AquaLang Research Report

Dataflow systems have emerged as the leading solution for running continuous analytics applications in the cloud, offering scalability, low-latency, and strong processing guarantees. However, the limitations imposed by their programming frontends, unsafe dataflow libraries and restrictive query languages, hinder their true potential. We propose AquaLang, a programming language for continuous analytics that offers the best of both worlds: flexibility, safety, efficiency, and conciseness. This report gives a technical overview of AquaLang, covering its motivation, design and applications.

ACM Reference Format:
. 2024. AquaLang Research Report. 1, 1 (February 2024), 30 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Contents

Author's address:

XXXX-XXXX/2024/2-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

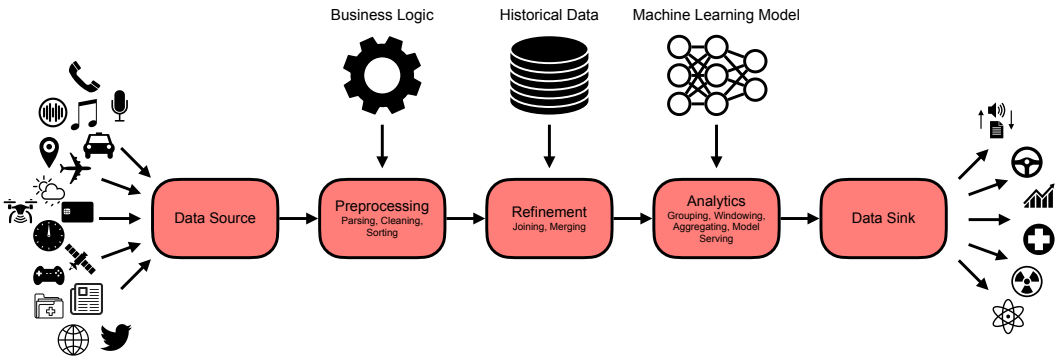

Fig. 1. The generic structure of a continuous analytics application.

Data is today being produced in massive volumes at an unprecedented rate, with sources ranging from social media user interactions to autonomous industrial sensors. This proliferation has resulted in a new domain of applications - *continuous analytics* - that requires data to be analyzed as soon as it is produced to support quick and accurate decision-making. Examples of continuous analytics applications include fraud detection of financial transactions, ride-sharing coordination, stock market prediction, online medical diagnosis, weather forecasting, cyber security, and speech recognition.

Continuous analytics applications are traditionally modelled as a sequence of stages that progressively derive meaningful information from data, as illustrated in 1. Data must typically be cleaned using application-dependent logic, transformed into a structured format, and refined with historical information to make it suitable for analysis. Analysis typically ranges from the calculation of basic summaries to advanced machine learning inference.

This report introduces AquaLang, a programming language specifically designed for expressing continuous analytics applications. We begin by describing the dataflow model, a computational model supported by modern systems for continuous analytics, which underpins AquaLang (**??**). We then give an overview of current programming frontends that support this model and pinpoint their advantages and shortcomings in terms of safety, efficiency and expressiveness (sec. **??**). We then progressively introduce AquaLang in a top-down manner **??**, showing how its features addresses the observed shortcomings. Finally, we conclude with prospects about future work.

2 THE DATAFLOW MODEL

Continuous Analytics applications today run on dedicated *dataflow systems* (e.g., Apache Flink, Apache Spark, and Cloud Dataflow) that implement the *dataflow model*. Dataflow is, in the most general sense, a computational model that represents programs as *dataflow graphs* modelling the flow of data between operators. These graphs are typically acyclic, meaning data flows in a unidirectional fashion from upstream operators to downstream operators. An *operator* is a concurrent entity that continuously consumes and produces data. Operators are independent, meaning the only means for communication between them is through their dataflow dependencies.

Dataflow can naturally be extended to support event-driven programming, meaning the flow of data is driven by external events such as user interactions (e.g., clicks on websites or mobile apps) or system events (e.g., log entries, error messages or performance metrics) [4]. An *event* is fundamentally a data point paired with a timestamp describing something that happened at a specific moment in time. This temporal dimension allows operators to aggregate information over time with respect to when the data was originally produced. The core abstraction for defining dataflow graphs are streams. A *stream* is an unbounded sequence of events, made available over time, that can be transformed through operators. Operators appear to process events in the order they were generated (i.e., timestamp order) up to a certain bound of disorder. A *source* is a special type of operator that ingests events to form a stream. Analogously, a *sink* is an operator that commits a stream to make its events observable. Together, sources and sinks serve as the only point of interaction dataflow applications have with the external world.

2.1 Capabilities

The dataflow model exhibits multiple capabilities that make it promising from an implementation standpoint. These are as follows:

- **Location Transparency**. No specific physical location is specified where operators must execute. It is up to the dataflow system to decide how dataflow graphs are mapped to a physical architecture.

Dataflow graphs can for example be projected to a network of machines, where each computer is responsible for executing a subset of the operators.

- **Supply- and Demand-Driven Execution**. Dataflow systems can support both supply-driven and demand-driven execution. Supply-driven execution executes operators eagerly as soon as their input data is available, pushing data through the system based on supply. Demand-driven execution executes operators lazily only when their output is requested, pulling data through the system based on demand. Generally a mix of both modes is used.
- **Streaming and Batch Execution**. Dataflow systems can support both streaming and batch execution. Streaming execution processes data incrementally, one event at a time, resulting in lower end-to-end latency. Batch execution processes data in bulk, multiple events at a time, resulting in higher end-to-end throughput.
- **Data Parallelism**. A stream can be sharded into logical partitions, allowing an operator to process different partitions in parallel as long as no communication is required between them.
- **Pipeline Parallelism**. An operator which consumes data can execute in parallel with the operator that is producing the data.
- **Concurrent I/O**. Sources and sinks are logical, and can represent an arbitrary number of physical endpoints (e.g., TCP connections). No event ordering is enforced between endpoints, allowing them to be handled concurrently.
- **Source Parallelism**. Events may be ingested to the system in parallel. This, for example, allows a source to read a file and ingest each line in parallel.
- **Out-of-order Processing**. Events can be processed out-of-order with respect to the time they were generated, as long as they appear to be processed in-order. A system may thus induce disorder by processing events with data parallelism, and later sort the events into the right order before an order sensitive operation.
- **Flow-Control**. Dataflow systems can support flow-control through techniques such as buffering, backpressure or rate-limiting. Flow control allows the rate of data production to be controlled by the rate of consumption between operators, ensuring the system does not run out of resources.
- **Deadlock-Freedom**. As a consequence of dataflow graphs being acyclic, flow-control cannot deadlock a system by making a consumer block itself from producing data. Additionally, since operators are independent, it is not possible for the system to deadlock by operators circularly waiting for shared resources.
- **Fault Tolerance**. Dataflow graphs can be persisted to a consistent state using a variety of snapshot protocols, providing both fast recovery and low runtime overhead.
- **Transactional Guarantees**. Dataflow systems can guarantee that their results are committed exactly once to the outside world. If an event was outputted by a sink, then the system must guarantee that its persisted state reflects this information. Exactly-once guarantees can additionally be weakened, to at-least-once or at-most-once, to improve performance.
- **System Composition**. Dataflow systems can be composed with other systems through the sources and sinks of their dataflow graphs. This for example allows dataflow systems to be used in microservice architectures, where each microservice can be a dataflow graph that communicates through their sources and sinks.

2.2 Alternatives

In addition to the dataflow model, there are more models being used today for complex analysis tasks, these include:

- **Relational Model**. The relational model is a data model that structures data in terms of relations, allowing it to be manipulated with declarative languages that build upon relational algebra. In

contrast, the dataflow model does not enforce a particular data model or algebra, providing flexibility to support unstructured data and user-defined functions at the expense of complexity.

- **Workflow Model**. The workflow model is a computational model that structures programs as workflow graphs modelling the flow of control between a system of tasks [6]. Workflows impose a strict order in which tasks must be completed. This is useful in applications where ordering plays a significant role, such as business processes [**?**], scientific processes [**?**], and ML training pipelines [**?**]. In contrast to the workflow model, the dataflow model has no ordering requirements or guarantees, and focuses on modelling applications that run continuously without completion.
- **Actor Model**. The actor model is a general model for concurrent computational which orients programs around actors. Actors are entities that make independent local decisions and communicate through asynchronous message passing. In response to receiving a message from another actor, an actor can perform some computation which possibly involves creating new actors and sending messages to other actors. Actors are addressable by reference, allowing for dynamic communication patterns where references are communicated using message passing. The dataflow model can be viewed as a subset of the actor model that imposes a static and acyclic messaging topology to enable the guarantees listed in 2.1.

2.3 Challenges

Despite the dataflow model's significant benefits in scalability and fault tolerance for large-scale data processing, its adoption and practical application come with challenges:

- **System Complexity**. Designing and implementing dataflow systems is difficult, requiring deep understanding in areas that include operating systems, distributed systems, databases, and hardware. This complexity can become a barrier to adoption if exposed to end-users. There is a pressing need for dataflow programming frontends that abstract their underlying system without leaking its implementation details.
- **Shift in Programming Paradigm**. Although dataflow programming is powerful at expressing data processing patterns, it requires a fundamental shift in thinking about how to write programs. Users must move away from a general purpose programming perspective, where anything is possible, to one that restricts programs to dataflow graphs.
- **Low-Level Representation**. While dataflow graphs capture the essence of functional data transformation, they can prove to be too low-level for end-users. There is a need for higher-level dataflow-based abstractions that more closely reflect the application domain.
- **Limited Observability**. The inherent parallel and distributed nature of dataflow systems make them considerably more difficult to introspect and understand when compared to traditional programs that run locally on a single thread. Developers cannot easily inspect intermediate states of the data as it flows through the dataflow graph, since the only observable components of a graph are sources and sinks, which is critical for debugging, testing, and monitoring. Conventional means for extracting information from running programs such as debug-printing to terminal are generally not possible.
- **Non-Determinism**. The non-deterministic nature of dataflow systems introduces challenges when it comes to reasoning about program behaviour. Without guarantees on the order of execution (e.g., ordering of event delivery) programs may exhibit unexpected results or bugs that are hard to reproduce.

3 DATAFLOW PROGRAMMING TOOLS

In this section we review and compare different tools for building dataflow applications that are offered by current dataflow systems and discuss their strengths and weaknesses.

3.1 Streaming Libraries

Streaming libraries such as Flink, Kafka Streams and Spark Streaming, are libraries hosted by General Purpose Languages (GPLs) that allow users to formulate dataflow graphs using a catalogue of functions that produce, transform, and consume data streams. These functions typically take a User-Defined Function (UDF) as input that is applied on individual events of a data stream which have a User-Defined Type (UDT). A program which, given a stream of temperature readings, calculates the maximum temperature per city per day, can be written as:

```
case class CityTemp(city: String, time: Long, temp: Double);
case class CityMaxTemp(city: String, time: Long, max_temp: Double);

def main() {
  val env = StreamExecutionEnvironment.getExecutionEnvironment

  env
    .readCsvFile("input.csv")
    .keyBy(_.city)
    .timeWindow(Time.days(1))
    .reduce((a, b) => CityMaxTemp(a.city, a.time, max(a.temp, b.temp))))
    .writeAsCsv("output.txt")

  env.execute()
}
```

This style of programming offers complete freedom in what users can programmatically express but comes at the cost of safety, efficiency, and ease of use:

- **Safety**: It is easy to write programs that type check but are still semantically incorrect with respect to the dataflow model. For example, nothing hinders a UDF inside an operator from accessing resources from the local file system. This can break location transparency, since an operator may produce different results depending on where it is executed.
- **Efficiency**: The runtime system views UDFs and UDTs as black boxes, meaning it has no perception or control over what code is being executed or what data is being manipulated. This prevents user code from being rewritten for further optimisation.
- **Ease of Use**: The GPL provides constructs aimed at general-purpose programming as opposed to dataflow programming, potentially resulting in boilerplate code that is unrelated to the problem being solved.

3.2 Streaming Query Languages

Streaming Query languages such as CQL [2], Calcite SQL [3] and KSQL [1] are high-level declarative languages that extend the relational model with support for stream processing. The code for calculating temperature statistics can be written in Calcite SQL as:

```
CREATE TABLE CityTemp (city STRING, time BIGINT, temp DOUBLE )
WITH ('path' = 'input.csv', 'format' = 'csv');

CREATE TABLE CityMaxTemp (city STRING, day TIMESTAMP(3), max DOUBLE )
```

```
WITH ('path' = 'output.csv', 'format' = 'csv');

INSERT INTO CityMaxTemp
SELECT city, TUMBLE_START(time, INTERVAL '1' DAY) AS day, MAX(temp) AS max
FROM CityTemp
GROUP BY city, TUMBLE(time, INTERVAL '1' DAY);
```

This style of programming restricts the user to a programming model that clearly defines what a legal program is. Programs that type check in this model are guaranteed to have well-defined behaviour. These language constraints additionally serve as a guide, helping programmers understand what programs are possible to write.

- **Expressiveness**. Query languages are declarative, meaning their programs are deterministic in how they produce their outputs. As a result, arbitrary computation is not feasible. Query languages, for example, cannot parse unstructured data, read configuration files, or contain structural control-flow, which are often necessary in end-to-end applications.
- **Extensibility**. While streaming libraries can import functionality from other
- **Modularity**. Additionally, query languages are high-level and as a result require no wide support for abstraction since the necessary abstractions are already provided out of the box.

In cofor this reason, often found embedded inside GPLs alongside streaming libraries to model declarative program segments such as data cleaning or descriptive statistics.

4 AQUALANG

In this section we provide a specification of AquaLang[1]. We begin by discussing requirements and high-level decisions behind AquaLang (sec. **??**). We then describe AquaLang's design by first introducing its dataflow operators that form the basis of dataflow applications (sec. **??**). Then, we describe a higher-level relational syntax and demonstrate how it can be layered over the dataflow operators to provide a more expressive and intuitive programming model (sec. **??**). Next, we describe AquaLang's sequential syntax which combines aspects of functional and imperative programming for building dataflow graphs and defining UDFs and UDTs **??**. Our explanation is top-down and assumes some prior knowledge in programming languages since we may refer to concepts before they are described with more detail. When formalising the syntax of the paradigms, we often use the notation $s_1, \cdots, s_n$ to represent a comma-separated sequence of symbols s with length n, where $n \geq 1$ unless specified otherwise.

Operators	Description
`source`	Ingest a stream from the outside world (sec. 4.1.2).
`sink`	Commit a stream to the outside world (sec. 4.1.3).
`flatmap`	Map each event to many new events using a UDF (sec. 4.1.4).
`keyby`	Group events by key, producing a keyed stream (sec. 4.1.5).
`unkey`	Ungroup keyed events, producing a stream (sec. 4.1.5).
`apply`	Apply a state machine on a stream to transform it (sec. 4.1.7).
`window`	Group events into windows and then aggregate them (sec. 4.1.8).
`merge`	Merge two streams into one (sec. 4.1.9).
`fork`	Fork one stream into two (sec. 4.1.10).

Table 1. An overview of the built-in dataflow operators of AquaLang.

[1]The source code of AquaLang is available on our GitHub repository: https://github.com/anonymous-aqualang/aqualang

4.1 Dataflow Subset

AquaLang programs are specifications of dataflow graphs that describe transformations of data streams. To construct dataflow graphs, AquaLang provides a set of dataflow operators, listed in 1 that operate on two types of data streams. This section gives an overview of these streams each operator, gives motivating examples that show how they can be used in real-world applications, and describes their individual semantics from a sequential execution perspective.

4.1.1 Streams. Streams provide the illusion of a infinite sequence of events that can be received and processed one-by-one. There are two types of streams in AquaLang, `Stream[T]` and `KeyedStream[K,T]`. These streams differ in how they enable parallelism.

- A `Stream[T]` is a stream whose events appear to be processed in sequential order. Operations on this type of stream can only be parallelised if they are commutative or associative (i.e., the order of operations or operands is insignificant).
- A `KeyedStream[K,T]` is a stream grouped by a key attribute into independent partitions, where the events of each partition appear to be processed in sequential order. Operations on this type of stream can be parallelised per-partition.

4.1.2 Source. A source is an operators that ingests data into a dataflow graph from the outside world to produce a stream. The `source` operator has the following signature.

```
def source[T](Reader, Encoding, TimeSource[T] = ingestion()): Stream[T];
```

Sources take:

- A **reader** that specifies from where data should be ingested. Currently supported readers are standard input, file, Kafka, TCP, and HTTP.

  ```
  def stdin_reader(): Reader;
  def file_reader(path: Path, watch: bool = false): Reader;
  def http_reader(sock: SocketAddress, path: Path): Reader;
  def tcp_reader(sock: SocketAddress): Reader;
  def kafka_reader(sock: SocketAddress, topic: String): Reader;
  ```

  For `file_reader`, an optional watch parameter can be passed as `true` to allow watching for changes in the file. When set to `false`, the reader emits an end-of-stream marker when the file is read to the end.
- An **encoding** specifying how data should be deserialized. Currently, supported encodings are CSV (Comma Separated Values) and JSON (Javascript Object Notation), and UTF8 (plaintext). In practice, more encodings could be added.

  ```
  def csv(sep: char = ','): Encoding;
  def json(): Encoding;
  def text(): Encoding;
  ```

- An optional **time source** specifying how timestamps should be derived from events. Timestamps are necessary for certain streaming operations, such as time-based window aggregation, which will be explained in section 4.1.8. The supported time sources are ingestion time (default if unspecified) and event time.
  - **Ingestion time** derives a timestamp for each event from the source's local physical clock at the point of ingestion. This is useful when the data itself does not contain timestamps.

– **Event time** derives a timestamp for each event from the data it carries using a timestamp extractor function.

```
def ingestion[T](): TimeSource[T],
def event[T](extractor: fun(T):time): TimeSource[T],
```

Following are examples of how to define data sources with different connectors, formats and time sources. Note that data sources, like other operators, are compositional, and data type annotations can be omitted as long as later operations can infer them.

```
type Item = {name:String, price:i32};
type Request = {id:u32, price:f64, items:Vec[Item]};

val s0: Stream[Item] = source(stdin_reader(), csv());
val s1: Stream[Item] = source(file_reader("logs/*.csv"), csv());
val s2: Stream[String] = source(tcp_reader("127.0.0.1", 8081), text());
val s3: Stream[Request] = source(kafka_reader("127.0.0.1", 8082, "requests"), json());

# Ingestion and event time
val s4: Stream[Item] = source(stdin_reader(), csv(), ingestion());
val s5: Stream[Item] = source(stdin_reader(), csv(), event(fun(e) = e.timestamp));
```

Algorithm 1 Ingestion-Time Source

```
1:  procedure ISOURCE(c, f_d, d_i)
2:      (tx', rx') ← CHANNEL()
3:      spawn
4:          rx_c ← CONNECT(c)
5:          rx_t ← TIMER(d_i)
6:          loop
7:              on event
8:                  case x ← RECV(rx_c)
9:                      v ← f_d(x)
10:                     t ← NOW()
11:                     SEND(tx', DATA(v, t))
12:                 case tick ← RECV(rx_t)
13:                     t ← NOW()
14:                     SEND(tx', WATERMARK(t))
15:     return rx'
16:
17:
18:
19:
20:
21:
```

Algorithm 2 Event-Time Source

```
1:  procedure ESOURCE(c, f_d, d_i, f_t, d_s)
2:      tx', rx' ← CHANNEL()
3:      spawn
4:          rx ← CONNECT(c)
5:          t_max ← 0
6:          t_w ← 0
7:          i ← INTERVAL(d_i)
8:          loop
9:              select
10:                 case x ← RECV(rx)
11:                     v ← f_d(x)
12:                     t ← f_t(x)
13:                     if t < t_w
14:                         continue
15:                     if t > t_max
16:                         t_max ← t
17:                     SEND(tx', DATA(v, t))
18:                 case TICK(i)
19:                     t_w ← t_max − slack
20:                     SEND(tx', WATERMARK(t_w))
21:     return rx'
```

The semantics of the source operators are illustrated in figure 1 and 2. In these algorithms, c is the connector, f_d is the deserialisation function specified by the data format, d_i is the watermark interval duration, f_t is the timestamp extractor function, and d_s is the slack duration.

4.1.3 Sink. Sinks are operators that commit results computed by the dataflow graph to the external world. The `sink` operator takes a stream, writer, and data format.

```
def sink[T](Stream[T], Writer, Encoding);
```

Analogous to a **reader**, a **writer** specifies how data should be committed to the external world:

```
def stdout_writer(): Writer;
def file_writer(path: Path): Writer;
def http_writer(sock: SocketAddress, path: Path): Writer;
def tcp_writer(sock: SocketAddress): Writer;
def kafka_writer(sock: SocketAddress, topic: String): Writer;
```

For `file_reader`, an optional watch parameter can be passed as `true` to allow watching for changes in the file. When set to `false`, the reader emits an end-of-stream marker when the file is read to the end.
An example pipeline that converts the data format of events from CSV to JSON can be written as follows:

```
val s: Stream[Item] = source(stdin_reader(), csv());
sink(s, stdout_writer(), json());
```

4.1.4 Flatmap. FlatMap is an operator that generalizes all stateless streaming operators. It is defined as follows:

```
def flatmap[I, O](Stream[I], fun(I): Vec[O]): Stream[O];
```

Using `flatmap`, it is possible to define other operators such as:

- `map`: Maps each event of a data stream to a new event using a user-defined mapping function.

```
def map(stream, f) = stream.flatmap(fun(x) = [f(x)].into_vec())
```

- `filter`: Filters out events of a data stream using a user-defined predicate.

```
def filter(stream, p) = stream.flatmap(fun(x) = if p(x) { [x].into_vec() } else { [].
    into_vec() })
```

- `flatten`: Flattens a stream of vectors of events into a stream of events.

```
def flatten(stream) = stream.flatmap(fun(x) = x)
```

Although `flatmap` is general enough to implement many kinds of operators, AquaLang provides specialized implementations of `map`, `filter` and `flatten` to avoid unnecessary computation. Below is an example of how these operators can be used, showing a pipeline which extracts all items from requests with a price greater than 10.0 USD.

```
val s0: Stream[Request] = source(stdin_reader(), json());
val s1: Stream[Request] = filter(s1, fun(request) = request.price > 10.0);
val s2: Stream[Item] = flatmap(s2, fun(request) = request.items);
val s3: Stream[String] = map(s3, fun(item) = item.name);
sink(s3, stdout_writer(), json());
```

Like dataflow libraries, the dot operator can be used to chain operators to avoid defining intermediate variables. Anonymous function syntax can additionally be used to avoid parameter declarations. Using these abbreviations, the previous example can be rewritten more concisely as:

```
source::[Request](stdin_reader(), json())
    .filter(_.price > 10.0)
    .flatmap(_.items)
    .map(_.name)
    .sink(stdout_writer(), json());
```

The syntax `a.f().g()` translates to `g(f(a))` and `f(_.id)` to `f(fun(_0) = _0.id)`.

Algorithm 3 Flatmap Semantics

```
1:  procedure FLATMAP(rx, f)
2:      (tx′, rx′) ← CHANNEL()
3:      spawn
4:          loop
5:              match RECV(rx)
6:                  case DATA(t, v)
7:                      for v ← f(v)
8:                          SEND(tx′, DATA(t, v′))
9:                  case WATERMARK(t)
10:                     SEND(tx′, WATERMARK(t))
11:     return rx′
```

The semantics of `flatmap` are illustrated in figure **??**. Notably the `flatmap` operator does not maintain information between processing events and does not rely on timestamps. Operators that are time-independent transparently forward timestamps and watermarks downstream.

4.1.5 Keyby and Unkey. When writing dataflow applications, it is often desirable to group events into distinct partitions that can be analyzed independently. For example, grouping item purchases by username or location and aggregating their total sum price. The operation for grouping in AquaLang is the `keyby` operator, which partitions data using a key extractor.

```
def keyby[K,T](Stream[T], fun(T):K): KeyedStream[K,T];
```

Grouping results in a `KeyedStream` supporting operations for aggregating information. Keys can be erased from streams using the `unkey` operator.

```
def unkey[K,T](KeyedStream[K,T]): Stream[T];
```

An example of using the keyby operator to identify price drops for specific items, with explicit type signatures to highlight type conversions, can be written as follows:

```
val s0: Stream[Item] = source(stdio(), csv());
val s1: KeyedStream[String, Item] = keyby(s0, _.name);
val s2: KeyedStream[String, Item] = scan(s1, argmin(_.price)); # Defined in next section
val s3: Stream[Item] = unkey(s2);
sink(s3, stdio(), csv());
```

Algorithm 4 Keyby Semantics

```
1:  procedure KEYBY(rx, f)
2:      (tx', rx') ← CHANNEL()
3:      spawn
4:          loop
5:              match RECV(rx)
6:                  case DATA(t, v)
7:                      k ← f(v)
8:                      SEND(tx', KDATA(k, t, v'))
9:                  case WATERMARK(t)
10:                     SEND(tx', WATERMARK(t))
11:     return rx'
```

Algorithm 5 Unkey Semantics

```
1:  procedure UNKEY(rx, f)
2:      (tx', rx') ← CHANNEL()
3:      spawn
4:          loop
5:              match RECV(rx)
6:                  case KDATA(k, t, v)
7:                      SEND(tx', DATA(t, v'))
8:                  case WATERMARK(t)
9:                      SEND(tx', WATERMARK(t))
10:     return rx'
11:
```

The semantics of keyby and unkey are defined in figure **??**.

4.1.6 Scan. The scan operator computes and outputs a rolling aggregate over a keyed data stream using an aggregator and a function for combining the aggregated result into the current event.

```
def scan[K,I,P,O](KeyedStream[K,I], Aggregator[I,P,O]): KeyedStream[K,O];
```

Aggregators are defined using the API proposed by [5]. An aggregator has three functions, a *lift* function that maps an input value to a partial aggregate, an associative *combine* function that merges two partial aggregates, and a *lower* function that maps a partial aggregate to an output value.

```
def aggregator[I,P,O](fun(I):P, fun(P,P):P, fun(P):O): Aggregator[I,P,O];
```

Aggregators such as sum, count, avg, min, max, and argmin can be defined as:

```
def sum() = aggregator(
    fun(x) = x,
    fun(a, b) = a + b,
    fun(x) = x
);

def count() = aggregator(
    fun(_) = 1,
    fun(a, b) = a + b,
    fun(x) = x
);
```

```
def avg() = aggregator(
    fun(x) = {sum:x, count:1},
    fun(a, b) = {sum: a.sum+b.sum, count: a.count+b.count},
    fun(x) = x.sum / x.count
);

def min() = aggregator(
    fun(x) = x,
    fun(a, b) = if a < b { a } else { b },
    fun(x) = x
);

def max() = aggregator(
    fun(x) = x,
    fun(a, b) = if a > b { a } else { b },
    fun(x) = x
);

def argmin(f) = aggregator(
    fun(x) = {data, arg:f(x)},
    fun(a, b) = if a.arg < b.arg { a } else { b },
    fun(x) = x.data
);
```

Given these aggregators, `scan` can be used as follows to compute the rolling revenue, heaviest and lightest items:

```
source::[Item](stdio(), csv())
    .keyby(_.name)
    .scan(sum(_.price), {revenue:_|_})
    .scan(max(_.mass), {heaviest:_|_})
    .scan(min(_.mass), {lightest:_|_})
    .unkey()
    .sink(stdio(), csv());
```

Here, `{revenue:_|_}` desugars to `fun(aggregate, item) = {revenue:agg|item}` which is a record-concatenation operation that creates a record containing the fields from `item` together with a field with label `revenue` and value `aggregate`. The semantics of `scan` are defined in figure **??**.

4.1.7 Process. In addition to the builtin operators, more specific user-defined operators (UDOs) can be defined using the `process` operator. This operator applies a state machine to a stream. For example, an operator that tracks price drops of items posted to a store, and resets the price as soon as there is a large price increase (> 10x), can be written as follows:

```
def process[T,O,S](Stream[T], S, fun(S,T):(S,Option[O])): Stream[O];
```

```
source::[Item](stdio(), json())
    .keyby(_.name)
    .process(0, fun(s, v) {
        if v < min or v > min*10 {
            (v, Some(v))
        } else {
```

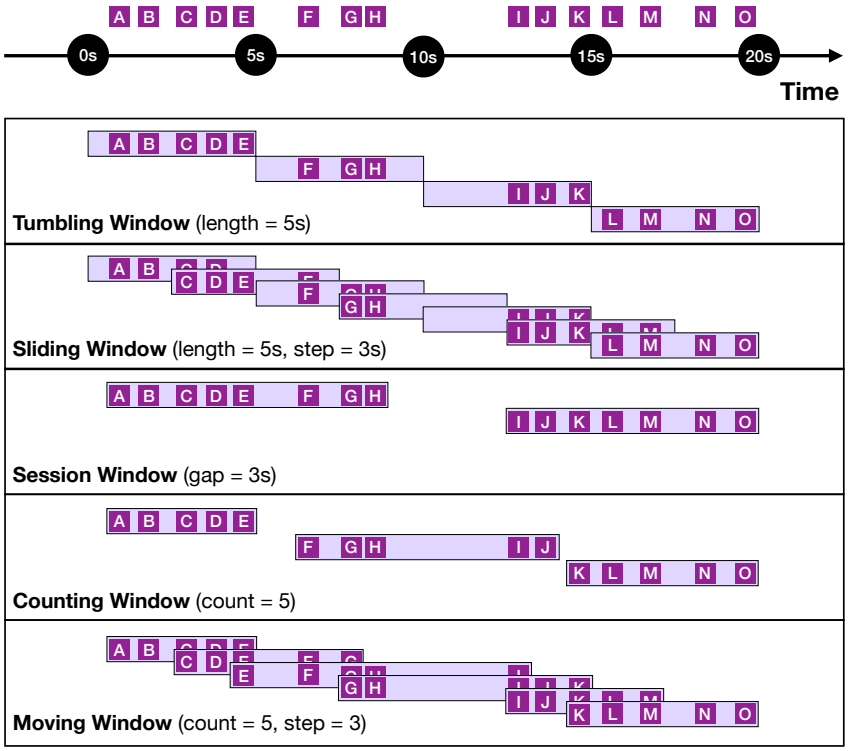

Fig. 2. Different types of window discretization strategies.

```
            (s, None)
        }
    })
    .sink(stdio(), json());
```

While the two previously introduced operators, `flatmap` and `scan`, can be defined in terms of `process`, they are builtin to enable higher-level optimisations. The `process` operator can be used in cases where the expressive power of these builtin operators is not enough.

4.1.8 Window. Window operators group events into possibly overlapping windows, reducing each into a value to produce a new data stream. In AquaLang, window operators take:

- A **discretizer** that specifies the method for grouping events into windows. Supported discretizers are *tumbling*, *sliding* and *session* (time-based), and *counting* and *moving* (count-based). An example of how these work is illustrated in 2.

```
def tumbling(length: Duration): Discretizer;
def sliding(length: Duration, step: Duration): Discretizer;
def session(gap: Duration): Discretizer;
def counting(length: u32): Discretizer;
def moving(length: u32, step: u32): Discretizer;
```

- An **aggregator** that defines the method for reducing a window into a value, using the same interface as the one defined in 4.1.6.

Following is an example of how to use windows:

```
source::[Request](stdio(), csv())
    .window(moving(count:5, step:3), argmin(_.price)))
    .sink(stdio(), csv());
```

The syntax `5s` translates to `s(5)`. Similar postfix operators can be defined for any numeric literal. Following is an example of deriving multiple aggregates:

```
source::[Request](stdio(), csv())
    .window(rolling(count:5, step:3), sum(_.price).compose(count().compose(average(_.price))))
    .map(fun((sum, (count, average))) = {sum, count, average})
    .sink(stdio(), csv());
```

Note that output of multiple aggregators is nested and is here un-nested. Events in data streams may arrive out-of-order with respect to their timestamp. This makes windowing difficult to implement efficiently since aggregation is often an order-dependent operation. How window operators are implemented will be discussed in section **??**.

4.1.9 Merge. The `merge` operator combines a vector of streams into a single stream. All streams being merged must have the same event type.

```
def merge[T](Vec[Stream[T]]): Stream[T];
```

The operator can be used as:

```
val s0: Stream[Request] = source(stdio(), csv());
val s1: Stream[Request] = source(file("requests-1.csv"), csv());
val s2: Stream[Request] = source(file("requests-2.csv"), csv());
val s3: Stream[Request] = merge([s0, s1, s2]);
```

4.1.10 Fork. The `fork` operator broadcasts a stream to two streams. It can be used to create separate analytical pipelines that operate on the same stream.

```
def fork[T](Stream[T]): (Stream[T], Stream[T]);
```

The operator can be used as:

```
val s0: Stream[Request] = source(stdio(), csv());
val (s1: Stream[Request], s2: Stream[Request]) = fork(s0);

val s3 = s1.filter(_.price > 100.0);
val s4 = s2.filter(_.price < 100.0);
```

The above code could be written more concisely as:

```
val s0: Stream[Request] = source(stdio(), csv());

val s3 = s0.filter(_.price > 100.0);
```

```
val s4 = s0.filter(_.price < 100.0);
```

Implicitly, `fork` operations are wherever sharing is observed.

4.2 Relational Subset

In addition to the functional-style syntax, AquaLang provides a more concise relational-style queries for defining dataflow graphs. A query is an expression which manipulates a stream using a sequence of query operators. While similar to SQL, AquaLang's query syntax reads top-down and respects lexical scoping. Events are represented as records of arbitrary data. Since AquaLang queries are expressions as opposed to statements, it is possible to abstract a query using a function and assign its result to a variable. While SQL-style queries translate to relational algebra, AquaLang's queries translate to the functional operators introduced in section 4.1. In this section we introduce each query operator and its translation.

4.2.1 From. Queries begin with the **`from`** operator which iterates over a stream. The iteration variable is visible in the following statements of the query. Queries output all iteration variables which are in scope. The **`from`** translates to the functional `map` operator. For example:

```
type Item = {name: Text, price: F64, mass: F64, height: F64, width: F64};

val stream: Stream[Item] =
    from item: Item in source(stdio(), csv());

# Translates to
val stream: Stream[Item] =
    source(stdio(), csv())
        .map(fun(item:Item) = {item});
```

While the `map` operator might seem unnecessary, its presence becomes important when there is pattern matching on the iteration variable:

```
val stream: Stream[{name:Text, price:F64}] =
    from {name, price|_}: Item in source(stdio(), csv());

# Translates to
val stream: Stream[{name:Text, price:F64}] =
    source(stdio(), csv())
        .map(fun({name, price|_}: Item) = {name, price});
```

Multiple **`from`** operators can be chained to produce a Cartesian product.

```
type User = {id: U64, name: Text, requests: Vec[Request]};
type Request = {id: U64, items: Vec[Item]};

val stream: Stream[{user:User, request:Request, item:Item}] =
    from user: User in source(stdio(), csv())
    from request in user.requests
    from item in request.items;

# Translates to
val stream: Stream[{user:User, request:Request, item:Item}] =
  source(stdio(), csv())
```

```
    .map(fun(user: User) = {user})
    .flatmap(fun(user) = user.requests.map(fun(request) = {user, request})
    .flatmap(fun({user, request} = request.items.map(fun(item) = {user, request, item})))
```

While the initial from statement that starts the query iterates over a stream, successive from statements iterate over finite-sized data (e.g., arrays). This property is verified by the type system.

4.2.2 Into. The `into` operator can be used to write the stream produced by a query into a sink. It translates to the `to` operator. For example.

```
from user: User in source(stdio(), csv())
into sink(stdio(), json());

# Translates to
source(stdio(), csv())
    .map(fun(user: User) = {user})
    .sink(stdio(), json());
```

Note that the provided serializers and deserializers automatically unnest records only containing a single attribute.

4.2.3 Select. The `select` operator is used to select what output should be produced, replacing the current iteration variables in scope. This operator directly translates into a functional `map`.

```
from item: Item in source(stdio(), csv())
select {item.name, price: item.price.usd()}
into sink(stdio(), json());

# Explicit form
from item: Item in source(stdio(), csv())
select {name: item.name, price: item.price.usd()}
into sink(stdio(), json());

# Translates to
source(stdio(), csv())
    .map(fun(item: Item) = {item})
    .map(fun({item}) = {item.name, price: item.price.usd()})
    .sink(stdio(), json());
```

4.2.4 With. The `with` operator assigns intermediate results to variables. Unlike `select`, `with` does not hide any of the variables currently in scope.

```
from item in source(stdio(), csv())
with area = item.width * item.height
into sink(stdio(), csv());

# Translates to
source(stdio(), csv())
    .map(fun(item) = {item, area: item.width * item.height})
    .sink(stdio(), csv());
```

4.2.5 Where. The `where` operator is used to keep events that satisfy a predicate and discard the rest. This operator directly translates into a functional `filter`.

```
from item in source(stdio(), csv())
where item.price > 5.0 and item.mass > 1.0
into sink(stdio(), json());

# Translates to
source(stdio(), csv())
    .filter(fun(item) = item.price > 5.0 and item.mass > 1.0)
    .sink(stdio(), json());
```

4.2.6 Query. The `query` operator performs a sub-query, piping the current stream into a function and reading its output back as a stream.

```
def filter_expensive(s) =
    from {item} in s
    where item.price > 100;

from item: Item in source(stdio(), csv())
query cleaned in filter_expensive
into sink(stdio(), csv());

# Translates to
source(stdio(), csv())
    .map(fun(item: Item) = {item})
    .filter_expensive()
    .map(fun(cleaned) = {cleaned})
    .sink(stdio(), json());
```

4.2.7 Roll. The `roll` operator derives incremental rolling aggregates from a data stream. It translates to the `scan` operator.

```
from item in source(stdio(), csv())
roll sum of item.price as sum
roll max of item.mass as max
select {item, sum, max}
into sink(stdio(), json());

from item in source(stdio(), csv())
roll sum of item.price
roll max of item.mass
into sink(stdio(), json());


# Translates to
source(stdio(), csv())
  .map(fun(item) = {item})
  .scan(sum(fun({item})=item.price),fun(revenue,{item})={revenue,item})
  .scan(max(fun({item})=item.mass),fun(heaviest,{revenue,item})={heaviest,revenue,item})
  .sink(stdio(), json());
```

4.2.8 Group. The `group` operator partitions events into distinct groups and performs a query on each, combining the results into one stream. This operator translates into a composition of the functional keyby and unkey operators.

```
from item: Item in source(stdio(), csv())
group item.id {
    roll min of item.price as cheapest
}
where item.price == cheapest
into sink(stdio(), json());

# Translates to
source(stdio(), csv())
    .keyby(fun(item) = item.id)
    .scan(fun(item) = item.price, sum, fun(revenue, item) = {revenue, item})
    .unkey()
    .filter(fun({revenue, item}) = revenue > 0.0)
    .sink(stdio(), json());
```

4.2.9 Compute. The `compute` operator derives a single aggregate from a finite data collection. Like `select`, `compute` introduces new variables and hides any previously defined ones.

```
val items: Vec[Item] = [
    {name:"Pizza", price:100},
    {name:"Pizza", price:300},
    {name:"Burger", price:200},
    {name:"Sushi", price:200}
];

val result: Vec[{revenue:f32, cheapest:f32, priciest:f32}] =
    from item: Item in items
    compute {
        sum of item.price as revenue,
        min of item.price as cheapest,
        max of item.price as priciest,
    };

assert(result == [{revenue:800, cheapest:100, priciest:300}]);

# Translates to
val result: Vec[{revenue:f32, cheapest:f32, priciest:f32}] =
    items
        .map(fun(item: Item) = {item})
        .reduce(
            sum(_.item.price).compose(
            min(_.item.price).compose(
            max(_.item.price))));
        .map(fun((revenue, (cheapest, priciest))) = {revenue, cheapest, priciest})
```

4.2.10 Limit. The `limit` operator returns a number of elements from a finite data collection. It translates to the take function.

```
val result: Vec[Item] =
```

```
    from item: Item in items
    limit 10;

# Translates to
val result: Vec[Item] =
    items
        .map(fun(item: Item) = {item})
        .take(10);
```

4.2.11 Order. The `order` operator sorts elements of a finite data collection in ascending (by default) or descending order. It translates to the `sort` function.

```
val result: Vec[Item] =
    from item: Item in items
    order item.price desc
    order item.name;

# Translates to
val result: Vec[Item] =
    items
        .map(fun(item: Item) = {item})
        .sort(fun({item}) = item.price, ascending: false)
        .sort(fun({item}) = item.name, ascending: true);
```

4.2.12 Over. The `over` operator partitions events into windows and performs a query on each, reducing it into a single value. Results from multiple windows are combined into one data stream. This operator translates to the functional `window` operator.

```
from item:Item in source(stdio(), json(), ingestion())
over tumbling(length:5s) {
    avg of item.price as average,
    min of item.price as cheapest,
    max of item.price as priciest
}
where average > 10.0
select {average, cheapest, priciest}
into sink(stdio(), json());

# Translates to
source::[Item](stdio(), csv())
    .window(
        tumbling(length:5s),
        avg(_.item.price).compose(
            min(_.item.price).compose(
               max(_.item.price)
            )
        )
    )
    .sink(stdio(), json());
```

4.2.13 View. The `view` operator makes the latest value of a stream visible to all downstream operators.

```
from item: Item in source(stdio(), csv())
view model: Model in source(file("model.txt"), tensorflow())
val

# Translates to
val result: Vec[Item] =
    items
        .map(fun(item: Item) = {item})
        .sort(fun({item}) = item.price, ascending: false)
        .sort(fun({item}) = item.name, ascending: true);
```

4.2.14 Join-On. The `join-on` operator performs an inner-join between a stream and a data collection. It can naively be translated to a composition of `flatmap`, `map` and `filter`. For example:

```
val catalogue = [
    {id:0, name:"Boat", price:5.0},
    {id:1, name:"Bike", price:10.0},
    {id:2, name:"Car", price:20.0}
];

from id:u32 in source(stdio(), csv())
join item in catalogue on id == item.id
into sink(stdio(), csv());

# Translates to
source::[Item](stdio(), csv())
    .map(fun(id:u32) = id)
    .flatmap(fun(id) = catalogue
        .map(fun(id) = {item, id})
        .filter(fun({item, id}) = item.id == id))
    .sink(stdio(), csv());
```

Note that the predicate given to `on` must be an equality comparison (`id == item.id`). If a custom predicate is desired, then a `from-where` can be used. For example:

```
from id:u32 in source(stdio(), csv())
from name in catalogue
where udf(id, item.id)
into sink(stdio(), csv());
```

4.2.15 Join-Over. The `join` operator can additionally be used together with the `over` operator to join events of two streams that fall into the same window.

```
val items = source(stdio(), csv());
val users = source(tcp("127.0.0.1", 8080), csv());

from item:Item in items
join user:User in users over tumbling(length:5s) {
    compute {
```

```
        sum of item.price as revenue,
        max of item.mass as heaviest,
        min of item.mass as lightest
    }
}
into sink(stdio(), csv());

# Translates to
source(stdio(), csv())
    .map(fun(item:Item) = item)
    .map(fun(item) = Either::A(item))
    .merge(users
        .map(fun(user:User) = user)
        .map(fun(user) = Either::B(user))
    )
    .window(
        tumbling(length:5s),
        fun(rx) {
            val users = rx.left();
            val items = rx.right();
            rx.reduce(sum(_.price), fun(revenue) = {revenue})
        }
    )
    .sink(stdio(), csv());
```

4.2.16 Join-Over-On. The `join-over` operator can be followed by `on` to also perform an inner join between the events that fall into the same time window.

```
from item:Item in items
join user:User in users over tumbling(length:5s) on item.user_id == user.id {
    compute {
        sum of item.price as revenue
    }
}
into sink(stdio(), csv());

# Translates to
source(stdio(), csv())
    .map(fun(item:Item) = item)
    .keyby(fun(item) = item.user_id)
    .map(fun(item) = Either::A(item))
    .merge(users
        .map(fun(user:User) = user)
        .keyby(fun(user) = user.id)
        .map(fun(user) = Either::B(user))
    )
    .window(
        tumbling(length:5s),
        fun(rx) = rx.reduce(sum(_.price), fun(revenue) = {revenue})
    )
    .sink(stdio(), csv());
```

4.2.17 Left, Right, Full, and Anti-Join. Joins can be prefixed by `left`, `right`, `full`, and `anti` to specify which records should be outputted.

- `join`: Returns the records from the left input and right input which match on the join condition.

```
from id:u32 in source(stdio(), csv())
join item in items
    on id == item.id
into sink(stdio(), csv());

# Translates to
source(stdio(), csv())
    .map(fun(id:u32) = id)
    .flatmap(fun(id) = items
        .filter(fun(item) = id == item.id)
        .map(fun(item) = {id, item}))
    .sink(stdio(), csv());
```

- `left join`: Returns all the records from the left input, and optionally the records from the right input if there is a match.

```
from id:u32 in source(stdio(), csv())
left join item in items
    on id == item.id
into sink(stdio(), csv());

# Translates to
source(stdio(), csv())
    .map(fun(id:u32) = id)
    .flatmap(fun(id) = items
        .map(fun(item) = if id == item.id {
            {id, item: Some(item)}
        } else {
            {id, item: None}
        }))
    .sink(stdio(), csv());
```

- `right join`: Returns all the records from the right input, and optionally the records from the left input if there is a match.

```
from id:u32 in source(stdio(), csv())
right join item in items
    on id == item.id
into sink(stdio(), csv());

# Translates to
source(stdio(), csv())
    .map(fun(id:u32) = id)
    .flatmap(fun(id) = items
        .map(fun(item) = if id == item.id {
            {id: Some(id), item}
        } else {
            {id: None, item}
        }))
    .sink(stdio(), csv());
```

- `full join`: Returns the union of what would be produced by a left join and a right join.

```
from id:u32 in source(stdio(), csv())
full join item in items
    on id == item.id
into sink(stdio(), csv());

# Translates to
source(stdio(), csv())
    .map(fun(id:u32) = id)
    .flatmap(fun(id) = items
        .flatmap(fun(item) = if id == item.id {
                [{id:Some(id), item:Some(item)}]
            } else {
                [{id:Some(id), item:None}, {id:None, item:Some(item)}]
            }))
    .sink(stdio(), csv());
```

- `anti join`: Returns records from either input for which there are no matching records in the opposite input.

```
from id:u32 in source(stdio(), csv())
anti join item in items
    on id == item.id
into sink(stdio(), csv());

# Translates to
source(stdio(), csv())
    .map(fun(id:u32) = id)
    .flatmap(fun(id) = items
        .filter(fun(item) = not (id == item.id))
        .flatmap(fun(item) = [{id:Some(id), item:None}, {id:None, item:Some(item)}]))
    .sink(stdio(), csv());
```

These translations are naive and could be further optimised by replacing `flatmap` with more specialized operators. Note that, unlike SQL, where optionality is represented by `NULL`, AquaLang represents optionality using the `Option` type. This is necessary for error handling.

4.2.18 Union. The union operator unions two streams into one. It translates to the functional `merge` operator.

```
from item:Item in source(file("items-1.txt"), csv())
union source(file("items-2.txt"), csv())
into sink(stdio(), csv());

# Translates to
source(file("items-1.txt"), csv())
    .map(fun(item:Item) = item)
    .merge(source(file("items-2.txt"), csv()))
    .sink(stdio(), csv());
```

4.2.19 Union. The `union` operator unions two streams into one. It translates to the functional `merge` operator.

```
from item:Item in source(file("items-1.txt"), csv())
union source(file("items-2.txt"), csv())
into sink(stdio(), csv());

# Translates to
source(file("items-1.txt"), csv())
    .map(fun(item:Item) = item)
    .merge(source(file("items-2.txt"), csv()))
    .sink(stdio(), csv());
```

4.3 Sequential Subset

```
P ::= s₁ ··· sₙ                                   program

s ::=                                             statements:
    | val p = e;                                  immutable variable
    | var p = e;                                  mutable variable
    | def x(p₁,···,pₙ) = e;                       function
    | type x = t;                                 type alias
    | enum x { C₁(t₁),···,Cₙ(tₙ) }                enum

v ::=                                             values:
    | x                                           variable
    | C(v)                                        variant
    | {x₁:v₁,···,xₙ:vₙ}                           record
    | [v₁,···,vₙ]                                 array
    | fun(p₁,···,pₙ) = e                          function
    | c                                           constant

p ::=                                             patterns:
    | x                                           variable
    | {x₁:p₁,···,xₙ:pₙ}                           record
    | [p₁,·,pₙ]                                   array
    | C(p)                                        variant
    | c                                           constant
    | p:t                                         annotation
    | _                                           wildcard

e ::=                                             expressions:
    | v                                           value
    | e:t                                         annotation
    | match e { p₁=>e₁,···,pₙ=>eₙ }               conditional
    | loop b                                      loop
    | break | continue | return e                 control-flow
    | do b                                        block
    | {x₁:t₁,···,xₙ:tₙ} | e.x | {x:e|e}           record
    | C(e)                                        variant
    | [e₁,···,eₙ] | e[e]                          array
    | e(e₁,···,eₙ)                                call
    | a = e                                       mutation

a ::= x | a[e] | a.x                              place expressions

b ::= { s₁ ··· sₙ e }                             block

t ::=                                             types:
    | x                                           variable
    | {x₁:t₁,···,xₙ:tₙ}                           record
    | [t;N]                                       array
    | fun(t₁,···,tₙ):t                            function
    | ()                                          unit
    | _                                           wildcard

c ::=                                             constants:
    | ()                                          unit
    | B | Z | R | C | S                           literals
```

Fig. 3. Abstract syntax of AquaLang.

AquaLang's sequential subset provides features for imperative and functional programming that can be used to build programs that construct dataflow graphs as well as define UDFs and UDTs. The syntax is defined in figure 3 using a BNF notation. The terminals `x`, `T`, `C` are meta variables for different categories of identifiers. This section progressively introduces the syntax alongside syntactic abstractions that make it more convenient to use.

4.3.1 Statements. An AquaLang program is a list of statements which evaluate in sequential order. A statement either defines a variable, function, or type. These definitions are lexically scoped, meaning they can be referred to by later statements inside their surrounding scope.

4.3.2 Variable Definition. The `val` and `var` statements evaluate an expression and assigns its result to immutable or mutable variables respectively using pattern matching **??**. For example:

```
val x = 1; # Immutable
var y = 2; # Mutable
x = 1; # Error
y = 3; # OK
```

AquaLang offers exterior mutability, meaning the only way to mutate data is by re-assigning mutable variables. Expressions that can occur as the left operand of an assignment expression are referred to as place-expressions. A place-expression represents a memory location, which could be a mutable variable, record field **??** or array index **??**. While mutable variables add more expressiveness, they have no impact on the efficiency of program execution. The purpose of the distinction between mutable and immutable variables is to make programs easier to reason about for users. When a user sees an immutable variable, they can safely assume it will not be reassigned in later statements.

Syntactic extensions	*Desugaring rules*
`s ::= ···` *expressions:* `| e;` *expression*	`e;` `:= val _ = e;`

Fig. 4. Expression statements

4.3.3 Expression Statements. Expression statements evaluate an expression, potentially producing side effects, and throws the result away. This statement is equivalent to a variable definition statement that binds the right hand side to a wildcard pattern 4. Expression statements are useful when the evaluated expression only produces effects. For example:

```
print("Hello world");
```

4.3.4 Functions. Functions are defined using the `def` keyword. A function takes one or multiple arguments, destructures them into variables using pattern matching, and evaluates an expression. The result of the expression is then returned by the function. A function for computing the greatest common divisor of an integer can for example be written as:

```
def gcd(a, b) = match b {
    0 => abs(a),
    n => gcd(a, a % n)
}
```

Arguments are passed to functions by-value, meaning they are copied as opposed to being passed by-reference. The purpose is to make programs easier to reason about. Statements within a function

cannot reassign variables outside of the function. Passing by-value is known to be inefficient for large data. An implementation of AquaLang can avoid excessive copying in multiple ways. For immutable data (e.g., immutable strings), it is possible to pass a shallow copy that is shared through reference counting. For mutable data (e.g., vectors), it is possible to pass data directly without copying it when it only has a single reference.

Syntactic extensions		*Desugaring rules*
`s ::= ···`	*statements:*	
`| def` x^f`(`p_1`,···,`p_n`) b`	*procedure*	`def` x^f`(`p_1`,···,`p_n`) b` `:= def` x^f`(`p_1`,···,`p_n`) = do b`
`e ::= ···`	*expressions:*	
`| fun(`p_1`,···,`p_n`) b`	*procedure*	`fun(`p_1`,···,`p_n`) b` `:= fun(`p_1`,···,`p_n`) = do b`
`b ::= ...`	*blocks*	
`| { `$\overline{s}$` }`	*procedure block*	`{ `$\overline{s}$` }` `:= { `$\overline{s}$` () }`

Fig. 5. Syntactic abstractions for procedures. Procedures are functions that evaluate a block of statements, potentially producing side effects.

A procedure is a function that evaluates a block instead of an expression, possibly producing side effects. Figure **??** illustrates a set of syntactic abstractions for defining functions as procedures, and show how these translate into the surface syntax. Procedure syntax is useful for imperative programming, which involves mutation and loops. For example, the above code can be written imperatively as follows:

```
def gcd(a, b) {
    var x = a;
    var y = b;
    while y != 0 {
        val temp = y;
        y = x % y;
        x = temp;
    }
    abs(x)
}
```

4.3.5 *Declare Before Use.*

4.3.6 *Variable Capture.* Functions can refer to variables defined in outside scopes. For example:

```
val x: Vec[i32] = Vec::new();
def f(v: i32) = x.push(v);
```

```
val length = 10;
def f[T](): Vec[T] {
  val x = Vec::with_capacity(length);
  x
}
```

Syntactic extensions		*Desugaring rules*		
`e ::= ···`	*expressions:*	$e.x^f(e_1,\cdots,e_n)$		
`| e.x^f(e1,···,en)`	*method call*		`:=`	$x^f(e,e_1,\cdots,e_n)$

Table 2. Syntactic abstractions for methods. Methods are regular functions, where the first argument to the function is passed through dot-chaining syntax.

Syntactic extensions *A - associativity, P - precedence*		*Desugaring rules*		
`e ::= ···`	*expressions:*			
`| not e`	*logic negation (P=1)*	`not e`	`:=`	`_not_(e)`
`| -e`	*arithmetic negation (P=1)*	`-e`	`:=`	`_neg_(e)`
`| e * e`	*multiplication (A=left, P=2)*	e_1 `*` e_2	`:=`	`_mul_(e1,e2)`
`| e / e`	*division (A=left, P=2)*	e_1 `/` e_2	`:=`	`_div_(e1,e2)`
`| e + e`	*addition (A=left, P=3)*	e_1 `+` e_2	`:=`	`_add_(e1,e2)`
`| e - e`	*subtraction (A=left, P=3)*	e_1 `-` e_2	`:=`	`_sub_(e1,e2)`
`| e < e`	*less than (A=left, P=4)*	e_1 `<` e_2	`:=`	`_lt_(e1,e2)`
`| e > e`	*greater than (A=left, P=4)*	e_1 `>` e_2	`:=`	`_gt_(e1,e2)`
`| e <= e`	*less than or equal (A=left, P=4)*	e_1 `<=` e_2	`:=`	`_leq_(e1,e2)`
`| e >= e`	*greater than or equal (A=left, P=4)*	e_1 `>=` e_2	`:=`	`_geq_(e1,e2)`
`| e == e`	*equal (A=left, P=5)*	e_1 `==` e_2	`:=`	`_eq_(e1,e2)`
`| e != e`	*not equal (A=left, P=5)*	e_1 `!=` e_2	`:=`	`_neq_(e1,e2)`
`| e and e`	*conjunction (A=left, P=6)*	e_1 `and` e_2	`:=`	`_and_(e1,e2)`
`| e or e`	*disjunction (A=left, P=6)*	e_1 `or` e_2	`:=`	`_or_(e1,e2)`

4.3.7 Methods. AquaLang does not make a distinction between methods and functions. As illustrated in **??**, functions can be called as if they were methods using dot syntax. For example.

```
val n = 123;
val g0 = gcd(n); # Function call syntax
val g1 = n.gcd(); # Method call syntax
```

4.3.8 Builtins. AquaLang programs are structured around a closed set of builtin data types and functions that are expected in the domain of continuous analytics. This section gives a brief introduction to the main builtin data types using examples. We refer to the documentation for a complete specification of all builtins.

Unit is a type that can only have one value. The unit value carries no information, indicating that no value of interest was produced.

```
():();
print("Hello"):();
```

Booleans can be either `true` or `false`. Their supported operations are `not`, `and`, and `or` operators. Booleans additionally play a role in

```
true:bool;
false:bool;
(true and false or true):bool;
```

Numeric types encompass signed integers, unsigned integers, and floats. These come in both fixed-size format (e.g., `i32` and `f64`) and dynamic-size format (e.g., `Int` and `Float`).

```
12345:i32; # Also i8, i16, i64, i128
12345:u32; # Also u8, u16, u64, u128
123.0:f64; # Also f32

12345:Int;
123.0:Float;

(1*2+3/4-5):i32;
```

Characters and *strings* are used for representing textual data.

```
val _: char = 'c';
val _: String = "Hello";
```

Characters are encoded as UTF-8, meaning a character is represented by up to four bytes. This allows representing both byte strings as well as unicode characters. A consequence of using a variable-length encoding is that random access (e.g., character indexing) is not a constant-time operation.

Arrays represent statically-sized sequences of homogeneous data. Operations on arrays are internally performed in-place on contiguous sequences of bytes, allowing efficient indexing, mutation, and iteration.

```
val x: [i32;3] = [1,2,3];
val y = x[0];
x[0] = 1;
for z in x { print(z); }
```

Options

Vectors represent dynamically-sized sequences of homogeneous data. Like arrays, their operations are performed in-place.

```
var x: [i32;3] = vec([1,2,3]);
val y = x[0];
x[0] = 1;
for z in x { print(z); }

val _: () = x.push(1);
val _: Option[i32] = x.pop();
```

Records represent sets of heterogeneous data.

```
val _: {a:i32, b:i32} = {a:1, b:2};
val x: {a:i32, b:Text} = {a:1, b:"Hello"};
val _: i32 = x.a;
```

Tuples represent statically-sized sequences of heterogeneous data.

```
val _: (i32, i32) = (1, 2);
val x: (i32, String) = (1, "Hello");
val _: String = x.1;
```

Time and Duration

```
val _: Time = 2020-01-01T00:00;
val _: Duration = 5s;
```

4.3.9 User-Defined Data Types. AquaLang offers two ways of composing builtin data types into user-defined data types.

Enums (short for enumerated unions) represent data that could be one out of multiple variants.

```
enum Shape[T] { Rectangle(f32, f32), Circle(f32) }

val _: Shape[f32] = Rectangle(1.0, 2.0);
val _: Shape[f32] = Circle(3.0);

def area(s) = match s {
    Rectangle(w, h) => w * h,
    Circle(r) => r * 3.14,
}
```

4.3.10 Type Aliases. Type aliases can be defined using the `type` keyword. Type aliases are substituted for their right hand side when used in the code. A type alias cannot be self-referential, and may only refer to previously defined types.

```
type User = {name: String, age: i32};

val _: User = {name: "Bob", age: 20};
```

REFERENCES

[1] [n. d.]. ksqlDB - Event Streaming Database Built for Stream Processing. https://www.confluent.io/product/ksqldb/

[2] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2006. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal* 15, 2 (June 2006), 121–142. https://doi.org/10.1007/s00778-004-0147-z

[3] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. 2018. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, Houston TX USA, 221–230. https://doi.org/10.1145/3183713.3190662

[4] Viktor Mayer-Schönberger and Kenneth Cukier. 2013. *Big data: A revolution that will transform how we live, work, and think*. Houghton Mifflin Harcourt.

[5] Kanat Tangwongsan, Martin Hirzel, Scott Schneider, and Kun-Lung Wu. 2015. General incremental sliding-window aggregation. *Proceedings of the VLDB Endowment* 8, 7 (Feb. 2015), 702–713. https://doi.org/10.14778/2752939.2752940

[6] Ian J Taylor, Ewa Deelman, Dennis B Gannon, Matthew Shields, and others. 2007. *Workflows for e-Science: scientific workflows for grids*. Vol. 1. Springer.