Icicle: Write Once, Run Once

Amos Robinson[αβ] Ben Lippmeier[βγ]

[α]Ambiata (Australia) amos.robinson@ambiata.com
[β]UNSW (Australia) {amosr,benl}@cse.unsw.edu.au
[γ]Vertigo Technology (Australia) benl@vergo.co

Abstract

We present Icicle, a pure streaming query language which statically guarantees that multiple queries over the same input stream are fused. We use a modal type system to ensure that fused queries can be computed incrementalally, and a fold-based intermediate language to compile down to efficient C code. We present production benchmarks demonstrating significant speedup over existing queries written in R, and on par with the Unix tools `grep` and `wc`.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers; Optimization

Keywords Query, fusion, stream

1. Introduction

At Ambiata we perform feature generation for machine learning applications by executing many thousands of simple queries over terabytes worth of compressed data.[1] For such applications we must automatically fuse these separate queries and be sure that the result can be executed in a single pass over the input. We also ingest tens of gigabytes of new data per day, and must incrementally update existing features without recomputing them all from scratch. Our feature generation process is executed in parallel on hundreds of nodes on a cloud based system, and if we performed neither fusion or incremental update then the cost of the computation would begin to exceed the salaries of the developers.

For example queries, suppose we have a table, `stocks`, containing daily open and close prices for a set of companies. We want to compute the number of days where the open price exceeded the close price, and vice versa. We also want the mean of the open price for days in which the open price exceeded the close price. In Icicle we write the three queries as follows:

```
table stocks { open : Int, close : Int }
query
  more = filter open > close of count;
  less = filter open < close of count;
  mean = filter open > close of sum open / count;
```

[1] In 2016 this was a lot of data.

FHPC'16, September 22, 2016, Nara, Japan
ACM. 978-1-4503-4433-3/16/09...
http://dx.doi.org/10.1145/2975991.2975992

In the above code, `open > close` and `close < open` are filter predicates, and `count` counts how many times the predicate is true.

Such a joint query can be converted to a back-end language like SQL, but doing so by hand is tedious and error prone. As the three queries use different filter predicates we cannot use a single `SELECT` statement and a `WHERE` expression to implement the filter. We must instead lift each predicate to an expression-level conditional and compute the count by summing the conditional:

```
SELECT SUM(IF(open > close, 1,    0))
     , SUM(IF(open < close, 1,    0))
     , SUM(IF(open > close, open, 0))
     / SUM(IF(open > close, 1,    0))
FROM stocks;
```

As we see, the result of query fusion tends to have many common sub expressions, and we wish to guarantee that the duplicates in the fused result are eliminated.

Joint queries such as the stocks example can be evaluated in a streaming, incremental fashion, which allows the result to be updated as we receive new data. As a counter example, suppose we have a table with two fields `key` and `value`, and we wish to find the mean of values whose key matches the last one in the table. We might try something like:

```
table kvs { key : Date; value : Real }
query avg = let k = last key
            in  filter (key == k) of mean value;
```

Unfortunately, although the *result* we desire is computable, the *algorithm* implied by the above query cannot be evaluated incrementally. While streaming through the table we always have access to the last key in the stream, but finding the rows that match this key requires streaming the table again from the start. We need a better solution. The contributions of this paper are:

- We present a domain specific language that guarantees any set of queries on a shared input table can be fused, and allows the query results to be updated as new data is received (§3);
- We present a fold-based intermediate language, which allows the query fusion transformation to be a simple matter of appending two intermediate programs (§4);
- We present production benchmarks of Icicle compiled code which outperforms an existing R feature generation system by several orders of magnitude (§5).

Icicle is related to stream processing languages such as Lucy [12] and Streamit [18], except we forgo the need for clock and deadlock analysis. Icicle is also related to work on continuous queries [3], where query results are updated as rows are inserted into the source table, except we can also compute arbitrary reductions and do not need to handle deleted source rows. Our implementation is available at `https://github.com/ambiata/icicle`.

2. Elements and Aggregates

To allow incremental computation all Icicle queries must execute in a single pass over the input stream. Sadly, not all queries *can* be executed in a single pass: the key examples are queries that require random access indexing, or otherwise need to access data in an order different to what the stream provides. However, as we saw in the introduction, although a particular *algorithm* may be impossible to evaluate in a streaming fashion, the desired *value* may well be computable, if only we had a different algorithm. Here is the unstreamable example from the introduction again:

```
table kvs { key : Date; value : Real }
query avg = let k = last key
            in  filter (key == k) of mean value;
```

The problem is that the value of `last key` is only available once we have reached the end of the stream, but `filter` needs this value to process the very first element in the same stream. We distinguish between these two access patterns by giving them different names: we say that `last key` is an *aggregate*, because to compute it we must have consumed the *entire stream*, whereas the filter predicate is an *element*-wise computation because it only needs access to the current element in the stream.

The trick to compute our average in a streaming fashion is to recognize that `filter` is selecting a particular subset of values from the input, but the value computed from this subset depends only on the values in that subset, and no other information. Instead of computing the mean of a single subset whose identity is only known at the end of the stream, we can instead compute the mean of *all possible subsets*, and return the required one once we know what that is:

```
table kvs { key : Date; value : Real }
query avg = let k    = last  key in
            let avgs = group key of mean value
            in  lookup k avgs
```

Here we use the `group` construct to assign key-value pairs to groups as we obtain them, and compute the running mean of the values of each group. The `avgs` value becomes a map of group keys to their running means. Once we reach the end of the stream we will have access to the last key and can lookup the final result. Evaluation and typing rules are defined in §3, while the user functions `last` and `mean` are defined in §4.

2.1 The Stage Restriction

To ensure that Icicle queries can be evaluated in a single pass, we use a modal type system inspired by staged computation [5]. We use two modalities, `Element` and `Aggregate`. Values of type `Element` τ are taken from input stream on a per-element basis, whereas values of type `Aggregate` τ are available only once the entire stream has been consumed. In the expression `(filter (key == k) of mean value)`, the variable `key` has type `Element Date` while `k` has type `Aggregate Date`. Attempting to compile the unstreamable query in Icicle will produce a type error complaining that elements cannot be compared with aggregates.

Note that the types of pure values such as constants are automatically promoted to the required modality. For example, if we have `open == 1` and `open : Element Int` then the constant `1` is automatically promoted to have type `Element Int` as well.

2.2 Finite Streams and Synchronous Data Flow

In contrast to synchronous data flow languages such as LUSTRE [8], the streams processed by Icicle are conceptually finite in length. Icicle is fundamentally a query language, which queries finite tables of data held in a non-volatile store, but does so in a streaming manner. Lustre operates on conceptually infinite streams, such as those found in real-time control systems (like to fly airplanes). In Icicle, the "last" element in a stream is the last one that appears in the table on disk. In Lustre, the "last" element in a stream is the one that was most recently received. If the unstreamable query from §2 was converted to Lustre syntax then it would execute, but the filter predicate would compare the last key with the most recent key from the stream, which is the key itself. The filter predicate would always be true, and the query would return the mean of the entire stream. Applying the Icicle type system to our queries imposes the natural stage restriction associated with finite streams, so there are distinct "during" (element) and "after" (aggregate) stages.

2.3 Incremental Update

Suppose we query a large table and record the result. Tomorrow morning we receive more data and add it to the table. We would like to update the result without needing to process all data from the start of the table. We can do this by remembering the values of all intermediate aggregates that were computed in the query, and updating them as new data arrives. In the `avg` example from §2 these aggregates are `k` and `avgs`.

We also provide impure contextual information to the query such as the current date, by assigning it an aggregate type. As element-wise computations cannot depend on aggregate computations we ensure that reused parts of an incremental computation are the same regardless of which day they are executed.

2.4 Bounded Buffer Restriction

Icicle queries process tables of arbitrary size that may not fit in memory. Due to this, each query must execute without requiring buffer space proportional to the size of the input. As a counter example, here is a simple function which cannot be applied without reserving a buffer of the same size as the input:

```
unbounded (xs : Stream Int)
 = zip (filter (> 0) xs) (filter (< 0) xs)
```

This function takes an input stream `xs`, and pairs the elements that are greater than zero with those that are less than zero. This computation requires an unbounded buffer because if the stream contains n positive values followed by n negative values, then all positive values must be buffered until we reach the negative ones, which allow output to be produced.

In Icicle, queries that would require unbounded buffering are statically outlawed by the typesystem, with one major caveat that we will discuss in a moment. In Icicle, the stream being processed (such as `xs` above) is implicit in each query. Constructs such as `filter` and `fold` do not take the name of the stream as an argument, but instead operate on the stream defined in the context. Icicle language constructs describe *how elements from the stream should be aggregated*, but the order in which those elements are aggregated is implicit, rather than being definable by the body of the query. In the expression `(filter p of mean value)`, the term `mean value` is applied to stream values which satisfy the predicate `p`, but the values to consider are supplied by the context.

Finally, our major caveat is that the `group` construct we used in §2 uses space proportional to the number of distinct *keys* in the input stream. For our applications the keys are commonly company names, customer names, and days of the year. Our production system knows that these types are bounded in size and that maps from keys to values will fit easily in memory. Attempting to group by values of a type with an unbounded number of members, such as a `Real` or `String` results in a compile-time warning.

$$
\begin{array}{lcl}
T & ::= & \texttt{Int} \mid \texttt{Bool} \mid \texttt{Map}\ T\ T \mid (T \times T) \\
M & ::= & T \mid \texttt{Element}\ T \mid \texttt{Aggregate}\ T \\
F & ::= & \overline{(M)} \to M \\
 & & \\
\mathit{Table} & ::= & \texttt{table}\ x\ \texttt{\{}\ \overline{(x : T;)}\ \texttt{\}} \\
 & & \\
\mathit{Exp}, e & ::= & x \mid V \mid \mathit{Prim}\ \overline{\mathit{Exp}} \mid x\ \overline{\mathit{Exp}} \\
 & \mid & \texttt{let}\quad x = \mathit{Exp}\ \ \texttt{in}\ \mathit{Exp} \\
 & \mid & \texttt{fold}\ \ x = \mathit{Exp}\ \texttt{then}\ \mathit{Exp} \\
 & \mid & \texttt{filter}\quad \mathit{Exp}\ \ \texttt{of}\ \mathit{Exp} \\
 & \mid & \texttt{group}\quad \mathit{Exp}\ \ \texttt{of}\ \mathit{Exp} \\
 & & \\
\mathit{Prim}, p & ::= & (\texttt{+}) \mid (\texttt{-}) \mid (\texttt{*}) \mid (\texttt{/}) \mid (\texttt{==}) \mid (\texttt{/=}) \mid (\texttt{<}) \mid (\texttt{>}) \\
 & \mid & \texttt{lookup} \mid \texttt{fst} \mid \texttt{snd} \\
 & & \\
V, v & ::= & \mathbb{N} \mid \mathbb{B} \mid \{V \Rightarrow V\} \mid (V \times V) \\
 & & \\
\mathit{Def} & ::= & \texttt{function}\ f\ \overline{(x : M)} = \mathit{Exp} \\
 & \mid & \texttt{query}\ x = \mathit{Exp} \\
 & & \\
\mathit{Top} & ::= & \mathit{Table};\ \overline{\mathit{Def}}; \\
\end{array}
$$

Figure 1. Icicle Grammar

3. Source Language

The grammar for Icicle is given in Figure 1. Value types T include numbers, booleans and maps. Modal types M include the pure value types, and modalities associated with a value type. Function types F include functions with any number of modal type arguments to a modal return type. As Icicle is a first-order language, function types are not value types.

Table definitions *Table* define a table name and the names and types of columns. Expressions *Exp* include variable names, constants, applications of primitives and functions. The `fold` construct defines the name of an accumulator, the expression for the initial value, and the expression used to update the accumulator for each element of the stream. The `filter` construct defines a predicate and an expression to accumulate values for which the predicate is true. The `group` construct defines an expression used to determine the key for each element of the stream, and an expression to accumulate the values that share a common key.

Prim defines the primitive operators. V defines values. *Def* contains both function and query definitions. *Top* is the top-level program, which specifies a table, the set of function bindings, and the set of queries. All queries in a top-level program process the same table.

3.1 Type System

The typing rules for Icicle are given in Figure 2. The judgment form $\Gamma \vdash e : M$ associates an expression e with its type M under context Γ. The judgment form $p :_P F$ associates a primitive with its function type. The judgment form $F \bullet \overline{M} : M$ is used to lift function application to modal types: a function type applied to a list of modal argument types produces a result type and matching mode. The judgment form $\Gamma \vdash \mathit{Def} \dashv \Gamma$ takes an input environment and function or query, and produces an environment containing the function or query name and its type. Finally, $\vdash \mathit{Top} \dashv \Gamma$ takes a top-level definition with a table, functions and queries, and produces a context containing the types of all the definitions.

Rules TcNat, TcBool, TcMap and TcPair assign types to literal values. Rule TcVar performs variable lookup in the context. Rule TcBox assigns an expression either `Element` or `Aggregate` type.

Rules TcPrimApp and TcFunApp produce the type of a primitive or function applied to its arguments. Rule TcLet is standard.

In rule TcFold the initial value has value type T. The type of the current element from the stream is added to the context of e_k as an `Element`, and the result of the overall fold is an `Aggregate`. Rules TcFilter and TcGroup are similar.

Rules PrimArith, PrimRel, PrimLookup, PrimFst and PrimSnd assign types to primitives. Rule AppArgs produces the type of a function or primitive applied to its arguments. Rule AppRebox is used when the arguments have modal type m — applying a function to arguments of mode m produces a result of the same mode.

Rule CheckFun builds the type of a user defined function, returning it as an element of the output context. Rule CheckQuery is similar, noting that all queries return values of `Aggregate` type. Finally, rule CheckTop checks a whole top-level program.

3.2 Evaluation

Evaluation rules for Icicle are given in Figure 3. Grammar N defines the modes of evaluation, including pure computation. Grammar Σ defines a heap containing stream values. Grammar V' defines the results that can be produced by evaluation, depending on the mode:

- `Pure` computation results are a single value;
- `Element` computation results are stream transformers, which are represented by meta functions that take a value and produces a new value; and
- `Aggregate` computation results consist of an initial (zero) state, an update (konstrukt) meta function to be applied to each stream element and current state, and an eject meta function to be applied to the final state.

In the grammar V' we write $\overset{\bullet}{\to}$ to highlight that the objects in those positions are meta-functions, rather than abstract syntax. To actually process data from the input table we will need to apply the produced meta-functions to this data.

The judgment form $N \mid \Sigma \vdash e \Downarrow V'$ defines a big-step evaluation relation: under evaluation mode N with heap Σ, expression e evaluates to result V'. The evaluation mode N controls whether pure values should be promoted to element (stream) or aggregate (fold) results. We assume that all functions have been inlined into the expression before evaluation.

Rule EVal applies when the expression is already a completed result. Rule EVar performs variable lookup in the heap. Rule ELet evaluates the bound expression under the given mode.

Rules EBoxStream and EBoxFold lift constant values to stream results and aggregate results respectively. To lift a constant to a stream result we produce a meta-function that always returns the value. To lift a constant to an aggregate result we set the update meta-function to return a dummy value, and have the eject meta-function return the value of interest.

Rules EPrimValue, EPrimStream and EPrimFold apply primitive operators to constant values, streams and aggregations respectively. In EPrimStream the result is a new stream transformer that applies the primitive to each of the elements gained from the input streams. In EPrimFold the result consists of new update and eject functions that get their input values by applying the update and eject functions gained by evaluating the arguments.

Rule EFilter first evaluates the predicate e to a stream transformer f, and the body e' to an aggregation. The result is a new aggregation where the update function applies the predicate stream transformer f to the input element s to yield a boolean flag which specifies whether the current aggregation state should be updated.

Rule EGroup is similar to EFilter, except that the stream transformer f produces group keys rather than boolean flags, and we maintain a finite map of aggregation states for each key. In the result

$$\boxed{\Gamma \vdash e : M}$$

$$\frac{}{\Gamma \vdash \mathbb{N} : \texttt{Int}}\;(\text{TcNat}) \qquad \frac{}{\Gamma \vdash \mathbb{B} : \texttt{Bool}}\;(\text{TcBool}) \qquad \frac{\{\Gamma \vdash v_i : T\} \quad \{\Gamma \vdash v'_i : T'\}}{\Gamma \vdash \{v_i \Rightarrow v'_i\} : \texttt{Map}\; T\; T'}\;(\text{TcMap}) \qquad \frac{\Gamma \vdash v : T \quad \Gamma \vdash v' : T'}{\Gamma \vdash v \times v' : T \times T'}\;(\text{TcPair})$$

$$\frac{(x : T) \in \Gamma}{\Gamma \vdash x : T}\;(\text{TcVar}) \qquad \frac{\Gamma \vdash e : T \quad m \in \{\texttt{Element}, \texttt{Aggregate}\}}{\Gamma \vdash e : m\, T}\;(\text{TcBox})$$

$$\frac{p :_P F \quad \{\Gamma \vdash e_i : M_i\} \quad F \bullet \{M_i\} : M'}{\Gamma \vdash p\,\{e_i\} : M'}\;(\text{TcPrimApp}) \qquad \frac{(x : F) \in \Gamma \quad \{\Gamma \vdash e_i : M_i\} \quad F \bullet \{M_i\} : M'}{\Gamma \vdash x\,\{e_i\} : M'}\;(\text{TcFunApp})$$

$$\frac{\Gamma \vdash e : M \quad \Gamma, x{:}M \vdash e' : M'}{\Gamma \vdash \texttt{let}\, x = e\; \texttt{in}\, e' : M'}\;(\text{TcLet}) \qquad \frac{\Gamma \vdash e_z : T \quad \Gamma, x{:}\texttt{Element}\, T \vdash e_k : \texttt{Element}\, T}{\Gamma \vdash \texttt{fold}\, x = e_z\; \texttt{then}\, e_k : \texttt{Aggregate}\, T}\;(\text{TcFold})$$

$$\frac{\Gamma \vdash e : \texttt{Element Bool} \quad \Gamma \vdash e' : \texttt{Aggregate}\, T}{\Gamma \vdash \texttt{filter}\, e\; \texttt{of}\, e' : \texttt{Aggregate}\, T}\;(\text{TcFilter}) \qquad \frac{\Gamma \vdash e : \texttt{Element}\, T \quad \Gamma \vdash e' : \texttt{Aggregate}\, T'}{\Gamma \vdash \texttt{group}\, e\; \texttt{of}\, e' : \texttt{Aggregate}\, (\texttt{Map}\; T\; T')}\;(\text{TcGroup})$$

$$\boxed{p :_P F}$$

$$\frac{p \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}\}}{p :_P (\texttt{Int}, \texttt{Int}) \to \texttt{Int}}\;(\text{PrimArith}) \qquad \frac{p \in \{\texttt{==}, \texttt{/=}, \texttt{<}, \texttt{>}\}}{p :_P (\texttt{Int}, \texttt{Int}) \to \texttt{Bool}}\;(\text{PrimRel})$$

$$\frac{}{\texttt{lookup} :_P (\texttt{Map}\; T\; T', T) \to T'}\;(\text{PrimLookup}) \qquad \frac{}{\texttt{fst} :_P (T \times T') \to T}\;(\text{PrimFst}) \qquad \frac{}{\texttt{snd} :_P (T \times T') \to T'}\;(\text{PrimSnd})$$

$$\boxed{F \bullet \overline{M} : M}$$

$$\frac{}{(\{M_i\} \to M') \bullet \{M_i\} : M'}\;(\text{AppArgs}) \qquad \frac{}{(\{T_i\} \to T') \bullet \{m\, T_i\} : m\, T'}\;(\text{AppRebox})$$

$$\boxed{\Gamma \vdash \mathit{Def} \dashv \Gamma}$$

$$\frac{\Gamma \cup \{x_i : M_i\} \vdash e : M' \quad F = \{M_i\} \to M'}{\Gamma \vdash \texttt{function}\, x\, \{x_i : M_i\} = e \dashv \Gamma, x : F}\;(\text{CheckFun}) \qquad \frac{\Gamma \vdash e : \texttt{Aggregate}\, T}{\Gamma \vdash \texttt{query}\, x = e \dashv \Gamma, x : \texttt{Aggregate}\, T}\;(\text{CheckQuery})$$

$$\boxed{\vdash \mathit{Top} \dashv \Gamma}$$

$$\frac{\Gamma_0 = \{x_i : \texttt{Element}\, T_i\} \quad \{\, \Gamma_{j-1} \vdash d_j \dashv \Gamma_j \,\}}{\vdash \texttt{table}\, x\, \{x_i : T_i\};\ \{\, d_j \,\} \dashv \Gamma_j}\;(\text{CheckTop})$$

Figure 2. Types of expressions

aggregation the update function updates the appropriate element in the map, and the eject function is applied to every accumulator.

Rule EFold introduces a new accumulator which is visible in the context of the body k. Evaluating the body k produces a body stream transformer k' whose job is to update this new accumulator each time it is applied. In the conclusion of EFold we pass this stream transformer a tuple (v,s) where v is the new accumulator and s is the current element of the stream we get from the context of the overall `fold` expression. The heap used when evaluating k is updated so that references to either the stream elements or new accumulator access the appropriate side of the tuple.

The judgment form $t \mid e \Downarrow V$ evaluates an expression over a table input: on input table t, aggregate expression e evaluates to value V. The input table t is a map from column name to a list of all the values for that column. Rule ETable creates an initial heap where each column name x_i is bound to an expression which projects out the appropriate element from a single row in the input table. Evaluating the expression e produces an aggregation result where the update function k accepts each row from the table and updates all the accumulators defined by e. The actual computation is driven by the *fold* meta-function.

4. Intermediate Language

The Icicle intermediate language is similar to a physical query plan for a database system. We convert each source level query to a query plan, then fuse together the plans for queries on the same table. Once we have the fused query plan we then perform standard optimisations such as common subexpression elimination and partial evaluation.

The grammar for the Icicle intermediate language is given in Figure 4. Expressions *PlanX* include variables, values, applications of primitives and anonymous functions. The *Plan* itself is split into a five stage *loop anatomy* [15]. First we have the name of the table and the names and element types of each column. The `before` stage then defines pure values which do not depend on any table data. The `folds` stage defines element computations and how they are converted to aggregate results. The `after` stage defines aggregate computations that combine multiple aggregations after the entire table has been processed. Finally, the `return` stage specifies the output values of the query; a single query will have only one output value, but the result of fusion can have many outputs.

$$\boxed{N \mid \Sigma \vdash e \Downarrow V'}$$

$$\frac{}{n \mid \Sigma \vdash V' \Downarrow V'}\ \text{(EVal)} \qquad \frac{x = V' \in \Sigma}{n \mid \Sigma \vdash x \Downarrow V'}\ \text{(EVar)} \qquad \frac{n' \mid \Sigma \vdash e \Downarrow v \quad n \mid \Sigma, x = v \vdash e' \Downarrow v'}{n \mid \Sigma \vdash \texttt{let}\,(x : n'\,\tau') = e\,\texttt{in}\,e' \Downarrow v'}\ \text{(ELet)}$$

$$\frac{\texttt{Pure} \mid \Sigma \vdash e \Downarrow \texttt{Value}\,v}{\texttt{Element} \mid \Sigma \vdash e \Downarrow \texttt{Stream}\,(\lambda s.\,v)}\ \text{(EBoxStream)} \qquad \frac{\texttt{Pure} \mid \Sigma \vdash e \Downarrow \texttt{Value}\,v}{\texttt{Aggregate} \mid \Sigma \vdash e \Downarrow \texttt{Fold}\,()\,(\lambda s\,().\,())\,(\lambda ().\,v)}\ \text{(EBoxFold)}$$

$$\frac{\{\texttt{Pure} \mid \Sigma \vdash e_i \Downarrow \texttt{Value}\,v_i\}}{\texttt{Pure} \mid \Sigma \vdash p\,\{e_i\} \Downarrow \texttt{Value}\,(p\,\{v_i\})}\ \text{(EPrimValue)} \qquad \frac{\{\texttt{Element} \mid \Sigma \vdash e_i \Downarrow \texttt{Stream}\,v_i\}}{\texttt{Element} \mid \Sigma \vdash p\,\{e_i\} \Downarrow \texttt{Stream}\,(\lambda s.\,p\,\{v_i\,s\})}\ \text{(EPrimStream)}$$

$$\frac{\{\texttt{Aggregate} \mid \Sigma \vdash e_i \Downarrow \texttt{Fold}\,z_i\,k_i\,j_i\}}{\texttt{Aggregate} \mid \Sigma \vdash p\,\{e_i\} \Downarrow \texttt{Fold}\,(z_0 \times \cdots \times z_i)\,(\lambda s\,(v_0 \times \cdots \times v_i).\,k_0\,s\,v_0 \times \cdots \times k_i\,s\,v_i)\,(\lambda (v_0 \times \cdots \times v_i).\,p\,\{j_i\,v_i\})}\ \text{(EPrimFold)}$$

$$\frac{\texttt{Element} \mid \Sigma \vdash e \Downarrow \texttt{Stream}\,f \quad \texttt{Aggregate} \mid \Sigma \vdash e' \Downarrow \texttt{Fold}\,z\,k\,j}{\texttt{Aggregate} \mid \Sigma \vdash \texttt{filter}\,e\,\texttt{of}\,e' \Downarrow \texttt{Fold}\,z\,(\lambda s\,v.\,\texttt{if}\,f\,s\,\texttt{then}\,k\,s\,v\,\texttt{else}\,v)\,j}\ \text{(EFilter)}$$

$$\frac{\texttt{Element} \mid \Sigma \vdash e \Downarrow \texttt{Stream}\,f \quad \texttt{Aggregate} \mid \Sigma \vdash e' \Downarrow \texttt{Fold}\,z\,k\,j}{\texttt{Aggregate} \mid \Sigma \vdash \texttt{group}\,e\,\texttt{of}\,e' \Downarrow \texttt{Fold}\,\{\_ \Rightarrow z\}\,(\lambda s\,m.\,m[f\,s \Rightarrow k\,s\,(m[f\,s])])\,(\lambda m.\,\{k_i \Rightarrow j\,v_i \mid k_i \Rightarrow v_i \in m\})}\ \text{(EGroup)}$$

$$\frac{\texttt{Pure} \mid \Sigma \vdash z \Downarrow \texttt{Value}\,z' \quad \texttt{Element} \mid \{x_i = \texttt{Stream}\,(f_i \cdot \texttt{snd}) \mid x_i = \texttt{Stream}\,f_i \in \Sigma\},\, x = \texttt{Stream}\,\texttt{fst},\, \Sigma \vdash k \Downarrow \texttt{Stream}\,k'}{\texttt{Aggregate} \mid \Sigma \vdash \texttt{fold}\,x = z\,\texttt{then}\,k \Downarrow \texttt{Fold}\,z'\,(\lambda s\,v.\,k'\,(v,\,s))\,(\lambda v.\,v)}\ \text{(EFold)}$$

$$\boxed{\{x \Rightarrow \overline{V}\} \mid e \Downarrow V}$$

$$\frac{\texttt{Aggregate} \mid \{x_i = \texttt{Stream}\,(\texttt{fst} \cdot \texttt{snd}^i) \mid x_i \Rightarrow v_i \in t\} \vdash e \Downarrow \texttt{Fold}\,z\,k\,j}{t \mid e \Downarrow j\,(\mathit{fold}\,k\,z\,\{v_0 \times \cdots \times v_i \times () \mid x_i \Rightarrow v_i \in t\})}\ \text{(ETable)}$$

$$V' ::= \texttt{Value}\,V \mid \texttt{Stream}\,(V \overset{\bullet}{\to} V) \mid \texttt{Fold}\,V\,(V \overset{\bullet}{\to} V \overset{\bullet}{\to} V)\,(V \overset{\bullet}{\to} V)$$

$$N ::= \texttt{Pure} \mid \texttt{Element} \mid \texttt{Aggregate} \qquad \Sigma ::= \cdot \mid \Sigma,\, x = V'$$

Figure 3. Evaluation rules

$$\begin{array}{lll}
\mathit{PlanX} & ::= & x \mid V \mid \mathit{PlanP}\ \overline{\mathit{PlanX}} \mid \lambda x.\ \mathit{PlanX} \\
\mathit{PlanP} & ::= & \mathit{Prim} \mid \texttt{mapUpdate} \mid \texttt{mapEmpty} \mid \texttt{mapMap} \mid \texttt{mapZip} \\
 & & \\
\mathit{Plan} & ::= & \texttt{plan}\,x\ \ \{\ \overline{x : T;}\ \} \\
 & & \texttt{before}\ \{\ \overline{x : T = \mathit{PlanX};}\ \} \\
 & & \texttt{folds}\ \ \{\ \overline{x : T = \mathit{PlanX}\ \texttt{then}\ \mathit{PlanX};}\ \} \\
 & & \texttt{after}\ \ \{\ \overline{x : T = \mathit{PlanX};}\ \} \\
 & & \texttt{return}\ \{\ \overline{x : T = x;}\ \}
\end{array}$$

Figure 4. Query Plan Grammar

Before we discuss an example query plan we first define the `count`, `sum`, `mean` and `last` functions used in earlier sections. Both `count` and `sum` are simple folds:

```
function count
 = fold c = 0 then c + 1;

function sum (e : Element Real)
 = fold s = 0 then s + e;
```

The `mean` function then divides the `sum` by the `count`.

```
function mean (e : Element Real)
 = sum e / count;
```

The `last` function uses a `fold` that initializes the accumulator to the empty date value `NO_DATE`[2], then updates it with the date gained from the current element in the stream.

```
function last (d : Element Date)
 = fold l = NO_DATE then d;
```

Inlining the above functions into the query from §2 yields the following:

```
query avg
 =    let lst = (fold l = NO_DATE then key)
   in let map = group key of
                ( (fold s = 0 then s + value)
                / (fold c = 0 then c + 1) )
   in let ret = lookup lst map
   in     ret
```

To convert this source query to a plan in the intermediate language we convert each of the let-bindings separately then simply concatenate the corresponding parts of the loop anatomy. The `lst` binding becomes a single fold, initialized to `NO_DATE` and updated with the current `key`.

```
plan kvs { key : Date; value : Real;      }
folds    { fL  : Date = NO_DATE then key  }
after    { lst : Date = fL                }
```

[2] In our production compiler, `last` returns a `Maybe`.

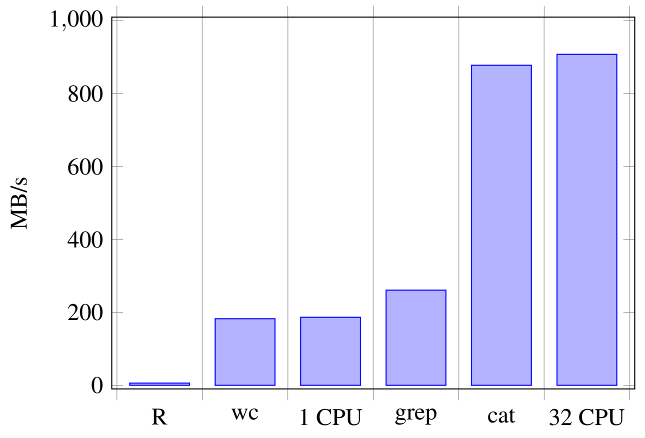

Figure 5. Throughput comparisons of Icicle (1 CPU and 32 CPU) against existing R code and standard Unix utilities; higher is faster.

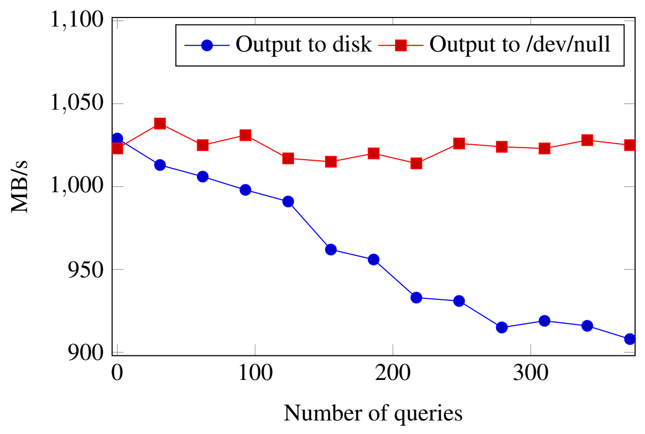

Figure 6. Decrease in read throughput as queries are added, comparing writing the output to disk and writing to /dev/null.

For the `map` binding, each fold accumulator inside the body of the `group` construct is associated with its own finite map. The `s` accumulator is associated with map `gS`, and the `c` accumulator with `gC`. Each time we receive a row from the table the accumulator associated with the `key` is updated, using the default value `0` if an entry for that key is not yet present. After we have processed the entire table we divide each sum with its corresponding count to yield a map of means for each key.

```
folds  { gS  : Map Date Real = mapEmpty
           then mapUpdate gS key 0 (λs. s + value)

       ; gC  : Map Date Real = mapEmpty
           then mapUpdate gC key 0 (λc. c + 1) }

after  { map : Map Date Real
            = mapMap (λs c. s / c) (mapZip gS gC) }
```

Finally, the `ret` binding from the original query is evaluated in the `after` stage. In the `return` stage we specify that the result of the overall query `avg` is the result of the `ret` binding.

```
after  { ret : Real = lookup 1st map }
return { avg : Real = ret }
```

To combine the plans from each binding we simply concatenate the corresponding parts of the anatomy. To fuse multiple plans we freshen the names of each binding and also concatenate the corresponding parts of the anatomy. The single-pass restriction on queries makes the fusion process so simple, because it ensures that there are no fusion-preventing dependencies between any two query plans.

Given a fused query plan we then convert it to an imperative loop nest in a similar way to our prior work on flow fusion [10].

5. Benchmarks

At Ambiata we are currently using Icicle in production over medium-sized data sets that fit on a single disk. We are also currently implementing a scheduler to distribute larger data sets across multiple nodes. The data we are working with is tens of terabytes compressed, which in 2016 does not fit on a single disk. However, each row has a natural primary key and the features we need to compute depend only on the data within single key groups, which makes the workload very easy to distribute.

In our proof of concept testing we replaced an existing R script that performed feature generation with new Icicle code. The R script computed features from a 317GB data set. It computed 12 queries over each of 31 input tables, for 372 query evaluations in total. The R script took 15 hours to run and consisted of 3,566 lines of code. The replacement Icicle version is only 191 lines of code and takes seven minutes to run.

The table in Figure 5 shows the throughput in megabytes per second. We compared the throughput of several programs over the same dataset:

- our original R implementation (R);
- Icicle running single-threaded (1 CPU);
- Icicle running on multiple processors (32 CPU);
- finding empty lines with `grep "^$"`;
- counting characters, words and lines with `wc`;
- reading and throwing away the results with `cat > /dev/null`.

We ran all the Unix utilities with unicode decoding disabled using `LANG=C LC_COLLATE` for maximum performance. The input data does not contain unicode characters. We used an Amazon EC2 `c3.8xlarge` with 32 CPUs, 60GB of RAM, and striped, RAIDed SSD storage. The fused Icicle version significantly outperformed the R version of the queries, and the single-threaded version was on par with `wc`, while only a little slower than `grep`. This is despite the fact that the Icicle queries perform more computational work than `wc` and `grep`. By using multiple processors, we were able to scale up to perform as well as `cat`, approaching the disk speed. The memory usage of Icicle starts at around 200MB of RAM for a single thread, but as more threads are added approaches 15MB per thread. The memory usage is constant in the input size and depends on the number of queries. The R code is single threaded and would require at least 150 processors to reach similar speeds, assuming perfect scaling. These results give us confidence that our distributed implementation will be fast as well as scalable [13].

Figure 6 shows how the total read througput scales as the number of fused queries is increased. For each number of queries, we ran two versions of the fused result: one version that wrote the output to disk, and the other that piped the result to `/dev/null`. The graph shows the throughput of the disk version decreasing roughly linearly in the number of queries, while the version ignoring the output remains constant. This suggests that we are IO bound on the write side. The time spent evaluating the queries themselves is small relative to our current IO load, and we are curently scaling up our system to use a larger query set.

6. Related Work

In Icicle there is only one stream, sourced from the input table, which is implicit in the bodies of queries. This approach is intentionally simpler than existing synchronous data flow languages such as Lucy [12], as well as our prior work on flow fusion [10]. Synchronous data flow languages implement Kahn networks [19] that are restricted to use bounded buffering [9] by clock typing and causal analysis [17]. In such languages, stream combinators with multiple inputs, such as `zip`, are assigned types that require their stream arguments to have the same clock — meaning that elements always arrive in lockstep and the combinators themselves do not need to perform their own buffering. In Icicle the fact that the input stream is implicit and distributed to all combinators means that we can forgo clock analysis. All queries in a program execute in lockstep on the same element at the same moment, which ensures that fusion is a simple matter of concatenating the components of the loop anatomy of each query.

Short-cut fusion techniques such as foldr/build [6] and stream fusion [4] rely on inlining to expose fusion opportunities. In Haskell compilers such as GHC, the decision of when to inline is made by internal compiler heuristics, which makes it difficult for the programmer to predict when fusion will occur. In this environment, array fusion is considered a “bonus” optimization rather than integral part of the compilation method. In contrast, for our feature generation application we really must ensure that multiple queries over the same table are fused, so we cannot rely on heuristics.

StreamIt [18] is an imperative streaming language which has been extended with dynamic scheduling [16]. Dynamic scheduling handles data flow graphs where the transfer rate between different stream operators is not known at compile time. Dynamic scheduling is trade-off: it is required for stream operators such as grouping and filtering where the output data rate is not known statically, but using dynamic techniques for graphs with static transfer rates tends to have a performance cost. Icicle includes grouping and filtering operators where the output rates are statically unknown, however the associated language constructs require grouped and filtered data to be aggregated rather than passed as the input to another stream operator. This allows Icicle to retain fully static scheduling, so the compiled queries consist of straight line code with no buffering.

Icicle is closely related to work in continuous and shared queries. A continuous query is one that processes input data which may have new records added or removed from it at any time. The result of the continuous query must be updated as soon as the input data changes. Shared queries are ones in which the same sub expressions occur in several individual queries over the same data, and we wish to share the results of these sub expressions among all individuals that use them. For example, in Munagala *et al* [14], input records are filtered by a conjunction of predicates, and the predicates occur in multiple queries. Madden *et al* [11] uses a predicate index to avoid recomputing them. Andrade *et al* describes a compiler for queries over geospacial imagery [1] that shares the results of several pre-defined aggregation functions between queries. Continuous Query Language (CQL) [2, 7] again allows aggregates in its queries, but they must be builtin aggregate functions. Icicle addresses a computationally similar problem, except that our input data sets can only have new records added rather than deleted, which allows us to support general aggregations rather than just filter predicates. It is not obvious how arbitrary aggregate functions could be supported while also allowing deletion of records from the input data — other than by recomputing the entire aggregation after each deletion.

Acknowledgements

We are grateful to the entire team at Ambiata; in particular Tran Ma and Jacob Stanley have been indispensable in compiler engineering and performance optimisations.

References

[1] Henrique Andrade, Suresh Aryangat, Tahsin Kurc, Joel Saltz, and Alan Sussman. Efficient execution of multi-query data analysis batches using compiler optimization strategies. In *Languages and Compilers for Parallel Computing*. 2003.

[2] Arvind Arasu, Shivnath Babu, and Jennifer Widom. An abstract semantics and concrete language for continuous queries over streams and relations. Technical report, Stanford InfoLab, 2002.

[3] Arvind Arasu, Shivnath Babu, and Jennifer Widom. CQL: A language for continuous queries over streams and relations. In *Database Programming Languages*, 2003.

[4] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: From lists to streams to nothing at all. In *ACM SIGPLAN Notices*, 2007.

[5] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *Journal of the ACM*, 2001.

[6] Andrew Gill, John Launchbury, and Simon L Peyton Jones. A short cut to deforestation. In *Proceedings of the conference on Functional programming languages and computer architecture*, 1993.

[7] STREAM Group et al. Stream: The Stanford stream data manager. *IEEE Data Engineering Bulletin*, 2003.

[8] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language Lustre. *Proceedings of the IEEE*, 1991.

[9] Wesley M Johnston, JR Hanna, and Richard J Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 2004.

[10] Ben Lippmeier, Manuel MT Chakravarty, Gabriele Keller, and Amos Robinson. Data flow fusion with series expressions in Haskell. In *ACM SIGPLAN Notices*, 2013.

[11] Samuel Madden, Mehul Shah, Joseph M Hellerstein, and Vijayshankar Raman. Continuously adaptive continuous queries over streams. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, 2002.

[12] Louis Mandel, Florence Plateau, and Marc Pouzet. Lucy-n: a n-synchronous extension of Lustre. In *Mathematics of Program Construction*, 2010.

[13] Frank McSherry, Michael Isard, and Derek G Murray. Scalability! But at what COST? In *Hot Topics in Operating Systems*, 2015.

[14] Kamesh Munagala, Utkarsh Srivastava, and Jennifer Widom. Optimization of continuous queries with shared expensive filters. In *Principles of database systems*, 2007.

[15] Olin Shivers. The anatomy of a loop: A story of scope and control. *SIGPLAN Notices*, 40(9), 2005.

[16] Robert Soule, Michael I. Gordon, Saman Amarasinghe, Robert Grimm, and Martin Hirzel. Dynamic expressivity with static optimization for streaming languages. In *The 7th ACM International Conference on Distributed Event-Based Systems*, June 2013.

[17] Robert Stephens. A survey of stream processing. *Acta Informatica*, 1997.

[18] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A language for streaming applications. In *Compiler Construction*, 2002.

[19] Zeljko Vrba, Pål Halvorsen, Carsten Griwodz, and Paul Beskow. Kahn process networks are a flexible alternative to MapReduce. In *High Performance Computing and Communications*, 2009.