# Arc: An IR for Batch and Stream Programming

Lars Kroll
KTH Royal Institute of Technology
Stockholm, Sweden
lkroll@kth.se

Klas Segeljakt
KTH Royal Institute of Technology
Stockholm, Sweden
klasseg@kth.se

Paris Carbone
RISE SICS
Stockholm, Sweden
paris.carbone@ri.se

Christian Schulte
KTH Royal Institute of Technology
Stockholm, Sweden
cschulte@kth.se

Seif Haridi*
KTH Royal Institute of Technology
Stockholm, Sweden
haridi@kth.se

## Abstract

In big data analytics, there is currently a large number of data programming models and their respective frontends such as relational tables, graphs, tensors, and streams. This has lead to a plethora of runtimes that typically focus on the efficient execution of just a single frontend. This fragmentation manifests itself today by highly complex pipelines that bundle multiple runtimes to support the necessary models. Hence, joint optimization and execution of such pipelines across these frontend-bound runtimes is infeasible. We propose Arc as the first unified Intermediate Representation (IR) for data analytics that incorporates stream semantics based on a modern specification of streams, windows and stream aggregation, to combine batch and stream computation models. Arc extends Weld, an IR for batch computation and adds support for partitioned, out-of-order stream and window operators which are the most fundamental building blocks in contemporary data streaming.

***CCS Concepts*** • **Software and its engineering → Context specific languages**; *Compilers*; • **Computing methodologies** → Distributed computing methodologies.

***Keywords*** stream processing, intermediate representation, data analytics

**ACM Reference Format:**
Lars Kroll, Klas Segeljakt, Paris Carbone, Christian Schulte, and Seif Haridi. 2019. Arc: An IR for Batch and Stream Programming. In *Proceedings of the 17th ACM SIGPLAN International Symposium on Database Programming Languages (DBPL '19), June 23, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3315507.3330199

*Also with RISE SICS.



*DBPL '19, June 23, 2019, Phoenix, AZ, USA*

ACM ISBN 978-1-4503-6718-9/19/06.
https://doi.org/10.1145/3315507.3330199

## 1 Introduction

Data-driven programming has gained critical traction in the past decade resulting in a large family of data programming models or "frontends" each of which addresses a very specific class of operations (e.g., CQL [3], Tensors [1], Dataflow [2]) as well as scalable runtimes that employ and optimise distributed execution over a single frontend (e.g., Spark [22], TensorFlow [1], Flink [5]). Modern pipelines typically require several frontends to express a complete end-to-end logic for continuous applications and services. For example, a typical recommendation pipeline contains ML feature selection, data cleaning, training and serving. Hence, data engineers have to bundle together code for multiple frontends, and set up multiple runtimes to execute each respective part efficiently. The end result is often highly complex and inefficient since functions do not share computational resources or hardware (e.g., GPUs) across systems. Moreover, all intermediate results have to be materialized across runtime boundaries, leading to heavy yet unnecessary IO overhead.

This paper contributes a new Intermediate Representation (IR) specifically designed to combine data-centric operations on stream and batch analytics, using a common translation layer. This enables efficient cross-platform compilation, optimization, and execution on shared hardware.

Several existing approaches in database and ML optimization have considered the use of IRs for sharing execution strategies and avoiding materialization, such as Halide [16] for ML and image processing, River [17] for Dataflow programming, and more recently Weld [13] which targets the broader spectrum of batch data analytics. Weld is the most general approach so far to express more than a single data programming frontend. In addition, our IR, Arc, attempts to support continuous and scalable operations on streams.

Data stream processing frameworks such as Apache Beam/-Google Dataflow [2], Flink [5] and Heron [9] can scale today to very large persistent state and offer strong consistency guarantees, making them a very attractive choice for executing the crucial parts of a data pipeline (e.g., ETL, model building, feature selection, model serving, anomaly detection etc.). At the same time, data stream models are known for core differences and peculiarities compared to other data

```
1 |input: Vec[i32]|
2   result(for(input, Appender[i32], |app, element, index|
3     merge(app, element + 5)))
```

**Listing 1.** A simple add-5-mapper function in Weld.

processing models. We identify three of the most distinct semantics across data stream frontends: out-of-order processing [2, 10, 11, 18], window discretization [3, 14] and sliding window aggregation [4, 19]. Despite the ongoing standardization and unification efforts in data streaming, no existing intermediate representation to date provides support for these semantics. IBM's River [17] is the only known data streaming IR, however, it falls short of supporting out-of-order processing or any other frontend whatsoever.

In this paper we present the fundamentals of Arc, a generalization of the Weld IR which supports the expression and compilation of long-running stream operators. The contributions of this work are twofold: We first distill the underlying principles of data streaming, namely unordered streams, window discretization and aggregation. In addition, we present an IR that can incorporate these semantics and allow the generation of static stream operator graphs and further open up the prospect of truly unified future runtimes for hardware-accelerated continuous batch and stream analytics.

## 2 Preliminaries: Weld IR Model

Most models for data programming typically involve different types of side-effect free transformations on data collections and are adequately covered by prior work on Weld [12, 13] among others. The purpose of Arc is to complement Weld with data stream semantics. To clarify this relationship we summarize the concepts provided by Weld in this section.

Weld has been designed to improve the performance of data analytics applications that rely on multiple libraries. Libraries are optimized internally, but typically not with respect to each other. In consequence, pipelining functions of different libraries may require materialization of intermediate results, causing expensive data movement. Weld confronts this problem by introducing an IR which acts as a common ground for different computations, allowing optimizations across function boundaries.

The Weld IR expresses side-effect free transformations over finite-sized data. The IR's programming model is based on monadic comprehensions [7] to enable data parallelism and vectorization, while remaining hardware agnostic. A sketch of relevant syntactical constructs of Weld can be seen in listing 2, which also contains the new elements added by Arc specially highlighted.[1] Data types in Weld are either *value types* or *builder types* (lines 7 and 12 in listing 2). Value types are read-only data types, in the form of scalars, simd types, and collections. Builder types are linear [15] write-only data types. Values of a builder's *merge type* may be

[1] All types in Weld are lower case, while Arc prefers them to be upper case.

```
1  program     ::= { declaration } lambda
2  declaration ::= macro id ( { id , } ) = expr ;
3                | type id = type ; // Type alias
4                | fn id | { type , } | ( type ) = lambda ;
5  lambda      ::= | { id : type , } | expr
6  type        ::= id | valueType | builderType | struct type
7  valueType   ::= Unit | bool | i8 | i16 |...
8                | Simd [ type ]
9                | Vec [ type ]
10               | Dict [ type , type ]
11               | Stream [ type ]
12 builderType ::= Appender [ type ]
13               | Merger [ type , binop ]
14               | StreamAppender [ type ]
15               | Windower [ type , type ]
16               |...
17 struct type ::= { { type , } }
18 expr         ::= opExpr | letExpr
19 opExpr      ::= ( expr )
20               | id
21               | literal
22               | type ( expr ) // Type cast
23               | for ( iterator , expr , lambda )
24               | merge ( expr , expr )
25               | result ( expr )
26               | if ( expr , expr , expr )
27               | cudf [ id , type ] ( { expr , } )
28               | drain ( expr , expr )
29               | builderConstr
30               | opExpr binop opExpr
31               |...
32 letExpr      ::= let id : type = opExpr ; expr
33 binop        ::= + | - | * | / |...
34               | id
35 literal      ::= scalarLiteral
36               | [ { expr , } ] // Vec literal
37               | { { expr , } } // Struct literal
38               | () // Unit literal
39 iterator     ::= expr | iter ( expr , expr , expr )
40               | next ( expr )
41               | keyby ( expr , lambda )
42               |...
43 builderConstr ::= Appender [ type ]
44               | Merger [ type , binop ]
45               | StreamAppender [ type ]
46               | Windower [ type , type ] ( lambda , lambda,
                     lambda )
47               |...
```

**Listing 2.** Sketch of Arc's syntax in EBNF. Elements added to Weld by Arc are highlighted by grey background.

written to it with a `merge` operation (line 24) and values of a builder's *result type* are materialized from it through a **result** operation (line 25). As builders are linear, they may only be used once. While the `merge` operation returns a new builder, the **result** operation simply consumes it. In addition to the types described above, Weld also offers a structural product type called *struct* (line 17). For iteration, Weld provides parallel for-loops (line 23), over collections or ranges, into builders which must have certain properties. Listing 1 gives an example implementation for a Weld function which receives as input a vector of integers and returns the result of adding the value 5 to each element. The for-loop in the example takes as input a collection, a builder, and a lambda expression. The collection is iterated over, and each element is passed along with the builder as input to the lambda expression, which merges the element, with value-5 added, into the builder. Note that the `merge` invocation returns a new builder, which is

```
|source: Stream[i32], sink: StreamAppender[i32]|
  for(source, sink, |sink, element|
    merge(sink, element + 5))
```

**Listing 3.** A simple add-5-mapper function in Arc.

```
|source: Stream[i32],
 odd_sink: StreamAppender[i32],
 even_sink: StreamAppender[i32]|
  let mapped = result(
    for(source, StreamAppender[i32], |sink, element|
        merge(sink, element + 5)));
  for(mapped,  odd_sink, |sink, e|
    if(e % 2 != 0, merge(sink, e), sink));
  for(mapped, even_sink, |sink, e|
    if(e % 2 == 0, merge(sink, e), sink))
```

**Listing 4.** An odd-even filter function in Arc.

passed to the next iteration of the loop. Moreover, the last iteration returns the builder from the for-loop. The final value is given by calling the result operation on the **Appender** returned from the for-loop, which materializes it into a vector of integers.

## 3 Arc IR Model

We present a language called Arc that extends Weld for streaming. Whereas Weld's builders construct values, Arc adds builders to describe streaming pipelines of operators with sources and sinks, connected by channels. An important observation is that stream sources are naturally read-only, and stream sinks are write-only, making them analogous in this regard to Weld's collections and builders respectively. Following this idea, Arc introduces *streams* (**Stream**) and *stream builders* (**StreamAppender**) as two new data types (lines 11 and 14 in the EBNF grammar of listing 2). A stream operator is created by using a parallel for-loop over a **Stream** and using a **StreamAppender** in the builder position (line 45 in listing 2). A simple example of a map operation in Arc, which adds the value 5 to each element in a stream of integers, is shown in listing 3. The input `source` and output `sink` are passed from the outside and information about physical sources and sinks is treated as metadata to the application. Such metadata could be partition identifiers of Kafka topic sinks and sources, for example. We use the syntactic structure of a Weld for-loop to describe a dataflow operator that maps every input value – or *record* – `element` to `element+5` and emits it to the output `sink` by merging.

Listing 4 illustrates another aspect of Arc, where we extend the mapping example from listing 3 with two for-loops, which are combined with if-expressions to filter odd and even numbers from an input source into separate sinks. In both for loops, the current element being iterated over is only merged into the respective sink if the condition of the if-expression evaluates to true. Consequently, the branches of the if-expressions return a sink which is passed to the for loop's next iteration. Calling **result** on the mapping for-loop, which itself returns a **StreamAppender[i32]** sink, creates a new **Stream**, i.e. a source, and instructs an implementation to create a channel between the sink and the new source. Thus, multiple operators can be connected to form a pipeline. As **Stream**s are immutable collections, they can be shared by multiple operators in a pipeline, with a channel for each connection, while **StreamAppender**s are linear and may thus not be shared.

### 3.1 Windowed Streams

In addition to the streaming operators described above, Arc also provides a mechanism to work with finite subranges, or *windows*, over a stream. In order facilitate this mechanism we introduce another new builder type, called a **Windower** (line 15 in listing 2). This builder type is the only "higher-order" builder type in Arc, as in addition to having a *merge* and *result* type, as a regular builder would, a **Windower** has two new state types $D$ and $A$, for *discretization* and *aggregation* respectively, where $A$ must be another builder type. Both types and functions that operate on them are explained in this section.

In order to describe which records fall into any given window $w$, it is necessary to define an *assigner* function, which we call $\vartheta_{assign}$. In listing 5 line 12, for example, $\vartheta_{assign}$ takes every record in a ten minute window and maps it to the same window id. In general, it is also possible to assign a record to multiple windows, thereby describing overlapping windows. In Arc, assigners are given the set of currently open windows, as well as the current instance of an arbitrary state type $D$. Both may be used to make assignment decisions based on (some function over) the already observed prefix of the stream.

While $\vartheta_{assign}$ would be sufficient to describe stream discretization in a model where we can observe the full stream before having to describe the resulting windowed stream, in reality we must be able to decide when a window is complete – we have assigned it all the values we will ever assign to it – within a finite amount of time (or number of records). If the stream is in-order this can be made decision based on records alone. However, in the more general case were records can arrive out-of-order, an additional *control event* is needed to ascertain window completion. We call such an event $c(t)$ with a timestamp $t$ a *low watermark* (or just *watermark* if the context is clear), if it asserts that no record with a lower timestamp than $t$ will arrive after it. Based on this, we define a *trigger* function $\vartheta_{trigger}$, that for each watermark $c(t)$ marks all windows ending in records with timestamp $t_r < t$ as "closed". For example, in listing 5 line 13, the $\vartheta_{trigger}$ returns all open windows with ids smaller than the watermark timestamp (scaled to ten minutes). Together $\vartheta_{assign}$ and $\vartheta_{trigger}$ describe the discretization of a stream into a windowed stream and both share the same state type $D$, as described above. $\vartheta_{assign}$ and $\vartheta_{trigger}$ are also the first two arguments that must be provided to instantiate a **Windower**, as seen in listing 5 lines 12,13.

```
1  type Timestamp, WindowId, ItemId, Watermark = u64;
2  type Price = u32;
3  type Item = { ItemId, Price };
4  type Element = { Timestamp, Element };
5  type MaxAggregator = Merger[Item, max_by_price];
6
7  fn max_by_price |Item, Item|(Item) = |a, b| if(a.$1 > b.$1, a, b);
8
9  |source: Stream[Element], sink: StreamAppender[Element]|
10   let windowed_stream = for(source,
11     Windower[Unit, MaxAggregator](
12       |e: Element, _, _| {[(e.$0)/600L], ()},
13       |wm: Watermark, open: Vec[WindowId], _| {filter(open, |id: WindowId| id < wm/600L), ()},
14       |id: WindowId, window: MaxAggregator| {id, result(window)}),
15     |e: Element, w: Windower[Unit, MaxAggregator]| merge(w, e));
16   drain(windowed_stream, sink)
```

**Listing 5.** NEXMark Query 7 in Arc.

At this point we could emit a (timestamp) ordered list (**Vec**) of all records within a window to the downstream operators. However, in most applications it is not this list that is desired, but the result of applying some *aggregation* function $\alpha : Vec(R_{in}) \rightarrow R_{out}$ to it. For example, in listing 5 we are only interested in the value with the highest price. Clearly, in such a scenario we would only want to maintain the highest value observed so far (and its price) as the window state, not all records in the window, which we are simply going to discard after closing the window. Generalizing this example, we want to maintain only an *aggregation state A*. To do so, we must define functions $\lambda_{\uparrow}, \bullet_A, \lambda_{\downarrow}$ with the following properties: $\lambda_{\uparrow}$ converts – we say "lifts" – a record of the input type $R_{in}$ to the aggregation type $A$. $\bullet_A$ is a binary operation on $A$, that combines new values with the current aggregation state. Since in Weld (and thus in Arc) only builders allow this kind of behaviour, $A$ must be a builder type. Depending on the concrete properties of $\bullet_A$ (e.g., associativity, commutativity) a compiler based on Arc will generate different implementations of the **Windower** and actual state maintained to deal with out-of-orderness, for example. More details on the window aggregation strategies adopted by Arc can be found in recent work on shared window aggregation [6, 20], and the out-of-order stream processing paradigm [2, 10]. Finally, when the window is closed, $\lambda_{\downarrow}$ converts – or "lowers" – the final aggregate into the output record type $R_{out}$. In listing 5 lines 5,11,14,15, for example, $A$ is a custom builder type called `MaxAggregator`, which determines the largest values based on its price. As Weld's `merge` function already implicitly provides a $\lambda_{\uparrow}$ from any builder's merge type $M$, our implementation does not require a special $\lambda_{\uparrow}$-component since $R_{in} = M$. If these types are different, $R_{in}$ values can be mapped in the for-loop's function, before merging into the **Windower**. Finally, we also provide the window id to our $\lambda_{\downarrow}$, which itself just calls **result** on the aggregation builder, because emitted values must have a timestamp as well, which can typically be derived from the window id.

It should be noted that in Weld, structs over builders form builders with their merge type and result type also being structs. In this way, the approach described above for windows also supports algebraic aggregators, such as averaging, for example.

Observe that every window aggregation operation in Arc supports the full expressiveness of Weld's batch processing facilities.

***Stream Windows Example:*** To demonstrate Arc, we picked the Stream Window Query 7 from the NEXMark benchmark, defined as: *"Query 7 monitors the highest price items currently on auction. Every ten minutes, this query returns the highest bid (and associated itemid) in the most recent ten minutes."* [21], returning a single result. The CQL representation of this query is given in the following listing:

```
SELECT Rstream(B.price, B.itemid)
FROM   Bid [RANGE 10 MINUTE SLIDE 10 MINUTE] B
WHERE  B.price = (SELECT MAX(B1.price)
          FROM BID [RANGE 10 MINUTE SLIDE 10 MINUTE] B1)
LIMIT 1;
```

The Arc version of this query is in listing 5. The program accepts an input source and output sink of tuples that contain a timestamp, item id and price. A windowed stream is created by merging the input stream into a **Windower** over a for-loop (lines 11-15). Each record is assigned to its window based on its timestamp (line 12), and windows are triggered as watermarks with higher timestamps arrive (line 13). The **Windower**'s aggregation type is a **Merger** that yields the highest bid of all records in a window using the `max_by_price` function. Windows are lowered in line 14 by materializing the **Merger** through the **result** operation. Finally, the **drain** operation transfers records of the windowed stream into the output sink without applying any transformations.

### 3.2 Partitioned Streams

While it is sometimes sufficient to be able to process a stream in purely sequential fashion, this approach does not scale well. For this reason, most stream processors provide mechanism to partition incoming streams into multiple infinite substreams, the union of which would recover the original stream. Substreams are then processed independently and

```
|source: Stream[i32],
 even_sink: StreamAppender[i32],
 odd_sink: StreamAppender[i32]|
  for(source, {even_sink, odd_sink}, |sink, element|
    let mapped = element+5;
    if(mapped % 2 == 0,
      {merge(sink.$1, mapped), sink.$2},
      {sink.$1, merge(sink.$2, mapped)}))
```

**Listing 6.** Optimized version of listing 4 where the pair of filters have been transformed into a filter over pairs.

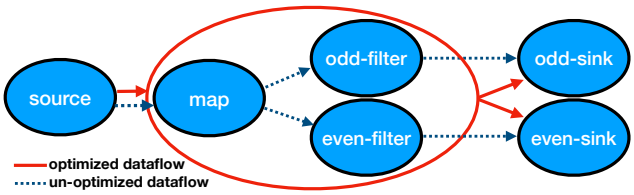


**Figure 1.** The generated dataflow graph for the optimized example.

in parallel. A partitioning function $f_p$ describes which substream each record belongs to. A streaming runtime them assigns certain ranges of partitioned space to certain nodes, thus creating a substream locally observed by each node. No state is typically shared between operators of different partitions and the only global (cross partition) guarantees provided are those given by watermarks.

***Partitioned operators*** can be created in Arc via a special `keyby` iterator (line 41 in listing 2), which replaces the implicit `next` iterator (line 40), that streams use by default, in the first argument of a for-loop. The `keyby` implementation takes a user-defined partitioning function as a parameter. Channels created between partitioned operators are implicitly partitioned and may lead to shuffles if the partitioning at the two ends does not match. A **Windower** used as a builder with a partitioned stream must describe a *local* discretizer, that is it may only act on information from its partition with the exception of watermarks.

### 3.3 Other Weld Extensions

There are a few other small extensions to Weld, that are required for Arc to be sufficiently expressive include the following: declaring *named functions* (line 4 in listing 2) and using such functions as aggregators in **Merger**-type builders (line 34); declaring and pattern-matching *tagged union* types, and an appropriate **union** iterator, are required to support operators with multiple input streams; defining *custom builder types* either inside Arc or as external C or Rust UDFs will be unavoidable. Custom builders should be able to annotate their semantics, in order to know if they are associative, or commutative.

### 3.4 Optimizations

Optimizations enabled by Arc IR include common dataflow optimizations such as *operator reordering*, *redundancy elimination*, *operator separation*, and *fusion* [8]. Operator reordering switches the order of two operators, such that the work of the second one is reduced by the first one. We support redundancy elimination such as avoiding duplicate computation after a split, for example by changing two separate for-loops with a single builder each to a single for-loop with a struct over the two builders. Operator separation splits an operator into multiple operators to enable reordering, reduce per-operator state, and increase the possibility for pipelining parallelism. Operator fusion combines multiple operators into a single operator in order to avoid unnecessary materialization. Listing 6 together with Fig.1 illustrate an example of operator fusion, where the pair of filters from listing 4 are reformulated during optimization to be a filter over pairs, requiring only a single if-expression. The resulting filter is also fused together with the map operator under the same for-loop, effectively unifying all three operators. This way, Arc avoids the materialization of an intermediate stream, as well as the unnecessary duplicate condition check in both branches of the filter.

## 4 Implementation

We implemented a prototype[2] of a compiler frontend for Arc in Scala. The implementation is backwards-compatible with pure Weld. At this point, the frontend includes lexing, parsing, name resolution, macro expansion, and type inference.

To support dynamically typed frontend languages, Arc, like Weld, provides a global type inference mechanism. It is split into two phases: the first traverses the syntax tree and derives type constraints while the second uses a constraint solver to find a solution. If a solution is found, it is applied to the syntax tree. This particular implementation is more powerful, but often slower, than the one found in Weld.

## 5 Conclusion

This paper describes Arc, which is as an extension to the Weld IR with streaming primitives, allowing different DSLs to be expressed in a common abstraction layer. Every window in Arc has the same properties as a a finite data set, thus allowing arbitrary Weld computations to be performed on it. In this way, we support a true combination of batch and stream computations. The design choices we described are motivated by a general model of stream computations. In addition to describing non-blocking stream operators, Arc allows the expression of a wide variety of discretization and window aggregation techniques. This expressiveness allows a number of streaming optimizations to be applied at the IR level, and thereby across different frontends.

We intend to use Arc as a hardware-independent first stage in a stream-processing compiler pipeline that allows

[2]The sources can be found at https://github.com/cda-group/arc.

operator implementations, leveraging a variety of backends, such as CPUs, GPUs, and FPGAs.

## Acknowledgments

This material is based upon work on the Continuous Deep Analytics project granted by the Swedish Foundation for Strategic Research (SSF) under Grant No.: BD15-0006.

## References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: a system for large-scale machine learning.. In *OSDI*, Vol. 16. 265–283.

[2] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, et al. 2015. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1792–1803.

[3] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2006. The CQL continuous query language: semantic foundations and query execution. *VLDBJ* (2006).

[4] Arvind Arasu and Jennifer Widom. 2004. Resource sharing in continuous sliding-window aggregates. In *VLDB*.

[5] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015).

[6] Paris Carbone, Jonas Traub, Asterios Katsifodimos, Seif Haridi, and Volker Markl. 2016. Cutty: Aggregate Sharing for User-Defined Windows. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. ACM.

[7] Peter MD Gray, Larry Kerschberg, Peter JH King, and Alexandra Poulovassilis. 2013. *The functional approach to data management: modeling, analyzing and integrating heterogeneous data*. Springer Science & Business Media.

[8] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. 2014. A catalog of stream processing optimizations. *ACM Computing Surveys (CSUR)* 46, 4 (2014), 46.

[9] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. 2015. Twitter Heron: Stream Processing at Scale. In *ACM SIGMOD*.

[10] Jin Li, Kristin Tufte, Vladislav Shkapenyuk, Vassilis Papadimos, Theodore Johnson, and David Maier. 2008. Out-of-order processing: a new architecture for high-performance stream systems. *Proceedings of the VLDB Endowment* 1, 1 (2008), 274–288.

[11] David Maier, Jin Li, Peter Tucker, Kristin Tufte, and Vassilis Papadimos. 2005. Semantics of data streams and operators. In *International Conference on Database Theory*. Springer, 37–52.

[12] Shoumik Palkar, James Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimajan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman Amarasinghe, et al. 2018. Evaluating end-to-end optimization for data analytics applications in weld. *Proceedings of the VLDB Endowment* 11, 9 (2018), 1002–1015.

[13] Shoumik Palkar, James J Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, Matei Zaharia, and Stanford InfoLab. 2017. Weld: A common runtime for high performance data analytics. In *Conference on Innovative Data Systems Research (CIDR)*.

[14] Kostas Patroumpas and Timos Sellis. 2006. Window specification over data streams. In *Current Trends in Database Technology–EDBT 2006*. Springer, 445–464.

[15] Benjamin C Pierce. 2005. *Advanced topics in types and programming languages*. MIT press.

[16] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices* 48, 6 (2013), 519–530.

[17] Robert Soulé, Martin Hirzel, Buğra Gedik, and Robert Grimm. 2016. River: an intermediate language for stream processing. *Software: Practice and Experience* 46, 7 (2016), 891–929.

[18] Utkarsh Srivastava and Jennifer Widom. 2004. Flexible time management in data stream systems. In *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 263–274.

[19] Kanat Tangwongsan, Martin Hirzel, Scott Schneider, and Kun-Lung Wu. 2015. General incremental sliding-window aggregation. In *VLDB*.

[20] Jonas Traub, Philipp Grulich, , Alejandro Rodriguez Cuellar, Sebastian Breß, Asterios Katsifodimos, Tilmann Rabl, and Volker Markl. 2019. Efficient Window Aggregation with General Stream Slicing. In *EDBT*. ACM.

[21] Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. 2008. *NEXMark–A Benchmark for Queries over Data Streams (DRAFT)*. Technical Report. Technical report, OGI School of Science & Engineering at OHSU, Septembers.

[22] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. *HotCloud* (2010).