  } }
  def map[P](f: R => P) = {
    unstream { () =>
      stream().map(f)
  } }
  def fold[S](z: S)(f: (R, S) => S): S = {
    unstream { () =>
      var result = zero[S]
      var done = false
      while(!done){
        stream().fold(
          e => { result = f(e, result) },
          () => ,
          () => { done = true }
        )
      }
      result
  } }
  def take(n: Int) = {
    var count = 0
    unstream { () =>
      if(count < n) {
        stream().map(e => {
          count += 1
          e
        })
      } else {
        Done
} } } }
```

(b)

Fig. 7. Correspondence between stream-fusion query engine and the stream fusion technique. (a) Stream-fusion query Engine. (b) Stream fusion of collections.

to the `continue` construct in an imperative language like C. Table 3 summarizes the differences among the aforementioned query engines.

Consider a relation of two elements for which we select its first element and the second element is filtered out. The first call to the `stream` method of the selection operator in the stream-fusion engine produces a `Yield` element, which contains the first element of the relation. The second invocation of the same method returns a `Skip` element, specifying that this element, which is the second element of the relation, is filtered out and should be ignored. The next invocation of this method results in a `Done` element, denoting that there is no more element to be produced by the selection operator. The `Done` value has the same role as the `null` value in the pull engine.

The specialized version of the example query (which was introduced in Figure 1) based on the stream-fusion engine is shown in Figure 8(a). The code is as compact

Table 3. *The supported looping constructs by each pipelined query engine*

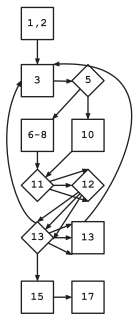
Pipelined query engines	Looping constructs
Push engine	`while` + `continue`
Pull engine	`while` + `break`
Stream-fusion engine	`while` + `break` + `continue`

```
1  var index = 0
2  var sum = 0.0
3  while(true) {
4    val step1 =
5      if(index < R.length) {
6        val rec = R(index)
7        index += 1
8        Yield(rec)
9      } else
10       Done
11   step1.filter(x => x.A < 10)
12   .map(x => x.B)
13   .fold(x => sum += x,
14         () => ,
15         () => break)
16 }
17 return sum
```

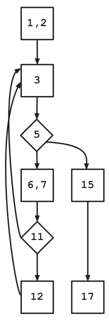
(a)

```
var index = 0
var sum = 0.0
while(true) {

  if(index < R.length) {
    val rec = R(index)
    index += 1


    if(rec.A < 10)
      sum += rec.B
  }
  else
    break
}
return sum
```

(b)



(c)



(d)

Fig. 8. Specialized version of the example query in stream-fusion engine and the corresponding control-flow graph (CFG). (a) Inlined query in stream-fusion engine without further specializations. (b) Inlined query in stream-fusion engine by inlining the visitor model of `Step`. (c) The CFG of the inlined query in stream-fusion engine without further specializations. (d) The CFG of the inlined query in stream-fusion engine by inlining the visitor model of `Step`.

as the push engine code. However, the CFG is similar to (or even more complicated than) the one of a pull engine (c.f. Figure 8(c)). Furthermore, the specialized stream-fusion engine suffers from more performance problems due to the intermediate `Step` objects created. The next section discusses implementation aspects and the optimizations needed for tuning the performance of the stream-fusion engine.

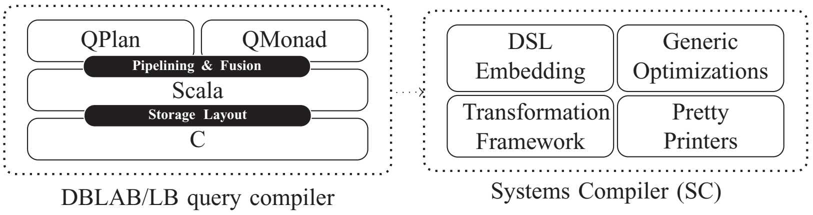


Fig. 9. The architecture of the DBLAB/LB query compiler and Systems Compiler (SC).

6 Implementation

In this section, we discuss the implementation of the presented query engines. First, we show the architecture of our query compiler. Then, we discuss how the fusion rules are implemented for each approach. Finally, we show how the problem associated with intermediate objects is resolved for the stream-fusion engine.

6.1 Architecture

We have implemented different query engines and the associated optimizations in the DBLAB/LB query compiler (Shaikhha *et al.*, 2016). This query compiler is a component of DBLAB,[2] a framework for building efficient database systems in the high-level programming language Scala.

The DBLAB/LB query compiler uses the compilation facilities provided by Systems Compiler (SC)[3] in order to implement several intermediate languages (through *language embedding* (Hudak, 1996)), the transformations inside and across these languages (using the transformation framework), and finally unparsing the transformed program into Scala or C code (using the pretty printers). Furthermore, SC provides several generic optimizations out-of-the-box, which DBLAB/LB uses during query compilation. These optimizations include Common-Subexpression Elimination, Dead-Code Elimination, Partial Evaluation, and Scalar Replacement.

The architecture of DBLAB/LB and SC is shown in Figure 9. The input programs can either be expressed using physical (relational algebra-style) query plans in the QPlan language or collection programming using the QMonad language. Depending on the input language, the query compiler performs either pipelining or loop fusion. These transformations result in a low-level Scala program, which does not have the high-level constructs of the QPlan and QMonad languages.[4] In order to transform this Scala program into a C program, the storage layout for records should be specified. DBLAB/LB provides both row and columnar storage layouts for relations (Shaikhha *et al.*, 2016). Finally, the DBLAB/LB query compiler uses the C pretty printer provided by SC to generate C code.

We have implemented the collection programming operations and the corresponding loop fusion techniques described earlier in this article. Thanks to

[2] http://github.com/epfldata/dblab
[3] http://github.com/epfldata/sc-public
[4] Note that DBLAB/LB defines more intermediate languages in its compilation stack (Shaikhha *et al.*, 2016), which we skipped for the sake of brevity.

```scala
type Cont[T] = (T => Unit) => Unit

class Consumer[T](cont: Cont[T])
  extends QueryMonad[T] {
  def foreach(f: T => Unit): Unit =
    cont(f)
}

def build[T](cont: Cont[T])
  : QueryMonad[T] =
  new Consumer[T](cont)
```

(a)

```scala
type Gen[T] = () => T

class DestroyGen[T](gen: Gen[T])
  extends QueryMonad[T] {
  def destroy[S](f: Gen[T] => S): S =
    f(gen)
}

def generate[T](gen: Gen[T])
  : QueryMonad[T] =
  new DestroyGen[T](gen)
```

(c)

```
build(f1).foreach(f2)

 |  (inline build definition)
 ↓

new Consumer(f1).foreach(f2)

 |  (inline foreach definition)
 ↓

f1(f2)
```

(b)

```
generate(f1).destroy(f2)

 |  (inline generate definition)
 ↓

new DestroyGen(f1).destroy(f2)

 |  (inline destroy definition)
 ↓

f2(f1)
```

(d)

Fig. 10. Constructs and derivation of fold fusion and unfold fusion. (a) The constructs for fold fusion. (b) The derivation of the fold-fusion rule. (c) The constructs for unfold fusion. (d) The derivation of the unfold-fusion rule.

the equivalence which was shown in Section 4 between query engines and collection programming, it is clear how different pipelining techniques can be implemented for query engines. As a result, it is not surprising that the experimental results presented in the next section for different fusion techniques match the results for the corresponding pipelined query engines. Next, we discuss how the fusion rules for different loop fusion algorithms can be expressed in this framework.

6.2 Fusion by inlining

As mentioned in Section 4, a fusion rule is expressed as a local transformation rule which is applied as an extension to the host language compiler (which is the Glasgow Haskell Compiler (GHC) (Jones *et al*., 1993) in the case of the mentioned papers). In this section, we show how these fusion rules are implemented by only using inlining. This was proposed for implementing fold fusion in Scala (Jonnalagedda & Stucki, 2015). Here, we use a similar approach for other fusion techniques.

Figure 10(a) shows the definition of the `build` operator. By inlining the definition of this operator, an object of type `QueryMonad` is created. The `foreach` method of this object applies the higher order function passed to the `build` method (`f1`) to the input parameter of the `foreach` method (`f2`). By inlining this `foreach` method, we derive the same rule as the fold-fusion rule which was introduced in Section 3. This derivation is shown in Figure 10(a). The constructs and derivation of unfold fusion are shown in Figure 10(c) and 10(d). Stream fusion follows a similar pattern which is given

```
type GenStream[T] = () => Step[T]

class Streamer[T](gen: GenStream[T]) extends QueryMonad[T] {
  def stream(): Step[T] = gen()
}

def unstream[T](gen: GenStream[T]): QueryMonad[T] =
  new Streamer[T](gen)
```

Fig. 11. The constructs for stream fusion.

```
unstream(() => e).stream()

  |
  |  (inline unstream definition)
  ↓

new Streamer(() => e).stream()

  |
  |  (inline stream definition)
  ↓

e
```

Fig. 12. The derivation of the stream-fusion rule.

in Figures 11 and 12. Figures B1 and B2 show the fusion process for the working example using fold and unfold fusion, respectively.

Next, we discuss the problematic creation of intermediate objects by the stream-fusion engine, as well as our solution.

6.3 Removing intermediate results

Although the stream-fusion engine removes intermediate relations, it creates intermediate `Step` objects. There are two problems with these intermediate objects. First, the `Step` data type operations are virtual calls. This causes poor cache locality and degrades performance. Second, normally these intermediate objects lead to heap allocations. This causes higher memory consumption and worse running times. This is why the original stream fusion approach is dependent on optimizations provided by its source language compiler (i.e., the GHC (Jones *et al.*, 1993) compiler). Implementing an effective version of it for other languages requires supporting similar optimizations supported by the GHC compiler.

The first problem with virtual calls can be solved by rewriting the `Step` operations by enumerating all cases for the `Step` object. This is possible because there are only three possible concrete cases (1. `Yield` 2. `Skip` 3. `Done`) for this data type. To do so, one can use if-statements. In functional languages, the pattern matching feature can be used. Although this approach solves the first problem, still there are heap allocations which are not removed.

The good news is that these heap allocations can be converted to stack allocations. This is because the created objects are not escaping their usage scope. For example, these objects are not copied into an array and not used as an argument to a function. This fact can be verified by the well-known compilation technique of

```
trait StepVisitor[T] {
  def yld(e: T): Unit
  def skip(): Unit
  def done(): Unit
}

trait Step[T] { self =>
  def __match(v: StepVisitor[T]): Unit
  def filter(p: T => Boolean): Step[T] =
    new Step[T] {
      def __match(v: StepVisitor[T]): Unit =
        self.__match(new StepVisitor[T] {
          def yld(e: T): Unit =
            if (p(e)) v.yld(e) else v.skip()
          def skip(): Unit = v.skip()
          def done(): Unit = v.done()
        })
    }
  def map[S](f: T => S): Step[S] =
    new Step[S] {
      def __match(v: StepVisitor[S]): Unit =
        self.__match(new StepVisitor[T] {
          def yld(e: T): Unit = v.yld(f(e))
          def skip(): Unit = v.skip()
          def done(): Unit = v.done()
        })
    }
}

case class Yield[T](e: T) extends Step[T] {
  def __match(v: StepVisitor[T]): Unit = v.yld(e)
}
case object Skip extends Step[Nothing] {
  def __match(v: StepVisitor[Nothing]): Unit = v.skip()
}
case object Done extends Step[Nothing] {
  def __match(v: StepVisitor[Nothing]): Unit = v.done()
}
```

Fig. 13. Step data type implemented using the Visitor pattern.

escape analysis (Choi *et al.*, 1999). Based on that, the heap allocations can be converted to stack allocations.

The compiler optimizations can go further and remove the stack allocations as well. Instead of the stack allocation for creating a `Step` object, the fields necessary to encode this type are converted to local variables. Hence, the `Step` abstraction is completely removed. This optimization is known as *scalar replacement* in compilers.

From a different point of view, removing the intermediate `Step` objects is a similar problem to removing the intermediate relations and collections in query engines and collection programming. Hence, one can borrow similar ideas and apply it for the `Step` objects in a fine-grained granularity.

To do so, we implement a variant of the `Step` data type using the Visitor pattern. As we discussed in Section 4, this is similar to the Church-encoding of data types. This encoding results in *pushing* `Step` objects down the pipeline. Hence, the stream-fusion engine implements a pull engine on a coarse-grained level (i.e., relation level) and pushes the individual elements on a fine-grained level (i.e., tuple level). The Visitor pattern version of the `Step` data type is shown in Figure 13.

The result of applying this enhancement to our working example is shown in Figure 8(b). By comparing this code to the code produced by a push engine, we see a clear similarity. First, there are no more additional virtual calls associated with the `Step` operators. Second, there is no more materialization of the intermediate `Step` objects. Finally, similar to push engines, the produced code does not contain any additional nested `while` loop for selection. This leads to a simpler CFG, which is shown in Figure 8(d).

As an alternative implementation, one can implement the `Step` data type as a *sum* type, a type with different distinct cases in which an object can be one and only one of those cases. Hence, the implementation of the `Step` methods can use the pattern matching feature of the Scala programming language. However, it has been proven that the Visitor pattern is a way to encode the sum types in object-oriented languages (Buchlovsky & Thielecke, 2006). On the other hand, pattern matching in Scala is a way to express the Visitor pattern (Emir *et al.*, 2007). Hence, from a conceptual point of view there is no difference between these implementations (Hofer & Ostermann, 2010).

7 Experimental results

We use a server-type x86 machine equipped with two Intel Xeon E5-2620 v2 CPUs running at 2 GHz each, 256 GB of DDR3 RAM at 1600 Mhz and two commodity HDDs of 2TB. The operating system is Red Hat Enterprise 6.7.

Our query compiler uses the same set of transformations for different pipelining techniques to allow for a fair comparison. These transformations consist of *dead code elimination*, *common subexpression elimination* or *global value numbering*, and *partial evaluation* (inlining and constant propagation). These transformations are provided out-of-the-box by DBLAB (Shaikhha *et al.*, 2016), which we use as our testbed. Also, the scalar replacement transformation is always applied unless otherwise specified. We do not use any data-structure specialization transformations or inverted indices for these experiments. Finally, all experiments use DBLAB's in-memory row-store representation.

For compiling, the generated programs throughout our evaluation we use version 3.9.1 of the CLang compiler. We use the most aggressive optimization strategy provided by the CLang compiler (the `–O3` optimization flag).[5] Finally, for C data structures we use the GLib library (version 2.42.1).

Our evaluation consists of two parts. First, by using micro-benchmarks, we clearly demonstrate the differences between different query engines. Then, for more complex queries, we use the TPC-H (Transaction Processing Performance Council, 2017) benchmark to demonstrate how different query engines behave in more complicated scenarios.

[5] We observed similar performance results with the `–O1` optimization flag. The `–O1` optimization flag provides all the transformation passes used in HyPer (Neumann, 2011) except global value numbering (GVN). This transformation is not needed in our case, as it is already provided by DBLAB (Shaikhha *et al.*, 2016).

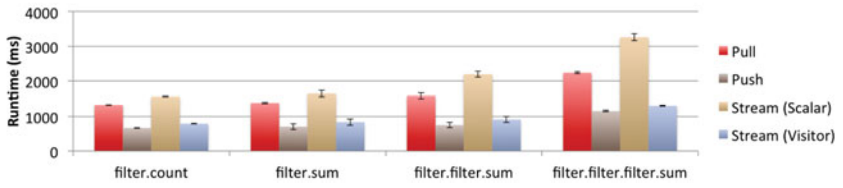


Fig. 14. Single-pipeline queries compiled without any optimization flags specified for CLang.

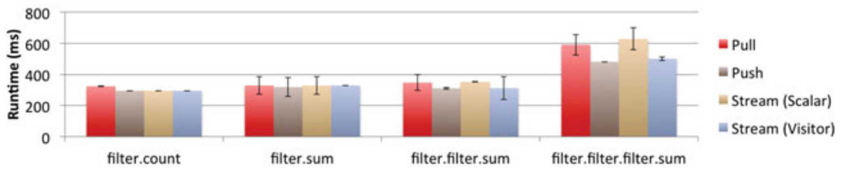


Fig. 15. Single-pipeline queries compiled with the -O3 optimization flag for CLang.

7.1 Micro-benchmarks

The micro-benchmarks belong to three categories, (1) queries consisting of only selection and aggregation without group by attributes leading to a single result, (2) queries consisting of a limit operator, which return a list of results, and (3) queries with selection and different join operators, such as hash join, merge join, and hash semi-join, which are followed by an aggregation operator resulting to a single result. All these queries use generated TPC-H databases at scaling factor 8, unless otherwise specified. The corresponding SQL queries for all these micro-benchmarks are shown in Table A1.

Aggregated single pipeline. Next, we measure the performance obtained by each engine for queries with a single pipeline, which aggregate into a single result. Figure 14 shows the performance of different engines when the generated C code is compiled without any optimization flags. The push engine is behaving 2X better than the pull engine in most cases. The visitor-based stream-fusion engine hides this limitation of the pull engine, and has a similar performance to push engines. However, the stream-fusion engines that use scalar replacement perform worse than pull engines.

The difference is more obvious whenever there are chains of selection operations. A similar effect was shown in HyPer (Neumann, 2011) in the case of using up to four consecutive selection operations. Again the visitor-based stream-fusion engine is resolving this practical limitation of pull engines. From a practical point of view, as the query optimizer is merging all conjunctive predicates into a single selection operator, the case in which a chain of several selection operators are followed by each other never happens in practice.

The difference among all types of engines can be removed by using more aggressive optimizations of the underlying optimizing compiler. Figure 15 shows that using the -O3 optimization flag of CLang, the performance of all types of engines is similar. This is mainly thanks to the CFG simplification performed by

```
1  var sum = 0.0
2  var index = 0
3  var count = 0
4  while(true) {
5    if(count < 1000) {
6      var rec = null
7      if(index < R.length) {
8        rec = R(index)
9        index += 1
10     } else {
11       rec = null
12     }
13     if(rec == null) break
14     else sum += rec.B
15     count += 1
16   } else {
17     break
18 } }
19 return sum
```

(a)

```
var sum = 0.0
var index = 0
var count = 0


while(index < R.length) {
  val rec = R(index)
  index += 1



  if(count < 1000) {
      sum += rec.B
      count += 1
  }

}
return sum
```

(b)

Fig. 16. Compiled version of the `take.sum` query in pull and push engines. (a) Inlined query in pull engine. (b) Inlined query in push engine.

CLang. However, queries with more complicated selection predicates (e.g., user-defined functions or external functions such as `strcmp`) make the reasoning hard for the underlying optimizing compiler. Hence, CFG simplification cannot be applied, and push-based engine, and stream-fusion engine have a superior performance in comparison with a pull-based engine. The impact of the optimizations provided by an underlying optimizing compiler is discussed in more details in Section 9.

Single pipeline with limit. Next, we examine the results for single pipeline queries that have a limit operator at the end of the pipeline. In all three queries, the push engine is performing worse than the pull-based engine and the stream-fusion engine. This is because the standard push engine cannot perform early loop-termination when using the limit query operator (c.f. Section 2.2).

To better illustrate the mentioned behavior, Figure 16 shows the generated code for the `take.sum` query for pull and push-based query engines. The pull engines do not require traversing all the elements and can stop immediately after reaching the limit operator (c.f. line 18 of Figure 16(a)). However, the push engine should wait until all elements are produced to be able to finish the execution (c.f. Figure 16(b)). A more detailed explanation on a similar query is given in Section 9. A similar behavior has been observed for pull-based and push-based fusion techniques for Java 8 streaming API in Biboudis *et al.* (2015).

Single join. Finally, we investigate the performance of different join operations, which is demonstrated in Figure 17. In the case of hash join and left-semi hash-join operators, there is no obvious difference among the engines. However, in the case of merge join, there is a great advantage for pull engines in comparison with the push engine. This is mainly because the push engine cannot pipeline both inputs of a merge join. Hence, it is forced to break the pipeline in one of the inputs (c.f.

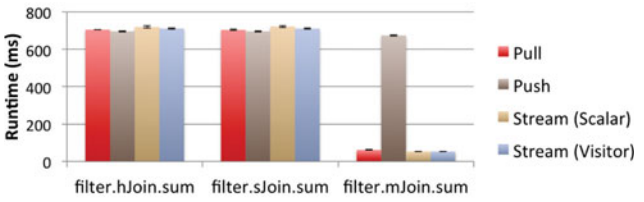


Fig. 17. Single-join queries using hash join (hJoin), left-semi hash join (sJoin), and merge join (mJoin) operators.

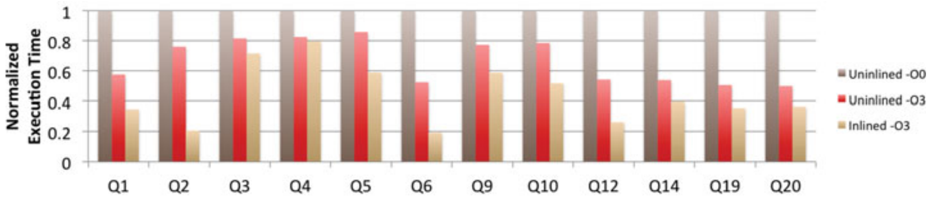


Fig. 18. The impact of inlining and low-level optimizations of CLang on a pull-based engine for TPC-H queries.

Section 2.2).[6] A more detailed investigation of the merge join operator is given in Section 9.

7.2 Macro-benchmarks

In this section, we investigate scenarios that are happening more often in practice. To do so, we use the larger and more complicated analytical queries defined in the TPC-H benchmark. First, we investigate the difference between an inlined and an uninlined version of a pull-based query engine on 12 TPC-H queries. Then, we show the impact of fine-grained optimizations as well as a inline-friendly way of implementing pull engines on one of the TPC-H queries. Finally, we demonstrate the performance difference among different types of query engines for 12 TPC-H queries. The remaining 10 TPC-H queries require features, which are not supported by all our query engines. All these experiments use 8 GBs of TPC-H generated data.

The impact of inlining on pull engine. As it was explained in Section 2.3, we expect inlined (compiled) query engines to perform better than their corresponding uninlined (interpreted) version. Figure 18 demonstrates the normalized execution time for 12 TPC-H queries for interpreted and compiled pull-based query engines. The compiled query engine inlines the `next` function invocations of a pull-based query engine, whereas the interpreted query engine invokes the (virtual) functions during run time. Performing aggressive compilation of the interpreted query using the `–O3` flag of the CLang compiler, improves the performance of the interpreted

[6] The stream-fusion engine should have a special case for handling merge joins followed by filter operations. By skipping, the elements in the main loop of merging, many CPU cycles are wasted for retrieving the next satisfying element. However, accessing them by using a similar approach to the Iterator model (keep iterating until the next satisfying element is found in a tight loop) gives a better performance.

Table 4. *The performance comparison of several variants of different engines on TPC-H query 19*

Type of engine	Run time (ms)
Pull (Interpreted)	3,486
Pull (Naïve)	2,405
Pull (Inline-Friendly)	2,165
Stream (Scalar replacement for `Step` objects)	2,447
Stream (Visitor model for `Step` objects)	2,217
Stream (No removal of `Step` objects)	6,886

query. However, the best performance is achieved by generating C code using query compilation, and then compiling the generated code using the `–O3` flag of the CLang compiler. On average, inlining the pull-based engine gives 67% improvement. In particular, for TPC-H query 2, we observe a 4 times speedup. This considerable performance improvement is the result of the removal of intermediate object allocations, which is achieved after inlining the operators of the query by DBLAB/LB. One exception is TPC-H query 4, which we see a negligible performance improvement after inlining. This query uses a semi hash join operator for implementing the functionality required for the EXISTS clause. The cost required for building the intermediate hash table (which is implemented using the GLib library) dominates the cost of (virtual) function calls. Hence, we do not see a significant improvement by inlining those function calls. The absolute execution times for all these queries can be found in Table 5. Note that, the performance difference with LegoBase (Klonatos *et al.*, 2014a; Shaikhha *et al.*, 2018) and DBLAB/LB (Shaikhha *et al.*, 2016) is due to the lack of additional optimizations provided by these systems such as data-structure specialization.

Inline-Friendly pull engine implementation. A naïve implementation of the selection operator in a pull-based query engine, invokes the `next` method of its source operator twice. This can exponentially grow the code size in the case of a chain of selection operators. This case is not frequent in practice, since the selection operator is mainly used right after the scan operator. However, in the case of TPC-H query 19 the selection operator is used after a join.[7] Table 4 shows that the inline-friendly implementation of the selection operator in pull engines, improves performance by 15%. One of the main reasons is that the inline-friendly implementation generates around 40% less query processing code in comparison with the naïve implementation for query processing in these two queries. This improves instruction cache locality, as a larger part of the code can fit into the instruction cache.

Removing intermediate object allocations. Table 4 shows the impact of intermediate object allocations on performance. Overall, removing heap allocations of

[7] An alternative implementation is to fuse the selections happening after joins in the join operator itself. The experiments performed in (Schuh *et al.*, 2016) are based on this assumption for join operators. This means that the join operator is not a pure join operator, but a super operator containing a join operator followed by a selection operator. For the purposes of this paper, we do not consider this case.

Table 5. *Execution times (in milliseconds) of different compiled query engines for TPC-H queries*

Query engine	Q1	Q2	Q3	Q4	Q5	Q6	Q9	Q10	Q12	Q14	Q19	Q20
Pull uninlined -O0	10,249	3,872	9,022	14,574	6,925	1,748	19,918	5,155	3,039	4,205	6,882	2,674
Pull uninlined -O3	5,911	2,949	7,371	12,042	5,947	919	15,420	4,055	1,652	2,271	3,486	1,338
Pull -O0	8,344	1,907	8,810	14,834	6,528	1,470	16,655	4,979	2,788	2,749	6,853	2,378
Push -O0	6,982	1,166	7,521	13,690	5,460	868	15,045	3,994	2,326	2,097	6,022	1,605
Stream (visitor) -O0	7,287	1,314	7,730	13,560	5,827	1,077	15,278	4,087	2,188	2,358	6,168	1,833
Pull -O3	3,540	769	6,387	11,474	3,970	338	10,612	2,588	921	1,263	2,165	869
Push -O3	3,652	622	6,429	11,557	3,927	359	10,809	2,742	1,121	1,447	2,218	868
Stream (visitor) -O3	3,552	704	6,652	11,260	4,640	376	10,703	2,659	928	1,334	2,217	881

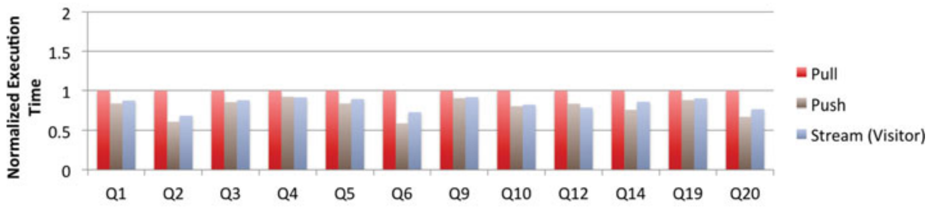


Fig. 19. Performance of different compiled query engines for TPC-H queries, when using the `–O0` flag with the CLang compiler.

intermediate `Step` objects improves the performance of a stream-fusion engine up to three times. More specifically, the visitor model for `Step` objects improves performance by 50% in comparison with the case in which Scalar Replacement is used for removing intermediate heap allocations (c.f. Section 6.3). Furthermore, our experiments show that removing intermediate `Step` objects (either by visitor model or Scalar Replacement) decreases the memory consumption from 14 GBs to 11 GBs for TPC-H query 19.

Different engines on analytical queries. Figures 19 shows the performance of several TPC-H queries using different engines, when they are not using any optimizations provided by CLang. Overall, this figure shows that the difference between engines is not in terms of "orders of magnitude;" in most cases, improvements are minor. This is because the comparison is performed in a fair scenario in which specialization is performed on all engines, in contrast with previous work in which operator inlining was not applied to pull engines (Klonatos *et al*., 2014a).

Based on this figure we make the following observations. First, in all cases, the push-based engine is outperforming both pull-based engines. This is justified by the simplified control flow produced by push-based engines. Second, the visitor-based stream-fusion engine has a similar performance to a push-based engine, thanks to the simplified control flow offered by its visitor-based tuple-level implementation. Finally, even for queries with limit and merge join the pull-based engine is not performing better than the push-based engine. For queries with limit, as the limit operator is followed by an ordering operator, the cost of sorting outweighs the cost of the final scan. As a result, pipelining the limit operator does not have a considerable impact. For TPC-H query 12, which has a merge join operator, the performance penalty caused by the complicated CFG of pull-based engine hides the improvement of pipelining this join operator. However, the stream-fusion engine has a better performance than both pull- and push-based engines, thanks to pipelining the merge join operator, while keeping the control flow simple.

Now, we answer the following question: to which extent the underlying optimizing compiler can hide the limitations of each engine? Figures 20 shows the performance of several TPC-H queries using different engines, when the CLang compiler is used with the `–O3` flag. Overall, this figure shows that for most queries all types of engines have a similar performance. This means that the underlying optimizing compiler (CLang) successfully optimizes the code generated by pull-based engines, to the extent that the generated machine code behaves similarly to the generated machine code for push-based engines.

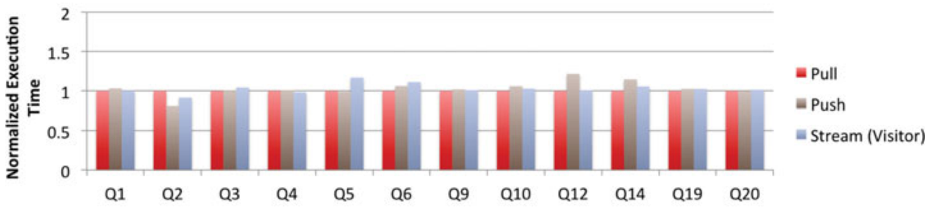


Fig. 20. Performance of different compiled query engines for TPC-H queries, when using the -03 flag with the CLang compiler.

However, there are still some cases for which CLang cannot compensate the limitation of a particular engine. Query 12 falls into this category because of its use of the merge join operator. This query has an average 70% speed up for a pull engine in comparison with a push engine. It is important to note that in this query, the query plan that uses a merge join is almost two times faster than the one that uses hash join. This is because both input relations are already sorted on the join key. Hence, the merge join implementation can perform the join on the fly, as opposed to the hash join implementation which needs to construct an intermediate hash table while joining the two input relations.

The stream-fusion engine always uses the Visitor pattern throughout this experiment. Interestingly, it is performing as well as push engines and significantly better than pull engines, whenever one does not rely on an underlying optimizing compiler to simplify the control flow. Furthermore, in the cases where push engines require to break the pipeline (the limit and merge join operators) the stream-fusion engine is as efficient as pull engines.

In this section, we have shown the experimental results for different design choices for building complied query engines. Although in realistic workloads there is no considerable advantage for either form of query engine, in certain edge cases each of the pull- and push-based engines have their own advantages. We have seen how the stream-fusion engine combines the individual advantages of both approaches in such edge cases, while avoiding their weaknesses. This makes the stream-fusion engine a good alternative choice for building compiled query engines.

8 Discussion: Column stores and vectorization

Column stores. For analytical workloads, there is merit in column-store databases (Idreos *et al*., 2012; Stonebraker *et al*., 2005). The PL community is using a similar representation (Peyton Jones *et al*., 2008), known as structs of arrays. Furthermore, database systems can leverage the compression opportunities provided by the column-stores that can improve performance and space consumption (Abadi *et al*., 2006; Zukowski *et al*., 2006; Binnig *et al*., 2009).

As it was explained in Section 6.1, DBLAB/LB supports both row layout and columnar layout representations. The columnar layout representation is achieved by translating the array of records coming from a row layout representation, each one containing N fields, into N arrays, where each array corresponds to the values of a particular column. Figure 21 shows the generated code of our running example for

```
 1 var sum = 0.0                                    var sum = 0.0
 2 var index = 0                                    var index = 0
 3 while(true) {
 4   var recNull = true
 5   do {
 6     if(index < R_length) {                       while(index < R_length) {
 7       recNull = false
 8       index += 1                                   index += 1
 9     } else {
10       recNull = true
11     }
12   } while(!recNull && !(R_A(index) < 10))        if(R_A(index) < 10)
13   if(recEmpty) break
14   else sum += R_B(index)                             sum += R_B(index)
15 }                                                }
16 return sum                                       return sum
```

(a) (b)

Fig. 21. Specialized version of the example query in column-store pull and push engines. (a) Inlined query in a column-store pull engine. (b) Inlined query in a column-store push engine.

push and pull-based engines. Lines 12 and 14 of this figure show how the accesses to the fields `A` and `B` of a record of the relation `R` are transformed into the accesses to the corresponding column arrays `R_A` and `R_B` in the columnar layout representation.[8]

Although the experiments shown in this paper use row layout, we have found similar results when comparing different engines using columnar layout representation. More specifically, Figure 22 shows the performance of the single-pipeline aggregated queries for different types of compiled columnar and row layout query engines. The performance of column-store engines is better than the performance of the row-store counterparts. Furthermore, the relative speedup of different engines over a pull-based engine using the columnar layout representation is similar to the speedup of the corresponding engines over a pull-based engine for the row layout representation. The only considerable difference is the first query, which counts the number of the filtered elements. This query gives a better performance for column-store push and stream-fusion engines, thanks to the automatic vectorization performed by CLang. However, the generated code of a column-store pull-based engine cannot be automatically vectorized by the compiler due to the existence of data-dependent exit conditions (Park *et al.*, 2012).

Supporting hash-based join operators for column-stores requires careful consideration of where to construct the tuples (a.k.a. the materialization strategies (Abadi *et al.*, 2007)), and even implementing other join operators (such as JIVE join (Li & Ross, 1999)), which we leave for future work.

Vectorization. Using SIMD operations for implementing query operators has been extensively investigated by the DB community (Zhou & Ross, 2002; Chhugani *et al.*, 2008; Polychroniou *et al.*, 2015). MonetDB (Zukowski *et al.*, 2005) implemented a

[8] Note that in most real-world database systems a column-store consists of other abstractions such as pages. However, for the purposes of this paper, we have presented a simplified form which only uses column arrays. Furthermore, as the column arrays have a primitive type (i.e., cannot have a null value), the check for termination in a pull-based engine (i.e., checking the equality to null) is handled through the intermediate boolean variable `recNull`.

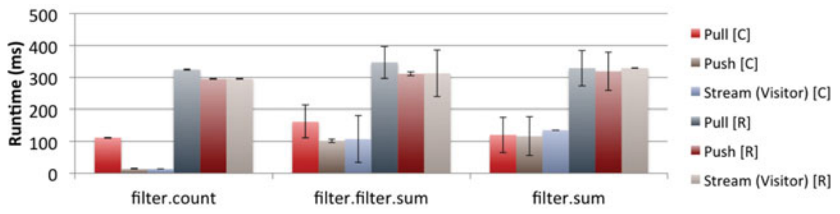


Fig. 22. Performance of different compiled query engines with columnar layout and row layout representations (denoted by [C] and [R], respectively), when using the `-O3` flag with the CLang compiler.

vectorized query engine by performing block-wise data processing instead of tuple-wise processing, through transferring a block of elements in the iterator model instead of a single element. Generalized stream fusion (Mainland *et al.*, 2013) followed a similar idea and showed how, by exploring vectorization opportunities, a high-level functional language can beat handwritten C code for collection programs. We can follow a similar idea to perform vectorization for the stream-fusion engine. On the other hand, push engines can also benefit from vectorization by pushing a block of elements and processing them using SIMD operations as explained in (Neumann, 2011).

9 Conclusion

If one effects a fair comparison of push and pull-based query processing – particularly if one attempts to inline and optimize code in both approaches as much as possible – neither approach clearly outperforms the other. We have discussed the reasons for this, and indeed, when considered closely how each approach fundamentally works, it should seem rather surprising if either approach dominated the other performance-wise.

We have also drawn close connections to three fundamental approaches to loop fusion in programming languages – fold, unfold, and stream fusion. As it turns out, there is a close analogy between pull engines and unfold fusion on one hand and push engines and fold fusion on the other.

Finally, we have applied the lessons learned about the weaknesses of either approach and propose a new approach to building query engines which draws its inspiration from stream fusion and combines the individual advantages of pull and push engines, avoiding their weaknesses.

References

Abadi, D., Madden, S. & Ferreira, M. (2006) Integrating compression and execution in column-oriented database systems. In Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data. ACM, pp. 671–682.

Abadi, D. J., Myers, D. S., DeWitt, D. J. & Madden, S. R. (2007) Materialization strategies in a column-oriented DBMS. In Proceedings of the IEEE 23rd International Conference on Data Engineering, ICDE 2007. IEEE, pp. 466–475.

Ahmad, Y. & Koch, C. (2009) DBToaster: A SQL compiler for high-performance delta processing in main-memory databases. *PVLDB* **2**(2), 1566–1569.

Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., Meng, X., Kaftan, T., Franklin, M. J., Ghodsi, A. & Zaharia, M. (2015) Spark SQL: Relational data processing in spark. In Proceedings of the SIGMOD '15. New York, NY, USA: ACM.

Biboudis, A., Palladinos, N., Fourtounis, G. & Smaragdakis, Y. (2015) Streams à la carte: Extensible pipelines with object algebras. In Proceedings of the 29th European Conference on Object-Oriented Programming, p. 591.

Binnig, C., Hildenbrand, S., & Färber, F. (2009) Dictionary-based order-preserving string compression for main memory column stores. In Proceedings of the SIGMOD '09. ACM, pp. 283–296.

Böhm, C. & Berarducci, A. (1985) Automatic synthesis of typed λ-programs on term algebras. *Theor. Comput. Sci.* **39**, 135–154.

Breazu-Tannen, V. & Subrahmanyam, R. (1991) *Logical and Computational Aspects of Programming with Sets/Bags/Lists*. Springer.

Breazu-Tannen, V., Buneman, P. & Wong, L. (1992) *Naturally Embedded Query Languages*. Springer.

Buchlovsky, P. & Thielecke, H. (2006) A type-theoretic reconstruction of the visitor pattern. *Electron. Notes Theor. Comput. Sci.* **155**, 309–329.

Chhugani, J., Nguyen, A. D., Lee, V. W., Macy, W., Hagog, M., Chen, Y.-K., Baransi, A., Kumar, S. & Dubey, P. (2008) Efficient implementation of sorting on multi-core SIMD CPU architecture. *PVLDB* **1**(2), 1313–1324.

Choi, J.-D., Gupta, M., Serrano, M., Sreedhar, V. C. & Midkiff, S. (1999) Escape analysis for java. *ACM SIGPLAN Notices* **34**(10), 1–19.

Coutts, D., Leshchinskiy, R. & Stewart, D. (2007) Stream fusion. From lists to streams to nothing at all. In Proceedings of the ICFP '07.

Crotty, A., Galakatos, A., Dursun, K., Kraska, T., Çetintemel, U. & Zdonik, S. B. (2015) Tupleware: "Big" data, big analytics, small clusters. In Proceedings of the CIDR.

Diaconu, C., Freedman, C., Ismert, E., Larson, P.-A., Mittal, P., Stonecipher, R., Verma, N. & Zwilling, M. (2013) Hekaton: SQL server's memory-optimized OLTP engine. In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, SIGMOD '13. New York, NY, USA: ACM, pp. 1243–1254.

Emir, B., Odersky, M. & Williams, J. (2007) Matching objects with patterns. In Proceedings of the ECOOP'07. Berlin, Heidelberg: Springer-Verlag.

Fegaras, L. & Maier, D. (2000) Optimizing object queries using an effective calculus. *TODS* **25**(4), 457–516.

Gedik, B., Andrade, H., Wu, K.-L., Yu, P. & Doo, M. (2008) SPADE: The system S seclarative stream processing engine. In Proceedings of the SIGMOD.

Gibbons, J. & Oliveira, B. C. d S. (2009) The essence of the iterator pattern. *J. Funct. Program.* **19**(3–4), 377–402.

Gill, A., Launchbury, J. & Peyton Jones, S. L. (1993) A short cut to deforestation. In Proceedings of the FPCA. ACM.

Graefe, G. (1994) Volcano–an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.* **6**(1), 120–135.

Graefe, G. (1993) Query evaluation techniques for large databases. *CSUR* **25**(2), 73–169.

Grust, T. & Scholl, M. (1999) How to comprehend queries functionally. *J. Intell. Inform. Syst.* **12**(2–3), 191–218.

Grust, T., Mayr, M., Rittinger, J. & Schreiber, T. (2009) FERRY: Database-supported program execution. In Proceedings of the SIGMOD 2009. ACM.

Grust, T., Rittinger, J. & Schreiber, T. (2010) Avalanche-safe LINQ compilation. *PVLDB* **3**(1–2), 162–172.

Hellerstein, J. M., Stonebraker, M. & Hamilton, J. (2007) Architecture of a database system. *Found. Trends® Databases* **1**(2), 141–259.

Hinze, R., Harper, T. & James, D. W. H. (2011) Theory and practice of fusion. In Proceedings of the 22Nd International Conference on Implementation and Application of Functional Languages, IFL'10. Berlin, Heidelberg: Springer-Verlag, pp. 19–37.

Hirzel, M., Soulé, R., Schneider, S., Gedik, B. & Grimm, R. (2014) A catalog of stream processing optimizations. *ACM Comput. Surv.* **46**(4), 46:1–46:34.

Hofer, C. & Ostermann, K. (2010) Modular domain-specific language components in scala. In Proceedings of the 9th International Conference on Generative Programming and Component Engineering, GPCE '10. New York, NY, USA: ACM, pp. 83–92.

Hudak, P. (1996) Building domain-specific embedded languages. *ACM Comput. Surv.* **28**(4es), 196.

Idreos, S., Groffen, F., Nes, N., Manegold, S., Mullender, S. K., Kersten, M. L., (2012) MonetDB: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.* **35**(1), 40–45.

Jones, S. L. P., Hall, C., Hammond, K., Partain, W. & Wadler, P. (1993) The glasgow Haskell compiler: A technical overview. In Proceedings of the UK Joint Framework for Information Technology, Technical Conference, vol. 93. Citeseer.

Jonnalagedda, M. & Stucki, S. (2015) Fold-based fusion as a library: A generative programming pearl. In Proceedings of the 6th ACM SIGPLAN Symposium on Scala. ACM, pp. 41–50.

Karpathiotakis, M., Alagiannis, I., Heinis, T, Branco, M. & Ailamaki, A. (2015) Just-in-time data virtualization: Lightweight data management with ViDa. In Proceedings of the CIDR.

Karpathiotakis, M., Alagiannis, I. & Ailamaki, A. (2016) Fast queries over heterogeneous data through engine customization. In Proceedings of the VLDB Endowment **9**(12), 972–983.

Klonatos, Y., Koch, C., Rompf, T. & Chafi, H. (2014a) Building efficient query engines in a high-level language. *PVLDB* **7**(10), 853–864.

Klonatos, Y., Koch, C., Rompf, T. & Chafi, H. (2014b) Errata for "Building efficient query engines in a high-level language" PVLDB 7(10):853-864. *PVLDB* **7**(13), 1784–1784.

Koch, C. (2010) Incremental query evaluation in a ring of databases. In Proceedings of the PODS 2010. ACM.

Koch, C. (2014) Abstraction without regret in database systems building: A manifesto. *IEEE Data Eng. Bull.* **37**(1), 70–79.

Koch, C., Ahmad, Y., Kennedy, O., Nikolic, M., Nötzli, A., Lupei, D. & Shaikhha, A. (2014) DBToaster: Higher-order delta processing for dynamic, frequently fresh views. *Vldbj* **23**(2), 253–278.

Krikellas, K., Viglas, S. & Cintra, M. (2010) Generating code for holistic query evaluation. In Proceedings of the ICDE, pp. 613–624.

Li, Z. & Ross, K. A. (1999) Fast joins using join indices. *VLDB J.* **8**(1), 1–24.

Lorie, R. A. (1974) *XRM: An Extended (N-ary) Relational Memory*. IBM.

Mainland, G., Leshchinskiy, R. & Peyton Jones, S. (2013) Exploiting vector instructions with generalized stream fusion. In Proceedings of the ICFP '13. New York, NY, USA: ACM.

Meijer, E., Beckman, B. & Bierman, G. (2006) LINQ: Reconciling object, relations and XML in the .NET framework. In Proceedings of the SIGMOD '06. ACM.

Murray, D. G., Isard, M. & Yu, Y. (2011) Steno: Automatic optimization of declarative queries. In Proceedings of the PLDI '11. New York, NY, USA: ACM.

Nagel, F., Bierman, G. & Viglas, S. D. (2014) Code generation for efficient query processing in managed runtimes. *PVLDB* **7**(12), 1095–1106.

Neumann, T. (2011) Efficiently compiling efficient query plans for modern hardware. *PVLDB* **4**(9), 539–550.

Padmanabhan, S., Malkemus, T., Jhingran, A. & Agarwal, R. (2001) Block oriented processing of relational database operations in modern computer architectures. In Proceedings of the ICDE, pp. 567–574.

Paredaens, J. & Gucht, D. V. (1988) Possibilities and limitations of using flat operators in nested algebra expressions. In Proceedings of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 21–23, 1988, Austin, Texas, USA, pp. 29–38.

Park, Y., Seo, S., Park, H., Cho, H. K., & Mahlke, S. (2012) SIMD Defragmenter: Efficient ILP realization on data-parallel architectures. In Proceedings of the ACM SIGARCH Computer Architecture News, vol. 40. ACM, pp. 363–374.

Peyton Jones, S., Leshchinskiy, R., Keller, G. & MT Chakravarty, M.. (2008) Harnessing the multicores: Nested data parallelism in Haskell. In Proceedings of the LIPIcs-Leibniz International Proceedings in Informatics, vol. 2. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

Pierce, B. C. (2002) *Types and Programming Languages*. MIT press.

Polychroniou, O., Raghavan, A. & Ross, K. A. (2015) Rethinking SIMD vectorization for in-memory databases. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, SIGMOD '15. New York, NY, USA: ACM, pp. 1493–1508.

Schuh, S., Chen, X. & Dittrich, J. (2016) An experimental comparison of thirteen relational equi-joins in main memory. In Proceedings of the SIGMOD '16. New York, NY, USA: ACM, pp. 1961–1976.

Shaikhha, A., Klonatos, Y. & Koch, C. (2018) Building efficient query engines in a high-level language. *Trans. Database Syst.* **43**(1).

Shaikhha, A., Klonatos, Y., Parreaux, L., Brown, L., Dashti, M. & Koch, C. (2016) How to architect a query compiler. In Proceedings of the SIGMOD'16.

Shivers, O. & Might, M. (2006) Continuations and transducer composition. In Proceedings of the PLDI '06. ACM.

Stonebraker, M., Abadi, D. J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E., O'Neil, P., Rasin, A., Tran, N. & Zdonik, S. (2005) C-store: A column-oriented DBMS. In Proceedings of the VLDB '05. VLDB Endowment.

Svenningsson, J. (2002) Shortcut fusion for accumulating parameters & zip-like Functions. In Proceedings of the ICFP '02. ACM.

Tibbetts, R., Yang, S., MacNeill, R. & Rydzewski, D. (2011) StreamBase LiveView: Push-based real-time analytics. In Proceedings of the StreamBase Systems (Jan 2012).

Transaction Processing Performance Council. (2017) *TPC-H, a Decision Support Benchmark*. `http://www.tpc.org/tpch`.

Trinder, P. (1992) Comprehensions, a query notation for DBPLs. In Proceedings of the 3rd DBPL Workshop, DBPL3. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc, pp. 55–68.

Veldhuizen, T. L. (2014) Leapfrog triejoin: A simple, worst-case optimal join algorithm. In Proceedings of the 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24–28, 2014.

Viglas, S., Bierman, G. M., & Nagel, F. (2014) Processing declarative queries through generating imperative code in managed runtimes. *IEEE Data Eng. Bull.* **37**(1), 12–21.

Vlissides, J., Helm, R., Johnson, R. & Gamma, E. (1995) Design patterns: Elements of reusable object-oriented software. *Reading: Addison-Wesley* **49**(120), 11.

Wadler, P. (1988) Deforestation: Transforming programs to eliminate trees. In Proceedings of the ESOP'88. Springer, pp. 344–358.

Wadler, P. (1990) Comprehending monads. In Proceedings of the 1990 ACM Conference on LISP and Functional Programming, LFP '90. New York, NY, USA: ACM, pp. 61–78.

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., & Stoica, I. (2012) Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In Proceedings of the NSDI'12. USENIX Association.

Zhou, J. & Ross, K. A. (2002) Implementing database operations using SIMD instructions. In Proceedings of the SIGMOD '02. New York, NY, USA: ACM.

Zukowski, M., Boncz, P. A., Nes, N., & Héman, S. (2005) MonetDB/X100 – A DBMS In The CPU Cache. *IEEE Data Eng. Bull.* **28**, 17–22.

Zukowski, M., Heman, S., Nes, N., & Boncz, P. (2006) Super-scalar RAM-CPU cache compression. In Proceedings of the 22nd International Conference on Data Engineering, ICDE '06. Washington, DC, USA: IEEE Computer Society, p. 59.

Appendix A. MicroBenchmark Queries

In this section, we present the corresponding SQL queries for the micro-benchmarks used in Section 7. These queries are presented in Table A1. All these queries are using the `LINEITEM` and `ORDERS` tables of TPC-H.

Table A1. *SQL queries of micro-benchmark queries*

Name	SQL Query
`filter.count`	`SELECT COUNT(*) FROM LINEITEM WHERE L_SHIPDATE >= DATE '1995-12-01'`
`filter.sum`	`SELECT SUM(L_DISCOUNT * L_EXTENDEDPRICE) FROM LINEITEM WHERE L_SHIPDATE >= DATE '1995-12-01'`
`filter.filter.sum`	`SELECT SUM(L_DISCOUNT * L_EXTENDEDPRICE) FROM LINEITEM WHERE (L_SHIPDATE >= DATE '1995-12-01') AND (L_SHIPDATE < DATE '1997-01-01')`
`filter.filter.filter.sum`	`SELECT SUM(L_DISCOUNT * L_EXTENDEDPRICE) FROM LINEITEM WHERE (L_SHIPDATE >= DATE '1995-12-01') AND (L_SHIPDATE < DATE '1997-01-01') AND (L_SHIPMODE = 'MAIL')`
`filter.take.sum`	`SELECT SUM(L_DISCOUNT * L_EXTENDEDPRICE) FROM LINEITEM WHERE L_SHIPDATE >= DATE '1995-12-01' LIMIT 1000`
`filter.map.take`	`SELECT L_DISCOUNT * L_EXTENDEDPRICE FROM LINEITEM WHERE L_SHIPDATE >= DATE '1995-12-01' LIMIT 1000`
`take.sum`	`SELECT SUM(L_DISCOUNT * L_EXTENDEDPRICE) FROM LINEITEM LIMIT 1000`
`filter.XJoin(filter).sum`	`SELECT SUM(O_TOTALPRICE) FROM LINEITEM, ORDERS WHERE O_ORDERDATE >= DATE '1998-11-01' AND L_SHIPDATE >= DATE '1998-11-01' AND O_ORDERKEY = L_ORDERKEY`

Appendix B. Example: Fusion Process

This section demonstrates the transformations applied for performing push- and pull-based loop fusion on the working example. Note that, here, inlining a particular definition is always assumed together with β-reduction (inlining) of the accompanying function values. As an example, in Figure B1, inlining `fold` means that we also inline the function value passed as the input parameter to `fold`.

```
val l1 = fromArray(R)
val l2 = l1.filter(r => r.A < 10)
val l3 = l2.map(r => r.B)
return l3.fold(0.0)((s, r) => s + r)
```

$\downarrow$ (inline fromArray & filter)

```
val l1 = build { k1 =>
  var index = 0
  while(index < R.length) {
    k1(R(i))
  }
}
val l2 = build { k2 =>
  l1.foreach { e1 =>
    if(e1.A < 10)
      k2(e1)
  }
}
val l3 = l2.map(r => r.B)
return l3.fold(0.0)((s, r) => s + r)
```

$\downarrow$ (fuse l1.foreach)

```
val l2 = build { k2 =>
  var index = 0
  while(index < R.length) {
    val e1 = R(i)
    if(e1.A < 10)
      k2(e1)
  }
}
val l3 = l2.map(r => r.B)
return l3.fold(0.0)((s, r) => s + r)
```

$\downarrow$ (inline map)

```
val l2 = build { k2 =>
  var index = 0
  while(index < R.length) {
    val e1 = R(i)
    if(e1.A < 10)
      k2(e1)
  }
}
val l3 = build { k3 =>
  l2.foreach { e2 =>
    k3(e2.B)
  }
}
return l3.fold(0.0)((s, r) => s + r)
```

$\downarrow$ (fuse l2.foreach)

```
val l3 = build { k3 =>
  var index = 0
  while(index < R.length) {
    val e1 = R(i)
    if(e1.A < 10)
      k3(e1.B)
  }
}
return l3.fold(0.0)((s, r) => s + r)
```

$\downarrow$ (inline fold)

```
val l3 = build { k3 =>
  var index = 0
  while(index < R.length) {
    val e1 = R(i)
    if(e1.A < 10)
      k3(e1.B)
  }
}
var sum = 0.0
l3.foreach { e3 =>
  sum = sum + e3
}
return sum
```

$\downarrow$ (fuse l3.foreach)

```
var sum = 0.0
var index = 0
while(index < R.length) {
  val rec = R(index)
  index += 1
  if(rec.A < 10)
    sum += rec.B
}
return sum
```

Fig. B1. Transformations needed for applying fold fusion on the example query.

```
val l1 = fromArray(R)
val l2 = l1.filter(r => r.A < 10)
val l3 = l2.map(r => r.B)
return l3.fold(0.0)((s, r) => s + r)
```

↓ (inline fromArray & filter)

```
var index = 0
val l1 = generate { () =>
  if(index < R.length) {
    val elem = R(index)
    index += 1
    elem
  } else { null }
}
val l2 = l1.destroy { n1 =>
  generate { () =>
    var elem: R = null
    do {
      elem = n1()
    } while (elem!=null && !(elem.A<10))
    elem
  }
}
val l3 = l2.map(r => r.B)
return l3.fold(0.0)((s, r) => s + r)
```

↓ (fuse l1.destroy)

```
var index = 0
val l2 = generate { () =>
  var elem: R = null
  do {
    if(index < R.length) {
      elem = R(index)
      index += 1
    } else { elem = null }
  } while (elem != null && !(elem.A < 10))
  elem
}
val l3 = l2.map(r => r.B)
return l3.fold(0.0)((s, r) => s + r)
```

↓ (inline map)

```
var index = 0
val l2 = generate { () =>
  var elem: R = null
  do {
    if(index < R.length) {
      elem = R(index)
      index += 1
    } else { elem = null }
  } while (elem != null && !(elem.A < 10))
  elem
}
val l3 = l2.destroy { n2 =>
  generate { () =>
    val elem = n2()
    if(elem == null) null
    else elem2.B
  }
}
return l3.fold(0.0)((s, r) => s + r)
```

↓ (fuse l2.destroy)

```
var index = 0
val l3 = generate { () =>
  var elem: R = null
  do {
    if(index < R.length) {
      elem = R(index)
      index += 1
    } else { elem = null }
  } while (elem != null && !(elem.A <
      10))
  if(elem == null) null
  else elem2.B
}
return l3.fold(0.0)((s, r) => s + r)
```

↓ (inline fold)

```
var index = 0
val l3 = generate { () =>
  var elem: R = null
  do {
    if(index < R.length) {
      elem = R(index)
      index += 1
    } else { elem = null }
  } while (elem!=null && !(elem.A<10))
  if(elem == null) null
  else elem2.B
}
return l3.destroy { n3 =>
  var sum = 0.0
  while(true){
    val elem = n3()
    if(elem == null) break
    else sum = sum + elem
  }
  sum
}
```

↓ (fuse l3.destroy & partial evaluation)

```
var sum = 0.0
var index = 0
while(true) {
  var elem: R = null
  do {
    if(index < R.length) {
      elem = R(index)
      index += 1
    } else { elem = null }
  } while (elem != null && !(elem.A <
      10))
  if(elem == null) break
  else sum = sum + elem.B
}
return sum
```

Fig. B2. Transformations needed for applying unfold fusion on the example query.

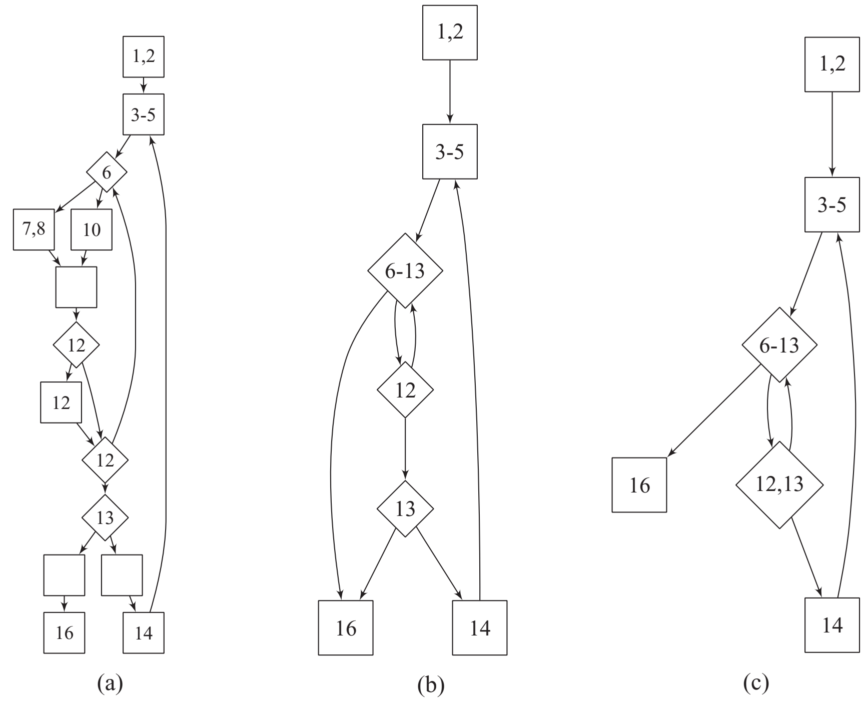


Fig. C1. Control flow graph of the specialized pull-based engine for the `filter.sum` query, compiled with different optimization flags in the CLang compiler. (a) Without any optimization flags. (b) With the memory to reference promotion and CFG simplification optimizations. (c) With the most aggressive optimization flags.

Appendix C. Impact of the Underlying Optimizing Compiler

In Section 2.3, we have seen that the control flow of pull engines is more complicated than push engines. However, a careful examination of the optimized machine code generated by the CLang compiler shows that the optimizing compiler partially compensates this limitation of pull engines.

Figure C1 shows the CFG of the specialized pull-engine for the `filter.map.sum` query. These graphs are obtained by using the `opt –dot–cfg` command for the generated LLVM code, which is generated by compiling C code using `clang –emit–llvm`.

The CFG of the generated machine code when one does not use any optimization is shown in Figure C1(a). This CFG is as complicated as the one shown in Section 2.3. Figure C1(b) shows the CFG of the generated machine code after performing the following two optimizations, which are both enabled by using the `–O1` and `–O3` optimization flags. First, the `–mem2reg` optimization is responsible for promoting memory references to register references. Second, the `–simplifycfg` is responsible for simplifying the CFG. The generated machine code has a much more simplified CFG than the machine code without any optimizations. Finally, Figure C1(c) shows the CFG of the generated machine code by using the `–O1` or `–O3` optimization flags. We

```
class LimitOp[R](n: Int) {
  var i = 0
  def consume(e: R): Unit =
    if(i < n) {
      dest.consume(e)
      i += 1
    }
}
```

(a)

```
class QueryMonad[R] {
  def take(n: Int) = build { k =>
    var i = 0
    for(e <- this)
      if(i < n) {
        k(e)
        i += 1
      }
} }
```

(b)

Fig. D1. Push-based query engine and fold fusion of collections for the Limit operator. (a) Push-based query engine. (b) Fold fusion of collections.

observed that adding the `–jump–threading` optimization flag, which is responsible for further simplification of CFG, to the existing set of optimization flags (`–mem2reg –simplifycfg`) achieves a similar CFG. This CFG is as simple as the CFG of the specialized push engine presented in Section 2.3.

Appendix D. Translating the limit operator

In this section, we investigate in more detail the translation of the limit operator in pull and push-based engines. The implementation of the limit operator for a pull-based engine is presented in Figure 5(a). If the limit threshold was not reached, the limit operator returns the next element of its source operator. Otherwise, if the limit threshold was reached, the limit operator produces a `null` value, specifying that the end of stream is reached. However, in a push-based engine there is no straightforward way for the destination operator to send a signal to the source operator specifying that the limit was reached. Hence, the limit operator is implemented by not passing the element to the destination operator if the limit was reached. This means that the source operator continues producing more elements, even though these elements will be ignored by the subsequent operators (c.f. Figure D1(a)).

Consider a query similar to the `take.sum` presented in Section 7. This query for a given collection of numbers (array of a thousand integers), returns the sum of the first five elements. The corresponding C code for pull- and push-based engines can be found in Figure D2(a) and D2(b), respectively. In a pull-based engine, when the fifth element is reached, no further element is processed thanks to the `break` expression in line 10 of Figure D2(a). However, a push-based engine ignores the elements after the fifth element, without early termination of the loop, as it can be observed in Figure D2(b).

One could argue that a smart enough compiler can compensate the mentioned limitation of a push-based engine. However, examining the generated assembly code by CLang 3.9.1, when the `–O3` optimization flag is used, shows that this claim is not necessarily true. Figure D2(c) demonstrates the generated assembly code for a pull-based engine. This figure shows that the optimizing compiler successfully unrolled the loop to process the sum of the first elements in five assembly instructions.

```
1  int mapTake(int* arr) {
2     const int N = 1000;
3     int res = 0;
4     int cnt = 0;
5     int i = 0;
6     while(1) {
7        if(i < N) {
8          int rec = 0;
9          if(cnt < 5) rec = arr[i];
10         else break;
11         res += rec;
12         cnt++;
13         i++;
14       } else break;
15    }
16    return res;
17 }
```

(a)

```
1  int mapTake(int* arr) {
2     const int N = 1000;
3     int res = 0;
4     int cnt = 0;
5     int i = 0;
6     while(i < N) {
7        if(cnt < 5) {
8           res += arr[i];
9           cnt ++;
10       }
11       i++;
12    }
13    return res;
14 }
```

(b)

```
1  mapTake(int*): # @mapTake(int*)
2     mov eax, dword ptr [rdi + 4]
3     add eax, dword ptr [rdi]
4     add eax, dword ptr [rdi + 8]
5     add eax, dword ptr [rdi + 12]
6     add eax, dword ptr [rdi + 16]
7     ret
```

(c)

```
1  mapTake(int*): # @mapTake(int*)
2     xor ecx, ecx
3     mov edx, 1
4     xor eax, eax
5  .LBB0_1: # =>This Inner Loop Header
6     cmp ecx, 4
7     jg .LBB0_3
8     add eax, dword ptr [rdi+4*rdx-4]
9     inc ecx
10 .LBB0_3: # in Loop: Header=BB0_1
11    cmp ecx, 5
12    jge .LBB0_5
13    add eax, dword ptr [rdi+4*rdx]
14    inc ecx
15 .LBB0_5: # in Loop: Header=BB0_1
16    add rdx, 2
17    cmp rdx, 1001
18    jne .LBB0_1
19    ret
```

(d)

Fig. D2. The generated C and assembly code for a simple query which returns the sum of the first five elements of an array of a thousand elements in pull and push-based engines. (a) C code for pull engine. (b) C code for push engine. (c) Generated assembly code for pull engine. (d) Generated assembly code for push engine.

However, the generated assembly code for a push-based engine is not as elegant as the one for a pull-based engine. The generated assembly code processes the elements of the array two-by-two, however, it does not perform the early termination that is happening in a pull-based engine. The 12th line of Figure D2(d) corresponds to the 7th line of Figure D2(b), which continues iterating the main loop, until *all* elements of the array have been processed (without terminating it early).

Appendix E. Translating the Merge Join Operator

In this section, we investigate in more detail the merge join operator in pull- and push-based engines. The implementation of this operator for these engines is given in Figure E1.

Figure E1(a) presents the implementation of the merge join operator in a pull-based engine. The elements of both relations are iterated in parallel until either the elements of both relations can be joined or one of the relations reaches the end.

```
1  class MergeJoinOp[R, S]
2   (cond: (R, S) => Int) {
3   var rec1: R = null
4   var rec2: S = null
5   var leftProceed = true
6   var rightProceed = true
7   def next(): (R, S) = {
8    var elem: (R, S) = null
9    while(true) {
10     if(leftProceed) rec1 = left.next()
11     if(rightProceed) rec2 = right.next()
12     if(rec1 != null && rec2 != null) {
13      leftProceed = cond(rec1, rec2) < 0
14      rightProceed = !leftProceed
15      if(cond(rec1, rec2) == 0) {
16       elem = rec1.concat(rec2)
17       rightProceed = true
18       break
19     } } else
20      break
21    }
22    return elem
23 } }
```

(a)

```
class MergeJoinOp[R, S]
 (cond: (R, S) => Int) {

 val leftBuf = new ArrayBuffer[R]()
 var leftIndex = 0

 def consumeLeft(e: R): Unit = {
  leftBuf += e
 }

 def consumeRight(e: S): Unit = {
  while(leftIndex < leftBuf.length &&
     cond(leftRel(leftIndex), e)<0) {
   leftIndex += 1
  }
  if(leftIndex < leftBuf.length &&
    cond(leftBuf(leftIndex), e) == 0) {
   val res =
      leftBuf(leftIndex).concat(e)
   dest.consume(res)
  }
 }
}
```

(b)

Fig. E1. Pull and push-based query engines for Merge Join operator. (a) Pull-based query engine for Merge Operator. (b) Push-based query engine for Merge Operator.

The local variables `leftProceed` and `rightProceed` are introduced in order to have only one invocation for the `next` method of the source operators (c.f. lines 10 and 11 of Figure E1(a)). This is in essence similar to the trick we used in the inline-friendly implementation of the Selection operator.

In a push-based engine, as opposed to a pull-based engine, one has to materialize the left relation. This is because there is no way for the destination operator to control which source operator should produce the next element. Hence, the merge join operator materializes the elements of the left source operator, when it is consuming those elements (c.f. line 8 of Figure E1(b)). This way, the merge join operator can control how to consume the (materialized) elements of the left source operator.

Consider the following query, which is similar to the `filter.mJoin(filter).sum` query presented in Section 7:

```
SELECT SUM(R.B * S.B) FROM R, S WHERE R.B > 10 AND R.A = S.A
```

The corresponding generated code for pull and push-based engines is presented in Figure E2. In a pull-based engine the elements of each relation are processed on the fly, without materializing the elements of any of the two relations (c.f. Figure E2(a)). However, in a push-based engine the filtered elements of the left relation are materialized into an intermediate collection (c.f. lines 6-11 of Figure E2(b)). The creation of this intermediate collection justifies the performance gap observed in Section 7 for the micro-benchmark of the merge join operator and TPCH query 12.

```
var sum = 0.0
var i1 = 0; var i2 = 0
var rec1 = null; var rec2 = null
var leftProceed = true
var rightProceed = true
while(true) {
  if(leftProceed) {
    do {
      if(i1 < R.length) {
        rec1 = R(i1)
        i1 += 1
      } else {
        rec1 = null
        break
    } } while (rec1.B <= 10)
  }
  if(rightProceed) {
    if(i2 < S.length) {
      rec2 = S(i2)
      i2 += 1
    } else
      rec2 = null
  }
  if(rec1 != null && rec2 != null) {
    leftProceed = rec1.A < rec2.A
    rightProceed = !leftProceed
    if(rec1.A == rec2.A)
      sum += rec1.B * rec2.B
  } else {
    break
} }
return sum
```

(a)

```
var sum = 0.0
var i1 = 0
var i2 = 0
val leftBuf = new ArrayBuffer[R]()
var leftIndex = 0


// Materialize an intermediate collection
while(i1 < R.length) {
  if(R(i1).B > 10) {
    RB += R(i1)
  }
  i1 += 1
}


// Use the intermediate collection
while(i2 < S.length) {
  val rec2 = S(i2)

  while(leftIndex < leftBuf.length &&
        leftBuf(leftIndex).A < rec2.A) {
    leftIndex += 1
  }

  if(leftIndex < RB.length &&
     leftBuf(leftIndex).A == rec2.A)
    sum += leftBuf(leftIndex).B * rec2.B
  i2 += 1
}

return sum
```

(b)

Fig. E2. Complied version of a query with a merge join operator in pull and push engines. Note that both versions are derived after several optimization passes. (a) Inlined query in pull engine. (b) Inlined query in push engine.

Appendix F. Experimental Results for Query Compilation Time

In this section, we show the query compilation times for the TPC-H queries that we have used in Section 7. The overall query compilation time consists of the time for DBLAB to generate C code, and the time CLang needs for compiling the generated C code into machine code. Figure F1 shows the results for the stream-fusion engine with the visitor model for individual tuples. We do not include the results for other types of engines, as all these engines show a similar behavior when it comes to compilation times. Based on this figure, the overall compilation time never exceeds 1.5 sec, which makes our compilation-based analytical processing system usable in practice, even for complicated, multi-way join queries. Moreover, the compilation time is equally divided between the C code generation time by DBLAB and C code compilation by CLang. We can further reduce the overall compilation time by directly generating LLVM code and invoking the necessary optimization passes manually from the LLVM framework. Furthermore, by porting our query compiler implementation from Scala to C++ and using the LLVM framework to implement transformation passes, the compilation time should improve even more. We leave both of these directions for future work.

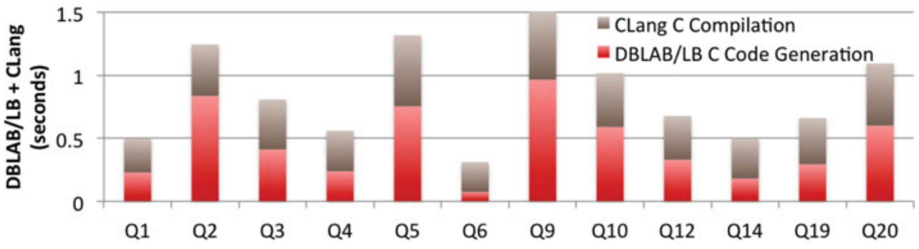


Fig. F1. Compilation time for generated C code of TPC-H queries for the stream-fusion (visitor) engine.