Perspectives on Stream Processing

Anthony Dario

June 3rd, 2025

Abstract

Stream processing is an evolving computing paradigm based around manipulating infinite sequences of data. The infinite size of streams presents a unique challenge that has been overcome through the development of incremental processing techniques. The effectiveness of these techniques has led to the spread of stream processing into the realms of big data and low latency applications. The semantic foundations that enabled this transition did not fully anticipate the needs of modern stream processing systems. The implementation of stream processing systems has evolved beyond their semantic foundations leading to a gap in theoretical understanding and actual capability. This gap manifests as ad hoc abstractions and low-level interfaces for stream processing systems that don't naturally capture the nature of stream processing as it is practiced today. In particular the notion of control scheme and the need to reason about distributed stream processing graphs are desirable properties of current stream processing systems that are not captured by existing semantic models.

1 Introduction

Streams are unbounded sequences of data that arise naturally in many areas of computer science. Examples of streams abound in embedded sensors [35], networking [44, 64, 92], business data [27], Astronomy [82], and more. Processing streams has been the subject of study by a variety of communities for decades. This work has produced effective ways of manipulating streams and birthed many stream processing systems. The success of past work has enabled stream processing to apply to unexpected areas. By conceptualizing finite data as infinite the techniques of stream processing can be used to tackle a problem incrementally, providing value before the entire dataset has been ingested. This reframing of finite data processing into stream processing has been successfully utilized in online machine learning [69], transportation analysis [75], and video streaming [41, 76]. The transition of stream processing from being specialized paradigm to a general purpose processing strategy has expanded the scope of stream processing systems. Early stream processing systems focused on querying and extracting insights from streams of data [1, 2, 16, 33, 36, 66]. Newer systems utilize stream processing to enable large-scale event-driven applications. Rising to meet this challenge are a suite of stream processing frameworks that focus on data throughput and distributed computation [6, 32, 89, 109, 117]. While these new systems are representative of the direction stream processing research is heading, streams continue to be a useful abstraction at smaller scales. Operating systems have long conceptualized instructions and I/O as streams with that work being extended to improving stream processing on GPUs [30, 118, 119]. Functional streaming abstractions have found their way into the standard libraries of popular object oriented languages Java [91] and C# [85].

Stream processing systems have raced ahead of the theoretical understanding streams producing a gap in our theoretical understanding. The early SQL variants were able to lift the strong fundamentals of relational algebra to provide a query language for stream processing but the needs of modern systems have outgrown that paradigm. Newer stream processing frameworks build their interfaces on general purpose programming languages. They require the user to consider details such as how state is divided in a system. The freedom provided by this approach allows the user to make costly mistakes – such as attempting to store the entire stream – and reduces the ability to automatically optimize the processor.

A strong semantic foundation for a language provides a myriad of benefits for users. Not only does a strong foundation lead to a natural way of expressing the desired task but it also allows an automated system to make safe transformations in the name of optimization. Taking SQl as an example, the semantic foundations of relational algebra provide a clean interface for querying databases. The user isn't responsible for managing individual rows and their transformations. Instead the user thinks at a declarative level and a query planner with a strong understanding of the abstraction can optimize their intent for them. It is a testament to the strength of the SQL abstraction that many newer stream processing frameworks still attempt to maintain a SQL interface despite the growing needs of the systems.

This document focuses on the semantic models of stream processing. By studying what aspects of stream processing these models capture we aim to understand why they do not meet the needs of contemporary stream processing systems. This discrepancy between theory and application indicates a lack of understanding of stream processing. This discrepancy is also an opportunity to improve upon the state of the art. This document identifies two key aspects of stream processing to be captured. First, the model should be able understand and transform the dataflow graph of the stream processor. This is necessary for many stream processing optimizations. Second, the model should capture where control is located in the system. This is an overlooked aspect of stream processing that has the potential to be exploited.

Figure 1 shows the scope of this document. The graph in the figure represents stream processing systems and their models. The graph is read left-to-right and the document follows that structure. We begin in Section 2 with models of streams. We first discuss the properties of streams in Section 2.1, then explore three models and how they capture those properties in Section 2.2. We then move onto stream processing in Section 3. To understand what stream processor models are capturing we study the requirements and desired properties of existing stream processors use in the first four subsections. These subsections classify various aspects from common patterns to optimizations. Section 3.5 describes four stream processing models and how they capture the different aspects mentioned earlier in the section. We move onto a discussion of existing stream processing systems in Section 4. To understand the aforementioned gap between existing models and existing systems we first discuss the evolution of stream processors from streaming databases to batch/stream processors in Section **??**. We then explore different languages for expressing stream processors in Section **??**. These languages represent ways to express the different models that have been discussed. They show how assuming a different semantic model can effect the interface for a stream processing system. We conclude in Section 5 with a discussion of the trade-offs made by stream processing models. We also reiterate the point that control schemes are an overlooked aspect of stream processing that have the potential to be exploited for performance and interface benefits.

It is worth noting the question marks in Figure 1. These represent gaps in the current stream processing landscape, where either semantic models or implemented systems are missing. The most obvious missing pieces are the semantic models of a stream and stream processor for modern stream processing systems. This is the gap created by the application of stream processing to big-data processing. The other gap is a lack of an implementation located at the bottom of the figure. This comes from a line of work that has developed an interesting model of stream processing based on mixed inductive types that has desirable properties. While there is no implementation yet, the benefits of the model should guide the design of any future semantic

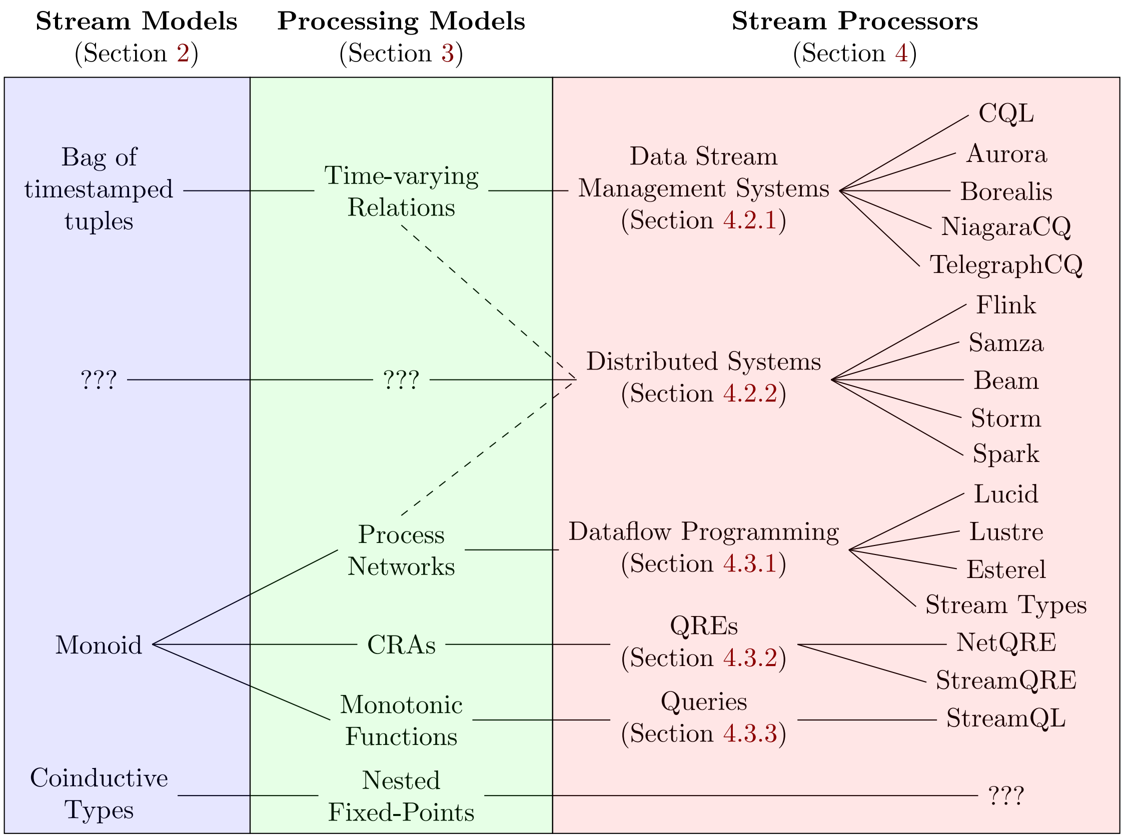


Figure 1: Stream processing models and their implementations. Underlying semantic models are on the left and higher level models are on the right ending with existing systems. Edges indicate that an underlying model is the foundation of a higher-level node. Dashed lines indicate that a model “inspired” a higher-level node. Question marks indicate a gap in the graph.

model.

Streams have always had their place in computer science. The increased rate of data ingestion has lifted stream processing from a specialized paradigm to a generally useful technique. The expansion of stream processing applications has outgrown the designs of existing stream processing abstractions. Systems have been developed to meet the challenges presented by the new applications but theory has fallen behind. This document explores the semantic foundations of stream processing in an attempt to understand where and why these models fall short of modern needs and what a new model would need to include in order to meet those needs.

2 Streams

Before discussing the existing work on stream processing it is worth spending some time considering what streams are. This is often an overlooked and underspecified area of stream processing but proper consideration can yield benefits. Some systems such as Apache Flink [32] and Aurora [2] provide a brief model of a stream while other systems like Lucid [115] and TelegraphCQ [33] elide a description of what a stream is and rely entirely on the users background knowledge.

By understanding the shape of streams we learn how they can and cannot be manipulated. These constraints on the structure of streams can then be built into a system and stream processors can be automatically checked for making incorrect assumptions. An example would be attempting a computation that depends on elements arriving in a certain order when that order is not guaranteed. These assumptions can be encoded into the type of a stream and a type checker can perform static checks for errors. Some work on typing streams has been done by Cutler *et al.* [45], Mamouras [83], and Saurin [101].

2.1 Properties of Streams

A stream is a potentially unbounded sequence of data items. The three parts of that definition (potentially unbounded, sequence, data items) are worth discussing as each of them requires special consideration during processing. We will discuss these in reverse order.

2.1.1 Data items

Streams are data structures. While the structure of streams demands most of the attention it is important to characterize the data contained within. By understanding the shape of the data contained within a stream compile time guarantees can be made for processing systems. Accurate types can inform strong bounds on space requirements when compiling a language to a streaming algorithm [84, 116]. In this document we will use the terms “data items” and “streaming elements” interchangeably.

2.1.2 Sequences

Streams are inherently sequential and ordered. Data items can be thought of as existing “earlier” or “later” in a stream and not all data is available at any one moment in time. This order may have meaning that is important for processing. As an example, the first step in a particular arrhythmia monitoring algorithm relies on identifying three short consecutive intervals between heartbeats to detect a potentially accelerating rhythm [3]. If the data items arrive out of order then the there is no assurance that the consecutive requirement will be met.

Order is typically a straightforward property reason about when stream processing is non-distributed and non-parallel. In the distributed and parallel cases differences in processing time and network latency cause changes in the order of streaming elements. Various systems utilize timestamps to provide an alternative to ordering elements based on when they are produced [2, 16, 74]. Other systems attach progress tracking information to streaming elements that ensures no later elements will arrive after [50]. A third technique is revision processing which produces updates to outputs when out-of-order data arrives. Samza practices revision processing by maintaining a history of window aggregations and updates them when an out-of-order elements arrives [89]. On the other hand the StreamIt system aggressively parallelizes stateless operations and uses its synchronous control scheme to ensure order is maintained after the parallelization [62]. Discussed more thoroughly in Section 3.4 StreamIt’s technique hints at the importance of control in maintaining order in a stream.

The significance of order in stream processing has been a concern in many stream processing systems. The commonality of distributed systems in contemporary computing has forced the development of dynamic out-of-order processing. Static methods for handling out-of-order computations are rarer. One example involves encoding out-of-order streams into a type systems that forces the user to handle all possible orderings of the stream at compile time.

2.1.3 Unbounded

The most notable property of streams is that they are potentially infinite. With finite resources dealing with infinite data becomes a perilous task. Special care must be taken to avoid infinite loops or overuse of available memory.

To get around the infinite size of the stream data structure stream processors must work on incremental data and produce incremental results. This frees the processor from having to store an unbounded amount of data to produce values and ensures that any consumers will eventually receive data to operate on. This paradigm is fundamental but inherently tricky. It is often desirable to process elements based on their pattern in the stream [116]. Without clairvoyant knowledge of the future the pattern may never appear and the system will have to ingest data forever.

The fundamental paradigm of incremental processing has been recognized as a useful technique beyond infinite data structures. Big-data systems have found it useful to conceptualize their data as infinite and process incrementally [40]. Incremental processing has also been used with smaller amounts of data as an optimization technique for producing results quickly. Examples can be found in decoding media byte streams over the internet [41] and incrementally training machine learning models over large datasets [69]. This utility beyond infinite streams is the source of the qualifier *potentially* in the above definition. Streams are conceptually infinite data structures even if there is an end to the data.

Some static techniques have been developed for managing the infinite size of streams. QRE-based languages (discussed in Section 4.3.2) use the concept of an *unambiguously iterable* pattern to ensure that a stream is divided into finite subsections. Cutler *et al.* [45] force aggregations to be the concatenation of a finite stream and a potentially infinite one. More generally, infinite structures defined coinductively can be considered *productive* if they always produce data in finite time [72].

Even when the stream processor is well defined the rate of produced data may overwhelm the consumer. If the rate of incoming data is greater than it can be processed the processor needs to store a growing amount of data. Two common techniques for managing this problem are load shedding [113], where incoming data is selectively dropped, and exploiting parallelism, where a consumer is split into multiple consumers to increase throughput [68].

The size of streams presents their fundamental challenge and gives the fundamental benefit of stream processing. Despite the success of incremental processing in tackling this problem challenges remain and defining stream processors remains a tricky task.

2.2 Models of Streams

Modeling streams semantically typically receives less attention than modeling stream processors (discussed in Section 3.5). Often stream processing systems only provide a brief high-level description of a stream. Despite this neglect a good stream model can inform and guide the design of a stream processor by capturing the desired properties of a stream and enforcing only valid operations.

The classic model of a stream is a function $s : \mathcal{T} \to A$ from some totally ordered time domain $\mathcal{T}$ to the type of streaming elements A [108]. This model allows for extreme flexibility when defining a stream but does not provide much structure for the shape or guide how a stream can be used. We present three more recent stream processing models that have been used for working with and reasoning about streams.

2.2.1 Databases: Ordered Tuple Structures

Streaming databases consider streams to be ordered structures containing database tuples. Special attention is paid to how elements in a stream are ordered and this is also the source of subtle differences between streaming database models. The focus on ordering stems from a desire to adapt order dependent relational queries (e.g. join) to a streaming setting.

In CQL and derived languages the stream is considered a bag (unordered multiset) of timestamped tuples written $\langle s, \tau \rangle$ where s is the tuple and $\tau \in \mathcal{T}$ is the timestamp [15, 16, 74]. The domain of timestamps $\mathcal{T}$ is not necessarily related to physical time but must be totally ordered. By considering a stream a bag, the order of elements is not built into the data structure itself, but rather it is imposed by the timestamps. This has the effect that query languages do not directly consider certain elements as unavailable. Instead queries are given with respect to the current timestamp and only the elements from the bag with a lesser timestamp are returned. Depending on how the timestamps are defined (e.g. creation time within the system [16]) it is possible for tuples to be given identical timestamps. These identically timestamped tuples are considered to have occurred simultaneously. This has implications for the ordering of tuples. If the most recent n tuples are requested but the most recent $m > n$ tuples all have the same timestamp then n of them are chosen non-deterministically.

In an attempt to avoid the non-determinism of simultaneously occurring stream elements Jain *et al.* [74] provides a mechanism for user-defined suborderings. Intuitively the tuples are ordered by timestamp and another property, such as arrival order or some semantic value within the tuple, is used to order simultaneous tuples. More formally the model defines an equivalence relation $_s$ and ordering $<_s$ on tuples. This new relation is a *refinement* on the induced timestamp equivalence relation $_t$ in that whenever $x\ _s y$ then $x\ _t y$. Similarly, the order $<_s$ is a refinement of the timestamp order $<_t$ in that whenever $x <_t y$ then $x <_s y$. The refined relation and ordering provides an ordering on tuples with the same timestamp. This ordering can be defined by the user on any attribute of a tuple using an interesting `SPREAD` operator which separates out simultaneous tuples by their ordering via $<_s$.

The Aurora [2] and Borealis [1] database systems also provide a method for user defined orderings. All tuples are given a timestamp when they enter the system but this is used internally and is invisible to the user. Instead, order-sensitive queries accept an order specification which identifies the order that tuples are expected to arrive in. Out of order tuples are ignored and discarded with some leeway given by a user-provided slack.

Modeling a stream as a bag of timestamped tuples proved to be a successful abstraction. Most streaming database systems developed in the 00's relied on this model either explicitly or implicitly. Despite this success there are some clear shortfalls in the model. By modeling the stream as an unordered data structure (bag) and requiring systems to inspect individual data items to determine order excess complexity is introduced to maintain this property. The enduring influence of these systems on more contemporary developments means the benefits and detriments of the model have continued to be propagated.

2.2.2 Streams as Coinductive Data Types

In programming language theory streams are often seen as a coinductive data type [72, 73, 98]. Coinduction is a proof technique that allows for reasoning about circular structures. The ability to reason circularly makes coinduction a powerful tool for reasoning about infinite structures and a natural fit for defining streams.

When defining a datatype coinductively attention is paid to how the type is used. For streams the means defining the first elements of the stream, the head, and the rest of the stream, the tail. This can also be thought of as "destructing" [1] a datatype into its constituent parts. For a stream there are two cases to consider: either the stream is finite and empty, or it contains more data. When destructing an empty stream an special value is returned indicating that there is no more data. When destructing a non-empty stream the head and the tail are returned. This behavior is captured in the type

$$A^* = \nu X.1 + (A \times X)$$

which represents streams of data items of type A. In the type definition the νX indicates that this is a coinductive type (allowing for circularities) and that the following signature should

[1]Here destruction does not refer to the resources of the object being cleaned up but rather the interface for taking it apart.

be read as the type that can be returned after destructing the team. The return type after destruction is a either a 1 indicating that the stream was empty or a pair of the head and the tail, $A \times X$.

Allowing for circularities provides a way to write an object that produces an infinite stream of data. A simple example is the infinite stream of 1s. When taking apart the stream of ones the first element, a 1, is given as the head, and the tail of the stream is the infinite list of 1s again. This structure can be written in Haskell as:

```
ones :: [Nat]
ones = 1 : ones
```

Here we have defined `ones` as a list of `Nat` (natural numbers). The first element is a `1` while the rest of the stream is given recursively as the list we are defining. If one were to attempt to build this structure inductively then they would continuously append 1s to the end of the stream in an infinite loop. Haskell allows this definition because it is lazy and does not compute the entire list until it is requested. Exploiting the laziness in Haskell to treat lists like streams has been developed into a stream processing style in its own right [78].

The programming languages community has developed ways for working with coinductive structures. A technique for defining a coinductive structure is referred to a *copattern matching* [4]. Copattern matching tasks the programmer with defining how a stream acts in any situation. We can define the infinite list of 1s using copattern matching in Agda as:

```
ones : List Nat
head (ones) = 1
tail (ones) = ones
```

Here the definition consists of two clauses, one for each of the possible uses of the stream. The first defines how `ones` acts when the `head` of the stream is requested, in which case a 1 is returned. The second defines how `ones` acts when the `tail` of the stream is requested, in which case `ones` is returned.

To guard against the possibility of infinite loops definitions must be *productive*. If a piece of a stream is requested the stream must produce a result in finite time. The above definition of `ones` is considered productive because it will either produce a natural number or another stream immediately. An example of a non-productive definition is one that continues to recurses an unknown number of times:

```
unproductive : List Nat
head (unproductive) = 1
tail (unproductive) = tail (unproductive)
```

If one were to request the tail of `unproductive` then the system would continue to request its tail ad infinitum and never return a useful result. By ensuring a productive definition a coinductive definition can be made safe in the face of an infinite structure.

The powerful tools developed to work with coinductive definitions have proven useful for reasoning about streams. Coinductive streams have been useful for modeling real numbers [94] and performing exact arithmetic operations on them [24, 102] as well as reasoning about modeling temporal logic modalities [72]. The usefulness of the coinductive model is rarer to see in general purpose stream processing systems. The main reason for this is that this line of work involves *defining* streams. Stream processing applications are typically concerned with *consuming* streams. It is common for the stream to be produced outside of the stream processing system requiring no need for an internal definition. However the techniques built around coinduction show promise in stream processing. A stream processor is its own circular structure that reacts to stream elements as they arrive, producing streams in their own right. The tools developed to work with coinductive definitions to grapple with the infinite should not be ignored but rather adapted to the needs of stream processing.

2.2.3 Streams as Monoids

Work done by Mamouras *et al.* [83] focuses on capturing the essence of incremental processing. Streams are modeled with an algebraic structure known as a monoid. This structure captures the ability to process incomplete input and has a method for understanding the order of elements in a stream.

Definition 2.1 (Monoid)**.** A monoid is a tuple $(A, \cdot, 1)$ where A is a set, $\cdot : A \times A \to A$ is a binary operation on elements of the set, and $1 : A$ is a distinguished element of A such that for all x, y, z in A:

- $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ (associativity)
- $1 \cdot a = a = a \cdot 1$ (identity)

A typical example of a monoid is the natural numbers with addition as the binary operation. The identity element is then 0.

In the stream case the elements of the set A are finite subsequences of the stream. The binary operation is concatenation of these subsequences and the identity is the empty subsequence. A preorder $\leq$ is imposed on the subsequences in the stream monoid.

Definition 2.2 (Preorder)**.** A preorder $\leq$ is a binary relation on a set A such that for all $x, y, z \in A$:

- $x \leq x$ (reflexivity)
- $x \leq y$ and $y \leq z$ implies $x \leq z$ (transitivity)

The preorder is defined so that any subsequence $x \in A$ is less than another subsequence $y \in A$ if x is a prefix of y. Formally $x \leq y$ iff $\exists z \in A$ such that $x \cdot z = y$. In this way finite subsequences represent data in the stream that is available and new data that arrives is concatenated in a strictly increasing fashion.

By considering finite prefixes and their concatenation, this model is able to reason about the order of elements in the stream. Prefixes are always before the rest of the sequence and understanding the structure of the prefix allows decisions to be made such as if an element in the prefix will have a value below a certain threshold. Similar reasoning on prefixes has been used to assign types to aggregating stream operations ensuring that they only collect a finite amount of elements for processing [45].

Mamouras demonstrates the generality of his model by translating many existing stream processing systems into the monoidal model [83]. The model allows him to apply algebraic reasoning and ensure the correctness of his translations. While the model provides nice properties for reasoning about finite operations on infinite streams, the generality of monoids lacks structure that is useful for stronger reasoning and understanding the computational component of stream processing. This model is not explicitly used in any existing stream processing system (although its influence is clearly felt in Cutler *et al.* [45]).

2.3 Summary

Table 1 summarizes the discussed models and how they each capture the three different properties of streams. The different methods have various trade-offs for how that property is handled. These trade-offs will become more apparent in the discussion of stream processing models (Section 3.5) and stream processors (Section 4). Table 1 summarizes the discussed models and how they capture the

Model	**Property** Data	Order	Unbounded
Bag of Timestamped Tuples	Tuple Schema	Timestamps	Infinite Sized Bag
Monoids	Carrier Set	Concatenation	Finite Prefixes
Coinductive	Type	Only Access The Head	Productivity

Table 1: Stream models and how they capture the properties of streams.

3 Stream Processing

Stream processing is the act of transforming one or more streams into some output. Often, the output is another stream, but even if not intermediate transformations between streams are useful for producing a final result. As noted in Section 2.1.3 incremental processing is a fundamental component of stream processing. Producing some output before the entire stream is read is necessary to deal with a potentially infinite input.

To capture the proper abstraction for stream processing it is necessary to see how what requirements are demanded of stream processing systems. To that end, this section starts in 3.1 by describing two methods of describing a stream processor, differing in their level of detail. This lays the groundwork for understanding what stream processors do. Section 3.2 categorizes different tasks required of stream processor. These categorizations are used to make safe optimizations, discussed in Section 3.3. These optimizations lead into a discussion of an under-utilized aspect of stream processing, the location of control. Control is discussed in Section 3.4 as a promising avenue for exploitation. Evidence of the benefits of exploiting control are given from different areas. Finally four models of stream processors are described in Section 3.5.

3.1 Interface Granularity

There are roughly two levels of detail with which to consider a stream processing system. At a higher and coarser level, a stream processor is a graph of processing nodes we will call *transformers* (also referred to in the literature as *operators* or *transducers*) connected by streams. In this document, this graph is referred to as the “stream processing graph” or “dataflow graph” interchangeably. The finer-grained viewpoint is to think about the transformers themselves. Both viewpoints can fully express stream processing but they have reasoning and optimization consequences. Figure 2 shows the relationship between the two viewpoints. A cost register automata (Discussed in Section 3.5.2) represents an individual transformer, while a graph shows how transformers are connected.

Many stream processing systems expose a particular level of detail in their interface. A diverse range of systems such as Apache Beam [6], Brooklet [105], and Aurora [2] provide a graph-level interface while languages like Lucid [115] and Lustre [65] center around defining transformers directly. Not all systems are so opinionated and many allow for defining individual transformers and using them in the declarative definition of a stream processing graph. IBM’s Stream Processing Language [66] allows for inline transformer definitions while Apache Samza [99] provides distinct interfaces for both viewpoints.

It is important to understand that while these two viewpoints are *conceptually* different they are not *functionally* different. A complex transformer defined directly can be represented as a subgraph in a stream processor while a large graph of predefined transformers can be seen as one large transformer.

Given the overlap in functionality between the graph and transformer level of granularity

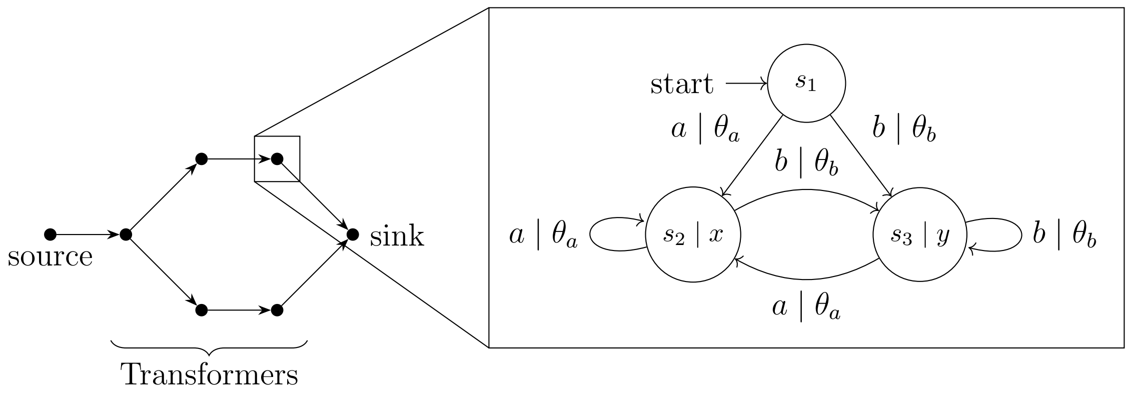


Figure 2: The different granularities for stream processing. The graph of transformer is displayed on the left, while the inset shows a cost register automota (see Section 3.5.2) associated with an transformer.

it can be difficult to identify what interface is being presented to the user. Is a streaming SQL query a user-defined transformer or a graph defined out of the constituent clauses? One rule of thumb is that if an interface requires the direct manipulation of stream elements then the user is defining a transformer, if the interface provides methods for manipulating entire streams then the user is defining a graph. As an example, consider the following CQL [16] query which continually produces the number of packets sent from IP address `1.0.0.1` in the last two minutes.

```
SELECT Istream(Count(*))
FROM packetStream[Range 2 Minutes]
WHERE senderIP == "1.0.0.1"
```

This query never exposes an individual element to the user. Instead the user defines a window on the stream (`FROM packetStream[Range 2 Minutes]`) and uses SQL-esque syntax to filter (`WHERE senderIP == "1.0.0.1"`) and count (`SELECT Istream(Count(*))`) all the elements in that window.

3.1.1 Graph View

In the graph-level view a stream processor is a directed graph with nodes representing transformers and directed edges between them representing data streams. The direction of the edges indicates the direction of the dataflow. Transformers without incoming edges are called *sources* and represent an external streaming data source. Transformers without output edges are called *sinks* and represent the output of the stream processor. The stream processing graph is not necessarily acyclic. Cyclic graphs are used in certain dynamic optimization techniques [19] and as a feedback mechanism for producing subsequent outputs [111].

Systems that present the stream processor as a graph typically have a fixed set of predefined transformers. These transformers provide common stream processing functionality and can be categorized in the framework provided in Section 3.2. Often, this set of transformers is intentionally not minimal. Redundancy provides opportunities for optimizations [2]. While the set of transformers is fixed, the transformers themselves are somewhat dynamic in their definition allowing for the user to configure logic within some behavioral boundaries. An example is a filter transformer which selectively passes along data that satisfies a user supplied predicate. The general shape of a filter is always the same but the predicate is variable depending on the needs of the programmer.

With the fixed set of transformers the programmer is then tasked with composing the graph to perform the desired processing. This composition is done through a declarative language such as CQL [16], SPL [66], or StreamQRE [84]. This model allows for optimizations to occur at the graph level. Interactions between the transformers are studied beforehand and safe rearrangements of the graph are identified and automatically applied in the pursuit of optimization [68]. The graph-level interface also maps nicely to a distributed setting. Nodes that are logically separate can be physically be separated by an automated scheduler [32]. Optimizations that transform the graph then transform the distributed topology of the system.

The nice reasoning properties and natural mapping to a distributed setting have made graph-level interfaces popular in the modern distributed stream processing systems. While many systems find it necessary to expose a lower-level interface for directly defining transformers they often guide people to using the graph-level interface first.

3.1.2 Transformer View

The more detailed view of stream processors provides a way to define stream transformations directly. When a stream processing system provides this interface to a programmer, the programmer is responsible for lower-level details such as the manipulation of individual stream elements, when a transformer should produce output, and the management of state. This style of stream processing is exemplified by dataflow languages such as Lustre [65] or Lucid [115].

In the spirit of code modularity user-defined stream transforms are often packaged into reusable pieces of code that can be composed to produce larger stream transformations. In dataflow programming stream transformations are defined similar to functions [45, 115] that consume and produce streams. Like ordinary functions, the streaming functions can be reused and composed together to build the final stream processor. These functions have direct access to the stream and can wait for the number of stream elements required to perform the transformation.

The transformer view gives greater control to the programmer to define their desired operations. It can be freeing to not have to compose a set of predefined transformers to achieve the desired functionality. The flexibility comes with a cost. Optimizing graph-level manipulations become more difficult to automatically perform. Without previous knowledge of the transformers behavior the code has to be analyzed to determine if it is safe to make an optimization, a significantly more difficult task.

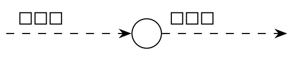
It is telling that most stream processing systems simultaneously provide a transformer-level interface and still promote a more declarative graph-level view. The advantages of the graph-level view are desirable but are incomplete with gaps in expressive power being plugged a secondary low-level interface.

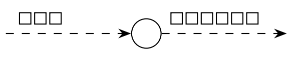
3.2 Common Stream Processing Tasks

To understand the needs of a stream processor it is informative to look at common tasks a stream processor accomplishes. This section describes four categories of stream transformers: transforms (stateless), folds (stateful), splits, and joins. These classifications of transformers into broader classes allows for easier reasoning on the graph-level of the stream-processor. Instead of understanding the properties of each individual node in the graph, placing nodes into categories enables certain rearrangements of the graph and informs placement of the computations, optimizations discussed in the next section.

3.2.1 Transforms

Transforms are stateless transformations of elements. Three transforms are diagrammed in Figure 3. The simplest is a `map` which applies a function to each element of a stream, producing a single output. The `filter` checks every stream element against a user-defined predicate


(a) One-To-One Transform


(b) One-To-Many Transform (ConcatMap)

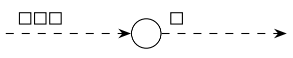

(c) Filter

Figure 3: Example transforms.

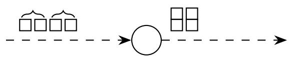


Figure 4: An example fold.

and only produces those that satisfy the predicate. A filter can be thought of as "shrinking" the input stream. This shrinking has optimization implications as it reduces the amount of communication between transformers and it is preferable to place a filter as early in a stream processor as possible. Filters also have an interesting pitfall, it is possible to provide a predicate that is always false. This acts as a method of blocking the stream. While this may not always be an error it is uncommon to want to end a streaming computation. Identification of a blocked stream can allow a system to free resources earlier in the pipeline.

Of particular note is the `concatmap` which produces many elements from a single input. This operation "expands" the size of the stream by increasing the number of elements flowing downstream. While this may seem like a simple variant of `map` which produces a list, the separation of the elements of the list into individual streaming elements gives it a different character. The cost register automata [9] and tree stream processing [59] models both have non-obvious implementations of this transformation (see Section 3.5).

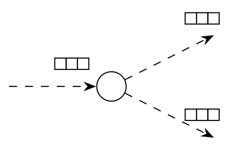
3.2.2 Folds

Folds (also called aggregations) are stateful transformations of stream prefixes. A fold is diagrammed in Figure 4. The key difference between a fold and a transform is the requirement of state. A downstream effect is that a fold can rely on multiple input elements to produce an output element. Folds all require an initial state and a function that updates that state given a new element. This scheme is reified in the `Initialize` clause of the Expressive Stream Language's aggregation syntax [20]. Similar to `filter`s, folds are capable of "shrinking" the size of a stream. This is not a necessary condition as some functions will fold over a single element (utilizing the initial state) or produce an element for each element consumed.

The shrinking behavior of folds provides one of their main benefits. They are able to compress the information in a stream providing aggregate insights. This functionality made folds the focus of study for early stream processing systems that were directed towards querying streaming data. Particularly attention was paid towards defining a finite *window* of input elements that the fold could access to produce output [16]. By ensuring a strong bound on the size of the window, a stream processing system can precisely allocated memory for a fold [2,84].

Windows are either considered *tumbling* or *sliding*. A tumbling window splits a stream into discrete non-overlapping chunks, effectively consuming the elements in the window whenever output is produced. A sliding window adds an drops elements as they become available, creating overlapping windows. Tumbling windows act similarly to a fold from functional programming. Sliding windows act more like a functional `scan`, producing output at every step given the elements within the window. Elegant tumbling window definitions over patterns are a core component of the QRE-based languages discussed in Section 4.3.2.

An interesting example of a stateful transformer is a `take`. The `take` transformer only passes on a certain number of streaming elements before blocking the stream. Similar to a filter


(a) Duplicating Split

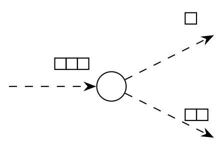
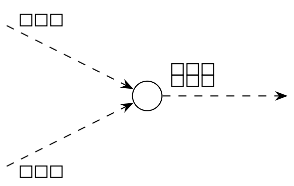

(b) Partitioning Split

Figure 5: Example splits


(a) Combining Join

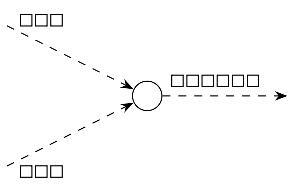

(b) Sequencing Join

Figure 6: Example Join Combinators

which has an unsatisfiable predicate, identifying when a `take` has blocked the input stream is an opportunity for reducing the usage of system resources. Through the inclusion of state `take` allows the user to control when the input stream is blocked (i.e. after n elements or when a certain threshold has been met) and allows the system to easily identify when the input stream will no longer be needed.

3.2.3 Splits

Splits take a stream and create multiple different streams. Splits can be divided into two different styles. A *duplicating* split forwards copies of every element of the input stream to all outputs, while a *partitioning* split selectively sends elements to specific output streams. Figure 5 diagrams the different styles of splits.

Splits can be used for logical transformations when data might be needed for two different downstream processing stages. An example would be a duplicating split over a stream of logs where one output sends logs towards some long-term storage, while the other output is analyzed for predetermined events to alarm on. Splits are often used for optimizations. A split can be introduced before an expensive transformer to scale up the operation by producing parallel transformers. A partitioning split followed by a join can be used to dynamically load-balance the system [61].

3.2.4 Joins

Joins merge multiple streams into one. There are two basic forms of joins. The first are *sequencing* joins which perform no processing of the individual elements and produce elements from both streams in a single stream. The second form is a *merge* join which combines elements of both streams into a single element of the output stream. Figure 6 shows both styles of join.

When using a join, order becomes a concern. Typically the transformers producing the input streams are not synchronized so there is no guarantee on the order that elements arrive in. This can be mitigated by synchronizing the entire system [62], or enforcing that downstream operators are not order dependent [45]. For merge joins order can have a large effect on the output stream, dictating which elements are combined. This appears in streaming databases that wish to use the `join` SQL operator over a sliding window [15, 36]. Depending on the order elements appear in the windows a SQL join will combine different elements.

Joins are useful when combining data from different sources or in conjunction with splits for adding parallelism to the stream processing graph. However the difficulties of parallelism rear there head giving less information about the structure of the stream to downstream transformers.

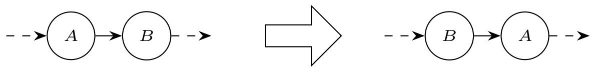
3.3 Optimizations

Optimization is a fundamental task of any computational system. How can the proper result be computed using fewer resources? Stream processing is no different. We group optimizations into three types: graph transformations, placement, and algorithmic. The first two use the graph-level interface to reason about the order of computations and how they are distributed in a topology. The previous study of stream processing tasks allows us to make an informed decision on if an optimization can be applied safely. The last type, algorithmic, deals with optimizing within a transformer. We present a representative (but not comprehensive) selection of these types of optimizations below.

3.3.1 Graph Transformations

Graph transformations manipulate the structure of the stream processing graph. These optimizations rely on knowledge of the properties of the transformers to safely modify the graph. This means they are mostly found in systems that expose graph-level interface.

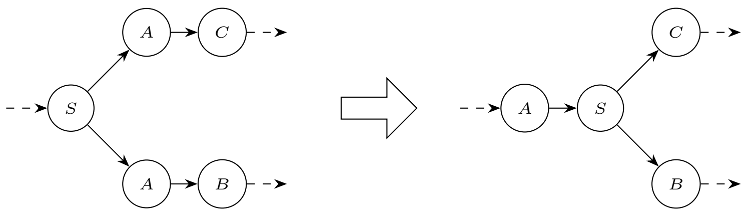
Reordering. Reordering flips the order of two transformers.



In the above example suppose transformer A is a filter while transformer A performs a costly computation. By swapping the two nodes transformer A does not have to process elements needlessly.

Reordering is commonly used in relational streaming systems. CQL [16] and Aurora [2] both use the semantics of relational algebra to perform reorderings of queries. These optimizations come from work on traditional (non-streaming) SQL query planners and can be found in standard textbooks such as Garcia-Molina et al. [54]. The eddy operator [18] dynamically reorders a set of commutative transformations by measuring their output rates to determine the most effective order.

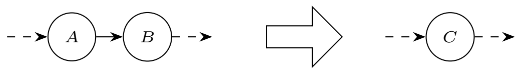
Redundancy Elimination. Redundancy elimination identifies redundant computations that can be stored after the first computation.



Here transformer A is on both branches directly after a duplicating split S. By moving transformer A before the split it only has to operate on the data once.

The above example is illustrative but redundancy elimination includes any optimization that caches a result instead of computing it many times. Redundancy elimination often occurs when a system is shared by different applications. NiagaraCQ [36] groups dynamically added queries based on signature and maintains a table of overlapping constants for each group that are joined together.

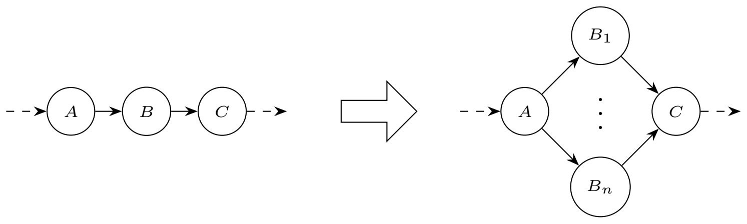
Fusion. Fusion is the merging of two adjacent transformers.



Here, transformer A is an operation over large stream elements while transformer B is a cheap filter that depends on the results of A's computation. While the two transformers remain separate all data that A produces will need to be marshaled and sent to B. By fusing the two transformers into one, C, only the elements that pass the filter will need to be communicated.

Fusion aims to reduce the cost of transmitting data between transformers. Depending on the system set up data may need to be transferred over a network, marshaled, or written to a shared data store to communicate intermediate values. If the transformers are located on physically separate nodes then fusion can eliminate transportation costs associated with sending the data over a network. Brooklet [105] makes it easy to identify fusible transformers by including any modifiable state in their signatures and only allowing output streams to be used once. Fusion is used in functional programming to remove intermediate structures in list processing pipelines [43].

Fission. Fission splits a single transformer into many parallel transformers.



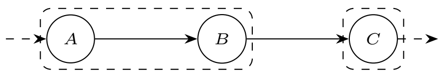
The above diagram shows transformer B being split into n parallel transformers with data from transformer A being distributed among the split transformers.

Fission enables parallelism by allowing simultaneous processes to perform the same transformation. It is necessary that the transformer can operate on elements independently. It also requires adding a partitioning split in order to distribute the streaming elements across the split transformers. Fission removes any guarantee that the order of elements arriving at the transformer is maintained on its output. StreamIt [61] gives a set of predefined transformers of which some are stateless and are identified as opportunities for fission. Sawzall [95] restricts its language to ensure commutativity before an aggregation stage ensuring safe parallelism.

3.3.2 Placement

Placement optimizations center around physically locating data and computations. These optimizations attempt to match the dataflow graph and the transformers to the underlying hardware in order maximize resource usage. Many of these optimizations target distributed systems where the underlying topology of the system can be utilized for performance benefits.

Layout. Layout is the assignment of transformers to physical locations such as hosts or processors.

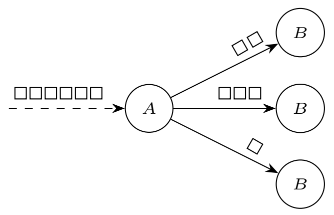


Imagine that transformers A and C are expensive computations while B is a cheap filtering transformer. By placing A and C on different hosts they will not starve each other for resources.

Because B is a cheap filter it makes sense to co-locate it with A (similar to fusion) to avoid communicating elements that would otherwise be filtered out.

The goal with layout optimization is to optimize the resource usage of a variety of hosts without incurring excessive communication costs. SPADE [57] uses a two-phase compilation where the first phase runs the stream processor under simulated data and uses the seen resource usage to assign transformers to hosts. StreamIt performs simulated annealing to optimize between core resource usage and communication overhead. A precise cost function for the optimization process can be given because of the heavy constraints on data rates [62].

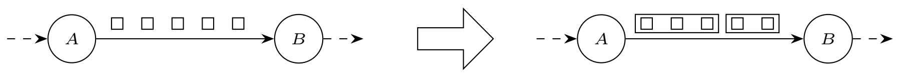
Load Balancing. Load balancing is selectively sending data to different transformers.



The above diagram shows transformer A selectively sending different amounts of stream elements to three downstream transformers that have undergone fission.

The goal of load balancing is to ensure that work is distributed among resources effectively, and avoiding overloading any one part of the system. River [17] uses a randomized credit-based scheme in which producers select a downstream consumer that has fewer outstanding messages than some given threshold. For large messages consumers request data from a producer in a pull-based scheme. StreamIt performs load balancing by reorganizing the stream processing graph through fusion and fission optimizations [62]. Knowledge of the data rates from transformers allows the compiler to map the graph to a processor architecture without overloading any particular node.

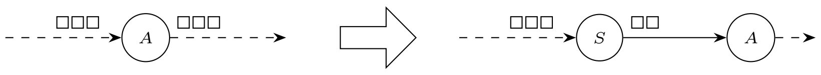
Batching. Batching is the aggregation of data before transferring between transformers.



Here, transformer A and B are not physically co-located and A needs to send six small elements to transformer B. The size of the elements means that the data transmitted between the two transformers is largely overhead. In this scenario it makes sense to combine multiple elements together as into a batch and send them as a single message to reduce overhead.

Batching identifies the tension between combining data into larger messages and incrementally processing smaller pieces of data. This can be seen through the intertwined histories of batch-processing and stream-processing systems. The MapReduce system [48] is a batch processing system that effectively manages large quantities of data. Condie *et al.* [40] propose a performance enhancement to MapReduce by streaming the batch data to allow for finer-grained incremental processing. Apache Spark [107], Apache Beam [6], and Apache Flink [32] all claim to be a unified model for batch and stream processing. Many Flink connectors (external sources or sinks of data) implement batching at the point of connection to increase throughput [51].

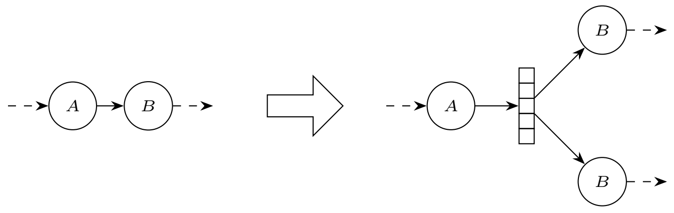
Load Shedding. Load shedding drops streaming elements when resources are stretched thin.



In the above diagram imagine transformer A is a fairly expensive process that is accepting all incoming streamed elements. If the rate of incoming elements is greater than the rate that A can process them then extra latency is introduced into the system. Transformer S is a shedder which dynamically drops incoming elements if it detects downstream processing delays in order to maintain a reasonable level of latency.

Load shedding is based on the principle that it is better to have incomplete quick results than slow precise results. This optimization is applicable in systems like streaming media where any loss of data is quickly superseded by newly produced data. Load shedding is inherently dynamic and relies on information about the stream processing graph as well as current rates of data between nodes. Feedback loops have been used as inputs to the shedder [113] The Borealis systems formulates load shedding as a linear optimization problem that can be solved via a centralized solver or a distributed metadata propagation approach [110].

Rate Synchronization. Rate synchronization fixes the rate of dataflow between transformers.



Above, transformer A is sending data to transformer B at an uncontrolled rate. If A's production is higher than B's consumption then latency increases in the system. This can be remedied by fixing A's rate of production and B's rate of consumption. With constant flow rates a fixed-sized buffer can be placed in-between the transformers without concern about overflow. The above diagram also splits B via fission to improve the rate of consumption.

Rate synchronization is a useful technique for limiting the memory usage of a system. The tight control it provides over a system enables many optimizations that are difficult in a more dynamic setting. StreamIt uses static flow rates to automatically fuse and split (via fission) transformers to allow for efficient placement onto CPU cores [61, 62]. The Lustre language [65] is a synchronous dataflow language which has had its synchronicity exploited for a number of optimizations. Gérard *et al.* [58] optimize memory usage by building an "interference graph" that identifies variables which cannot interact through an analysis of transformer clock rates and programmer annotations. The graph is then colored and non-interfering variables share a memory location and are updated in place. A type system on Lustre ensures that the flow rates between transformers do not require buffers guaranteeing no memory overhead [39].

3.3.3 Algorithmic Optimizations

Algorithmic optimizations rely on optimizing inside the transformer. These optimizations are dependent on the model of the transformer itself. We briefly discuss two algorithmic optimizations to provide an intuitive understanding of how to optimize a transformer.

Compiler transformations. Stream processing systems that present an transformer-level interface often benefit from standard compiler optimizations. SPADE [57] implements fusion by placing transformers on the same host and connecting them via function calls. A low-hanging optimization for any compiler is function inlining, further reducing the communication overhead of fused transformers. Shivers *et al.* [104] develop a transformer definition in the vein of continuation passing style that includes explicit continuations pointing up and down stream.

By including these continuations the compiler can fuse transformers by using existing, generally useful compiler optimization techniques.

CRA Minimization. Cost register automota (CRAs) are an automota that model streaming transductions. Discussed more thoroughly in Section 3.5.2 CRAs compute over input words and have a set of registers that store values and on each state transition the registers are updated based on the previous register values. A typical optimization for an automata is *minimization*, an algorithmic process for reducing the number of states and transitions in an automata without affecting the functionality. CRAs complexity is defined as the number of registers needed to computer the underlying function [14]. Minimizing the complexity of a CRAs with weights in group $\mathbb{G}$ over unique register operations $\otimes c$ denoted $\mathrm{CRA}_{\otimes c}(\mathbb{G})$ [46]. They introduce a "twinning property" of weighted automata that characterizes the distance between two computations and prove that automata that satisfy the twinning property compute functions that satisfy the "bounded variation property" which states that for a function f any two input words with a finitely bounded distance (differing in only a prefix) will produce outputs with a finitely bounded distance. They show that any function with the "bounded variation property can be computed by a $\mathrm{CRA}_{\otimes c}(\mathbb{G})$. They use this result to show that identifying a minimization of a $\mathrm{CRA}^{+}_{\otimes c}(\mathbb{M})$ to k registers is PSPACE-complete.

3.4 Pushing and Pulling

Stream processing systems behave differently depending on where control is located. This difference of behavior can be seen in how different stream processing tasks behave under different control schemes. Identifying the different control schemes and how they affect the stream processor and any potential optimizations opens up a promising avenue for new optimizations.

Stream processors can be given a "push" (data driven) or a "pull" (demand driven) control scheme. Intuitively a push perspective has transformers sending output values downstream when they are produced and a pull perspective has transformers requesting values from upstream when needed. The perspective chosen is determined by where control is located and the location of control has impacts on how the system operates. Regardless of whether a system explicitly identifies as push or pull all of them have an underlying model of control. There is evidence from other areas that making the choice of control scheme explicit and taking advantage of the characteristics of either technique can have benefits.

The choice of using a push or pull perspective in stream processing is often constrained by application. Push models are widespread when the processing system does not have control over the data source. An example is network packet analysis. The analyzer is unable to control when packets arrive in the network and must act reactively [44]. Pull models are more natural when the system controls the source but the requests are uncontrolled. This is a typical scenario when there is user input querying the source and the stream processor performs transformations based on the user queries. In scientific visualization the Visit software utilizes a pull-based model to render a dataset in response update requests from the renderer (e.g. when the user changes the camera angle) [38].

3.4.1 Push

In a push perspective control begins upstream. The source produces data and sends it downstream when it can. The downstream transformers react to data recieved and eagerly push any output they produce downstream. In this scheme unless batching is implemented (Section 3.3) each transformer reacts to a single value of input. In a single threaded context this amounts to passing control to the downstream transformer for processing.

Transformer	Push	Pull
Filter	No unnecessary elements passed downstream.	Unbounded recursion or skip elements (unnecessary requests upstream).
Take	Unnecessary elements passed downstream.	No unnecessary work requested from upstream.
Fold	Requires state to collect elements from upstream.	No state needed, all fold elements are collected in one pass.
Flatmap	No state required to produce many elements.	State required to store intermediate elements.
Split	No state required to push elements downstream.	State required to remember which elements to give downstream.
Merge	No ability to enforce order.	Can enforce order.

Table 2: Common stream transformers characterstics in push and pull control schemes

Push stream processors have difficulty with merging, take, and folds. For merging push streams the merge transformer has no control over when either stream produces a value. This makes imposing an order on the resulting merged stream costly. It can be done through a judicious use of state at the merge transformer or brittle multi-threading schemes [103]. The take transformer, which only reads a finite amount of data from an upstream source also incurs waste. After the desired amount of data is consumed no more data will be produced. The upstream transformer is unaware of this situation and will continue to push data downstream incurring unecessary resource usage. A fold transformer requires multiple inputs in order to produce an output. Push transformers "activate" every time an input is received. This discrepancy requires the transformer to maintain local state for an unknown amount of time until the fold is complete. Only once the fold produces an element can the state be removed. This can be mitigated through structured windowing which can guarantee an upperbound on the amount of state needed [84]. Table 2 lays out the transformer that are negatively impacted by a push-based control scheme.

There are other performance considerations to take into account. It is possible that data could arrive at a transformer before it is done processing the previous element. Unless the system is lossy this requires a buffer to exist at the consumer which can store data until the consumer is ready. The existence of this potentially unbounded buffer is undesirable as an overloaded consumer can cause the host to run out of memory. A typical solution to this is to perform load shedding (Section 3.3) by fixing a maximum size for the buffer and dropping elements with increasing frequency as the buffer fills up.

A stream processing system that adopts the push model typically fits well for applications in which the production of data is not controlled by the user. In a push-based model the user defines stream transformers as consumers that react to the actions of a producer which drives the execution of the stream processor. In push models the user is starting from some sink, whether its a rendering function or a data store and layering transforms in front of the sink, effectively building the pipeline backwards.

3.4.2 Pull

In a pull perspective control starts downstream. Instead of waiting for data to arrive, transformers wait for a request from downstream. Receiving a request will begin the computation and when upstream data is required the transformer will make a request of its own to the upstream transformer. In a single-threaded context this amounts to passing control upstream until data that satisfies the request is found. With each request the transformer is expected to produce its next value in its output stream.

Pull stream processors are less natural for the filter, split, and flatmap transformers. Filters pose a problem because downstream transformers will wait until the filter produces an element. If the input stream to the filter never satisfies its predicate then the downstream transformers will be waiting forever. Splits become less efficient in a pull setting as the two different downstream transformers may request elements at different rates. The split transformer then has to maintain state to remember which elements have only been read by a single downstream transformer. The flatmap transformer is expected to produce many values for a single input. The downstream transformer is only expecting a single value so the flatmap transformer must store the unread values that have been produced by an input. Table 2 shows the transformers which are negatively impacted by a pull-based control scheme.

In a pull stream the producer is blocked on producing until the consumer is ready. Unlike in push streams this blocking does not necessarily introduce a problem of unbounded memory usage. The source being defined by the users implies that there is some finite representation of a producer that can generate the values of the stream. This finite representation does not necessary “fill up” waiting for a request like an overloaded consumer in a push stream can. As an example, a stream of digits of pi can be produced by a representation that only contains a running sum of the current calculation that is updated on every request [96].

3.4.3 A False Dichotomy

It is not necessary to assume a single model of control for an entire stream processing system. It is possible for a system to utilize both push and pull control schemes. Indeed, there is nothing stopping a looping process from constantly requesting from a pull stream and pushing into a push stream consumer. Similarly a push stream can feasibly be converted (with careful consideration of available resources) into a pull stream by collecting produced elements into some state and waiting for requests to provide those elements. Many modern stream processing systems provide ways to connect with external sources in either a push or pull manner, depending on the source. Apache Flink [51] provides connectors to databases such as MongoDB which require pulling data at intervals. Evidence for benefits from utilizing both push and pull models can be seen in areas outside of stream processing which will be further discussed in Section 3.4.4.

Work by Shivers *et al.* [104] reifies the choice of control scheme by explicitly including continuations in the definition of transformers. Transformers contain continuations to their upstream producer(s) and downstream consumer(s) and invoke those continuations when data is either produced or read. In that work a push-based control scheme is analyzed by placing initial control at the source of the stream processor. They note that a pull-based stream could easily be created by placing initial control at the sink.

Another control scheme separate from pushing and pulling is synchronous processing exemplified by the StreamIt [111] system and the Lustre [58, 65] and Esterel [23]. In this scheme the transformers are designated fixed rates of data production and consumption. Each stream connecting transformers is given a buffer of a fixed size in which the producer pushes data to and the consumer pulls data from. This technique utilizes both a push and a pull from a shared data source. The constraint on the data rates allows the system to predetermine the buffer size needed for the transformation and stops any unbounded growth. It is tempting to consider this system one that gives control to both the producer and consumer. It is more accurate to think of control being placed at a higher level where the data rates are controlled. This control scheme can be seen in the StreaMIT [111] system as well as synchronous dataflow languages.

Where control is located in a stream processor is a necessary choice that all systems make.

This choice is constrained by the application but also has impacts on the performance of the stream processor. By making the choice of control scheme explicit and manipulating how control is used in a stream processor there is the potential to exploit the behavioral differences for efficiency benefits.

3.4.4 Beyond Streams

The push-pull dichotomy is not unique to stream processing. Where control is located in a computation is a deeper question that has been noticed and explored in a variety of different areas. Even outside of computer science push and pull effects have been noticed in migration patterns [49] and manufacturing [90]. There is evidence within computer science for the benefits of mixing push and pull in a control scheme.

Graph Processing Graph processing is the transformation of a graph into either another graph or a discrete value. Examples of graph processing algorithms are breadth first search, minimum spanning tree, page rank, and graph coloring. In graph processing pushing and pulling indicates how information is propagated through the graph. Pushing indicates that nodes send updates to their neighbors, in effect updating the neighbors state. In a pull setting nodes request updates from their neighbors and write their own state. In [25] push and pull implementations of classic graph processing algorithms are compared in a multithreaded setting for performance and behavioral properties. It is shown that choice of push or pull can have significant consequences on performance and the choice is algorithm dependent. The usefulness of both push and pull graph processing algorithms has led to work developing languages for expressing and optimizing both perspectives [114].

Rumor spreading is a technique for spreading information through a graph. It differs from graph processing by not focusing on producing an output from a graph but rather continuously disseminating updates throughout the nodes of a graph. Rumor spreading can be accomplished through a push method, in which a node with information selects a random neighbor to send the information to, or a pull method, in which a node requests an update from a random neighbor [5]. The two methods are combined in the standard push&pull protocol which has nodes performing both actions. It has been shown that the combined push&pull method disseminates updates to all nodes in the graph in logarithmic time while only utilizing a push or pull requires polynomially many rounds [37].

The graph algorithms mentioned above differ from stream processing in that the computation occurs in discrete “rounds”. All (relevant) nodes perform their push or pull updates during a round before the next round begins. This is in contrast to the streaming setting where whatever entity has control, the producer for push and the consumer for pull, may choose to engage another entity at any time. Despite this difference similar behaviors arise. The push method requires greater synchronization as there is the possibility for multiple nodes to write to the same neighbor. In contrast the pull method can require more communication as a node may request updates from neighbors without anything to share [25]. The push synchronization is reminiscent of the inability to order the elements of two joined push streams. To impose an order on the joined elements either some thread-level synchronization must occur or potentially unbounded state is maintained. The pull communication is similar to a pull stream continually requesting updates from a push stream, either by blocking or receiving empty responses [42].

Database Query Engines The performance of the push-pull dichotomy has been studied in database query engines. A pull-based technique was proposed early on [63] which was eventually superseded by push-based techniques [79, 87]. Query engines provide a good study for the different performance characteristics of pushing and pulling. Much of the early stream processing work was done by the database community and their techniques have continued to influence modern systems. Additionally, much like a graph-based interface, a query engine is an optimizer over a select set of operators.

The change from a pull-based to push-based dataflow in query engines was justified by improved cache locality and branch prediction [87]. This was shown to improve performance but required an API change for the different query operators. This change is delicate as the properties of the operators changed in different control schemes (similar to the discussion in Sections 3.4.1 and 3.4.2). To manage this change a mechanical transformation between the control schemes for the desired operators was developed [79]. The advantage of the push scheme is rejected by Shaikhha *et al.* [103]. They show that the push-based scheme was making use of standard compiler optimizations that were applicable in a pull-based scheme. They implement the same optimizations in both schemes and show comparable performance across queries.

The exploration of control schemes in database query engines is a good lesson in the difficulties of explicitly managing control. Interface design, performance, and compilers all come into play to determine the behavior of a system depending on its control scheme.

3.5 Models of Stream Processing

Now that we have sketched the shape of the stream processing problem we take a look at four different models of stream processors. We will analyze the presented models with respect to the properties laid out in the previous subsections. Does the model present a graph or transformer view of the problem? How are typical tasks defined? What optimizations are applicable? Is a control scheme captured by the model? These questions indicate what a model captures and what a model tasks the user with defining.

3.5.1 Time-Varying Relations

Earlier work in stream processing done by the database community modeled stream processors as *queries* over time-varying relations [15, 16]. This model has the benefit of fitting naturally with traditional database processing. The model transforms streams to time-varying relations using a window (stream-to-relation), then the relation is transformed using SQL queries (relation-to-relation), before finally converting back to a stream using predefined operations (relation-to-stream). The subsequent description of the time-varying relation model follows the CQL [16] semantics. This model is the most completely described and provides a good example of time-varying relations. Other streaming database systems use similar models to support relational queries [2, 36].

A time-varying relation is defined as a window over one or more input streams. The windowing scheme can be highly dependent on order. As mentioned in Section 2.2.1 streaming databases model a stream as an unordered bag of timestamped tuples, with the possibility that some tuples have the same timestamp. This creates a need for non-deterministic selection if a window cannot determine which tuples have occurred "earlier" [74].

Once the time-varying relation is defined typical SQL queries are used to manipulate the window. As the data in the stream moves in and out of the stream window, the query is rerun and updates the data in the time-varying relation. This step is where the advantage of the model really comes through. By modeling the data available to a transformer as a relation, the full power of SQL can be used for transforming the stream.

To produce the output stream the relation is transformed using a predefined relation-to-stream operation. Three example operations are `Istream`, `Dstream`, and `Rstream`. The `Istream` operation produces an output tuple whenever a new tuple is added to the relation, the `Dstream` operation produces an output tuple whenever a tuple leaves the relation, and the `Rstream` produces all the tuples in the relation at every time step.

As an example, consider the query that counts the number of packets sent from IP address `1.0.0.1`:

```
Select RStream(Count(*))
From Packets[Range 1 Minute]
Where ip_sender = "1.0.0.1"
```

Here, `Packets[Range 1 Minute]` Defines a time-varying relation over the stream of packets. The relation includes all packets seen in the last minute. The `Where` clause is standard SQL for selecting only the packets sent from address `1.0.0.1`. The `Select` clause contains a relation-to-stream operator `RStream` which will produce the count at every time step. If the `IStream` operator is used instead then output will only be produced whenever the count changes.

This model of a stream processor fits nicely in with the relational database way of thinking. It has the advantage of utilizing the powerful and known language of SQL for the transformers. By making windowing the primary concept of reading a stream, order became a primary concern of many streaming databases. Multiple different implementations have created ways for users to impose their own order when necessary on a stream. This is evidence that the model does not accurately capture the shape of the stream as ad hoc additions are needed to guarantee correct processing in the face of disorder.

3.5.2 Cost Register Automata

An interesting line of work involves modeling streaming transducers as automata. A streaming transducer is a map from input sequences to output sequences in a single pass [7]. The requirement that the transducer perform its computation in a single pass lends well to unbounded streams. Transducers only need to rely on historical data and any processed data does not need to be stored. The development of a computational model for streaming transducers led to the creation of the stream processing systems StreamQRE [12, 84] and NetQRE [116] both having similarities to regular expressions.

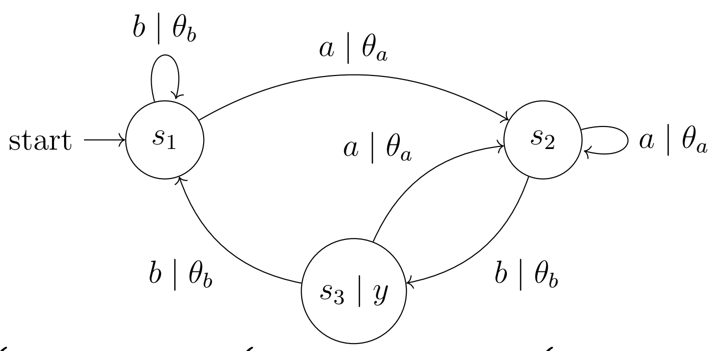
Streaming transductions were introduced by Alur in 2011 [7] and a machine-based formalization was given as cost register automata (CRAs) in 2013 [8]. CRAs are state machines that map finite inputs of tagged values to output values. A CRA has a finite number of states and data registers for storing intermediate and output values. The machine is equipped with a simple expression language for updating the values in its registers denoted $\mathbb{E}$. At every step the machine reads in a tagged value, updates it state, and updates its registers. When the machine lands in an final state, an ultimate set of register updates is performed and the output value is produced.

Definition 3.1 (Cost Register Automata)**.** A cost register automota $\mathcal{A}$ defined over a tag alphabet Σ, data values D, and expressions $\mathbb{E}$ is a tuple $\mathcal{A} = (Q, X, \Delta, I, F)$ where:

- Q is a finite set of states,
- X is a finite set of registers,
- $\Delta \subseteq Q \times \Sigma \times U_{\mathcal{O}} \times Q$ is a transition function where $U_{\mathcal{O}}$ is a set of register updates $X \to \mathbb{E}$
- $I : Q \rightharpoonup (X \to \mathbb{E})$ is an initialization function, and
- $F : Q \rightharpoonup \mathbb{E}$ is a partial finalization function.

The domain of F is the set of final or accepting states. The value of an expression $t \in \mathbb{E}$ is defined over a variable mapping $\alpha : X \to D$ denoted $[\![t]\!] : X \to D \to D$ which is the evaluation of the expression t after replacing any variables with their mapping in α.

The output value of a CRA over a data word is $[\![F(q_\omega)]\!](\alpha_\omega)$ where q_omega and α_ω are the final states of the CRA after consuming the data word.



$$\theta = \begin{cases} x := 0 \\ y := 0 \end{cases} \qquad \theta_a = \begin{cases} x := x + 1 \\ y := y \end{cases} \qquad \theta_b = \begin{cases} x := 0 \\ y := \max(y, x) \end{cases}$$

Figure 7: An example CRA over the tag alphabet $\{a, b\}$ that outputs the length of the longest seen sequence of a tagged values. The register x holds the length of the current sequence of a tagged values while register y holds the maximum length seen. State transitions are written $x \mid \theta$ with x being the tag of the next element in the data word and θ being the register update function executed on that transition.

One can view the execution of the CRA as building an expression piece by piece as a word is consumed and then evaluating that expression at the end of the computation. While the operation of a CRA is defined over a finite data word the semantics can be lifted a model of stream processing by outputting a value whenever the machine is in an accepting state. Figure 7 shows an example CRA that outputs the longest sequence of a particular tag seen.

CRAs provide a solid computational model of a stream processor and they have been used as the foundation for stream processing languages (Section 4.3.2). However there are some deficiencies. Because CRAs only produce a single value from a finite string, and the lifting of a CRA to a streaming setting does not fundamentally change this it is not obvious how to implement the concatmap operation which produces many elements from a single streaming element. To support this the systems built on top of CRAs tend to use a secondary slightly modified expression language for producing values [116], see Section 4.3.2.

Since their introduction CRAs have been extensively studied. CRAs are an extension of register automata [26,77,88] that loosen the restrictions on the expression language by providing a set of allowable operations. By restricting the expression language in various ways different classes of transductions are produced [9].

3.5.3 Monotonic Functions

Building on the model of streams as monoid discussed in Section 2.2.3 stream processors can be modeled as monotonic functions on stream prefixes, or elements of the monoid. This model is explored by Mamouras [83] and used as the semantics in the StreamQL language [80].

To capture the requirement that a transformer processes elements as they arrive, the underlying function must be monotonic. Monotonicity relies on the ordering prefixes by their extension. That is, prefix a is considered less than prefix b if a is a prefix of b. Formally, $a \leq b$ if there exists a z such that $a \cdot z = b$. A monotonic function is one that preserves order.

Definition 3.2 (Monotonic Function)**.** A function $f : A \to B$ with preorder $\leq$ on A and preorder $\preceq$ on B is monotonic if for all $x, y \in A$ $x \leq y$ implies $f(x) \preceq f(y)$.

The restriction to monotonic functions ensures that as the processor receives more input, it only appends to the output, or, a processor cannot retract any output.

In this model, stream processors (called *stream transductions*) are pairs of monotonic functions and *monotonicity witness functions*. The witness function provides information about how to compute the output without storing the entire history of the stream.

Definition 3.3 (Stream Transduction)**.** Given monoids A and B, a stream transduction from A to B is a pair (β, μ) where $\beta : A \to B$ is a monotonic function with respect to the prefix preorder and $\mu : \Pi_{x,y \in A} \mathit{prefix}(x, y) \to \mathit{prefix}(\beta(x), \beta(y)$ is the monotonicity witness function. Where $\mathit{prefix}(x, y)$ function computes the sequence that is concatenated to x to produce y. That is, given $x, y, z \in A$ and $x \leq y$ and $x \cdot z = y$ then $\mathit{prefix}(x, y) = z$.

With the monotonicity witness function a stream transducer does not need to store seen elements indefinitely.

Monotonic functions present a transformer-based interface for defining stream processors. Each function is reasoning about the individual elements of the carrier set of the input monoid. Basing stream processors on set functions provides an extremely flexible model but doesn't capture much about the computational component of the processor. Consequently, any optimizations that are done will be algorithmic, depending on the implementation of the function. Mamouras gives a computational model with a few transformers in the form of an abstract state machine and shows to use the denotational model of stream transductions to prove the correctness of optimizations algebraically.

Monotonic functions present a push model of control. The functions are consumers of streams (more specifically stream prefixes) and functions are defined with respect to the finite input prefixes. To ensure the properties of a stream are appropriately captured, restrictions are placed on the stream processor rather than the stream itself. The monoid does not provide structure about the order of produced elements.

The model of stream processors as monotonic functions over monoids provides nice algebraic reasoning properties and can be used as the basis for a stream type system.

3.5.4 Nested Fixed Points

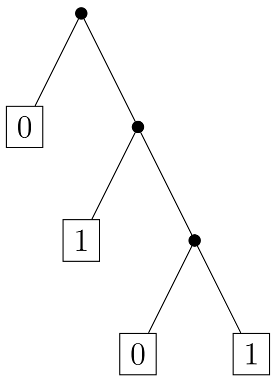
Ghani *et al.* [59] introduced a novel model for a stream processor represented by a mixed inductive, coinductive tree. The model uses a well-founded tree to represent a *discrete* function that operates on a finite prefix of the stream. The tree is wrapped in a coinductive type to enable productive repetition of the prefix computation, enabling *continuous* stream functions. Treating stream processors as trees provides a useful reasoning model for the productivity of the stream processor, ensuring that any processor will eventually produce some output.

Stream Processors are represented by a mix of inductive and coinductive types [59]. Stream processors are represented as continuous functions from a stream to an output type $f : A^* \Rightarrow X$. Here, the $\Rightarrow$ refers to continuous functions where continuity means finite information of the output is determined by finite information of the input. The output type X can either be a discrete space B or a continuous space of streams B^*. Continuity is an important property of these functions as it ensures the productivity of the function. Values must be produced after a finite amount of input has been consumed.

Functions into a discrete space can be inductively defined by well-founded trees. Beginning at the root of the tree, consume elements of the stream following the structure of the tree until a leaf containing the output is reached. Continuity is maintained by the requirement that the tree is well-founded therefore a leaf will be reached after a finite amount of input.

The tree representation of a finite function starts at the root and branches on successive elements of the stream. When you eventually reach a leaf you have the return value of your function.

Figure 8 shows an example function on a stream of 1s and 0s. At each node the next element of the stream is processed by branching to the left on 0s and to the right on 1s. When



$$
\begin{aligned}
&f : 2^\omega \to 2 \\
&f(0, \ldots) &&= 0 \\
&f(1, 0, \ldots) &&= 1 \\
&f(1, 1, 0, \ldots) &&= 0 \\
&f(1, 1, 1, \ldots) &&= 1
\end{aligned}
$$

Figure 8: A discrete function represented as a tree

a leaf node is reached the function returns the value of the node. A discrete function $A^* \Rightarrow B$ represented by a tree has type $T_A B \triangleq \mu X.B + X^A$. The μX indicates that this is an inductive type on X and that it can be constructed by either providing an element of type B (indicating a leaf in a tree) or by providing a function from A, the type of the stream elements, to a subtree (indicating a node).

These discrete functions correspond to transforming a prefix of the stream into a single element. These can be considered similar to a fold from Section 3.2. A discrete function is capable of ingesting many elements from a stream and producing a single output. The state of the fold is managed by the definition of the tree. A nice property of this definition of a fold is that the function is guaranteed to terminate by the necessity that the tree is finite.

Continuous functions transform a stream into another stream. To do this the well-foundedness property of the tree is removed and cycles are introduced. These are represented by nesting the discrete functions in a coinductive structure. Instead of the leaves of the tree containing a value they now contain a value as well as another coinductive tree. This is represented in the type: $P_A B \triangleq \nu X.T_A(B \times X)$. Here the νX indicates a coinductive type on X. The type is a discrete function over a stream of As. The discrete function produces an element and another stream processor $(B \times X)$.

Figure 9 shows a stream processor as a non-wellfounded tree. The tree is conceptually infinite, if one were to walk the tree they would cycle forever. However the presence of coinductive node at the leafs provides a stopping point for the evaluation of the stream processor. A new processor is returned along with a produced element. This new processor is then waiting until the caller evaluates it. This satisfies the coinductive definition of productivity discussed in Section 2.2.2. Every invocation of the processor will produce something in finite time (it is evaluating a productive discrete function).

When this model of stream processors was introduced Ghani *et al.* [59] gave a proof of completeness (all stream processors are representable) and define composition of stream processors. The tree model allows for an infinite number of representations for every function. Garner [55] addresses this by providing a characterization of *extensional* stream processors allowing for proofs of equivalence between stream processors.

The model of stream processors as non-wellfounded trees guarantees that the processor is productive. It also provides a way to flexibly define folds over arbitrary windows. These properties come with some downsides. The stream processor is entirely static, a user must define the tree up front accounting for all possible stream elements. Additionally, while it is easy to read multiple elements from the input stream. Producing multiple elements to the output stream is more tedious and relies on the composition of single node trees.

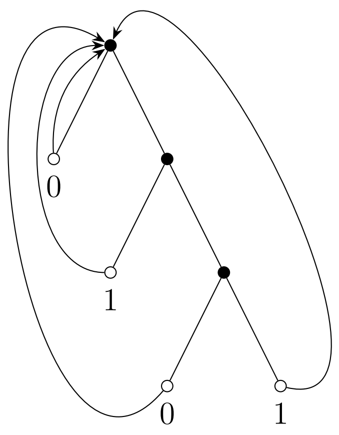


Figure 9: A stream processor represented by nested fixed points. The coinductive nodes are indicated by hollow circles that point back to the root of the tree.

3.5.5 Process Networks

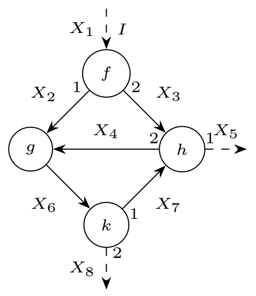
Process networks were introduced by Kahn as a way to model parallel programming [60]. They subsequently became the basis for the dataflow programming paradigm exemplified by the languages Lustre [28,58,65], Lucid [115] and Esterel [23]. Kahn's goal with introducing process networks was to produce a parallel programming paradigm that had good reasoning properties. By modeling a program as a set of transformers executing in parallel connected by streams he preempted modern distributed streaming frameworks by 30 years.

A process network is a dataflow graph of "computing stations" (transformers) and "channels" (streams). A transformer with n input streams is modeled by a function $f_i : D_1^\omega \times D_2^\omega \times ... D_n^\omega$ with D_i^ω being a finite sequence of the ith input stream over data elements D. Sequences D^ω are ordered in the usual way (prefixes of a sequence being less than the sequence. See Section 2.2.3). Any increasing chain ξ in D^ω, $X_1 \leq X_2 \leq ... \leq X_n \leq ...$, has a least upper bound called $\lim(\xi)$. Transformer functions are constrained to being continuous.

Definition 3.4 (Continuous Function)**.** A function $f : A \rightarrow B$ for complete partial orders A and B is continuous iff for any increasing chain ξ of A $f(\lim \xi) = \lim(f(\xi))$.

This constraint that the functions are continuous means that a transformer will send output before reading the entire stream, preventing a blockage in the dataflow. Continuous functions are also *monotonic* (Section 2.2.3) which ensures that the transformers cannot retract any output and future output only depends on future input. In other words, monotonicity enables incremental processing.

Process networks were designed for proving properties of parallel programs. To accomplish this the network is viewed as a system of transformer equations. Each output stream of a transformer is equal to the transformers function applied to its inputs. Figure 10 shows an example process network and the associated system of equations. This system of equations is then solved for the least fixed point of one of the output streams. Solving the system of equations provides a proof of the behavior of an output stream.



$$\Sigma_P = \begin{cases} X_1 = I \\ X_2 = f_1(X_1) \\ X_3 = f_2(X_1) \\ X_4 = h_1(X_3, X_7) \\ X_5 = h_2(X_3, X_7) \\ X_6 = g(X_2, X_4) \\ X_7 = k_1(X_6) \\ X_8 = k_2(X_6) \end{cases}$$

Figure 10: An example process network P and its system of equations Σ_P. Each stream is labeled X_i and its output is equal to the function for the associated computing station applied to the input streams of the computing station.

Model	Interface	Optimizations	Push/Pull
Time-Varying Relation	Graph	Graph/Placement	Push
CRAs	Transformer	Algorithmic	Either
Process Networks	Transformer	Graph/Algorithmic	Push
Monotonic Functions	Transformer	Algorithmic	Push
Nested Fixed Points	Transformer	Algorithmic	Pull

Table 3: Stream processing models and their properties.

Process networks are an early version of a graph-level interface for stream processing. They provide an example of how reasoning at the level of the dataflow graph can help with proving the correctness and safety of the system. In the original process networks the definition of the transformers is given in a dialect of Algol with keywords for waiting for and producing streaming data. This does not appropriately capture the abstraction of a streaming transformer as the reasoning properties require transformers to be limited to continuous functions but nothing in the expression of the stream process constrains the user.

3.6 Summary

Table 3 summarizes the discussed stream processing models and where they land in the framework outlined above. Each model presents either a graph or transformer based interface. With most modeling a stream processor as a transformer. Despite this, the stream processing systems that build on these models typically provide some sort of composition, allowing for reuse and the definition of a graph. This penchant for focusing at the transformer level means that few models are able to safely capture graph-level optimizations, with most relying on model, specific optimizations. None of the models explicitly focus on a control scheme but with the notable exception of CRAs they all require either a push or pull scheme. For CRAs it is possible to think of the automata as reacting to inputs from the stream or requesting data from a source.

4 Stream Processors

In this section we review stream processing systems as they have evolved with their expanding use-cases. While these systems have been in a process of continual reiteration to meet the new demands imposed by a more digital world, the expression of stream processing pipelines has fallen behind. Early systems relied on models of streaming adapted from relational databases while newer systems with a wider range of applications use preexisting languages and constructs to express their transformations.

Despite the lead gained by the systems community, there are promising avenues being explored in the expression of stream processing systems. The development of new systems has brought the requirements of stream processing into clearer view and new languages with solid semantic foundations are being developed. In this section we explore the changing landscape of stream-processing systems and how stream processing languages have evolved alongside to express the growing needs of these systems.

4.1 Related Surveys

The wide range of stream processing systems has led to a need for organization and categorization of the space. The increased production of research into stream processing has increased the number of surveys of the space. While early stream processing surveys could attempt to review the entire field [108] newer surveys tend to focus on specific aspects such as optimizations [47, 112] or how the field has evolved [31, 53].

Most surveys focus on the explosion of stream processing systems that have been developed since the early 00's. Fragkoulis *et al.* [53] and Carbone *et al.* [31] review the evolution of systems arising from the database community to the large-scale distributed frameworks we see today. Fragkoulis analyzes how the implementation of specific functionality provided by stream processing frameworks has evolved over time. Carbone explores how the newer distributed stream processing frameworks are designed and being used for different applications than the relational models expected arguing that the database community should attempt to modernize their understanding of stream processing. Fragkoulis also defines a "third generation" of stream processing systems, with the first two generations being streaming databases and distributed stream processing systems respectively. The third generation is not clearly defined but is characterized by a move to light-weight stream processors designed to run in a serverless setting or be hardware aware. This presents an interesting trend but the underlying stream processing model remains the same as the distributed stream processing systems.

Selecting a stream processing system is made difficult by the large number of available options. A few surveys attempt to alleviate this problem by focusing on the differences in functionality between systems. Isah *et al.* [70] classify distributed stream processing systems by a set of 12 features ranging from technical details such as messages guarantees to external aspects like community support. They use these features to provide recommendations for selecting a distributed stream processing framework given an application. Similarly Dayarathna *et al.* [47] organize stream processing systems by their intended use case in order to provide recommendations for selecting a system for practitioners.

Many surveys focus on specific aspects of stream processing systems. To *et al.* [112] review state management in distributed data processing systems, a notoriously tricky topic. They classify state by what elements of a stream processing system need access to the state (e.g. configuration data is shared among all transformers). Systems are inspected for how they implemented various operations on state. Hirzel *et al.* 2018 [67] explore languages for expressing stream processing pipelines. They provide three requirements divided into four principles which stream processing languages should meet. Languages are then compared with how they meet the given principles. Optimizations are an area of concern for stream processing systems. Hirzel *et al.* 2014 [68] explores a variety of stream processing optimizations and their implementation in the literature. Röger *et al.* [97] survey parallelization techniques in stream processing. They divide the parallelization problem into two challenges: parallelizing processing within transformers, and dynamically adjusting the level of parallelization based on the flow rate of data.

Stream processing systems are categorized by aspects which enable different parallelization techniques.

There are relatively fewer surveys on the theory of streams and stream processing. An early survey by Stephens *et al.* [108] written in 1997 reviews the existing systems in an push towards providing a general theory of stream processing. Special attention is paid to the semantic models of the systems. A in-depth collection of stream processing techniques is presented by Garofalakis *et al.* [56]. That collection contains a deep dive into various systems developed in the early 00's, before the rise of large distributed stream processing frameworks.

4.2 Stream Processing Systems

To understand how to correctly capture a stream processing abstraction we consider two existing paradigms of high-performance stream processing systems. We start with a discussion of the properties of streaming databases (Data Stream Management Systems) and see how the needs of stream processing have evolved beyond their model into today's large scale distributed streaming systems which lack a solid semantic foundation.

4.2.1 Data Stream Management Systems

Data stream management systems (DSMSs) are an adaptation of relational database systems and SQL-like languages to a streaming setting. A surge of work was done in the 2000s introducing influential systems such as Stanford's STREAM [86], Aurora [2], Borealis, TelegraphCQ [33], and NiagaraCQ [36] with accompanying languages [15, 16, 20, 74] and performance optimization techniques [18, 19, 81, 110].

DSMSs are based around the idea of *querying* streams to gain insights into the data contained within. This perspective views the eventual use of a stream processor as providing up-to-date aggregate information based off of continuous data streams. This viewpoint is seen throughout the examples given in DSMS papers from determining a group of soldiers "center of mass" [2] to updates whenever a stock price makes a dramatic shift [36]. This narrow view limited the applications that this early crop of stream processors targeted. Despite this these systems are still usable as general stream processors.

To express streaming queries SQL-based languages was a popular choice [16, 33]. SQL provided a known and powerful language for querying data that could be leveraged in a streaming setting. Both STREAM [86] and NiagaraCQ [34] focus on converting streams into relations using windows and utilizing SQL (or a slight modification) to perform the transformations. Other systems deviated further from SQL to give their own (SQL-reminiscent) languages for defining queries.

Bai *et al.* [20] proposed a query language "Expressive Stream Language" (ESL) that uses SQL to modify a transformers internal state based on the movement of tuples. ESL is split into `Initialize`, `Iterate`, `Terminate`, and `Expire` clauses. The `Initialize` clause initializes any necessary state for an aggregation (e.g setting a count to 0 or creating a temporary table). The `Iterate` clause contains standard SQL that defines the logic to be performed whenever a new tuple arrives. The `Terminate` clause is a blocking clause that contains SQL to return a final result when the entire input is read. The blocking nature of `Terminate` means it must be used on finite data or within a window. The `Expire` clause is used to update the state based on tuples leaving a window.

The Aurora system [2] deviates from SQL clauses but provides a set of operators familiar to anyone who uses SQL. These operators are treated as functions from stream to stream with extra parameters needed to configure the transformation logic. Familiar operators such as `filter`, `map`, and `join` are joined by more exotic ones. Worth noting are the `BSort` operator which attempts a best-effort sort on subsequences of a stream and `Resample` which attempts to align pairs of streams. Operators which rely on an ordering of the stream tuples are given a user defined ordering which is assumed to hold up to some bounded disordering. If the assumption is broken then the operator produces incorrect results.

The common thread of these stream processing languages is the predefinition of clauses and operators. All these systems present a **graph-based** interface for the user. The influence from SQL query engines is clear to see. By providing a set of operators up front, performance optimizations can be automatically applied in a query scheduler.

Generally, DSMSs follow a **push based** control scheme [1, 2, 16, 33]. The elements in a window updates whenever new data arrives and the relational queries reactively transform the data in the window producing the time-varying relation. NiagaraCQ presents an interesting alternative where data sources are declared as push-based or pull-based and NiagaraCQ will either react or periodically check the different source types respectively [36].

While research activity focusing on streaming databases has died down the influence of the work remains with many stream processing systems adopting the terminology and models. Stream transformations are often referred to as queries [84, 116] while systems such as Apache Spark view streams as append-only tables of data. SQL remains a popular basis for declarative stream languages with newer systems such as Apache Spark [107] and Apache Samza [89] maintaining a SQL interface.

4.2.2 Distributed Streaming Frameworks

As the scale of software systems increased the volume of data that those systems began to process rose dramatically. To handle the large quantities of data that were being processed it become helpful to conceptualize large data sources as potentially infinite. Traditional Large-scale data processing systems such as MapReduce [48] and Dryad [71] were replaced with streaming systems [40] that modeled infinite data sources and exploited incremental processing for performance benefits. The current crop of mature distributed big data processing systems are all built around streaming semantics. The use-cases of these systems differed from the older DSMSs. Instead of querying data streams for aggregate data, these systems are used for event-driven applications and continuous ETL operations [31].

The focus on performance and the distributed nature of these systems biases them towards exposing a **graph level** interface. The main interface exposed by these frameworks typically predefines a set of transformer schema, similar to configurable queries in a relational language, that constrict the behavior of the particular transformer [6, 51, 107, 109]. Users define the logic of the transformer in an existing language such as Java or Python by implementing a given interface. Examples are map functions which modify singular stream elements and aggregations which compress multiple input elements into single output elements. While the different frameworks provide slightly different sets of transformer schema they generally fit into the common operations described in Section 3.2. The set of possible transformers is not necessarily minimal as redundancy allows for optimization opportunities. This graph level interface also maps nicely to the assumed distributed setting for these frameworks. Different transformers can automatically placed on different hosts to distribute workload.

While the previously described API is the main interface for most systems many also expose separate APIs. Apache Samza has three APIs: a graph-level API as described above, a low level API that allows users to define custom transformers, and a SQL API [99]. Apache Spark exposes a graph-level API as well as has a SQL interface [106]. Apache Flink's Datastream API allows for user-defined transformers and it also exposes a SQL interface [52]. Apache Storm has two mature APIs one at the graph-level and another at the transformer level and two experimental APIs: a SQL interface and another graph-level interface.

It is telling that both the main programming interfaces for these systems are fairly similar and that many of them feel the need to expose different interfaces. The commonalities are suggestive that these systems are attempting to achieve similar goals of making common transformations easy to write and performant. The desire to add other interfaces hints at missing expressiveness with the current pattern. The ubiquity of SQL in particular shows the effectiveness of that abstraction at expressing complex transformations while maintaining performance.

In the name of performance these systems adopt a **push-based** control scheme. Data is pushed through the system as fast as possible and excess volume is handled through parallelization or load-shedding. These systems explicitly ignore the definition of sources and sinks and solely focus on the stream processing graph. Sources and sinks are considered external and must be configured to send or receive from the stream processing graph. Sources can be interacted with in either a push or a pull control scheme. Apache Flink allows static databases to be sources that must be pulled from [51]. Apache Beam provides built in functionality for repeatedly pulling from an arbitrary web API [21].

These systems expand on the DSMSs model of stream processing to approach a different scale of problem. Instead of needing to query continually updating datasets stream processing has become an enabler for large-scale distributed applications. While the programming model has started to converge towards providing common stream transformation patterns the support for auxiliary APIs shows opportunity for a strong abstraction that provides the desired flexibility and optimizations possibilities.

4.3 Stream Processing Languages

Expressing a stream processor depends on the underlying model being adopted. In this section we explore three different classes of streaming languages that are born from the previously discussed models of stream processors (Section 3.5). By comparing the different implementations of stream processing models it becomes clear where the models lack specification of the problem. If two stream processing languages using the same underlying model allow for different behavior then the model does not entirely capture that behavior. This is not necessarily a deficiency of the model as some aspects of implementation would be unnecessary to include in the model. However a comparative analysis allows us to see the limits of the model more clearly.

4.3.1 Dataflow Programming

Dataflow programming is a paradigm that rose out of Kahn's work on process networks [60]. Dataflow languages define reactive processes that change as their environment changes. The input data from the environment is treated as a stream of values represented by a variable. When the input data changes the process reacts performing some computation and producing output.

While the dataflow graph is used for proving the correctness of a dataflow program, these languages lack the ability to rearrange the graph for optimization purposes. By having the programmer define the transformers directly it becomes difficult to know statically if a transformer is stateful, stateless, and its relationship with its neighbors. This makes graph rearrangement and placement optimizations of a weakness of dataflow languages for streaming purposes.

The language Lucid [115] provides an early example of a reactive programming language. Later on, reactive systems where found to be effective models for lower level systems such as signal processing or process control. These systems placed great importance on correctness and the parallel nature of a reactive system was tamed with synchronous dataflow languages. Esterel [23,29] was the first of the synchronous dataflow languages and was quickly followed by others [22,65]. The Lustre language in particular is still being used and modified today [28,58].

In the following we discuss Lucid as an early example of dataflow programming as well as a recent language Delta [45] that focuses on typing in a dataflow programming paradigm.

Lucid. Lucid was born out of a desire to express typical programs, in particular iterative programs, could be expressed in a declarative manner to aid in verification [115]. In the process of developing the language it become a dataflow language in the style of Kahn's process networks.

In Lucid a variable represents a stream of values. Streams are defined by giving the first value of the stream followed by an expression for generating subsequent values. For example the stream of natural numbers is defined by:

```
nats = 0 fby nats + 1
```

The `fby` operator defines the stream by providing the left-hand side as the head of the stream and generating the rest of the stream via the right-hand operand. There is an interesting case of recursion in this definition. On the left side of the assignment `nats` is being defined as a stream but on the right side `nats` is being added to an number. This is possible because the value of `nats` in an expression is the first element of its stream. So `nats` is a `0` followed by the old value of `nats` plus one. The `fby` operator represents an iterative process for generating values based on expressions.

Lucid has a notion of synchronization where variables in the same clause are considered to be updated simultaneously. For example, the program to compute the Fibonacci numbers uses two variables that are updated at the same time.

```
f
where
    f = 1 fby g + f
    g = 0 fby f
end
```

This program computes the Fibonacci numbers in variable `f`. The previous two numbers are stored in the current value of `f` as well as the auxiliary variable `g`. It is important that `f` and `g` are both updated at the same time. Without this constraint the program will produce the wrong output. For example if `g` is updated first, then it will take on the value `1` (the current value of `f`) and then `f` will be updated with `2`. The stream represented by `f` would then be then $1, 2, 4, 8, ...$

Lucid presents a **transformer-level** interface. Transformers are defined by functions that update whenever their inputs update. The dataflow graph is created by through function calls in which the output stream of a transformer is then sent as input to another transformer. It is clear that lucid also uses a **push-based** control scheme as transformers react to changes in their variables, propagating the change through their computations.

Delta A newer language which can be seen as a type of dataflow programming is described by Cutler *et al* [45]. The paper presents a calculus and a type system for guaranteeing deterministic stream processing despite any non-determinism arising from parallel processing. These guarantees are made by capturing the structure of the stream in types and forcing the programmer to handle any disorder.

The calculus uses a *push* model in which syntactic expressions represent transformations on prefixes of input streams. Types represent the type of the input streams the expression uses as well as the type of the stream the expression outputs. A user can write then write an expression representing a stream processing pipeline and typecheck the expression to ensure that the processing is deterministic.

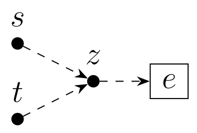


Figure 11: A parallel pair typed variable $z : s \mid\mid t$ being used in expression e

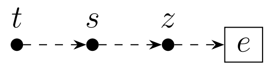


Figure 12: A sequential pair typed variable $z : s \cdot t$ being used in expression e

Figure 13: Stream Pair Types

The type system is based on two different pairings of streams. The first is a parallel pair (written $(s \mid\mid t)$) which represents two streams which have no guarantees on the relative order of their elements. The second type is a sequential pair (written $(s \cdot t)$) which ensures that every element of the first stream (of type s) will arrive before the second stream (of type t). The distinction between the pair types is used to ensure determinism. If an expression uses a parallel pair then it has to account for all possible interleavings of both streams prefixes.

Type contexts provide a stream type for all variables in a program. Contexts can also be paired either sequentially or in parallel. A parallel context pairing $(\Gamma; \Delta)$ indicates that the inputs described by Γ and Δ can arrive in any order. A sequential pairing (Γ, Δ) indicates that all the inputs (streams) described by Γ will arrive before any inputs described by Δ. These different pairings have different structural rules in the calculus. In general you have more freedom to reorder and duplicate parallel context pairings than sequential ones.

The operational semantics are based on *environments* which represent prefixes for all variables in the program. The semantics uses a judgement of the form $\eta \Rightarrow e \downarrow e' \Rightarrow p$. This judgement states that "running e on environment η will produce output prefix p and step to term e'. The syntactic transformation of terms is only used to remove streams that have been fully read. In a way the programming model has a context of stream variables. These variables can be included in any expression and the expression handles any elements that are produced by the free variables in the expression.

The core of the calculus only considers finite streams. However they extend the calculus with a "star" operator similar to a Kleene star from regular expressions. The type $s^\star$ indicates 0 or more finite streams of type s concatenated together in sequential pairs. The typing rules, environments, and operational semantics are updated for the star types.

For stateful transformations the calculus has to wait for the appropriate stream to push the desired element. This is accomplished through a historical context which contains a buffered "history" of the streams in the environment. The type of the history are plain inductive lists which have been built from the pushed elements. The waiting is accomplished through blocking operations. This has the unfortunate effect of creating potentially unbounded space requirements depending on how slow the stream being waited on is.

Delta's unique type system helps a programmer deal with the pitfalls of stream processing by identifying potentially problematic situations and alerting the programmer. Living in the dataflow paradigm Delta exposes a **transformer-level** interface to the user. However the type system shows promise for providing guarantees that can be used for graph-level optimizing transformations.

4.3.2 Quantitative Regular Expressions

Quantitative regular expressions (QREs) are a streaming query language built on top of CRAs. The definition of an automata for streaming transductions naturally leads to the question of if there is a regular language that can express the class of transductions defined by CRAs. Research into this question has led to *Quantitative Regular Expressions* [10, 11]. QREs are a language similar to regular-expressions that can be evaluated over infinite streams rather than just finite data words. QREs have been applied to stream processing in the StreamQRE [12,84] and NetQRE [116].

The strong semantic underpinnings of QRE-based languages enable a unique programming model that allows the user to ignore any explicit state-management and think exclusively in terms of patterns over stream elements. The languages also have attractive performance properties having shown to be faster than select distributed streaming frameworks such as Apache Flink [84], albeit on a single machine. The theoretical background also allows QRE compilers to have strong bounds on space usage. There has been modest uptake with StreamQRE being applied to health sensor data [3, 13].

StreamQRE. StreamQRE [84] is a declarative language for defining QREs over unbounded data streams. The language acts similarly to CQl in that a QRE defines a sliding window which is analyzed until an output is produced. However, instead of the QRE producing a time-varying table the output is immediately passed along to whatever system is consuming the data.

StreamQRE presents a **graph-view** interface which consists of a select set of combinators used for defining queries over unbounded streams. A QRE is then compiled into a streaming algorithm that computes the desired output stream. The selection of combinators is delicate as it has to maintain two properties. The first is that the QRE is unambiguous, that is there is a unique way to parse the expression. The second is that the data contained in the registers of the underlying CRA are maintained in constant space despite the size of the received input stream. Similarly to streaming SQL variants, StreamQRE Combinators can be seen as stream transformers and can be used as inputs to other combinators. Typical streaming operations are provided such as splits, windowing (iteration), and joins (combination).

As an example consider a query that counts the number of data items above a value `K`.

```
aboveK: QRE<Nat, Nat> = atom(x -> x > K)
count: QRE<Nat, Nat> = iter(aboveK, 0, (x,y) -> x + 1)
```

The query `aboveK` matches on single elements (`atom`) when the element (`x`) has a value above `K` (`x > K`). `aboveK` is used in the query `count` uses the builtin `iter` construct to continually match on `aboveK`. The second parameter to `iter` (`0`) is the initial value of an accumulator while the third parameter is an update function based on the matched element and the current value of the accumulator which adds one to the current accumulator (`(x,y) -> x + 1`).

Conceptually and technically the control scheme for StreamQRE is **push-based**. As new elements arrive the compiled algorithm appends to the data in its window until an output can be produced. Once an output is produced the contents of the window are discarded until new elements arrive. In the example above the `iter` construct produces counts as new stream elements arrive that match the `aboveK` query. The push-based control scheme places pressure on the compiled streaming algorithm. The goal of an upper bound on space needed for executing a query means the algorithms must process data faster than it arrives.

NetQRE. NetQRE [116] is a declarative language focused on quantitative network queries over incoming packets. Similar to StreamQRE, NetQRE exposes a **graph-level** interface through a set of builtin modular queries.

NetQRE implements and extension of QREs with parameters called *parameterized quantitative regular expressions* (PQREs). PQREs allow the expression matching language to contain typed parameters. If any value inhabiting the type of the parameter is matched then the expression matches. Expressions can be divided into conditional expressions of the form `exp ? exp1`. Conditional expressions check if `exp` matches on the input stream, if so then the `exp1` is used to produce an output, otherwise an undefined value `undef` is produced. As an example consider a query which only forwards packets with source IP: `1.0.0.1`.

```
/.*[srcip='1.0.0.1']/?last
```

The expression before the `?` consists of a Kleene star over any packets (`.*`) followed by a packet with a source of `1.0.0.1` (`[srcip='1.0.0.1']`). This matches any stream that ends with a packet coming from `1.0.0.1`. The expression after the `?` uses a builtin `last` which produces the last element of the stream.

Parameterization allows NetQRE queries to be expressed over a range of values. As an example of parameterization, consider a query that counts the number of distinct IP address in a stream.

```
sum{/.*[srcip=x].*/?1:0 | IP x}
```

Here the predicate `/.*[srcip=x].*/` iterates over all possible values of `x` (all possible IPs in this case) and matches if any of them exist in the stream. The expression after the `?` produces a `1` if the IP exists and a `0` otherwise (denoted by the `:` operator). Finally the builtin `sum` operator aggregates the stream into a single sum.

The NetQRE runtime manages the movement of data through the system with a **push-based** control scheme. The runtime receives and preprocesses network packets before pushing them to the NetQRE program. This control scheme fits well with the intended use-case of NetQRE which is to monitor a stream of packets act on specific events.

4.3.3 StreamQL

StreamQL [80] is a newer language based on combining and manipulating stream processors. While the language provides a comprehensive set of constructs for defining stream transformers, it distinguishes itself by providing powerful tools for composing the stream processing graph directly and allowing the user to manipulate the graph temporally.

StreamQL consists of 23 queries broken into 4 categories: relational, dataflow, temporal, and user-defined constructs. Through these queries a user can define standard transformers such as a map or a filter, or compose the dataflow graph with parallel and sequential composition. StreamQL also provides a `userDefined` query which gives the programmer fine-grained control to define very general functionality. This gives StreamQL both **graph-level** and a **transformer-level** interface.

The most unique part of StreamQL are the temporal constructs. These are queries that explicitly manipulate subsequences of the stream to modify the stream processing graph. The subsequences are indicated by a special end-of-stream marker $\triangleleft$ which indicates that the stream has halted. An example is the `take(n)` query which only produces the first n items of input and then halts, producing a finite stream. The `take` query can be combined with the temporal sequencing query `seq(f,g)` which runs the first query, f, until it halts then runs the second query, g.

Consider a query which attempts to identify IP addresses that are sending bursts of large packets.

```
groupBy(p -> p.senderIP,
        iter(seq(search(p -> p.numBytes > 1250),
                 takeUntil(p -> p.numBytes <= 1250 >>
                 reduce(0, (x,y) -> x + 1)))))
```

This query splits a packet stream by senderIP (`groupBy(p -> p.senderIP, ...)`) and waits until it detects a packet larger than 1250 bytes (`search(p -> p.bytes > 1250)`). Then it counts the number of packets sent over the byte threshold using `takeUntil` and `reduce`. This query makes

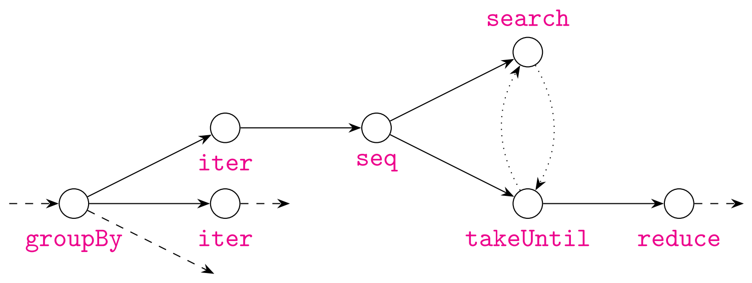


Figure 14: A dataflow graph of a streamQL query. The dotted lines indicate temporal sequencing.

use of a number of composition operations. The `>>` operator represents sequential composition and is used in this query to forward the packet stream above the threshold to the aggregation query `reduce`. The `seq` query runs the `search` until it finds a packet over the threshold then runs the composed `takeUntil` and `reduce` queries to produce a count. The `iter` query repeats the count of elements over the threshold continually. Figure 14 shows the stream processing graph for the above query. The dotted lines between the `search` and `takeUntil` transformers indicate temporal sequencing which shifts where the output of the `seq` transformers is sent.

The ability to manipulate the stream processing graph temporally is a powerful tool that allows the user to define *dynamic* transformations of the stream processing graph only using syntax. This is an exciting innovation that can be used to define much of the dynamic behavior currently being used in distributed stream processing systems such as load-balancing.

StreamQL's semantics are based on monotonic functions between monoids. Every built-in query has an analogous function between subsequences of a stream. These semantics allowed the language designers to identify and plug gaps in the language. The language provides a `flatten` query which processes a stream of lists and emits the list contents one at a time. Without the `flatten` query the language would not be able to produce multiple output elements given a single input element.

StreamQL uses a **push-based** control scheme. This has some consequences for its implementation. The paper implements StreamQL as a Java library. Within StreamQL there is no way to define a source (stream) so the user is responsible for passing each element to the query. In the paper this is done by defining a Java `Iterator` and using a loop to collect elements and send them through the query. The manual manipulation of the stream presents a weakness of the language. The language does not entirely stand on its own as it requires the user to reach into a general-purpose programming language to drive the stream processor.

Despite requiring a host language, StreamQL presents an exciting evolution of stream processing languages. By providing a way to manipulate stream processors temporally dynamic behavior can be captured in a natural way. There are still some aspects of stream processing that elude StreamQL such as defining streams directly.

4.3.4 Calculi

A number of calculi have been developed that focus on stream processing. Calculi provide a minimalist model of computation that provides a simple basis for studying properties of programming languages. Often calculi are used as intermediate languages in compilers as optimizing transformations are simpler to prove correct.

$\Lambda\mu$-calculus An interesting calculus of streams was discovered by Saurin while extending the $\lambda\mu$-calculus. The $\lambda\mu$ calculus was introduced by Parigot as a term language for classical

Syntax

$$M ::= x \mid \lambda x.M \mid \mu\alpha.M \mid (M)\alpha \mid (M)M$$

Reduction

$$\begin{array}{llll}
(\lambda x.M)(N) & \to_{\beta} & M[N/x] & \\
\lambda x.(Mx) & \to_{\eta} & M & x \notin FV_t(M) \\
(\mu\alpha.M)\beta & \to_{\beta_s} & M[\beta/\alpha] & \\
(\mu\alpha.M\alpha) & \to_{\eta_s} & M & \alpha \notin FV_s(M) \\
\mu\alpha.M & \to_{fst} & \lambda x.\mu\alpha.M[(U)x\alpha/(U)\alpha] & x \notin FV_t(M)
\end{array}$$

Figure 15: The $\Lambda\mu$-calculus

natural deduction a system for writing proofs in classical logic. [93]. It is an extension of lambda calculus in which terms can be labeled with a name and a μ binder allows substitution to occur at every term with a particular name. Parigot noted that because the reduction rule the μ binder did not remove the binder it acts like a function that accepts an infinite number of arguments. Saurin modified the calculus so that it satisfies the separation property [2] and can be considered a calculus of streams [100].

Figure 15 shows the syntax and semantics of the $\Lambda\mu$-calculus. The substitution $[(U)x\alpha/(U)\alpha]$ is read as replacing all occurrences $(U)\alpha$ with $(U)x\alpha$ for an arbitrary term U. The calculus extends λ-calculus with *continuations* (denoted by Greek letters) and a new μ binder which can be thought of as a function accepting an infinite stream of terms. The β and η reduction rules are lifted directly from the λ-calculus. Three reduction rules are added β_s and eta_s and fst. The β_s and η_s rules are extensions of the typical β and η rules from lambda calculus extended to streams accepted by the new μ binder. The reduction involved in the β_s rule is a simple renaming of a continuation while the η_s rule allows for the μ binder to be removed when its continuation is immediately applied to its body.

The main operational difference lies in the fst rule which allows for terms to be continually substituted into the body wherever the continuation parameter is located. The fst rule expands the *mu* binder with a *lambda* that accepts a term. It is intuitive to think about the fst rule as accepting the head (or first element) of a stream of terms as an argument. By expanding the *mu*-term the fst rule does not remove the binder after reduction. This enables *mu*-terms to continually accepting streams of terms.

Saurin gives two reduction strategies for $\Lambda\mu$. The head reduction strategy applies head reduction (reducing the outermost term) for the β, η, β_s, and η_s rules. When no longer possible reduce the (necessarily unique) fst-redux to create another head $\beta\eta\beta_s\eta_s$-redux. The left reduction strategy involves reducing the left most $\beta\eta\beta_s\eta_s$-redux and applying the fst rule when it creates a $\beta\eta\beta_s\eta_s$-redux to the left of any other $\beta\eta\beta_s\eta_s$-redux.

When introducing $\Lambda\mu$ Saurin typed the calculus according to classical natural deduction. This typing scheme satisfies the Curry-Howard correspondence but disallows certain computationally interesting terms. In an effort to expose more of the computational component of $\Lambda\mu$ Saurin developed a type system based on the stream interpretation of the calculus.

Figure 16 shows the stream type system for $\Lambda\mu$. The system deals with "pretypes" which are considered up to the equivalence relation $\equiv_{fst}$, the symmetric, reflexive, and transitive closure

[2]The separation property (Böhms theorem) is beyond the scope of this document. It states that for any two canonical normal forms M and N there exists a context $C[]$ which will reduce to two different variable

Term pretypes: $\mathcal{T}, A, B, ... ::= o_i \mid A \rightarrow B \mid S \Rightarrow \mathcal{T}$
Stream pretypes: $\mathcal{S}, P, Q, ... ::= \sigma_i \mid \mathcal{T} \rightarrow \mathcal{S} \mid \bot$

$$\frac{}{\Gamma, x : \mathcal{T} \vdash x : \mathcal{T} \mid \Delta} Var_{\mathcal{T}} \qquad \frac{\Gamma \vdash M : \mathcal{T} \mid \Delta}{\Gamma \vdash M : \mathcal{T}' \mid \Delta} \equiv_{fst} (\text{if } \mathcal{T} \equiv_{fst} \mathcal{T}')$$

$$\frac{\Gamma, x : \mathcal{T} \vdash M : \mathcal{T}' \mid \Delta}{\Gamma \vdash \lambda x.M : \mathcal{T} \rightarrow \mathcal{T}' \mid \Delta} Abs_{\mathcal{T}} \qquad \frac{\Gamma \vdash M : \mathcal{T} \rightarrow \mathcal{T}' \mid \Delta \qquad \Gamma \vdash M' : \mathcal{T} \mid \Delta}{\Gamma \vdash (M)M' : \mathcal{T}' \mid \Delta} App_{\mathcal{T}}$$

$$\frac{\Gamma \vdash M : \mathcal{T} \mid \Delta, \alpha : \mathcal{S}}{\Gamma \vdash \mu\alpha.M : \mathcal{S} \Rightarrow \mathcal{T} \mid \Delta} Abs_{\mathcal{S}} \qquad \frac{\Gamma \vdash M : \mathcal{S} \Rightarrow \mathcal{T} \mid \Delta, \alpha : \mathcal{S}}{\Gamma \vdash (M)\alpha : \mathcal{T} \mid \Delta, \alpha : \mathcal{S}} App_{\mathcal{S}}$$

Figure 16: Stream Types for the $\Lambda\mu$-calculus

of relation $>_{fst}$ which is defined as

$$(\mathcal{T} \rightarrow \mathcal{S}) \Rightarrow \mathcal{T}' >_{fst} \mathcal{T} \rightarrow (\mathcal{S} \Rightarrow \mathcal{T})'.$$

The typing rule $\equiv_{fst}$ allows for conversion between equivalent types.

This typing system provides some insight into what terms of $\Lambda\mu$ are. Stream *consumers* are defined using μ-terms and they are typed as functions from streams to terms. Because μ-terms are terms then they can produce themselves as output continually accepting data from the incoming stream.

By defining consumers of streams the writer of $\Lambda\mu$ programs is working with **push** streams. A μ-term is waiting for data to arrive before it begins calculating through standard $\beta\eta\beta_s\eta_s$ reductions. The output of processing a stream element pushes a new term into the program to be consumed by a subsequent stream consumer. The ability to implement the exact behavior of a stream consumer means that the calculus exposes a transformer-based interface to the user.

Brooklet Brooklet [105] is a calculus for stream processing languages that aims to be an universal intermediate language with the ability to reason about distributing stream processing computations and optimizing programs. Specifically, Brooklet focuses on explicitly capturing the state requirements and non-determinism found in stream-processing systems.

The Brooklet calculus is focused on defining the stream processing graph exposing a **graph-level** interface. Brooklet ignores the implementation of transformers (called operators in Brooklet parlance) only expressing how they are connected and the state that they modify.

Figure 17 shows the syntax of Brooklet. The $\overline{q}$ notation denotes a finite sequence. Programs are composed of a declaration of sinks (*out*) sources (*in*) and operators (*op*). Inputs and outputs are semantically FIFO *queues* and operators are connected by queues as well.

Consider an example program that counts the number of DNS requests that can be served via IPV4 and IPV6.

```
output IP4s, IP6s ;
input packets ;
(dns1, dns2) <- selectDNS(packets) ;
(ARecords) <- SelectAs(dns1) ;
(AAAARecords) <- SelectAAAAs(dns2) ;
```

P	$::=$	$out\ in\ \overline{op}$	Program
out	$::=$	`output` $\overline{q}$;	Output declaration
in	$::=$	`input` $\overline{q}$;	Input declaration
op	$::=$	$(\overline{q},\overline{v}) \leftarrow f(\overline{q},\overline{v})$	Operator
q	$::=$	id	Queue identifier
v	$::=$	`$`id	Variable identifier
f	$::=$	id	Function identifier

Figure 17: The syntax of Brooklet

```
(IP4s, \$cnt4) <- Count(ARecords, \$cnt4) ;
(IP6s, \$cnt6) <- Count(AAAARecords, \$cnt6) ;
```

The first two lines of the program declare that there are two output queues `IP4s` and `IP6s`. The second line declares that the program reads from input queue `packets`. The rest of the program defines the stream processing graph. First packets are filtered for those following the DNS protocol by passing the `packets` queue to the opaque function `selectDNS`. Brooklet intentionally does not define local deterministic functions. The output of the DNS filtering is placed into two queues `dns1` and `dns2`. It is necessary to use two duplicate queues as Brooklet queues can only be read from by one operator. These queues are filtered for A records and AAAA records and placed into the `ARecords` and `AAAArecords` queues respectively. Then the stateful function `Count` is used to updated running counts of the number DNS requests that are served via IPv4 and IPv6. The `Count` function reads from a queue as well as a stateful variable (e.g. `$cnt4`). The output queue and modified state are explicitly declared in the output of the operator.

The operational semantics of Brooklet is based around a single queue producing in a step of the program. At each step a *firing queue* is non-deterministically selected. The firing queue must be both non-empty, and the input to an operator. The requirement that each queue is only used as input once means that a single firing queue also selects a single operator. The operator's function is then called on a data item popped from the firing queue and any stateful variables declared as input. The function will then return a set of elements to push to the output queues and updates to any output variables.

The goal of the operational semantics is to encode the non-deterministic nature of stream production. Firing queues are selected randomly so there is no guarantee on the order at which streaming elements are produced. While a reading of the semantics shows that control is maintained by whatever is selecting the firing queue the semantics model a **push-based** control scheme. Even though an operator is given the time to fully process its input the output may not actually be produced until its output queue has been selected to fire.

Brooklet's main goals of explicitly declaring state and capturing non-determinism allow it to syntactically analyze programs for three optimization opportunities: parallelism, fusion, and reordering. For parallelism Brooklet identifies functions which don't access state as being parallelizable. For fusion Brooklet identifies groups of operators that are in a simple pipeline (the restriction on queue usage makes this simple) and that rely on state which is not used anywhere else in the program. For reordering two operators Brooklet requires both operators to be stateless and the downstream operator to not rely on modified data from the upstream operator. To determine the data dependencies of the two operators Brooklet relies on undefined static analysis of the abstracted function language.

Brooklet presents a pared down syntax for defining stream processing graphs that allows it to identify graph-level optimization opportunities. The lack of ability to define local function in the language limits Brooklet's practical usefulness. Instead it clearly shows the restrictions on state and topology that are needed to make optimizing transformations.

5 Discussion

The success of stream processing has caused it to outgrow its semantic roots. Benefits can be gained from conceptualizing a large but finite data source as infinite and using incremental processing. The effectiveness of this approach has led to the spread of stream processing into areas previously handled by other computational paradigms (e.g. batch processing). This has caused existing abstractions, originally developed as a way to query data in the stream, to miss aspects of modern stream processing. Contemporary stream processing systems now use ad hoc abstractions built on top of general purpose programming languages to serve their needs.

Creating a natural and expressive abstraction that meets the needs of contemporary stream processors requires a firm semantic model. A good semantic model should allow the user to define their stream processors without having to manipulate details like the organization of the stream processing graph or the placement of transformers in a distributed environment. To this end the proper model would allow the user to precisely define their transformers, be able to classify (and optimize!) the individual transformers, and build an efficient stream processing graph without needing to concern the programmer. This is a lofty goal! No current stream processing model is able to meet all these needs: Streaming SQL dialects provide the automation but lack in flexibility, dataflow languages give great flexibility but put a great deal of responsibility on the programmer to ensure the underlying graph is efficient, QRE-based languages provide flexible windowing but do not capture computations over those windows in the model. While all of these models have their deficiencies they have found uses for their particular purposes. It is not at all obvious that a perfect model exists and each of the discussed models is a excellent push towards better stream processing.

An underutilized aspect of stream processing is the control scheme. None of the discussed models explicitly captures where control is located in a stream processor. By capturing both a push and pull scheme a model could present an interface that is natural to more applications. It could also manipulate control in a stream processing graph to optimize certain transformers. Evidence for this approach can be found in other domains. This understudied aspect of stream processing is worth a closer look.

Abstraction is a fundamental task in computer science. When a good abstraction is found it can unlock new ways of approaching a problem and concrete performance benefits. Current stream processing systems are hampered by abstractions that never anticipated their needs. Defining the shape of the problem and making the proper trade-offs in an abstraction can have significant benefits for the wide array of areas that make use of stream processing.

6 Acknowledgment

This work was funded in part by Sandia National Laboratories. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

References

[1] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina,

et al. The design of the borealis stream processing engine. In *Cidr*, volume 5, pages 277–289, 2005.

[2] Daniel J Abadi, Don Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *the VLDB Journal*, 12:120–139, 2003.

[3] Houssam Abbas, Rajeev Alur, Konstantinos Mamouras, Rahul Mangharam, and Alena Rodionova. Real-time decision policies with predictable performance. *Proceedings of the IEEE*, 106(9):1593–1615, 2018.

[4] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. Copatterns: programming infinite structures by observations. *ACM SIGPLAN Notices*, 48(1):27–38, 2013.

[5] Huseyin Acan, Andrea Collevecchio, Abbas Mehrabian, and Nick Wormald. On the push&pull protocol for rumour spreading. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 405–412, 2015.

[6] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, et al. 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):1792–1803, 2015.

[7] Rajeev Alur and Pavol Černỳ. Streaming transducers for algorithmic verification of single-pass list-processing programs. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 599–610, 2011.

[8] Rajeev Alur, Loris DAntoni, Jyotirmoy Deshmukh, Mukund Raghothaman, and Yifei Yuan. Regular functions and cost register automata. In *2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 13–22. IEEE, 2013.

[9] Rajeev Alur, Dana Fisman, Konstantinos Mamouras, Mukund Raghothaman, and Caleb Stanford. Streamable regular transductions. *Theoretical Computer Science*, 807:15–41, 2020.

[10] Rajeev Alur, Dana Fisman, and Mukund Raghothaman. Regular programming for quantitative properties of data streams. In *Programming Languages and Systems: 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2–8, 2016, Proceedings 25*, pages 15–40. Springer, 2016.

[11] Rajeev Alur, Adam Freilich, and Mukund Raghothaman. Regular combinators for string transformations. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–10, 2014.

[12] Rajeev Alur and Konstantinos Mamouras. An introduction to the streamqre language. In *Dependable Software Systems Engineering*, pages 1–24. IOS Press, 2017.

[13] Rajeev Alur, Konstantinos Mamouras, and Caleb Stanford. Modular quantitative monitoring. *Proc. ACM Program. Lang.*, 3(POPL), January 2019.

[14] Rajeev Alur and Mukund Raghothaman. Decision problems for additive regular functions. In *Automata, Languages, and Programming: 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part II 40*, pages 37–48. Springer, 2013.

[15] Arvind Arasu, Shivnath Babu, and Jennifer Widom. Cql: A language for continuous queries over streams and relations. In *International Workshop on Database Programming Languages*, pages 1–19. Springer, 2003.

[16] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The cql continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15:121–142, 2006.

[17] Remzi H Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E Culler, Joseph M Hellerstein, David Patterson, and Kathy Yelick. Cluster i/o with river: Making the fast case common. In *Proceedings of the sixth workshop on I/O in parallel and distributed systems*, pages 10–22, 1999.

[18] Ron Avnur and Joseph M Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 261–272, 2000.

[19] Shivnath Babu, Rajeev Motwani, Kamesh Munagala, Itaru Nishizawa, and Jennifer Widom. Adaptive ordering of pipelined stream filters. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 407–418, 2004.

[20] Yijian Bai, Hetal Thakkar, Haixun Wang, Chang Luo, and Carlo Zaniolo. A data stream language and system designed for power and extensibility. In *Proceedings of the 15th ACM international conference on Information and knowledge management*, pages 337–346, 2006.

[21] Apache Beam. Web apis i/o connector. https://beam.apache.org/documentation/io/built-in/webapis/. Accessed: 2025.

[22] Albert Benveniste, Patricia Bournai, Thierry Gautier, and Paul Le Guernic. *SIGNAL: A data flow oriented language for signal processing*. PhD thesis, INRIA, 1985.

[23] Gérard Berry. The foundations of esterel. 2000.

[24] Yves Bertot. Coinduction in coq. *arXiv preprint cs/0603119*, 2006.

[25] Maciej Besta, Michał Podstawski, Linus Groner, Edgar Solomonik, and Torsten Hoefler. To push or to pull: On reducing communication and synchronization in graph computations. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, pages 93–104, 2017.

[26] Henrik Björklund and Thomas Schwentick. On notions of regularity for data languages. *Theoretical Computer Science*, 411(4-5):702–715, 2010.

[27] Eric Bouillet, Ravi Kothari, Vibhore Kumar, Laurent Mignet, Senthil Nathan, Anand Ranganathan, Deepak S Turaga, Octavian Udrea, and Olivier Verscheure. Processing 6 billion cdrs/day: from research to production (experience report). In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, pages 264–267, 2012.

[28] Timothy Bourke and Marc Pouzet. Lustre, fast first and fresh. *IEEE Embedded Systems Letters*, 2024.

[29] Frédéric Boussinot and Robert De Simone. The esterel language. *Proceedings of the IEEE*, 79(9):1293–1304, 1991.

[30] Ian Buck, Theresa Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. *ACM transactions on graphics (TOG)*, 23(3):777–786, 2004.

[31] Paris Carbone, Marios Fragkoulis, Vasiliki Kalavri, and Asterios Katsifodimos. Beyond analytics: The evolution of stream processing systems. In *Proceedings of the 2020 ACM SIGMOD international conference on Management of data*, pages 2651–2658, 2020.

[32] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *The Bulletin of the Technical Committee on Data Engineering*, 38(4), 2015.

[33] Sirish Chandrasekaran. Telegraphcq: Continuous dataflow processing for an uncertain world. In *Proc. of the Conference on Innovative Data Systems Research, 2003*, 2003.

[34] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J Franklin, Joseph M Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R Madden, Fred Reiss, and Mehul A Shah. Telegraphcq: continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 668–668, 2003.

[35] Chung-Min Chen, Hira Agrawal, Munir Cochinwala, and David Rosenbluth. Stream query processing for healthcare bio-sensor applications. In *Proceedings. 20th International Conference on Data Engineering*, pages 791–794. IEEE, 2004.

[36] Jianjun Chen, David J DeWitt, Feng Tian, and Yuan Wang. Niagaracq: A scalable continuous query system for internet databases. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 379–390, 2000.

[37] Flavio Chierichetti, Silvio Lattanzi, and Alessandro Panconesi. Rumor spreading in social networks. *Theoretical Computer Science*, 412(24):2602–2610, 2011.

[38] Hank Childs, Eric Brugger, Kathleen Bonnell, Jeremy Meredith, Mark Miller, Brad Whitlock, and Nelson Max. *A contract based system for large data visualization.* IEEE, 2005.

[39] Jean-Louis Colaço and Marc Pouzet. Clocks as first class abstract types. In *International Workshop on Embedded Software*, pages 134–155. Springer, 2003.

[40] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M Hellerstein, John Gerth, Justin Talbot, Khaled Elmeleegy, and Russell Sears. Online aggregation and continuous query support in mapreduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 1115–1118, 2010.

[41] Gregory J Conklin, Gary S Greenbaum, Karl Olav Lillevold, Alan F Lippman, and Yuriy A Reznik. Video coding for streaming media delivery on the internet. *IEEE Transactions on Circuits and Systems for Video Technology*, 11(3):269–281, 2001.

[42] Duncan Coutts. *Stream Fusion: Practical shortcut fusion for coinductive sequence types.* PhD thesis, University of Oxford, 2011.

[43] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: From lists to streams to nothing at all. *ACM SIGPLAN Notices*, 42(9):315–326, 2007.

[44] Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk. Gigascope: A stream database for network applications. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 647–651, 2003.

[45] Joseph W Cutler, Christopher Watson, Emeka Nkurumeh, Phillip Hilliard, Harrison Goldstein, Caleb Stanford, and Benjamin C Pierce. Stream types. *Proceedings of the ACM on Programming Languages*, 8(PLDI):1412–1436, 2024.

[46] Laure Daviaud, Pierre-Alain Reynier, and Jean-Marc Talbot. A generalised twinning property for minimisation of cost register automata. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 857–866, 2016.

[47] Miyuru Dayarathna and Srinath Perera. Recent advancements in event processing. *ACM Computing Surveys (CSUR)*, 51(2):1–36, 2018.

[48] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[49] Guido Dorigo and Waldo Tobler. Push-pull migration laws. *Annals of the Association of American Geographers*, 73(1):1–17, 1983.

[50] Omar Farhat, Khuzaima Daudjee, and Leonardo Querzoni. Klink: Progress-aware scheduling for streaming data systems. In *Proceedings of the 2021 International Conference on Management of Data*, pages 485–498, 2021.

[51] Apache Flink. Datastream connectors. https://nightlies.apache.org/flink/flink-docs-release-2.0/docs/connectors/datastream/overview/. Accessed: 2025.

[52] Apache Flink. Table api & sql. https://nightlies.apache.org/flink/flink-docs-release-2.0/docs/dev/table/overview/. Accessed: 2025.

[53] Marios Fragkoulis, Paris Carbone, Vasiliki Kalavri, and Asterios Katsifodimos. A survey on the evolution of stream processing systems. *The VLDB Journal*, 33(2):507–541, 2024.

[54] Hector Garcia-Molina. *Database systems: the complete book*. Pearson Education India, 2008.

[55] Richard Garner. Stream processors and comodels. *Logical Methods in Computer Science*, 19, 2023.

[56] Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. *Data stream management: processing high-speed data streams*. Springer, 2016.

[57] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S Yu, and Myungcheol Doo. Spade: The system s declarative stream processing engine. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1123–1134, 2008.

[58] Léonard Gérard, Adrien Guatto, Cédric Pasteur, and Marc Pouzet. A modular memory optimization for synchronous data-flow languages: application to arrays in a lustre compiler. *ACM SIGPLAN Notices*, 47(5):51–60, 2012.

[59] Neil Ghani, Peter Hancock, and Dirk Pattinson. Representations of stream processors using nested fixed points. *Logical methods in computer science*, 5, 2009.

[60] KAHN Gilles. The semantics of a simple language for parallel programming. *Information processing*, 74(471-475):15–28, 1974.

[61] Michael I Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. *ACM SIGPLAN Notices*, 41(11):151–162, 2006.

[62] Michael I Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S Meli, Andrew A Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, et al. A stream compiler for communication-exposed architectures. *ACM SIGPLAN Notices*, 37(10):291–303, 2002.

[63] Goetz Graefe. Volcano/spl minus/an extensible and parallel query evaluation system. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):120–135, 1994.

[64] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 conference of the ACM special interest group on data communication*, pages 357–371, 2018.

[65] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[66] Martin Hirzel, Henrique Andrade, Bugra Gedik, Gabriela Jacques-Silva, Rohit Khandekar, Vibhore Kumar, Mark Mendell, Howard Nasgaard, Scott Schneider, Robert Soulé, et al. Ibm streams processing language: Analyzing big data in motion. *IBM Journal of Research and Development*, 57(3/4):7–1, 2013.

[67] Martin Hirzel, Guillaume Baudart, Angela Bonifati, Emanuele Della Valle, Sherif Sakr, and Akrivi Akrivi Vlachou. Stream processing languages in the big data era. *ACM Sigmod Record*, 47(2):29–40, 2018.

[68] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. A catalog of stream processing optimizations. *ACM Computing Surveys (CSUR)*, 46(4):1–34, 2014.

[69] Steven CH Hoi, Doyen Sahoo, Jing Lu, and Peilin Zhao. Online learning: A comprehensive survey. *Neurocomputing*, 459:249–289, 2021.

[70] Haruna Isah, Tariq Abughofa, Sazia Mahfuz, Dharmitha Ajerla, Farhana Zulkernine, and Shahzad Khan. A survey of distributed data stream processing frameworks. *IEEE Access*, 7:154300–154316, 2019.

[71] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European conference on computer systems 2007*, pages 59–72, 2007.

[72] Bart Jacobs. *Introduction to coalgebra*, volume 59. Cambridge University Press, 2017.

[73] Bart Jacobs and Jan Rutten. *An introduction to (co) algebra and (co) induction.* Cambridge University Press, 2011.

[74] Namit Jain, Shailendra Mishra, Anand Srinivasan, Johannes Gehrke, Jennifer Widom, Hari Balakrishnan, Uğur Çetintemel, Mitch Cherniack, Richard Tibbetts, and Stan Zdonik. Towards a streaming sql standard. *Proceedings of the VLDB Endowment*, 1(2):1379–1390, 2008.

[75] Sachini Jayasekara, Srinath Perera, Miyuru Dayarathna, and Sriskandarajah Suhothayan. Continuous analytics on geospatial data streams with wso2 complex event processor. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, pages 277–284, 2015.

[76] Xiantao Jiang, F Richard Yu, Tian Song, and Victor CM Leung. A survey on multi-access edge computing applied to video streaming: Some research issues and challenges. *IEEE Communications Surveys & Tutorials*, 23(2):871–903, 2021.

[77] Michael Kaminski and Nissim Francez. Finite-memory automata. *Theoretical Computer Science*, 134(2):329–363, 1994.

[78] Oleg Kiselyov. Iteratees. In *International Symposium on Functional and Logic Programming*, pages 166–181. Springer, 2012.

[79] Ioannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. Building efficient query engines in a high-level language. *Proceedings of the VLDB Endowment*, 7(10):853–864, 2014.

[80] Lingkun Kong and Konstantinos Mamouras. Streamql: a query language for processing streaming time series. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–32, 2020.

[81] 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*, pages 49–60, 2002.

[82] Mahmoud S Mahmoud, Andrew Ensor, Alain Biem, Bruce Elmegreen, and Sergei Gulyaev. Data provenance and management in radio astronomy: A stream computing approach. In *Data Provenance and Data Management in eScience*, pages 129–156. Springer, 2013.

[83] Konstantinos Mamouras. Semantic foundations for deterministic dataflow and stream processing. In *European Symposium on Programming*, pages 394–427. Springer International Publishing Cham, 2020.

[84] Konstantinos Mamouras, Mukund Raghothaman, Rajeev Alur, Zachary G Ives, and Sanjeev Khanna. Streamqre: Modular specification and efficient evaluation of quantitative queries over streaming data. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 693–708, 2017.

[85] Microsoft. Language integrated query (linq). https://learn.microsoft.com/en-us/dotnet/csharp/linq/. Accessed: 2025.

[86] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query processing, resource management, and approximation in a data stream management system. In *CIDR 2003*. Stanford InfoLab, 2002.

[87] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment*, 4(9):539–550, 2011.

[88] Frank Neven, Thomas Schwentick, and Victor Vianu. Finite state machines for strings over infinite alphabets. *ACM Transactions on Computational Logic (TOCL)*, 5(3):403–435, 2004.

[89] Shadi A Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H Campbell. Samza: stateful scalable stream processing at linkedin. *Proceedings of the VLDB Endowment*, 10(12):1634–1645, 2017.

[90] Jan Olhager and Björn Östlund. An integrated push-pull manufacturing strategy. *European Journal of Operational Research*, 45(2-3):135–142, 1990.

[91] Oracle. Stream (java platform se 8). https://docs.oracle.com/javase/8/docs/api/java/util/stream/St Accessed: 2025.

[92] Francesco Paolucci, Andrea Sgambelluri, Filippo Cugini, and Piero Castoldi. Network telemetry streaming services in sdn-based disaggregated optical networks. *Journal of Lightwave Technology*, 36(15):3142–3149, 2018.

[93] Michel Parigot. $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction. In *Logic Programming and Automated Reasoning: International Conference LPAR'92 St. Petersburg, Russia, July 15–20, 1992 Proceedings 3*, pages 190–201. Springer, 1992.

[94] Dusko Pavlović and Vaughan Pratt. The continuum as a final coalgebra. *Theoretical Computer Science*, 280(1-2):105–122, 2002.

[95] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with sawzall. *Scientific Programming*, 13(4):277–298, 2005.

[96] Stanley Rabinowitz and Stan Wagon. A spigot algorithm for the digits of π. *The American mathematical monthly*, 102(3):195–203, 1995.

[97] Henriette Röger and Ruben Mayer. A comprehensive survey on parallelization and elasticity in stream processing. *ACM Computing Surveys (CSUR)*, 52(2):1–37, 2019.

[98] Jan JMM Rutten. A coinductive calculus of streams. *Mathematical Structures in Computer Science*, 15(1):93–147, 2005.

[99] Apache Samza. Programming model.

[100] Alexis Saurin. Separation with streams in the/spl lambda//spl mu/-calculus. In *20th Annual IEEE Symposium on Logic in Computer Science (LICS'05)*, pages 356–365. IEEE, 2005.

[101] Alexis Saurin. Typing streams in the $\lambda\mu$-calculus. *ACM Transactions on Computational Logic (TOCL)*, 11(4):1–34, 2010.

[102] Helmut Schwichtenberg and Franziskus Wiesnet. Logic for exact real arithmetic. *Logical Methods in Computer Science*, 17, 2021.

[103] Amir Shaikhha, Mohammad Dashti, and Christoph Koch. Push versus pull-based loop fusion in query engines. *Journal of Functional Programming*, 28:e10, 2018.

[104] Olin Shivers and Matthew Might. Continuations and transducer composition. *ACM SIGPLAN Notices*, 41(6):295–307, 2006.

[105] Robert Soulé, Martin Hirzel, Robert Grimm, Buğra Gedik, Henrique Andrade, Vibhore Kumar, and Kun-Lung Wu. A universal calculus for stream processing languages. In *Programming Languages and Systems: 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings 19*, pages 507–528. Springer, 2010.

[106] Apache Spark. Spark sql, dataframes and datasets guide. https://spark.apache.org/docs/latest/sql-programming-guide.html. Accessed: 2025.

[107] Apache Spark. Spark streaming programming guide. https://spark.apache.org/docs/latest/streaming-programming-guide.html. Accessed: 2025.

[108] Robert Stephens. A survey of stream processing. *Acta Informatica*, 34:491–541, 1997.

[109] Apache Storm. Apache storm. https://storm.apache.org/. Accessed: 2025.

[110] Nesime Tatbul, Uğur Çetintemel, and Stan Zdonik. Staying fit: Efficient load shedding techniques for distributed stream processing. In *Proceedings of the 33rd international conference on Very large data bases*, pages 159–170. Citeseer, 2007.

[111] William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications. In *Compiler Construction: 11th International Conference, CC 2002 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8–12, 2002 Proceedings 11*, pages 179–196. Springer, 2002.

[112] Quoc-Cuong To, Juan Soto, and Volker Markl. A survey of state management in big data processing systems. *The VLDB Journal*, 27(6):847–872, 2018.

[113] Yi-Cheng Tu, Song Liu, Sunil Prabhakar, and Bin Yao. Load shedding in stream databases: a control-based approach. 2006.

[114] Hans Vandierendonck. Graptor: Efficient pull and push style vectorized graph processing. In *Proceedings of the 34th ACM International Conference on Supercomputing*, pages 1–13, 2020.

[115] William W Wadge, Edward A Ashcroft, et al. *Lucid, the dataflow programming language*, volume 303. Academic Press London, 1985.

[116] Yifei Yuan, Dong Lin, Ankit Mishra, Sajal Marwaha, Rajeev Alur, and Boon Thau Loo. Quantitative network monitoring with netqre. In *Proceedings of the conference of the ACM special interest group on data communication*, pages 99–112, 2017.

[117] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, pages 423–438, 2013.

[118] Feng Zhang, Lin Yang, Shuhao Zhang, Bingsheng He, Wei Lu, and Xiaoyong Du. {FineStream}:{Fine-Grained}{Window-Based} stream processing on {CPU-GPU} integrated architectures. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 633–647, 2020.

[119] Yongpeng Zhang and Frank Mueller. Gstream: A general-purpose data streaming framework on gpu clusters. In *2011 International Conference on Parallel Processing*, pages 245–254. IEEE, 2011.