AquaLang: A Dataflow Programming Language

Klas Segeljakt[1] (klasseg@kth)
Seif Haridi[1,2] (haridi@kth.se)
Paris Carbone[1,2] (parisc@kth.se)

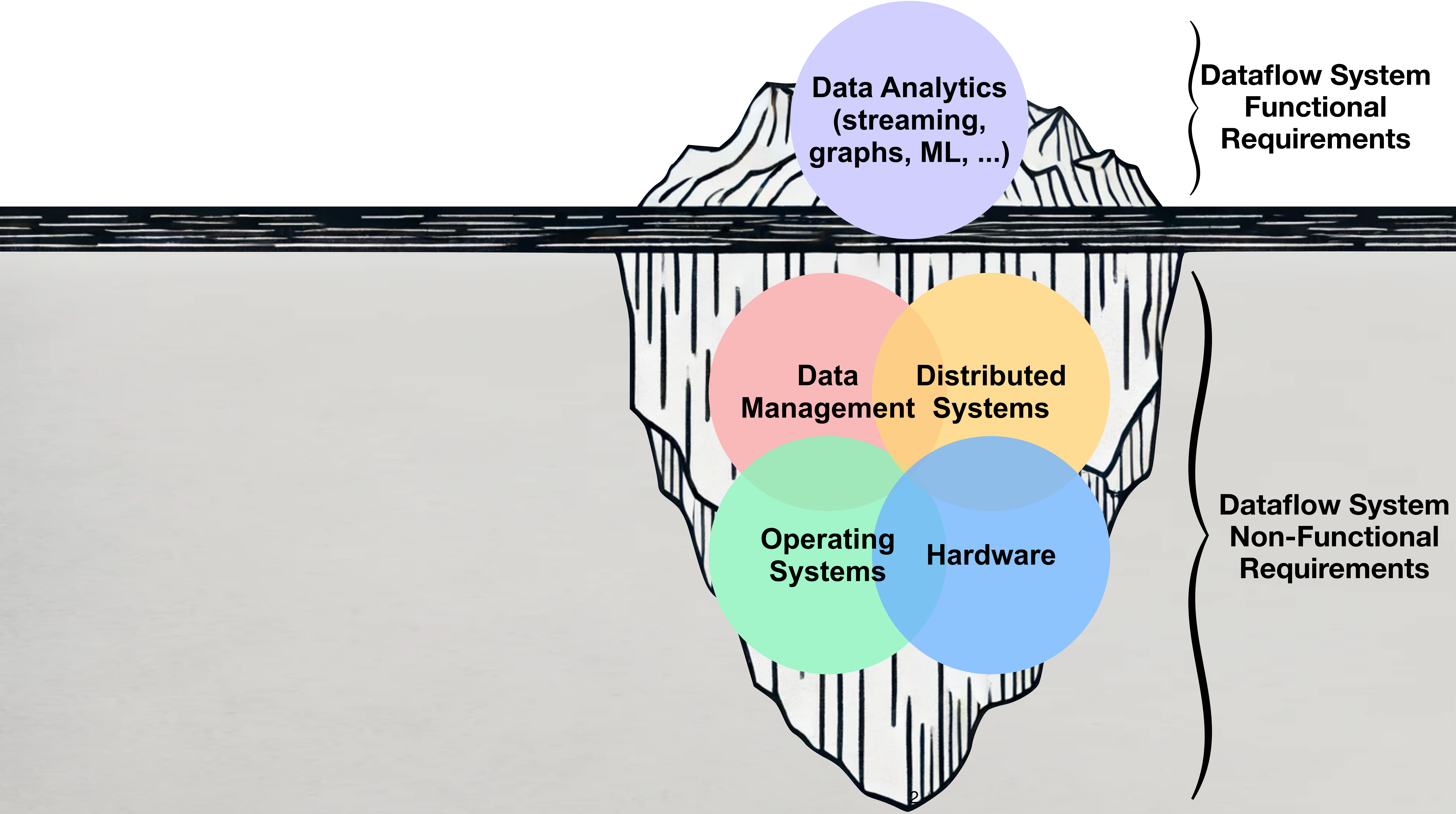
[1] KTH Royal Institute of Technology
[2] RISE Research Institutes of Sweden

DEBS 2024

The **Prospect** of Dataflow Systems

The **Prospect** of Dataflow Systems

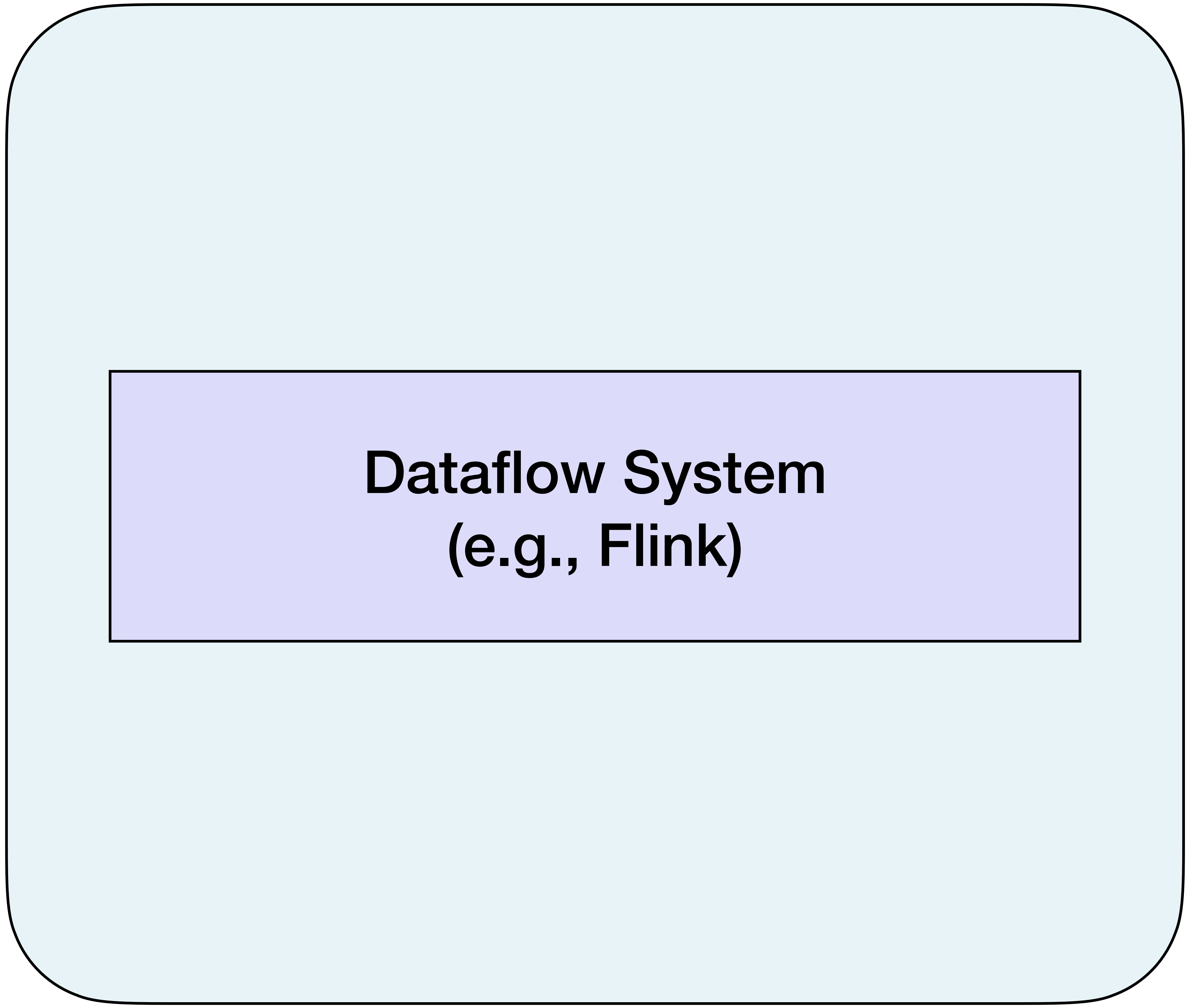
The **Prospect** of Dataflow Systems



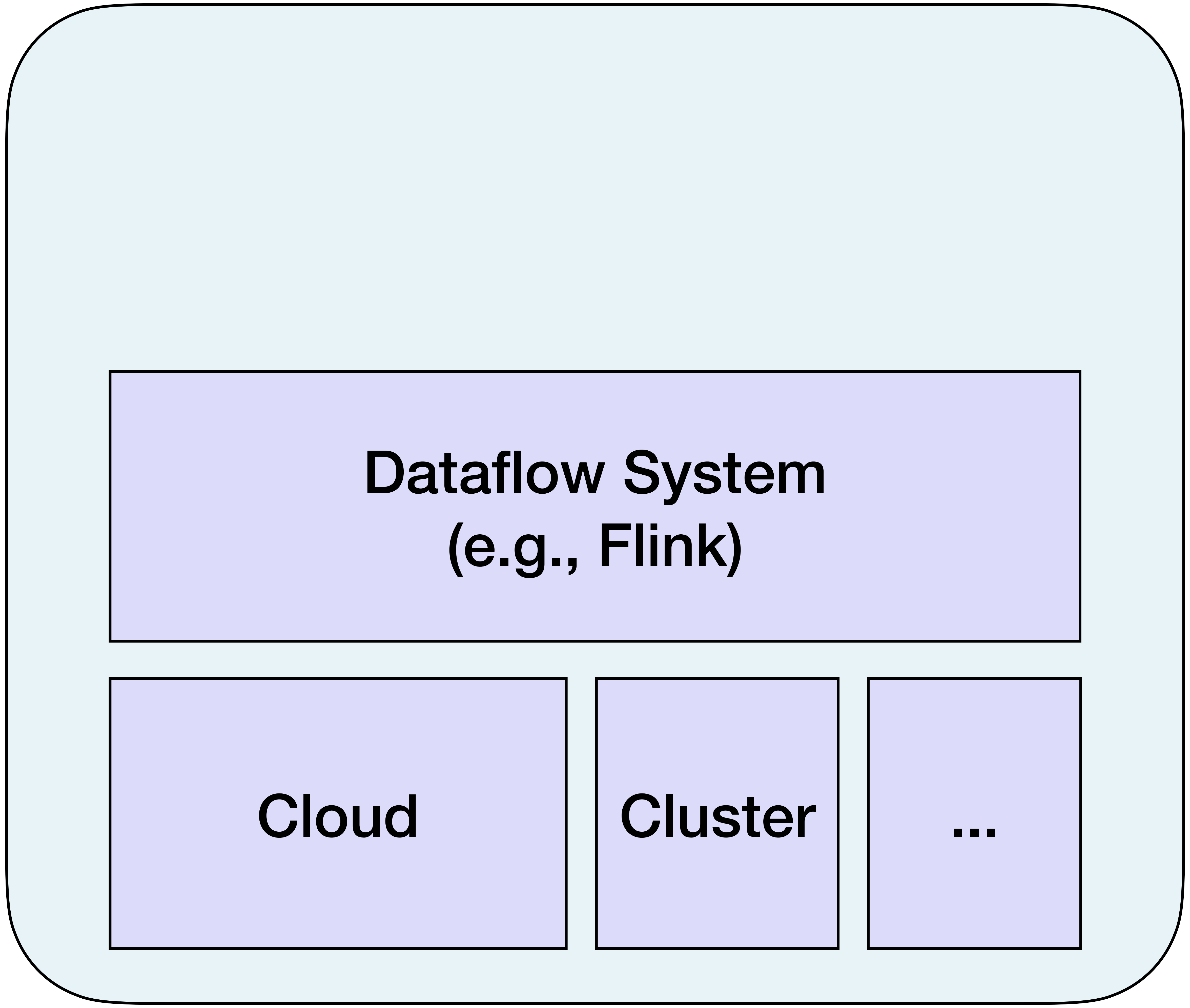
The **Prospect** of Dataflow Systems

What exists today

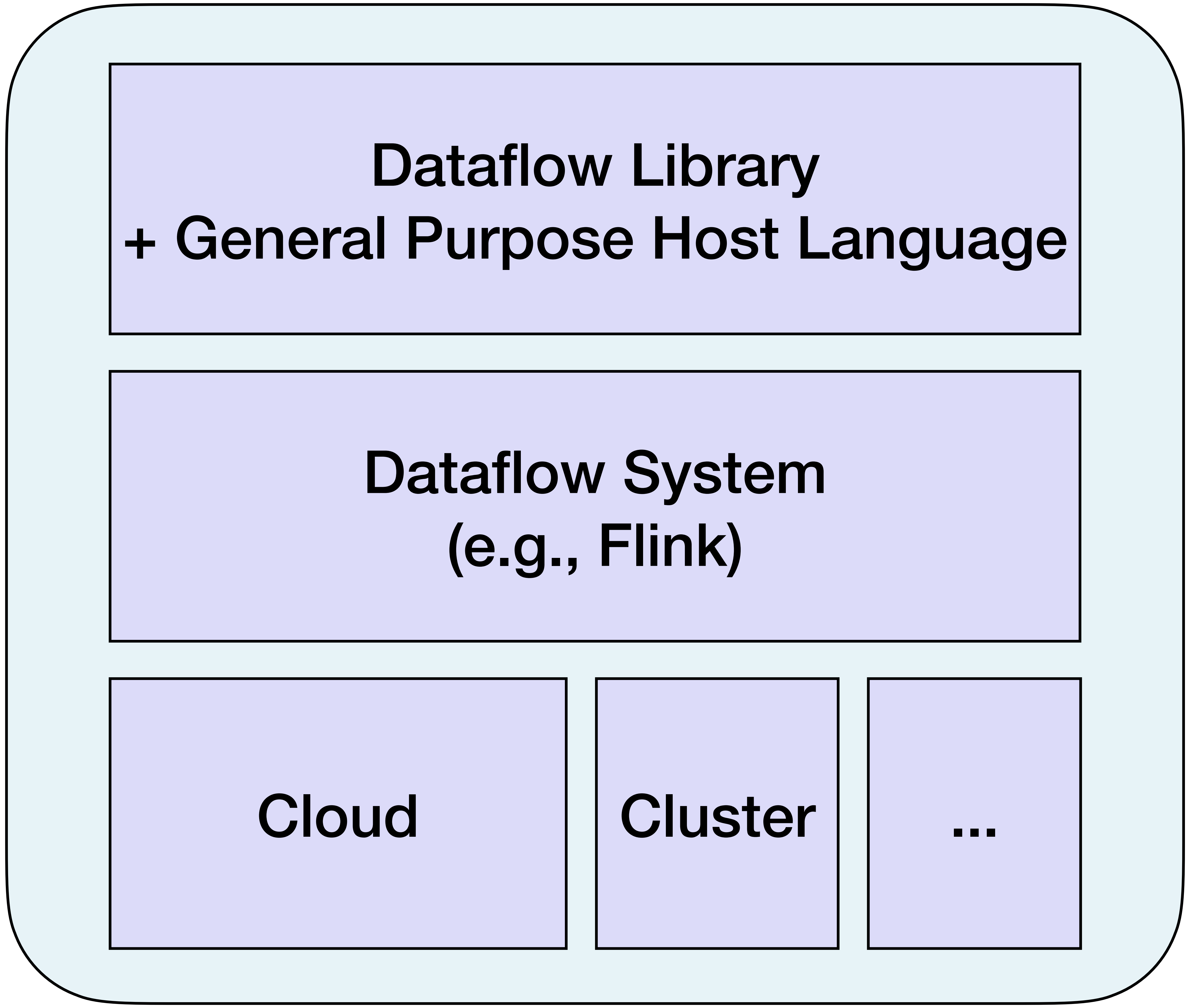
What exists today



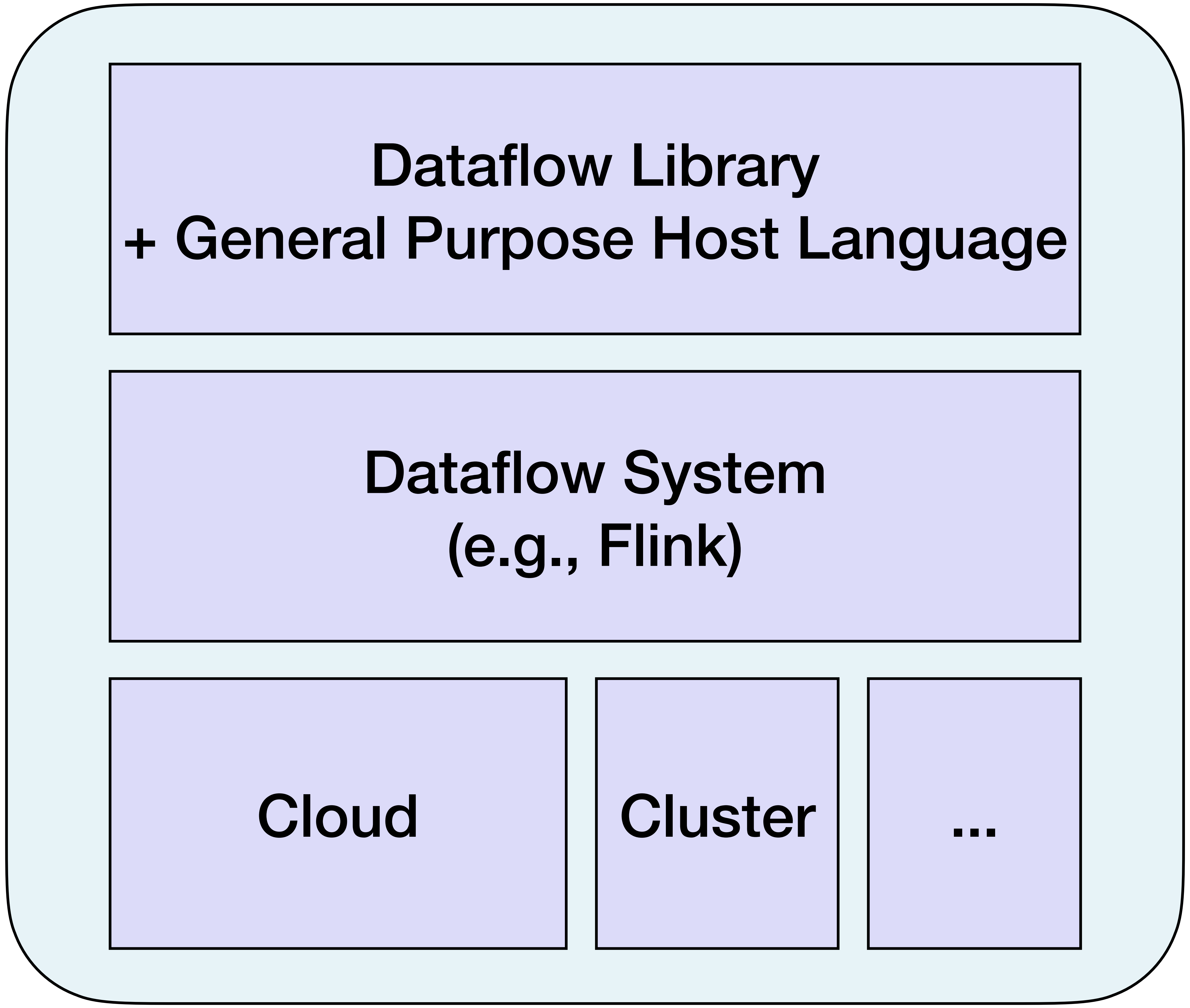
What exists today



What exists today



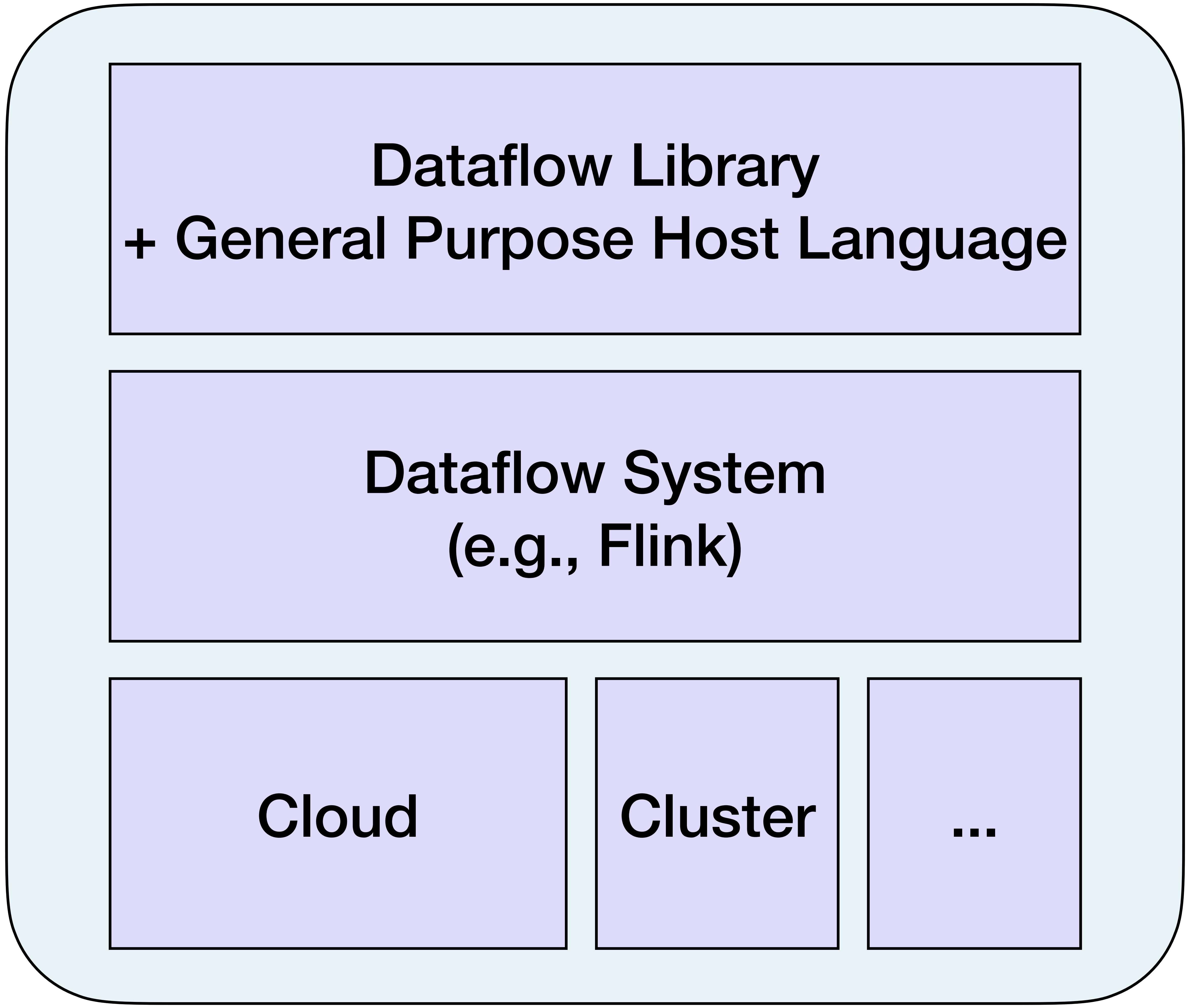
What exists today



Dataflow Libraries

Pros:
- Arbitrary code execution

What exists today

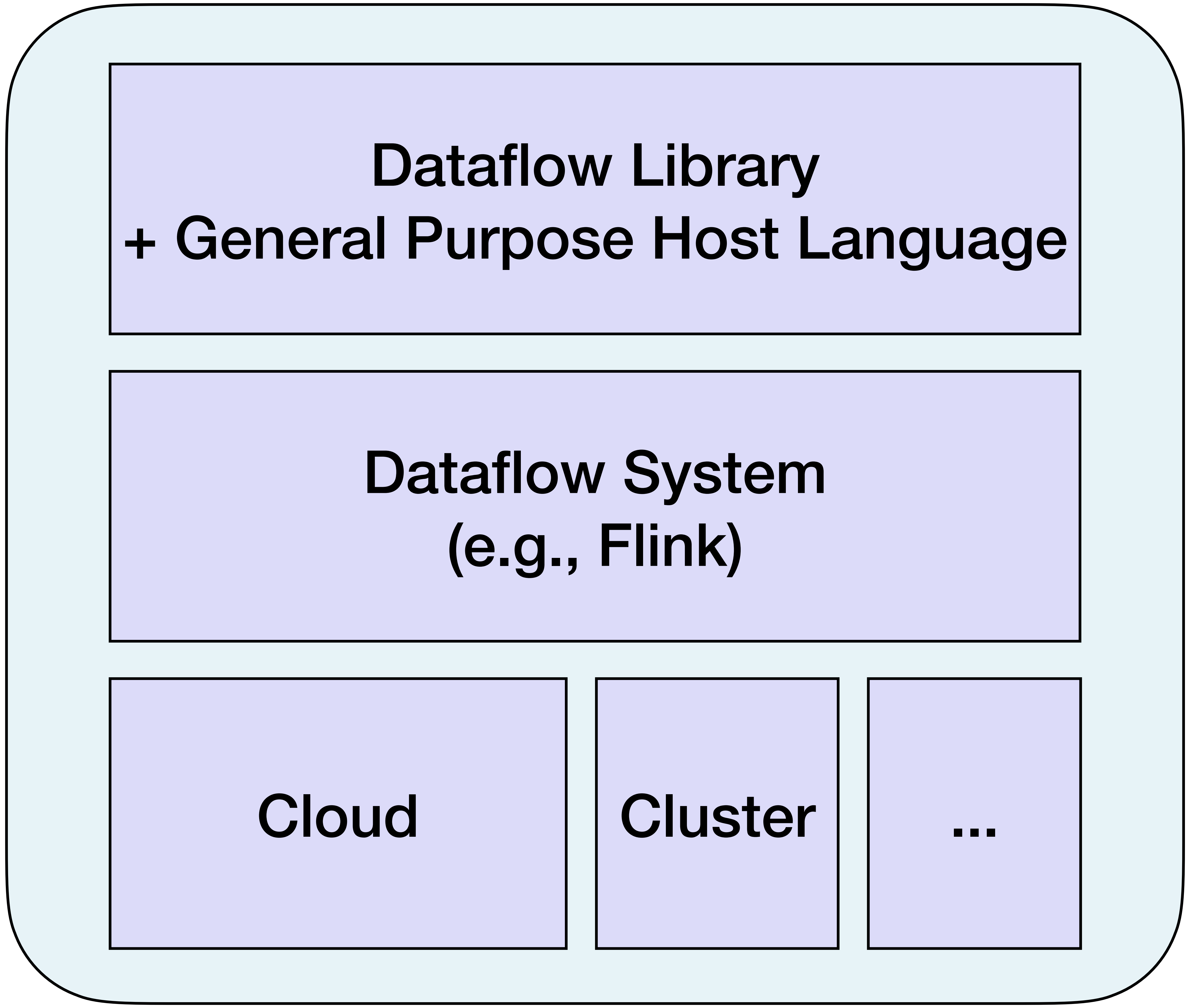


Dataflow Libraries

Pros:
- Arbitrary code execution

Cons:
- **Safety problems:**
  - Weak typing
  - Undefined behaviour
  - Untrusted code
  - ...

What exists today



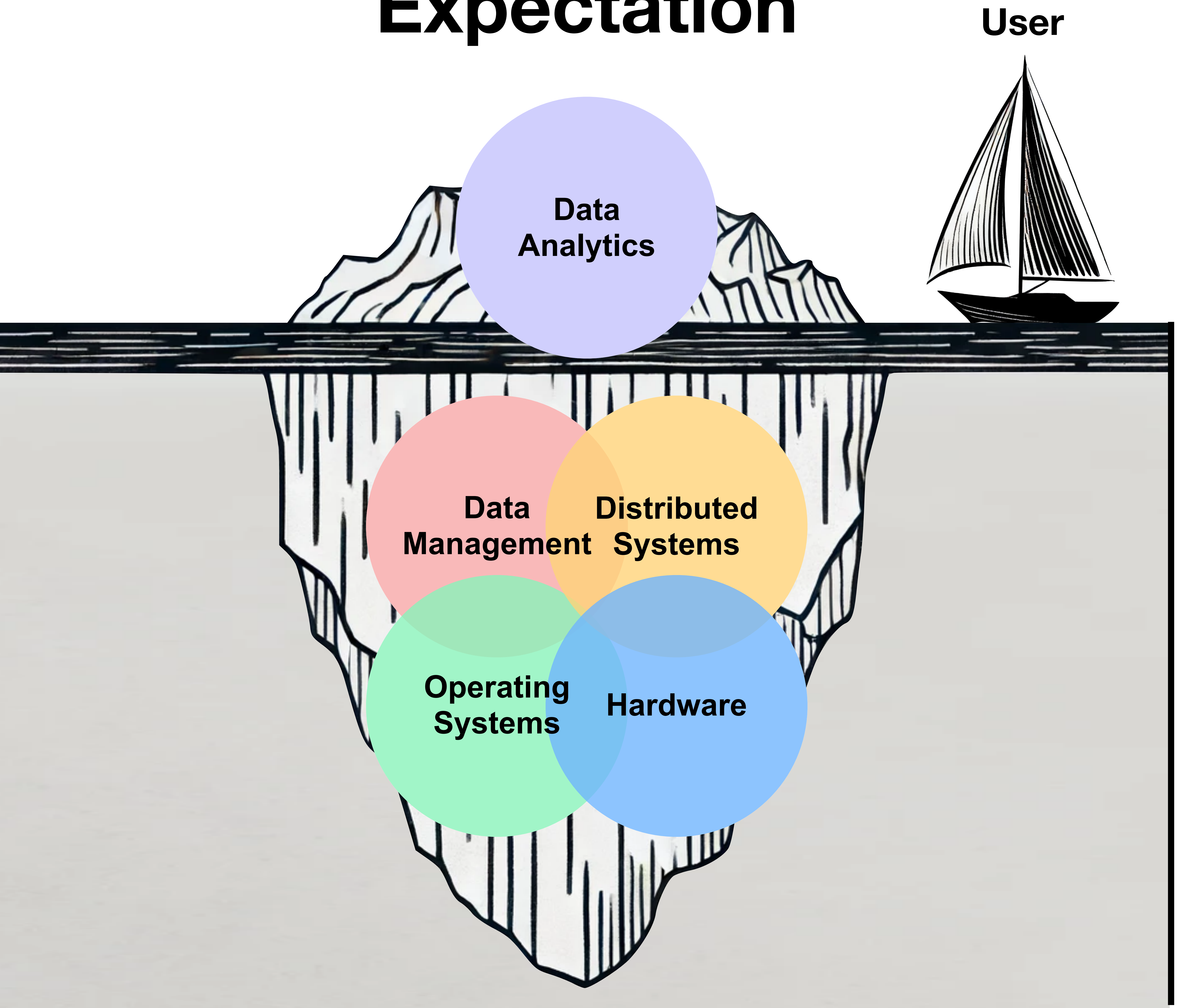
Dataflow Libraries

Pros:
- Arbitrary code execution

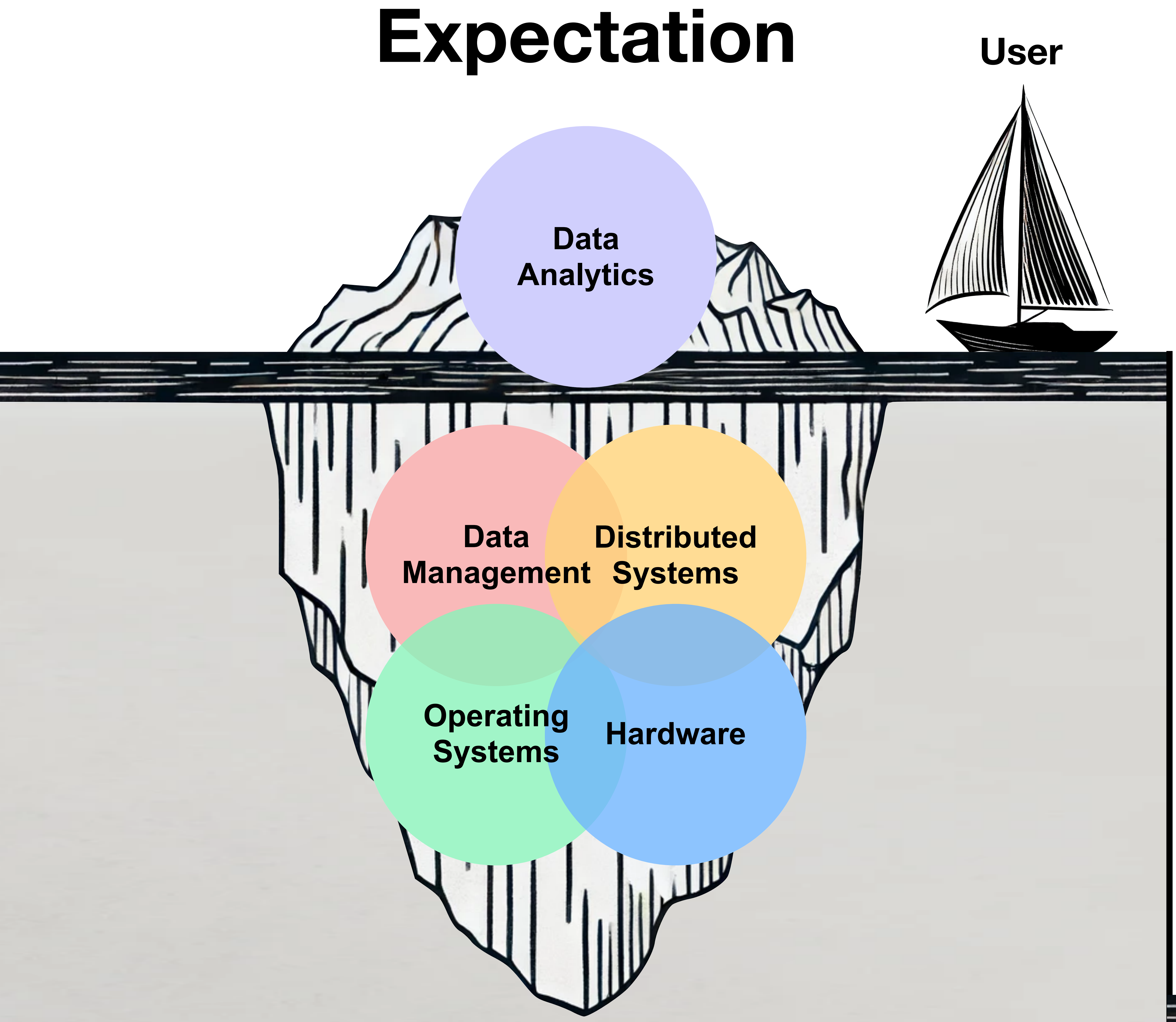
Cons:
- **Safety problems:**
  - Weak typing
  - Undefined behaviour
  - Untrusted code
  - ...
- **Performance problems:**
  - Unoptimisable Code
  - Performance hacks
  - Close coupling
  - ...

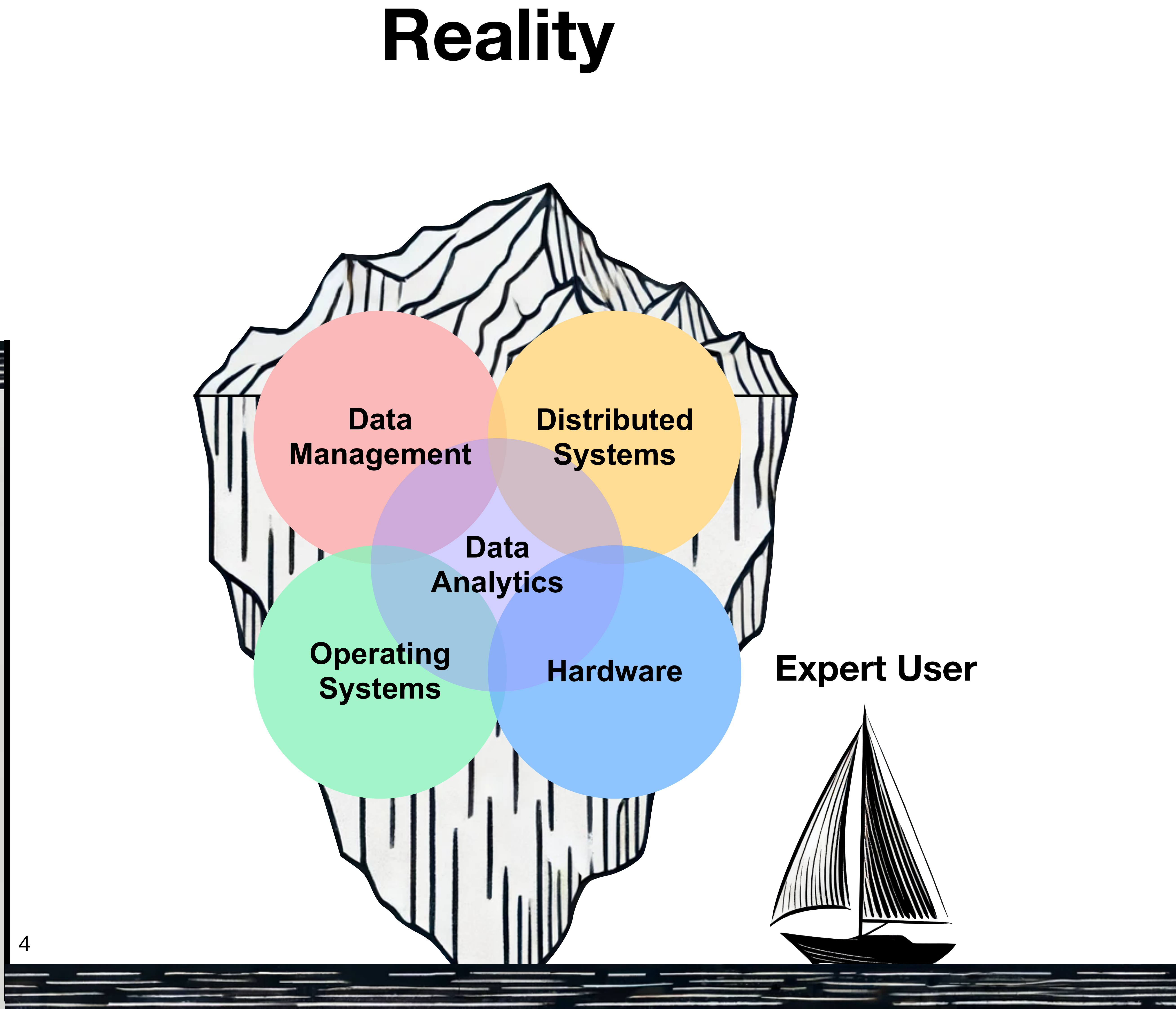
The **Reality** of Dataflow Systems

Expectation

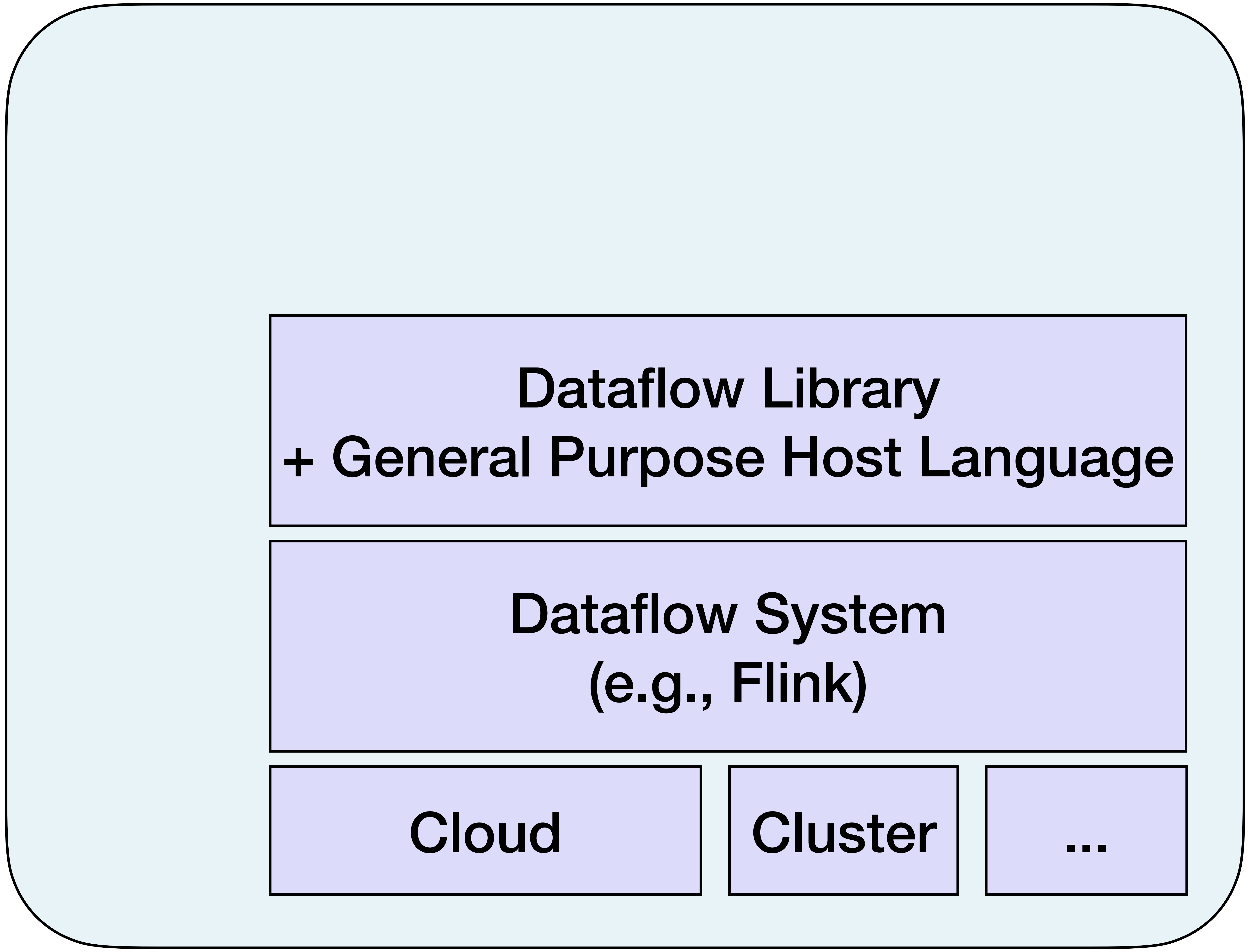


The **Reality** of Dataflow Systems

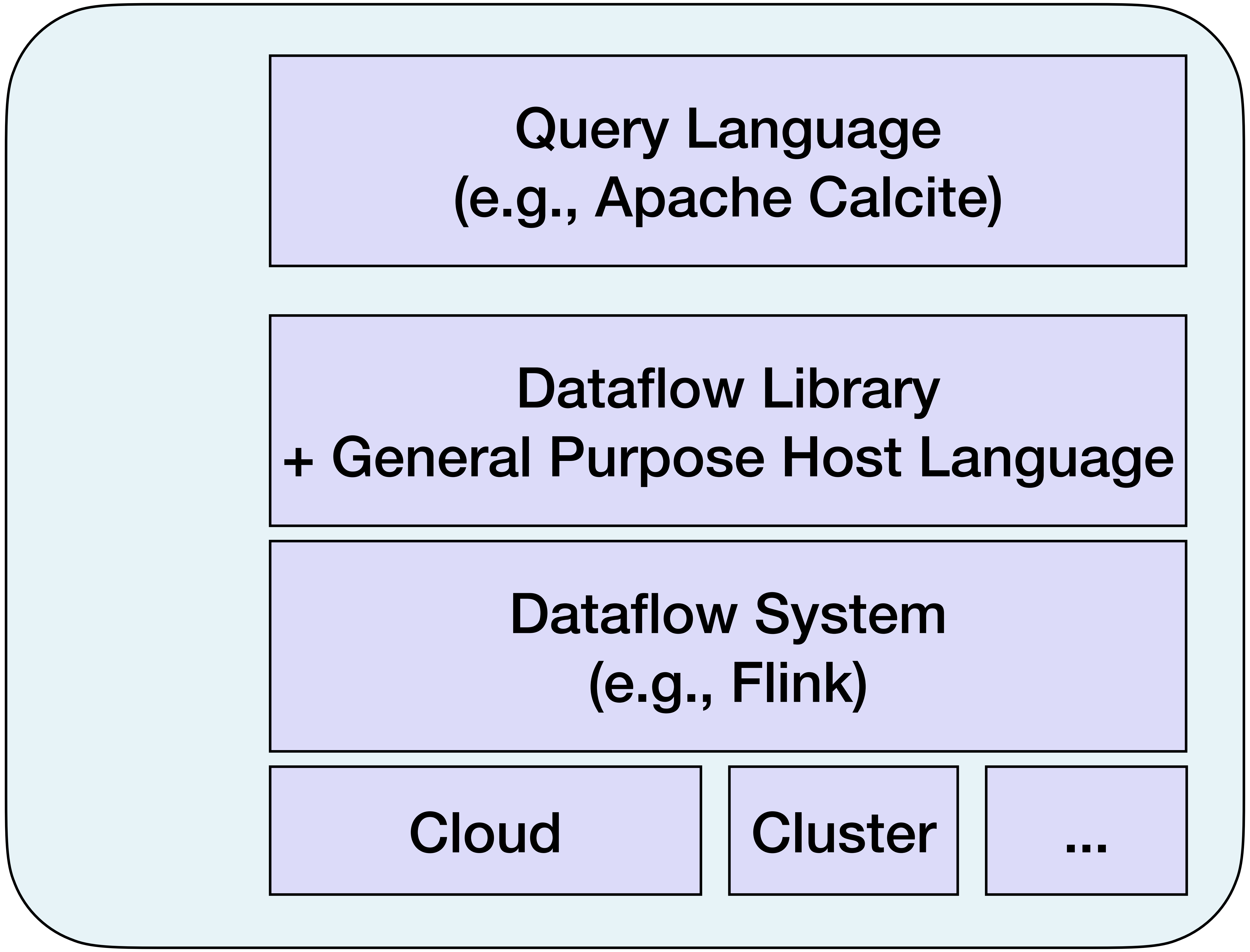




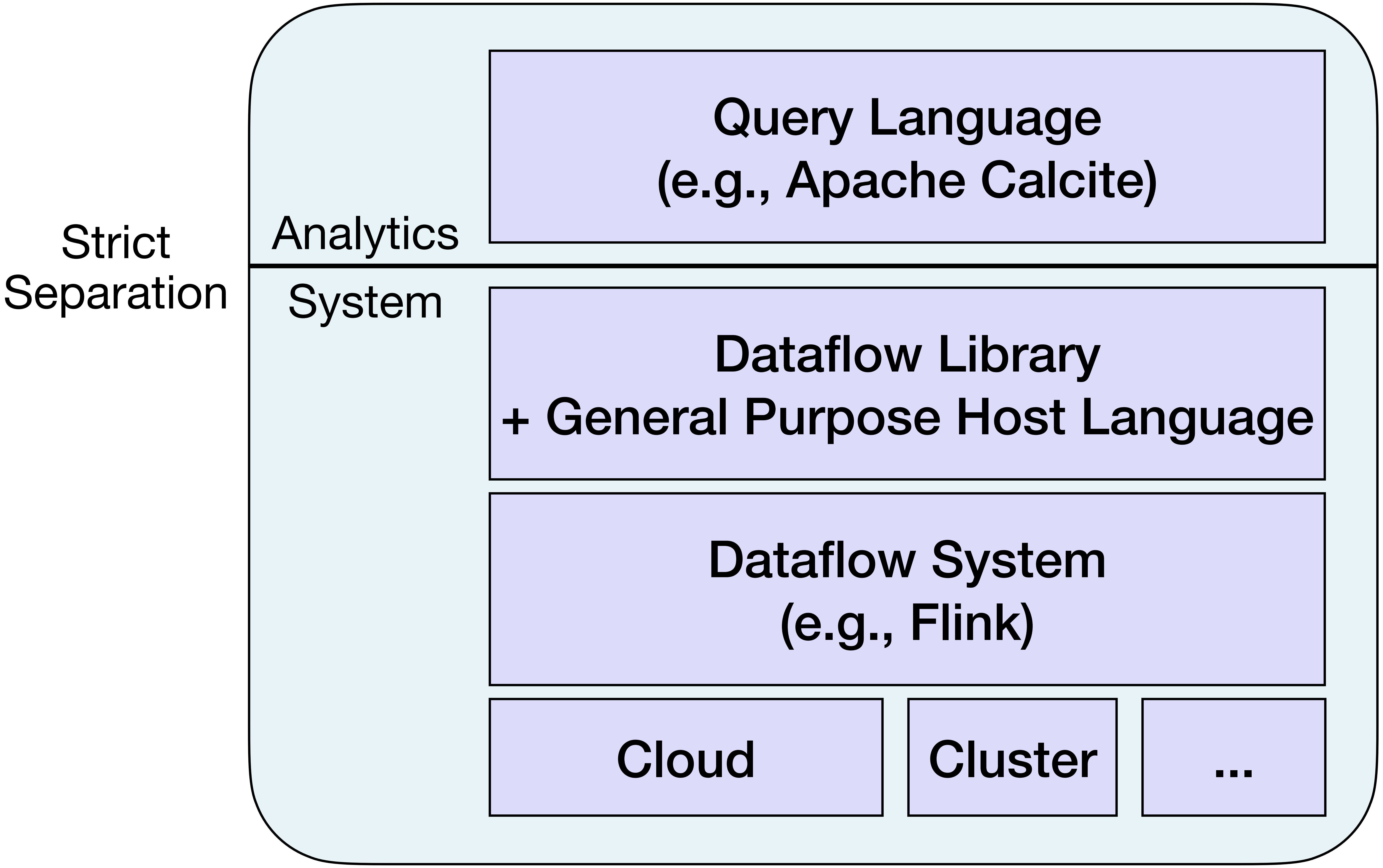
Other current options?



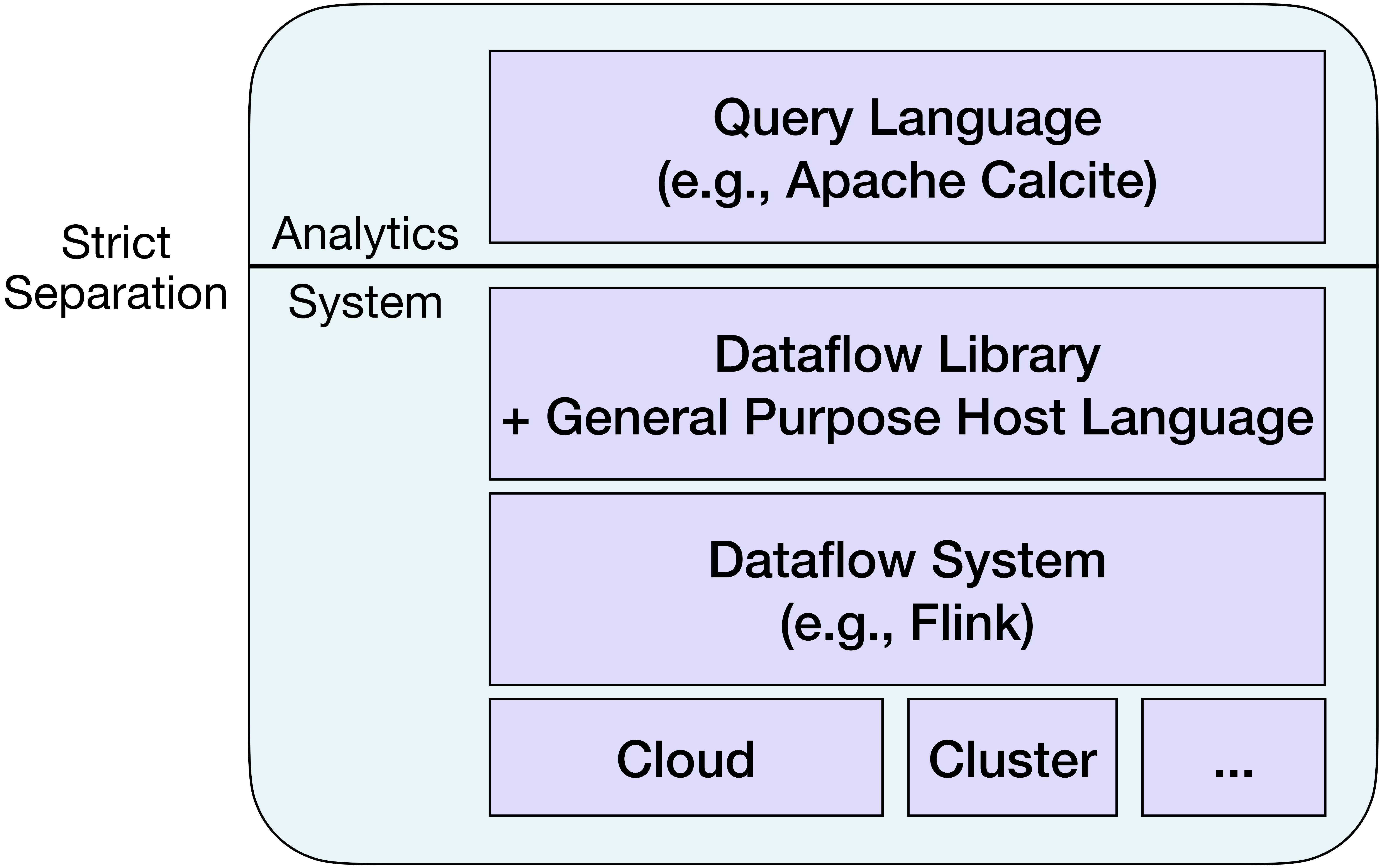
Other current options?



Other current options?



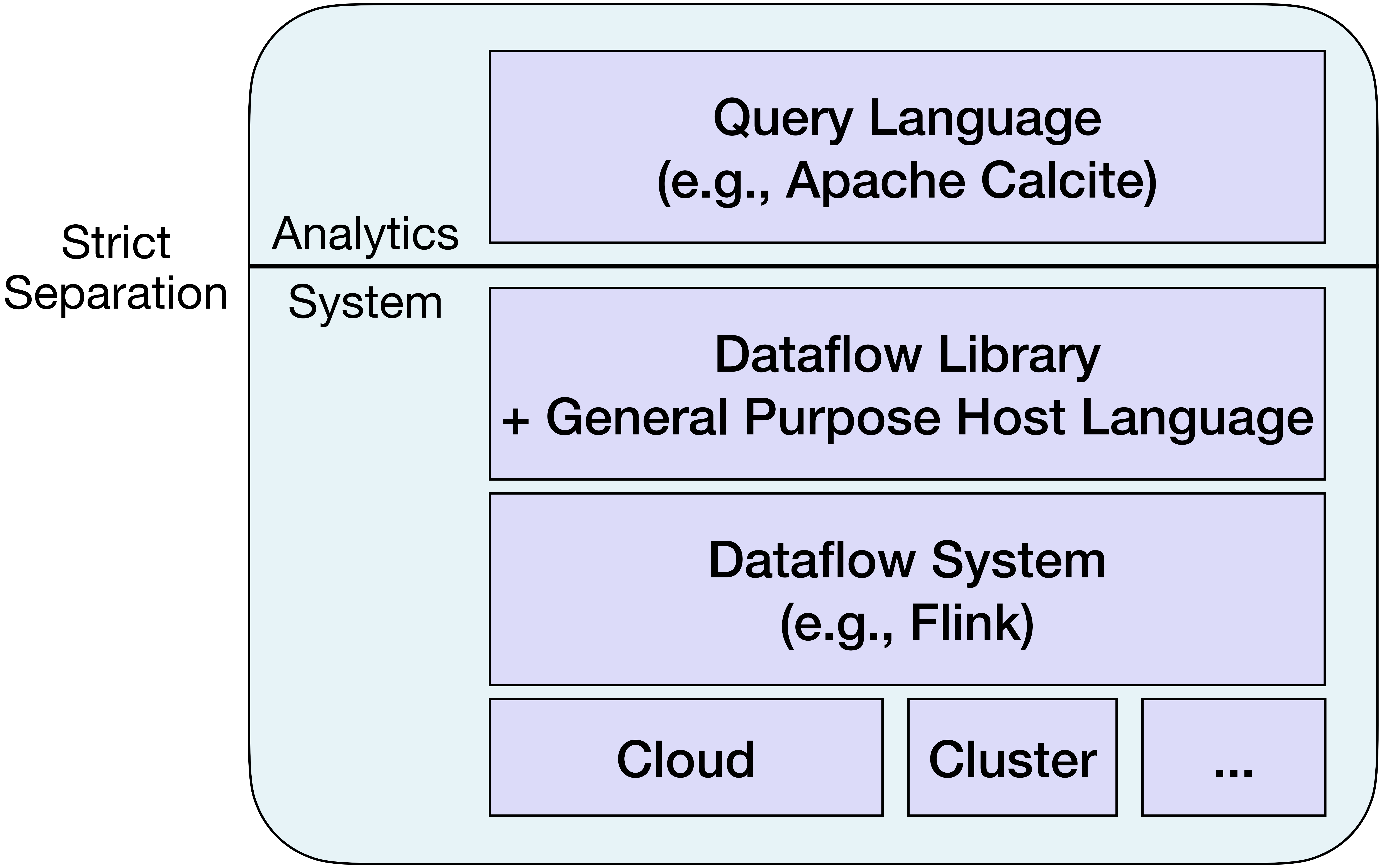
Other current options?



Query Languages

Pros:
- Safe, efficient and concise

Other current options?



Query Languages

Pros:
- Safe, efficient and concise

Cons:
- **Flexibility problems:**
  - Purely declarative code
  - Purely relational data
  - Limited extensibility
  - ...

Beyond Dataflow Libraries and Query Languages

Dataflow Library
+ General Purpose Host Language

+ Flexibility

???

Query Language

+ Safety, Conciseness, Efficiency

Beyond Dataflow Libraries and Query Languages

Best of both worlds

Dataflow Library
+ General Purpose Host Language

+ Flexibility

AquaLang

+ and more?

Query Language

+ Safety, Conciseness, Efficiency

AquaLang: Dataflow Programming

AquaLang: Dataflow Programming

```
struct Item(id:u32, usd:f64, kg:f64, time:Time);
```

AquaLang: Dataflow Programming

```
struct Item(id:u32, usd:f64, kg:f64, time:Time);

val s0: Stream[Item] = source(kafka("127.0.0.1", 8082, "us-items"), json(), _.time);
val s1: Stream[Item] = source(kafka("127.0.0.1", 8082, "uk-items"), json(), _.time);
```

AquaLang: Dataflow Programming

```
struct Item(id:u32, usd:f64, kg:f64, time:Time);

val s0: Stream[Item] = source(kafka("127.0.0.1", 8082, "us-items"), json(), _.time);
val s1: Stream[Item] = source(kafka("127.0.0.1", 8082, "uk-items"), json(), _.time);
```

AquaLang: Dataflow Programming

```
struct Item(id:u32, usd:f64, kg:f64, time:Time);

val s0: Stream[Item] = source(kafka("127.0.0.1", 8082, "us-items"), json(), _.time);
val s1: Stream[Item] = source(kafka("127.0.0.1", 8082, "uk-items"), json(), _.time);
```

AquaLang: Dataflow Programming

```
struct Item(id:u32, usd:f64, kg:f64, time:Time);

val s0: Stream[Item] = source(kafka("127.0.0.1", 8082, "us-items"), json(), _.time);
val s1: Stream[Item] = source(kafka("127.0.0.1", 8082, "uk-items"), json(), _.time);
```

AquaLang: Dataflow Programming

```
struct Item(id:u32, usd:f64, kg:f64, time:Time);

val s0: Stream[Item] = source(kafka("127.0.0.1", 8082, "us-items"), json(), _.time);
val s1: Stream[Item] = source(kafka("127.0.0.1", 8082, "uk-items"), json(), _.time);

val s2: Stream[Item] = merge(s0, s1);
```

AquaLang: Dataflow Programming

```
struct Item(id:u32, usd:f64, kg:f64, time:Time);

val s0: Stream[Item] = source(kafka("127.0.0.1", 8082, "us-items"), json(), _.time);
val s1: Stream[Item] = source(kafka("127.0.0.1", 8082, "uk-items"), json(), _.time);

val s2: Stream[Item] = merge(s0, s1);

val d0: Dataflow = sink(s2, kafka("127.0.0.1", 8082, "output1"), json());
val d1: Dataflow = sink(s2, kafka("127.0.0.1", 8082, "output2"), json());
```

AquaLang: Dataflow Programming

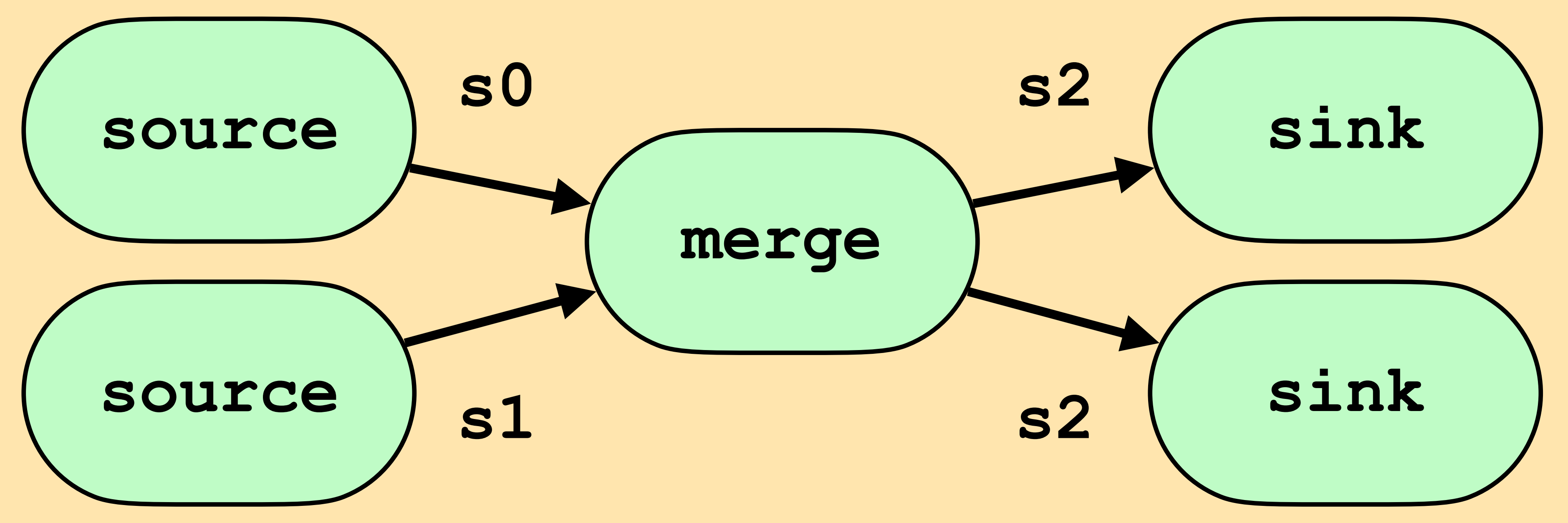
```
struct Item(id:u32, usd:f64, kg:f64, time:Time);

val s0: Stream[Item] = source(kafka("127.0.0.1", 8082, "us-items"), json(), _.time);
val s1: Stream[Item] = source(kafka("127.0.0.1", 8082, "uk-items"), json(), _.time);

val s2: Stream[Item] = merge(s0, s1);

val d0: Dataflow = sink(s2, kafka("127.0.0.1", 8082, "output1"), json());
val d1: Dataflow = sink(s2, kafka("127.0.0.1", 8082, "output2"), json());
```

AquaLang: Dataflow Programming

```
struct Item(id:u32, usd:f64, kg:f64, time:Time);

val s0: Stream[Item] = source(kafka("127.0.0.1", 8082, "us-items"), json(), _.time);
val s1: Stream[Item] = source(kafka("127.0.0.1", 8082, "uk-items"), json(), _.time);

val s2: Stream[Item] = merge(s0, s1);

val d0: Dataflow = sink(s2, kafka("127.0.0.1", 8082, "output1"), json());
val d1: Dataflow = sink(s2, kafka("127.0.0.1", 8082, "output2"), json());

val d2: Dataflow = compose([d0, d1]);
```

AquaLang: Dataflow Programming

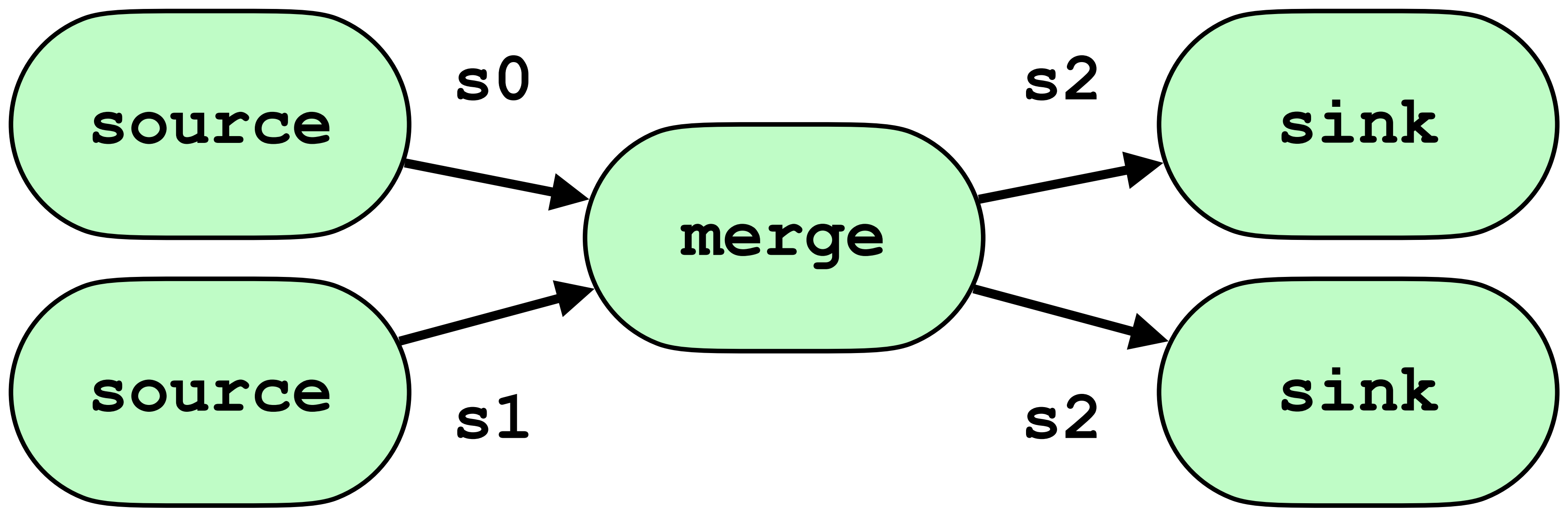
```
struct Item(id:u32, usd:f64, kg:f64, time:Time);

val s0: Stream[Item] = source(kafka("127.0.0.1", 8082, "us-items"), json(), _.time);
val s1: Stream[Item] = source(kafka("127.0.0.1", 8082, "uk-items"), json(), _.time);

val s2: Stream[Item] = merge(s0, s1);

val d0: Dataflow = sink(s2, kafka("127.0.0.1", 8082, "output1"), json());
val d1: Dataflow = sink(s2, kafka("127.0.0.1", 8082, "output2"), json());

val d2: Dataflow = compose([d0, d1]);
```



Dataflow graph d2

AquaLang: Dataflow Programming

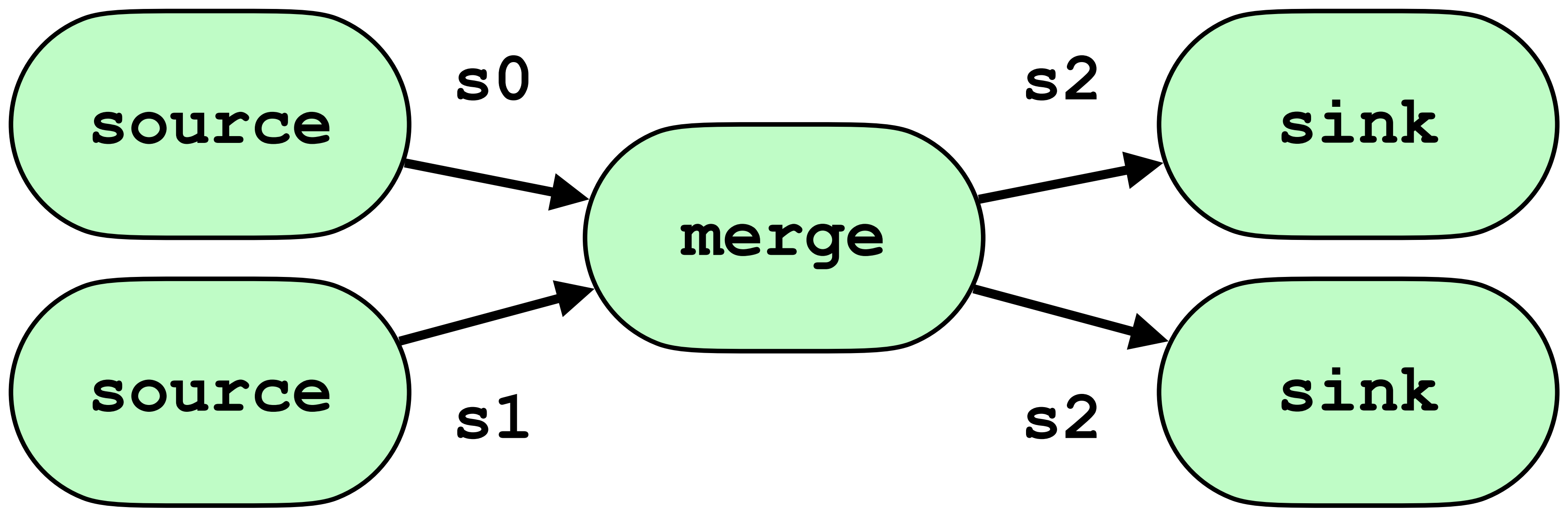
```
struct Item(id:u32, usd:f64, kg:f64, time:Time);

val s0: Stream[Item] = source(kafka("127.0.0.1", 8082, "us-items"), json(), _.time);
val s1: Stream[Item] = source(kafka("127.0.0.1", 8082, "uk-items"), json(), _.time);

val s2: Stream[Item] = merge(s0, s1);

val d0: Dataflow = sink(s2, kafka("127.0.0.1", 8082, "output1"), json());
val d1: Dataflow = sink(s2, kafka("127.0.0.1", 8082, "output2"), json());

val d2: Dataflow = compose([d0, d1]);

val i: Instance = run(d2, flink(…));
```



Dataflow graph d2

AquaLang: Dataflow Programming

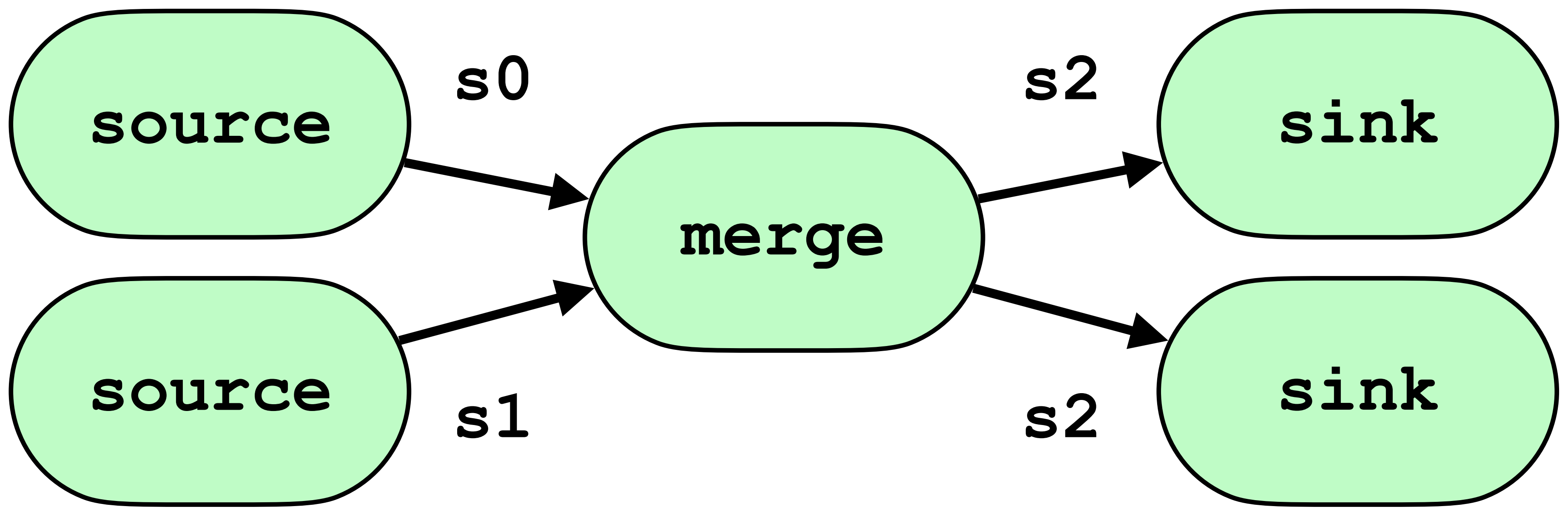
```
struct Item(id:u32, usd:f64, kg:f64, time:Time);

val s0: Stream[Item] = source(kafka("127.0.0.1", 8082, "us-items"), json(), _.time);
val s1: Stream[Item] = source(kafka("127.0.0.1", 8082, "uk-items"), json(), _.time);

val s2: Stream[Item] = merge(s0, s1);

val d0: Dataflow = sink(s2, kafka("127.0.0.1", 8082, "output1"), json());
val d1: Dataflow = sink(s2, kafka("127.0.0.1", 8082, "output2"), json());

val d2: Dataflow = compose([d0, d1]);

val i: Instance = run(d2, flink(…));
```



Dataflow graph d2

AquaLang: Dataflow Programming

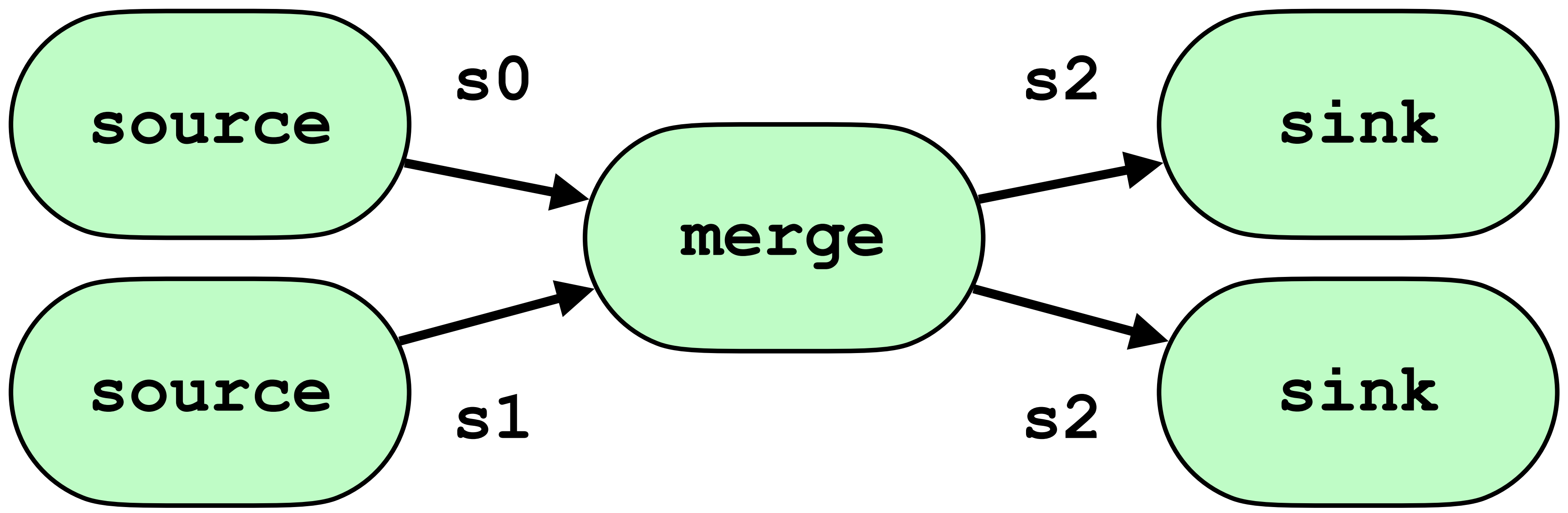
```
struct Item(id:u32, usd:f64, kg:f64, time:Time);

val s0: Stream[Item] = source(kafka("127.0.0.1", 8082, "us-items"), json(), _.time);
val s1: Stream[Item] = source(kafka("127.0.0.1", 8082, "uk-items"), json(), _.time);

val s2: Stream[Item] = merge(s0, s1);

val d0: Dataflow = sink(s2, kafka("127.0.0.1", 8082, "output1"), json());
val d1: Dataflow = sink(s2, kafka("127.0.0.1", 8082, "output2"), json());

val d2: Dataflow = compose([d0, d1]);

val i: Instance = run(d2, flink(…));
```



Dataflow graph d2

AquaLang: Dataflow Programming

```
struct Item(id:u32, usd:f64, kg:f64, time:Time);

val s0: Stream[Item] = source(kafka("127.0.0.1", 8082, "us-items"), json(), _.time);
val s1: Stream[Item] = source(kafka("127.0.0.1", 8082, "uk-items"), json(), _.time);

val s2: Stream[Item] = merge(s0, s1);

val d0: Dataflow = sink(s2, kafka("127.0.0.1", 8082, "output1"), json());
val d1: Dataflow = sink(s2, kafka("127.0.0.1", 8082, "output2"), json());

val d2: Dataflow = compose([d0, d1]);

val i: Instance = run(d2, flink(…));

stop(i); # Stop the running dataflow graph
```



Dataflow graph d2

AquaLang: Dataflow Operators

AquaLang: Dataflow Operators

```
# Sources and Sinks
def source[T](Reader, Encoding, fun(T):Time): Stream[T];
def sink[T](Stream[T], Writer, Encoding): Dataflow;
```

AquaLang: Dataflow Operators

```
# Sources and Sinks
def source[T](Reader, Encoding, fun(T):Time): Stream[T];
def sink[T](Stream[T], Writer, Encoding): Dataflow;

# Dataflow
def run(Dataflow, RuntimeConfig): Instance;
def stop(Instance);
# ...
```

AquaLang: Dataflow Operators

```
# Sources and Sinks
def source[T](Reader, Encoding, fun(T):Time): Stream[T];
def sink[T](Stream[T], Writer, Encoding): Dataflow;

# Dataflow
def run(Dataflow, RuntimeConfig): Instance;
def stop(Instance);
# ...

# Generic Transformations
def process[I,O](Stream[I], FSM[I,O]): Stream[O];
def keyBy[K,T](Stream[T], fun(T):K): KeyedStream[K,T];
# ...
```

AquaLang: Dataflow Operators

```
# Sources and Sinks
def source[T](Reader, Encoding, fun(T):Time): Stream[T];
def sink[T](Stream[T], Writer, Encoding): Dataflow;

# Dataflow
def run(Dataflow, RuntimeConfig): Instance;
def stop(Instance);
# ...

# Generic Transformations
def process[I,O](Stream[I], FSM[I,O]): Stream[O];
def keyBy[K,T](Stream[T], fun(T):K): KeyedStream[K,T];
# ...

# Specialized Transformations
def flatMap[I,O](Stream[I], fun(I):Vec[O]): Stream[O];
def window[I,O](Stream[I], Assigner, fun(Vec[I]):O): Stream[O];
# ...
```

AquaLang: Dataflow Programming

```
struct Item(id:u32, usd:f64, kg:f64, time:Time);
```

AquaLang: Dataflow Programming

```
struct Item(id:u32, usd:f64, kg:f64, time:Time);

# Functional Syntax Dataflow
source(…)
  .filter(fun(item:Item) = item.kg > 5.0)
  .map(fun(item) = struct(item.id, eur=item.usd*0.85))
  .sink(…).run(…);
```

AquaLang: Dataflow Programming

```
struct Item(id:u32, usd:f64, kg:f64, time:Time);

# Functional Syntax Dataflow
source(…)
  .filter(fun(item:Item) = item.kg > 5.0)
  .map(fun(item) = struct(item.id, eur=item.usd*0.85))
  .sink(…).run(…);
```

AquaLang: Dataflow Programming

```
struct Item(id:u32, usd:f64, kg:f64, time:Time);

# Functional Syntax Dataflow
source(…)
  .filter(fun(item:Item) = item.kg > 5.0)
  .map(fun(item) = struct(item.id, eur=item.usd*0.85))
  .sink(…).run(…);
```

AquaLang: Dataflow Programming

```
struct Item(id:u32, usd:f64, kg:f64, time:Time);

# Functional Syntax Dataflow
source(…)
  .filter(fun(item:Item) = item.kg > 5.0)
  .map(fun(item) = struct(item.id, eur=item.usd*0.85))
  .sink(…).run(…);

# Relational Syntax Dataflow
from item:Item in source(…)
where item.kg > 5.0
select item.id, eur=item.usd*0.85
into sink(…).run(…);
```

AquaLang: Dataflow Programming

```
struct Item(id:u32, usd:f64, kg:f64, time:Time);

# Functional Syntax Dataflow
source(…)
  .filter(fun(item:Item) = item.kg > 5.0)
  .map(fun(item) = struct(item.id, eur=item.usd*0.85))
  .sink(…).run(…);

# Relational Syntax Dataflow
from item:Item in source(…)
where item.kg > 5.0
select item.id, eur=item.usd*0.85
into sink(…).run(…);
```

AquaLang: Dataflow Programming

```
struct Item(id:u32, usd:f64, kg:f64, time:Time);

# Functional Syntax Dataflow
source(…)
  .filter(fun(item:Item) = item.kg > 5.0)
  .map(fun(item) = struct(item.id, eur=item.usd*0.85))
  .sink(…).run(…);

# Relational Syntax Dataflow
from item:Item in source(…)
where item.kg > 5.0
select item.id, eur=item.usd*0.85
into sink(…).run(…);
```

AquaLang: Dataflow Programming

```
struct Item(id:u32, usd:f64, kg:f64, time:Time);

# Functional Syntax Dataflow
source(…)
  .filter(fun(item:Item) = item.kg > 5.0)
  .map(fun(item) = struct(item.id, eur=item.usd*0.85))
  .sink(…).run(…);

# Relational Syntax Dataflow
from item:Item in source(…)
where item.kg > 5.0
select item.id, eur=item.usd*0.85
into sink(…).run(…);
```

AquaLang: Dataflow Programming

```
struct Item(id:u32, usd:f64, kg:f64, time:Time);

# Functional Syntax Dataflow
source(…)
  .filter(fun(item:Item) = item.kg > 5.0)
  .map(fun(item) = struct(item.id, eur=item.usd*0.85))
  .sink(…).run(…);

# Relational Syntax Dataflow
from item:Item in source(…)
where item.kg > 5.0
select item.id, eur=item.usd*0.85
into sink(…).run(…);
```

AquaLang: Dataflow Programming

```
struct Item(id:u32, usd:f64, kg:f64, time:Time);

# Functional Syntax Dataflow
source(…)
  .filter(fun(item:Item) = item.kg > 5.0)
  .map(fun(item) = struct(item.id, eur=item.usd*0.85))
  .sink(…).run(…);

# Relational Syntax Dataflow
from item:Item in source(…)
where item.kg > 5.0
select item.id, eur=item.usd*0.85
into sink(…).run(…);
```

Semantically Equivalent

AquaLang: Dataflow Programming

```
struct Item(id:u32, usd:f64, kg:f64, time:Time);

# Functional Syntax Dataflow
source(…)
  .filter(fun(item:Item) = item.kg > 5.0)
  .map(fun(item) = struct(item.id, eur=item.usd*0.85))
  .sink(…).run(…);

# Relational Syntax Dataflow
from item:Item in source(…)
where item.kg > 5.0
select item.id, eur=item.usd*0.85
into sink(…).run(…);
```

Semantically Equivalent

Equivalent Program
in **Apache Flink Table API:**

AquaLang: Dataflow Programming

```
struct Item(id:u32, usd:f64, kg:f64, time:Time);

# Functional Syntax Dataflow
source(…)
  .filter(fun(item:Item) = item.kg > 5.0)
  .map(fun(item) = struct(item.id, eur=item.usd*0.85))
  .sink(…).run(…);

# Relational Syntax Dataflow
from item:Item in source(…)
where item.kg > 5.0
select item.id, eur=item.usd*0.85
into sink(…).run(…);
```

Semantically Equivalent

Equivalent Program
in **Apache Flink Table API:**

```
val env = ExecutionEnvironment
   .getExecutionEnvironment()
val tEnv = StreamTableEnvironment.create(env)

tEnv.from("Input")
    .where($"kg" > 5.0)
    .select($"id", ($"usd" * 0.85).as("eur"))
    .insertInto("Output")
    .execute()
```

AquaLang: Dataflow Programming

```
struct Item(id:u32, usd:f64, kg:f64, time:Time);

# Functional Syntax Dataflow
source(…)
  .filter(fun(item:Item) = item.kg > 5.0)
  .map(fun(item) = struct(item.id, eur=item.usd*0.85))
  .sink(…).run(…);

# Relational Syntax Dataflow
from item:Item in source(…)
where item.kg > 5.0
select item.id, eur=item.usd*0.85
into sink(…).run(…);
```

Semantically Equivalent

Equivalent Program
in **Apache Flink Table API:**

```
val env = ExecutionEnvironment
   .getExecutionEnvironment()
val tEnv = StreamTableEnvironment.create(env)

tEnv.from("Input")
    .where($"kg" > 5.0)
    .select($"id", ($"usd" * 0.85).as("eur"))
    .insertInto("Output")
    .execute()
```

Weak string-based typing

AquaLang: Relational Syntax

AquaLang: Relational Syntax

```
struct Item(id:u32, usd:f64, kg:f64);
struct Order(id:u32, status:String, items:Vec[Item], time:Time);
```

AquaLang: Relational Syntax

```
struct Item(id:u32, usd:f64, kg:f64);
struct Order(id:u32, status:String, items:Vec[Item], time:Time);

from order:Order in source(…)
from item:Item in order.items
group orderId = order.id
  over tumbling(10min)
  compute sumItemUsd = sum of item.usd,
          maxItemUsd = max of item.usd
into sink(…).run(…);
```

AquaLang: Relational Syntax

```
struct Item(id:u32, usd:f64, kg:f64);
struct Order(id:u32, status:String, items:Vec[Item], time:Time);

from order:Order in source(…)
from item:Item in order.items
group orderId = order.id
  over tumbling(10min)
  compute sumItemUsd = sum of item.usd,
          maxItemUsd = max of item.usd
into sink(…).run(…);
```

AquaLang: Relational Syntax

```
struct Item(id:u32, usd:f64, kg:f64);
struct Order(id:u32, status:String, items:Vec[Item], time:Time);

from order:Order in source(…)
from item:Item in order.items
group orderId = order.id
  over tumbling(10min)
  compute sumItemUsd = sum of item.usd,
          maxItemUsd = max of item.usd
into sink(…).run(…);
```

AquaLang: Relational Syntax

```
struct Item(id:u32, usd:f64, kg:f64);
struct Order(id:u32, status:String, items:Vec[Item], time:Time);

from order:Order in source(…)
from item:Item in order.items
group orderId = order.id
  over tumbling(10min)
  compute sumItemUsd = sum of item.usd,
          maxItemUsd = max of item.usd
into sink(…).run(…);
```

AquaLang: Relational Syntax

```
struct Item(id:u32, usd:f64, kg:f64);
struct Order(id:u32, status:String, items:Vec[Item], time:Time);

from order:Order in source(…)
from item:Item in order.items
group orderId = order.id
  over tumbling(10min)
  compute sumItemUsd = sum of item.usd,
          maxItemUsd = max of item.usd
into sink(…).run(…);
```

AquaLang: Relational Syntax

```
struct Item(id:u32, usd:f64, kg:f64);
struct Order(id:u32, status:String, items:Vec[Item], time:Time);

from order:Order in source(…)
from item:Item in order.items
group orderId = order.id
  over tumbling(10min)
  compute sumItemUsd = sum of item.usd,
          maxItemUsd = max of item.usd
into sink(…).run(…);
```

AquaLang: Relational Syntax

```
struct Item(id:u32, usd:f64, kg:f64);
struct Order(id:u32, status:String, items:Vec[Item], time:Time);

from order:Order in source(…)
from item:Item in order.items
group orderId = order.id
  over tumbling(10min)
  compute sumItemUsd = sum of item.usd,
          maxItemUsd = max of item.usd
into sink(…).run(…);
```

AquaLang: Relational Syntax

```
struct Item(id:u32, usd:f64, kg:f64);
struct Order(id:u32, status:String, items:Vec[Item], time:Time);

from order:Order in source(…)
from item:Item in order.items
group orderId = order.id
  over tumbling(10min)
  compute sumItemUsd = sum of item.usd,
          maxItemUsd = max of item.usd
into sink(…).run(…);
```

AquaLang: Functional-Relational Composition

```
struct Item(id:u32, usd:f64, kg:f64);
struct Order(id:u32, status:String, items:Vec[Item], time:Time);

from order:Order in source(…)
from item:Item in order.items
group orderId = order.id
  over tumbling(10min)
  compute sumItemUsd = sum of item.usd,
          maxItemUsd = max of item.usd
into sink(…).run(…);

def query(orders: Stream[Order]): Stream[_] =
  from order:Order in orders
  from item in order.items
  group orderId = order.id
    over tumbling(10min)
    compute sumItemUsd = sum of item.usd,
            maxItemUsd = max of item.usd;
```

AquaLang: Functional-Relational Composition

```
struct Item(id:u32, usd:f64, kg:f64);
struct Order(id:u32, status:String, items:Vec[Item], time:Time);

from order:Order in source(…)
from item:Item in order.items
group orderId = order.id
  over tumbling(10min)
  compute sumItemUsd = sum of item.usd,
          maxItemUsd = max of item.usd
into sink(…).run(…);

def query(orders: Stream[Order]): Stream[_] =
  from order:Order in orders
  from item in order.items
  group orderId = order.id
    over tumbling(10min)
    compute sumItemUsd = sum of item.usd,
            maxItemUsd = max of item.usd;
```

AquaLang: Functional-Relational Composition

```
struct Item(id:u32, usd:f64, kg:f64);
struct Order(id:u32, status:String, items:Vec[Item], time:Time);

from order:Order in source(…)
from item:Item in order.items
group orderId = order.id
  over tumbling(10min)
  compute sumItemUsd = sum of item.usd,
          maxItemUsd = max of item.usd
into sink(…).run(…);

def query(orders: Stream[Order]): Stream[_] =
  from order:Order in orders
  from item in order.items
  group orderId = order.id
    over tumbling(10min)
    compute sumItemUsd = sum of item.usd,
            maxItemUsd = max of item.usd;
```

AquaLang: Functional-Relational Composition

```
struct Item(id:u32, usd:f64, kg:f64);
struct Order(id:u32, status:String, items:Vec[Item], time:Time);

from order:Order in source(…)
from item:Item in order.items
group orderId = order.id
  over tumbling(10min)
  compute sumItemUsd = sum of item.usd,
          maxItemUsd = max of item.usd
into sink(…).run(…);

def query(orders: Stream[Order]): Stream[_] =
  from order:Order in orders
  from item in order.items
  group orderId = order.id
    over tumbling(10min)
    compute sumItemUsd = sum of item.usd,
            maxItemUsd = max of item.usd;

from event in query(source(…))
select event.orderId, sumItemEur = event.sumItemUsd * 0.85
into source(…);
```

AquaLang: Functional-Relational Composition

```
struct Item(id:u32, usd:f64, kg:f64);
struct Order(id:u32, status:String, items:Vec[Item], time:Time);

from order:Order in source(…)
from item:Item in order.items
group orderId = order.id
  over tumbling(10min)
  compute sumItemUsd = sum of item.usd,
          maxItemUsd = max of item.usd
into sink(…).run(…);

def query(orders: Stream[Order]): Stream[_] =
  from order:Order in orders
  from item in order.items
  group orderId = order.id
    over tumbling(10min)
    compute sumItemUsd = sum of item.usd,
            maxItemUsd = max of item.usd;

from event in query(source(…))
select event.orderId, sumItemEur = event.sumItemUsd * 0.85
into source(…);
```

More details in the paper:

e ::= ...	Expression
| from x in e $q_1 \ldots q_n$	Query Expression
| from x in e $q_1 \ldots q_n$ into x($e_1, \ldots, e_n$)	Query Composition
q ::= ...	Query clause
| from x in e | val x = e	Iteration, Variable Def.
| where e | select x_1=e_1,...,x_n=e_n	Selection, Projection
| over e compute $a_1, \ldots, a_n$	Window Agg.
| group x=e over e compute $a_1, \ldots, a_n$	Keyed Window Agg.
| join x in e_1 on e_2 == e_3	Equi-Join
| join x in e_1 over e_2 on e_3 == e_3	Window Equi-Join
a ::= x=e_1 of e_2	Aggregation

Figure 3: Subset of the relational syntax of AquaLang.

AquaLang: Effect System

Preventing Undefined Behaviour

AquaLang: Effect System

Preventing Undefined Behaviour

```
val filePaths: Stream[String] = source(…);

filePaths
  .flatMap(fun(path:String): Vec[String] = open(path).readString().split(" "))
  .sink(…).run(…);
```

AquaLang: Effect System

Preventing Undefined Behaviour

```
val filePaths: Stream[String] = source(…);

filePaths
  .flatMap(fun(path:String): Vec[String] = open(path).readString().split(" "))
  .sink(…).run(…);
```

AquaLang: Effect System

Preventing Undefined Behaviour

```
val filePaths: Stream[String] = source(…);

filePaths
  .flatMap(fun(path:String): Vec[String] = open(path).readString().split(" "))
  .sink(…).run(…);
```

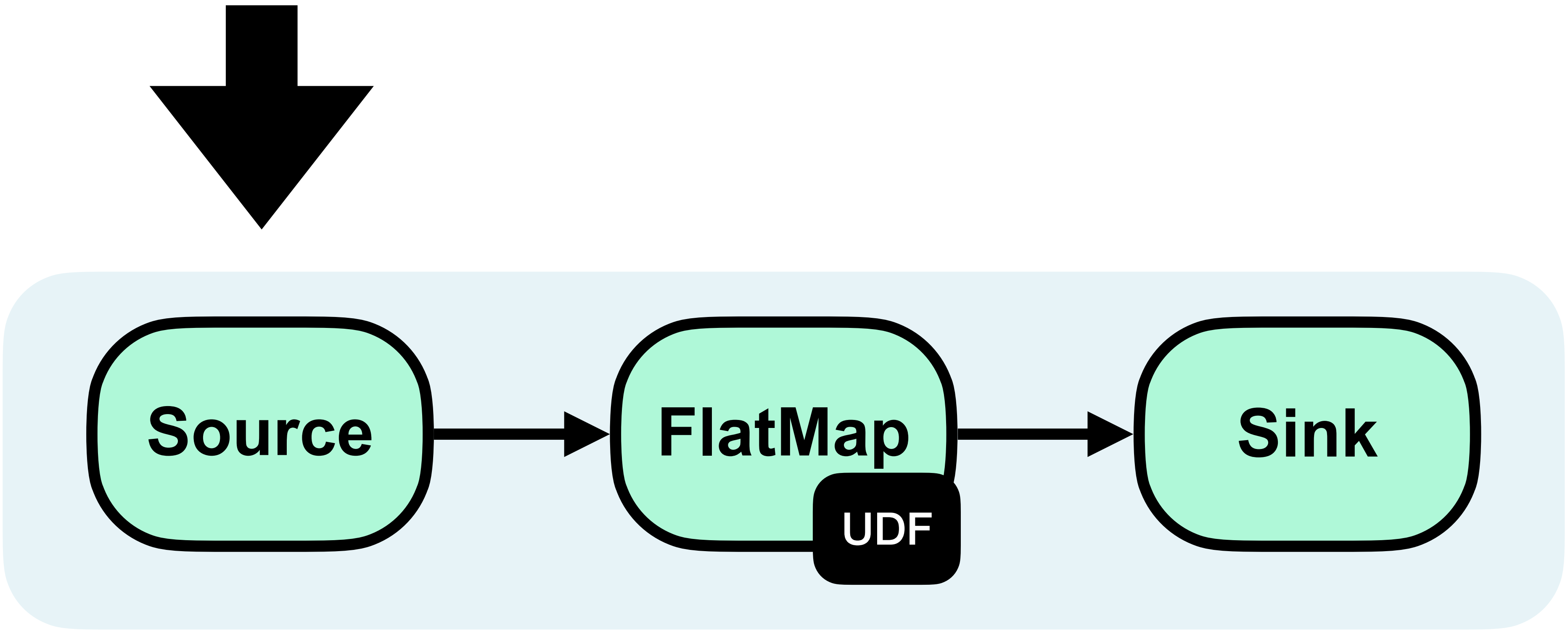
Semantics?

AquaLang: Effect System

Preventing Undefined Behaviour

```
val filePaths: Stream[String] = source(…);

filePaths
  .flatMap(fun(path:String): Vec[String] = open(path).readString().split(" "))
  .sink(…).run(…);
```

Semantics?



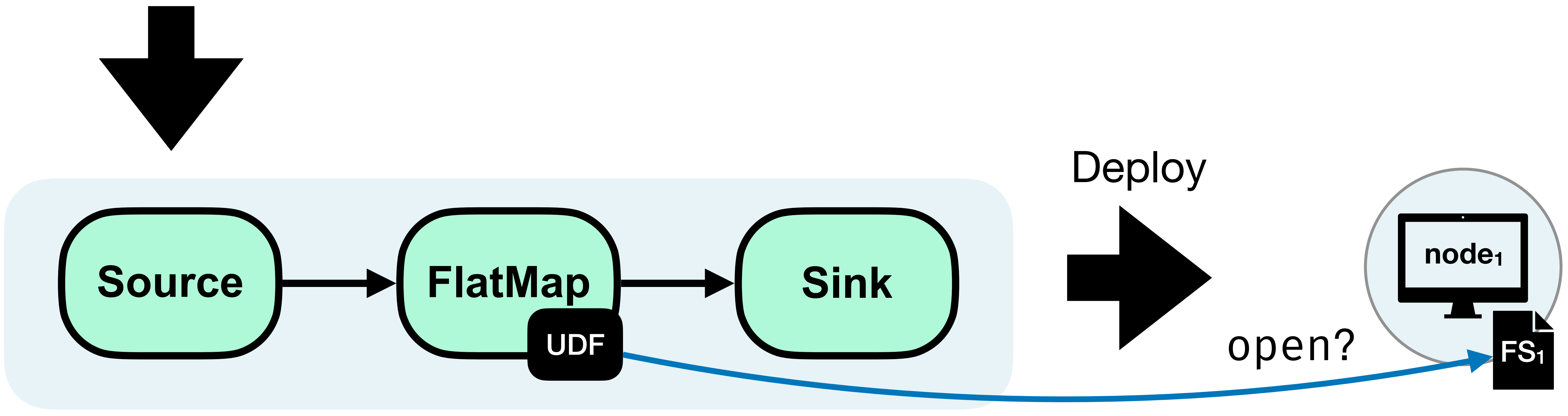
Logical Dataflow Graph

AquaLang: Effect System

Preventing Undefined Behaviour

```
val filePaths: Stream[String] = source(…);

filePaths
  .flatMap(fun(path:String): Vec[String] = open(path).readString().split(" "))
  .sink(…).run(…);
```

Semantics?



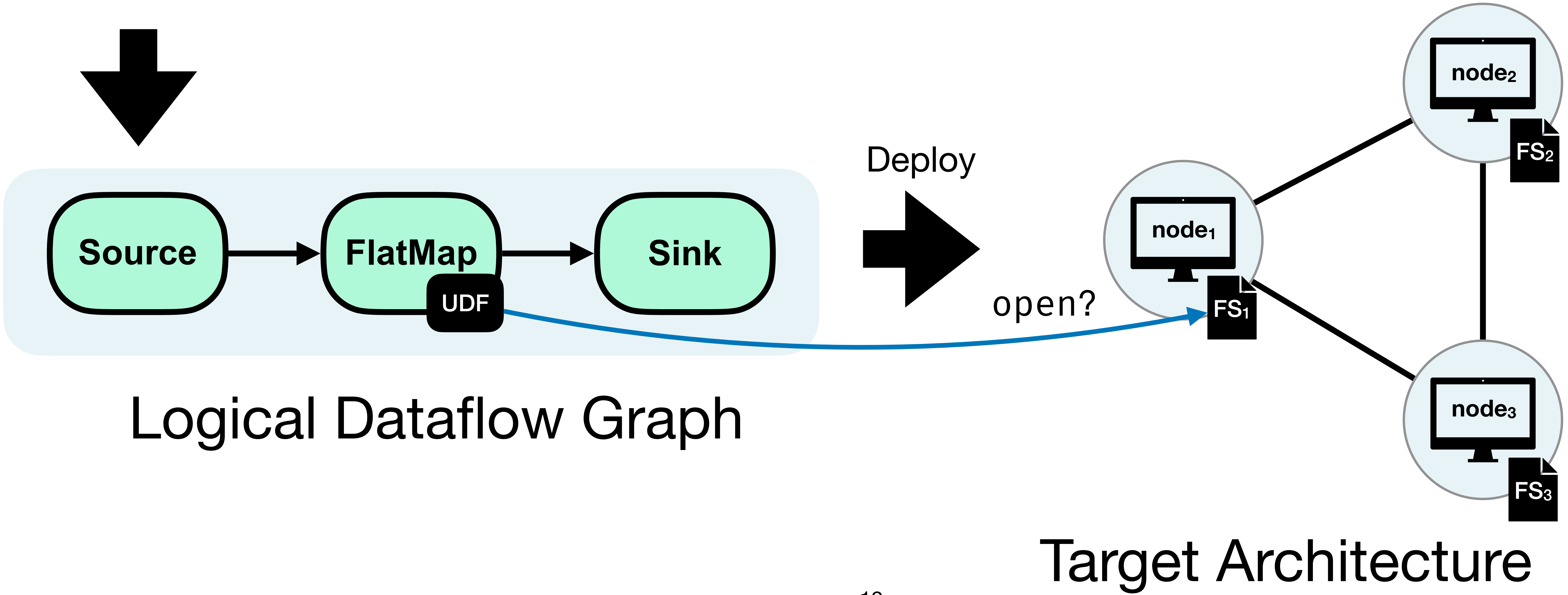
Logical Dataflow Graph

AquaLang: Effect System

Preventing Undefined Behaviour

Semantics?

```
val filePaths: Stream[String] = source(…);

filePaths
  .flatMap(fun(path:String): Vec[String] = open(path).readString().split(" "))
  .sink(…).run(…);
```

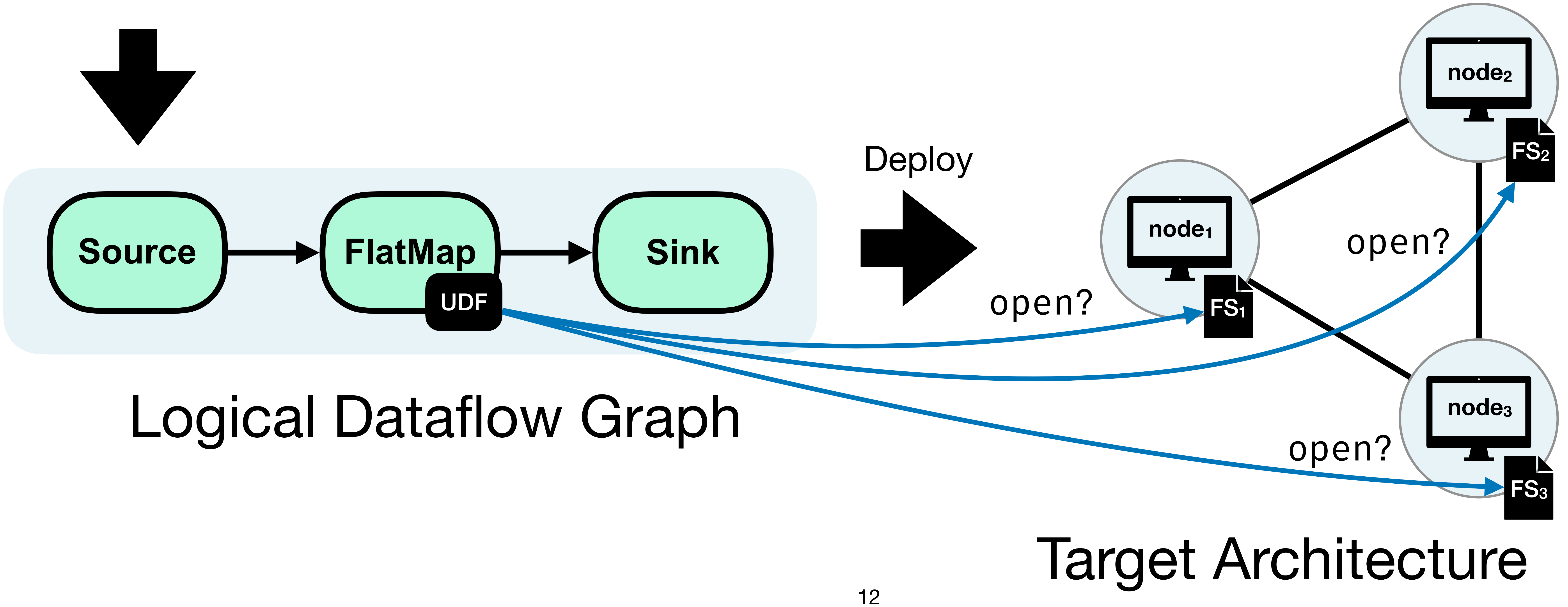


AquaLang: Effect System

Preventing Undefined Behaviour

Semantics?

```
val filePaths: Stream[String] = source(…);

filePaths
  .flatMap(fun(path:String): Vec[String] = open(path).readString().split(" "))
  .sink(…).run(…);
```

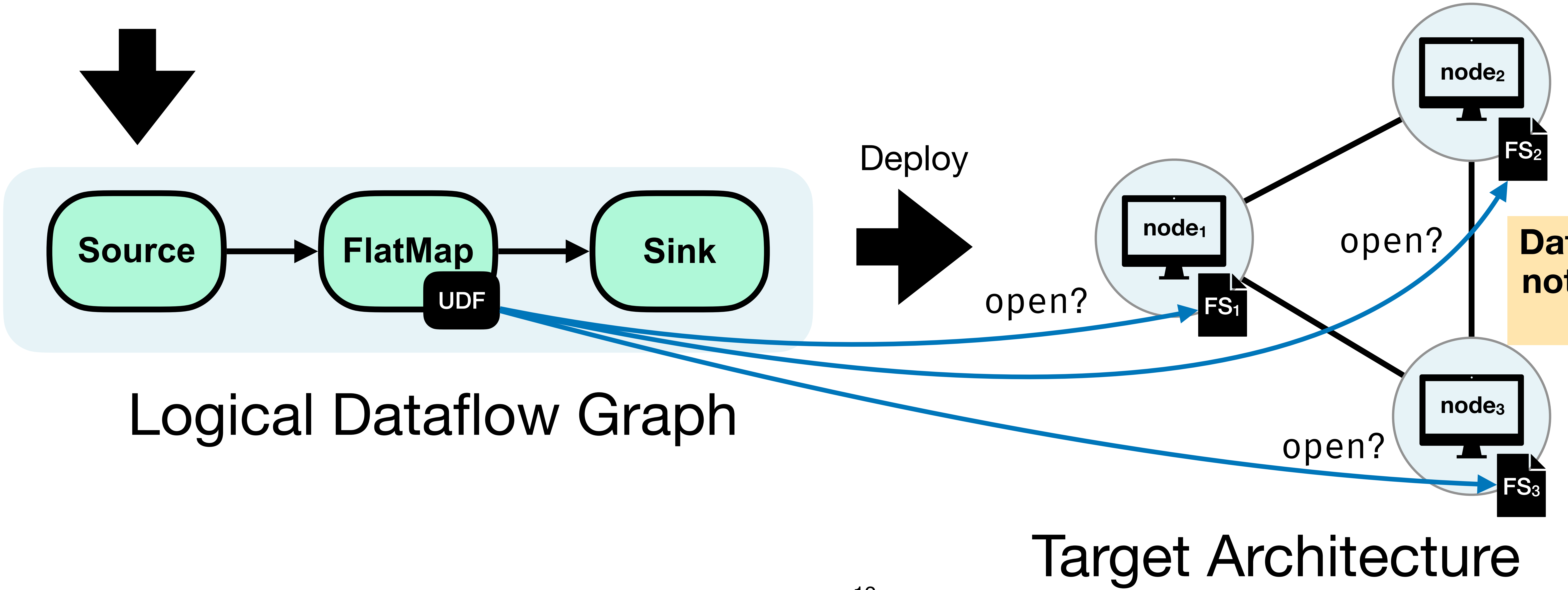


AquaLang: Effect System

Preventing Undefined Behaviour

```
val filePaths: Stream[String] = source(…);

filePaths
  .flatMap(fun(path:String): Vec[String] = open(path).readString().split(" "))
  .sink(…).run(…);
```

Semantics?



Dataflow behaviour must not depend on the target architecture

AquaLang: Effect System

Preventing Undefined Behaviour

```
val filePaths: Stream[String] = source(…);

filePaths
  .flatMap(fun(path:String): Vec[String] = open(path).readString().split(" "))
  .sink(…).run(…);
```

AquaLang: Effect System

Preventing Undefined Behaviour

```
val filePaths: Stream[String] = source(…);

filePaths
  .flatMap(fun(path:String): Vec[String] = open(path).readString().split(" "))
  .sink(…).run(…);



def open(String): File ~ {io};
def flatMap[I,O](Stream[I], fun(I):Vec[O] ~ {}): Stream[O] ~ {};
```

AquaLang: Effect System

Preventing Undefined Behaviour

```
val filePaths: Stream[String] = source(…);

filePaths
  .flatMap(fun(path:String): Vec[String] = open(path).readString().split(" "))
  .sink(…).run(…);



def open(String): File ~ {io};
def flatMap[I,O](Stream[I], fun(I):Vec[O] ~ {}): Stream[O] ~ {};
```

AquaLang: Effect System

Preventing Undefined Behaviour

```
val filePaths: Stream[String] = source(…);

filePaths
  .flatMap(fun(path:String): Vec[String] = open(path).readString().split(" "))
  .sink(…).run(…);
```

`open` produces an **`io`** effect

```
def open(String): File ~ {io};
def flatMap[I,O](Stream[I], fun(I):Vec[O] ~ {}): Stream[O] ~ {};
```

AquaLang: Effect System

Preventing Undefined Behaviour

```
val filePaths: Stream[String] = source(…);

filePaths
  .flatMap(fun(path:String): Vec[String] = open(path).readString().split(" "))
  .sink(…).run(…);
```

`open` produces an `io` effect

`flatMap` produces no effects

```
def open(String): File ~ {io};
def flatMap[I,O](Stream[I], fun(I):Vec[O] ~ {}): Stream[O] ~ {};
```

AquaLang: Effect System

Preventing Undefined Behaviour

```
val filePaths: Stream[String] = source(…);

filePaths
  .flatMap(fun(path:String): Vec[String] = open(path).readString().split(" "))
  .sink(…).run(…);
```

open produces an io effect

flatMap produces no effects

```
def open(String): File ~ {io};
def flatMap[I,O](Stream[I], fun(I):Vec[O] ~ {}): Stream[O] ~ {};
```

flatMap requires a UDF that produces no effects

AquaLang: Effect System

Preventing Undefined Behaviour

```
val filePaths: Stream[String] = source(…);

filePaths
  .flatMap(fun(path:String): Vec[String] ~ {io} = open(path).readString().split(" "))
  .sink(…).run(…);



def open(String): File ~ {io};
def flatMap[I,O](Stream[I], fun(I):Vec[O] ~ {}): Stream[O] ~ {};
```

Effect inference: Functions inherit the effects of the functions they call

AquaLang: Effect System

Preventing Undefined Behaviour

```
val filePaths: Stream[String] = source(…);

filePaths
  .flatMap(fun(path:String): Vec[String] ~ {io} = open(path).readString().split(" "))
  .sink(…).run(…);



def open(String): File ~ {io};
def flatMap[I,O](Stream[I], fun(I):Vec[O] ~ {}): Stream[O] ~ {};
```

Effect inference: Functions inherit the effects of the functions they call

≠

Error: Effect mismatch, expected {} but found {io}

AquaLang: Effect System

Preventing Undefined Behaviour

```
val filePaths: Stream[String] = source(…);

filePaths
  .flatMap(fun(path:String): Vec[String] ~ {io} = open(path).readString().split(" "))
  .sink(…).run(…);



def open(String): File ~ {io};
def flatMap[I,O](Stream[I], fun(I):Vec[O] ~ {}): Stream[O] ~ {};
```

Effect inference: Functions inherit the effects of the functions they call

$\neq$

Error: Effect mismatch, expected {} but found {io}

More details in the paper:

$$(\textsc{DefFun})\ \frac{\Gamma' \vdash e_a : t_a \,|\, f_a \qquad \Gamma' \vdash e_b : t_b \,|\, f_b}{\Gamma \vdash \mathtt{def}\ x(x_1 : t_1 \ldots x_n : t_n) : t_a \sim f_a = e_a\ \mathtt{in}\ e_b : t_b \,|\, f_b}$$

where Γ' is $x \mapsto \mathtt{fun}(t_1 \ldots t_n) : t \sim f, \Gamma$

$$(\textsc{DefVar})\ \frac{\Gamma \vdash e_1 : t_1 \,|\, f_1 \qquad x \mapsto t_1, \Gamma \vdash e_2 : t_2 \,|\, f_2}{\Gamma \vdash \mathtt{val}\ x = e_1\ \mathtt{in}\ e_2 : t_2 \,|\, f_1 \cup f_2}$$

$$(\textsc{Call})\ \frac{\Gamma \vdash e : \mathtt{fun}(t_1 \ldots t_n) : t \sim f_a \,|\, f_b \qquad \Gamma \vdash e_i : t_i \,|\, f_i,\ \text{for } i \in 1 \ldots n}{\Gamma \vdash \Gamma \vdash e(e_1 \ldots e_n) : t \,|\, f_a \cup f_b \cup f_1 \cup \ldots \cup f_n}$$

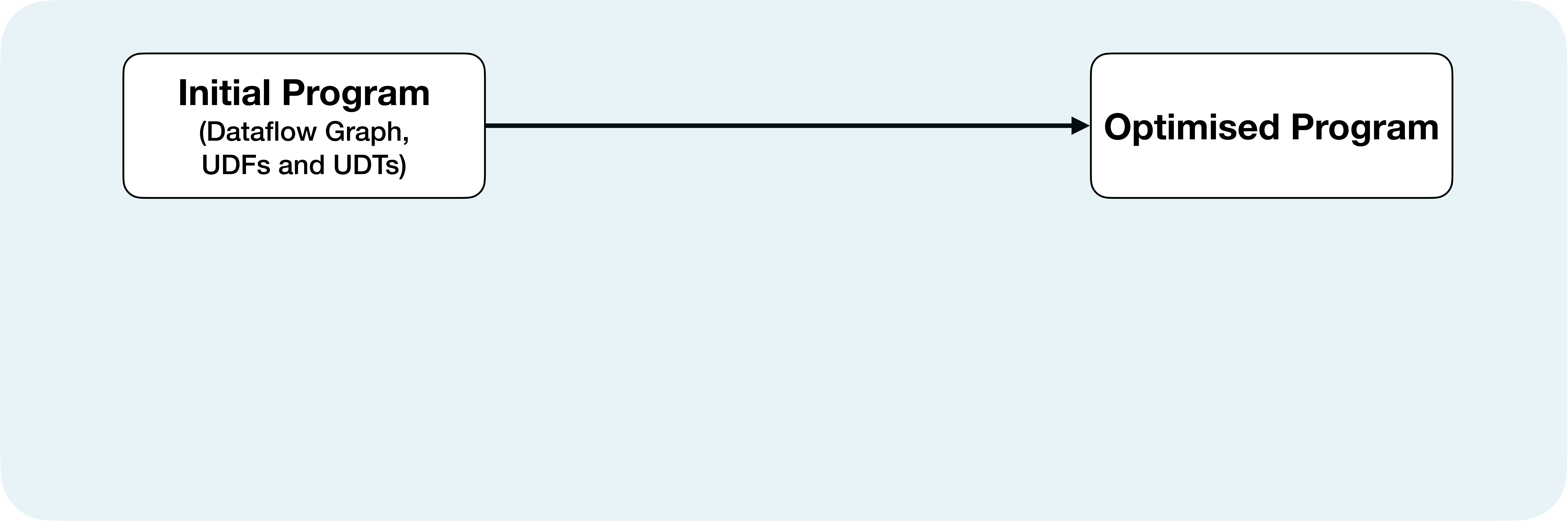
$$(\textsc{Var})\ \frac{t = \Gamma(x)}{\Gamma \vdash x : t \,|\, \varnothing} \qquad (\textsc{BaseVal})\ \frac{}{\Gamma \vdash v_\tau : \tau \,|\, \varnothing}$$

$$(\textsc{FunVal})\ \frac{}{\Gamma \vdash \mathtt{fun}(x_1 : t_1 \ldots x_n : t_n) : t \sim f = e : \mathtt{fun}(t_1 \ldots t_n) : t \sim f \,|\, \varnothing}$$

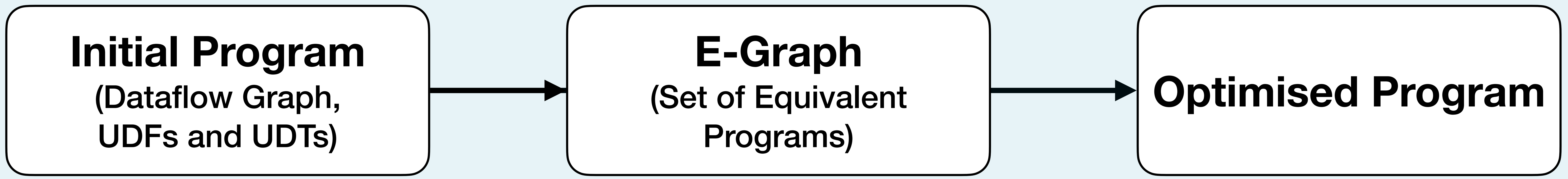
Figure 5: Type and effect checking of AquaLang.

AquaLang: Equality Saturation-Based Optimisation

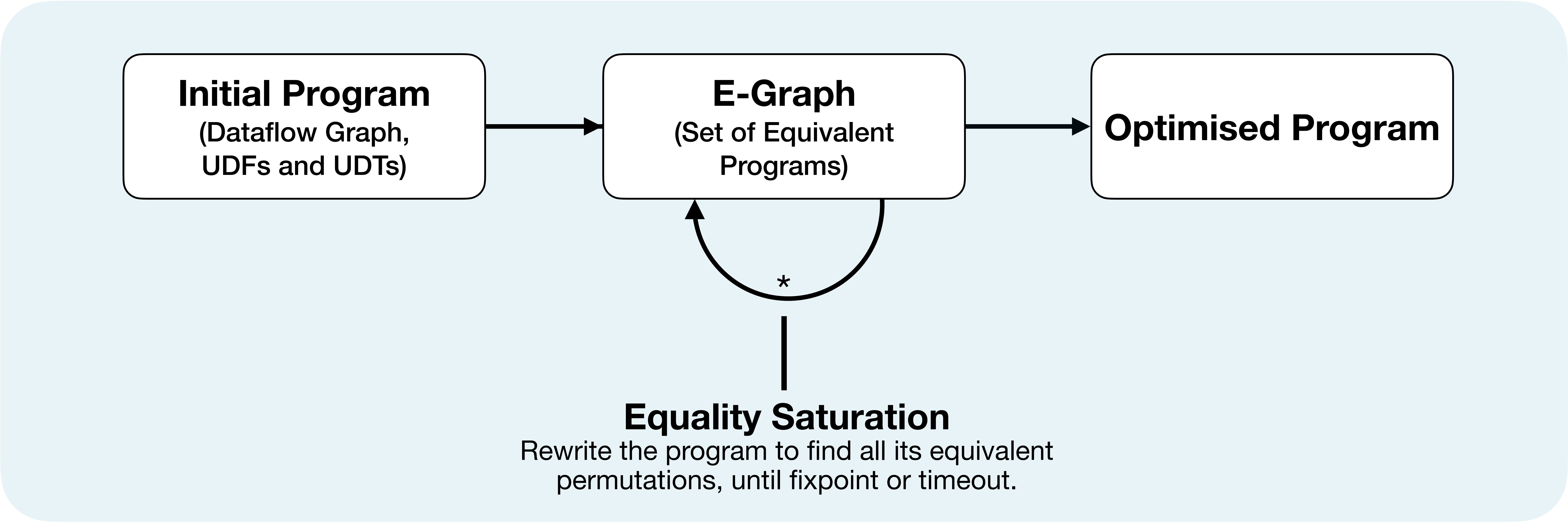
AquaLang: Equality Saturation-Based Optimisation



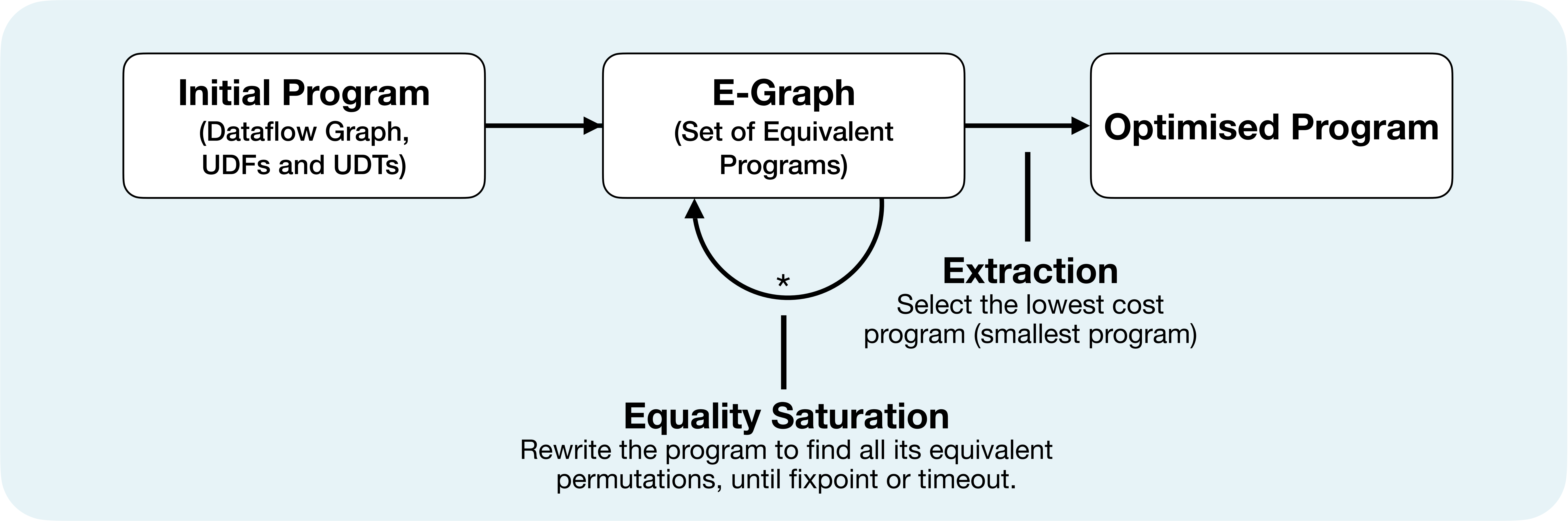
AquaLang: Equality Saturation-Based Optimisation



AquaLang: Equality Saturation-Based Optimisation



AquaLang: Equality Saturation-Based Optimisation



AquaLang: Optimisations

- **Relational Optimisations:** Projection Pushdown, Predicate Pushdown
- **Functional Optimisations:** Operator Fusion

AquaLang: Optimisations

- **Relational Optimisations:** Projection Pushdown, Predicate Pushdown
- **Functional Optimisations:** Operator Fusion
- **UDF Optimisations:** Incrementalisation, Algebraic Property Inference

AquaLang: From Windows to Incremental Windows

Basic Example of UDF optimisations

AquaLang: From Windows to Incremental Windows

Basic Example of UDF optimisations

```
source(…)
  .window(
    sliding(…),
    fun(items: Vec[Item]): f64 = {
      var s = 0; var c = 0;
      for item in items { s += item.usd; c += 1; }
      s / c })
  .sink(…).run(…);
```

AquaLang: From Windows to Incremental Windows

Basic Example of UDF optimisations

```
source(…)
  .window(
    sliding(…),
    fun(items: Vec[Item]): f64 = {
      var s = 0; var c = 0;
      for item in items { s += item.usd; c += 1; }
      s / c })
  .sink(…).run(…);
```

AquaLang: From Windows to Incremental Windows

Basic Example of UDF optimisations

```
source(…)
  .window(
    sliding(…),
    fun(items: Vec[Item]): f64 = {
      var s = 0; var c = 0;
      for item in items { s += item.usd; c += 1; }
      s / c })
  .sink(…).run(…);
```

Non-incremental arithmetic mean of item prices

AquaLang: From Windows to Incremental Windows

Basic Example of UDF optimisations

```
source(…)
  .window(
    sliding(…),
    fun(items: Vec[Item]): f64 = {
      var s = 0; var c = 0;
      for item in items { s += item.usd; c += 1; }
      s / c })
  .sink(…).run(…);
```

Non-incremental arithmetic mean of item prices

Full-pass

AquaLang: From Windows to Incremental Windows

Basic Example of UDF optimisations

```
source(…)
  .window(
    sliding(…),
    fun(items: Vec[Item]): f64 = {
      var s = 0; var c = 0;
      for item in items { s += item.usd; c += 1; }
      s / c })
  .sink(…).run(…);
```

Non-incremental arithmetic mean of item prices

Full-pass

Incrementalisation via Equality Saturation

```
source(…)
  .incrWindow(
    sliding(…),
    (0, 0),
    fun(item) = (item.usd, 1),
    fun((s1,s2),(c1,c2)) = (s1+s2,c1+c2),
    fun((s,c)) = s/c)
  .sink(…).run(…);
```

AquaLang: From Windows to Incremental Windows

Basic Example of UDF optimisations

Non-incremental arithmetic mean of item prices

```
source(…)
  .window(
    sliding(…),
    fun(items: Vec[Item]): f64 = {
      var s = 0; var c = 0;
      for item in items { s += item.usd; c += 1; }
      s / c })
  .sink(…).run(…);
```

Full-pass

Incrementalisation via Equality Saturation

```
source(…)
  .incrWindow(
    sliding(…),
    (0, 0),
    fun(item) = (item.usd, 1),
    fun((s1,s2),(c1,c2)) = (s1+s2,c1+c2),
    fun((s,c)) = s/c)
  .sink(…).run(…);
```

AquaLang: From Windows to Incremental Windows

Basic Example of UDF optimisations

```
source(…)
  .window(
    sliding(…),
    fun(items: Vec[Item]): f64 = {
      var s = 0; var c = 0;
      for item in items { s += item.usd; c += 1; }
      s / c })
  .sink(…).run(…);
```

Non-incremental arithmetic mean of item prices

Full-pass

Incrementalisation via Equality Saturation

```
source(…)
  .incrWindow(
    sliding(…),
    (0, 0),
    fun(item) = (item.usd, 1),
    fun((s1,s2),(c1,c2)) = (s1+s2,c1+c2),
    fun((s,c)) = s/c)
  .sink(…).run(…);
```

Initial aggregate (sum and count)

AquaLang: From Windows to Incremental Windows

Basic Example of UDF optimisations

```
source(…)
  .window(
    sliding(…),
    fun(items: Vec[Item]): f64 = {
      var s = 0; var c = 0;
      for item in items { s += item.usd; c += 1; }
      s / c })
  .sink(…).run(…);
```

Non-incremental arithmetic mean of item prices

Full-pass

Incrementalisation via Equality Saturation

```
source(…)
  .incrWindow(
    sliding(…),
    (0, 0),
    fun(item) = (item.usd, 1),
    fun((s1,s2),(c1,c2)) = (s1+s2,c1+c2),
    fun((s,c)) = s/c)
  .sink(…).run(…);
```

Initial aggregate (sum and count)

Map items to aggregates

AquaLang: From Windows to Incremental Windows

Basic Example of UDF optimisations

```
source(…)
  .window(
    sliding(…),
    fun(items: Vec[Item]): f64 = {
      var s = 0; var c = 0;
      for item in items { s += item.usd; c += 1; }
      s / c })
  .sink(…).run(…);
```

Non-incremental arithmetic mean of item prices

Full-pass

Incrementalisation via Equality Saturation

```
source(…)
  .incrWindow(
    sliding(…),
    (0, 0),
    fun(item) = (item.usd, 1),
    fun((s1,s2),(c1,c2)) = (s1+s2,c1+c2),
    fun((s,c)) = s/c)
  .sink(…).run(…);
```

Initial aggregate (sum and count)

Map items to aggregates

Combine two aggregates

AquaLang: From Windows to Incremental Windows

Basic Example of UDF optimisations

```
source(…)
  .window(
    sliding(…),
    fun(items: Vec[Item]): f64 = {
      var s = 0; var c = 0;
      for item in items { s += item.usd; c += 1; }
      s / c })
  .sink(…).run(…);
```

Non-incremental arithmetic mean of item prices

Full-pass

Incrementalisation via Equality Saturation

```
source(…)
  .incrWindow(
    sliding(…),
    (0, 0),
    fun(item) = (item.usd, 1),
    fun((s1,s2),(c1,c2)) = (s1+s2,c1+c2),
    fun((s,c)) = s/c)
  .sink(…).run(…);
```

Initial aggregate (sum and count)

Map items to aggregates

Combine two aggregates

Map aggregate to arithmetic mean

AquaLang: From Windows to Incremental Windows

Basic Example of UDF optimisations

```
source(…)
  .incrWindow(
    sliding(…),
    (0, 0),
    fun(item) = (item.usd, 1),
    fun((s1,s2),(c1,c2)) = (s1+s2,c1+c2),
    fun((s,c)) = s/c)
  .sink(…).run(…);
```

AquaLang: From Windows to Incremental Windows

Basic Example of UDF optimisations

```
source(…)
  .incrWindow(
    sliding(…),
    (0, 0),
    fun(item) = (item.usd, 1),
    fun((s1,s2),(c1,c2)) = (s1+s2,c1+c2),
    fun((s,c)) = s/c)
  .sink(…).run(…);
```

AquaLang: From Windows to Incremental Windows

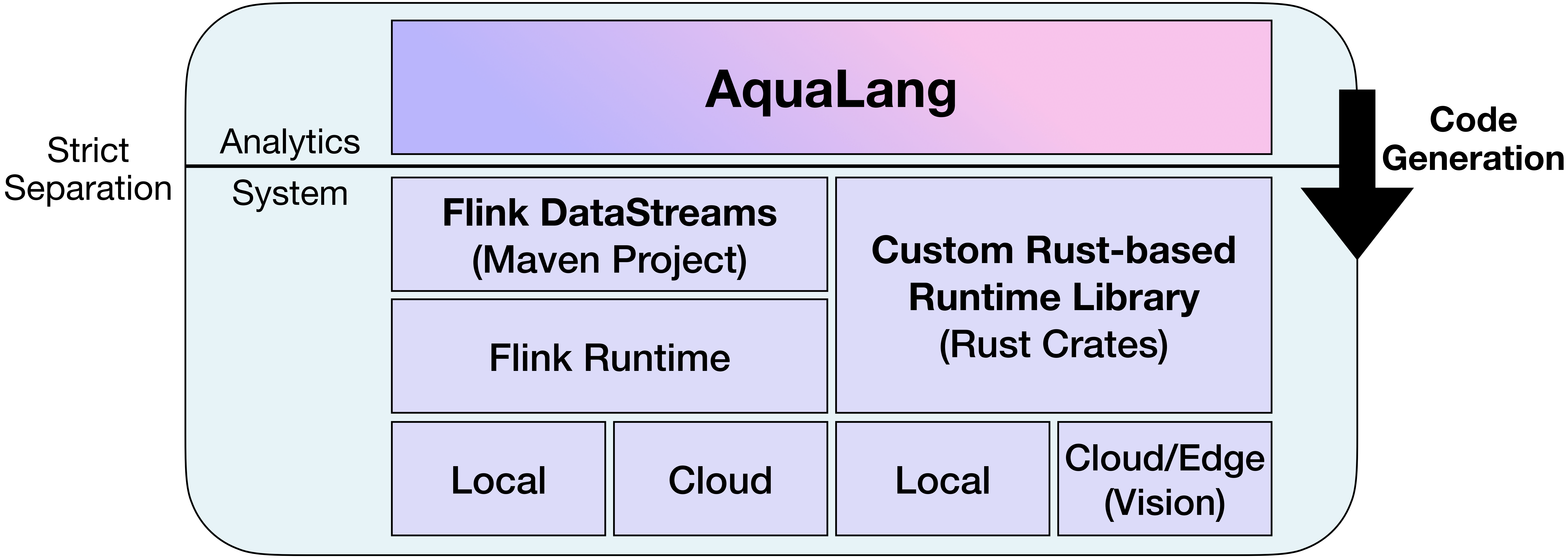
Basic Example of UDF optimisations

```
source(…)
  .incrWindow(
    sliding(…),
    (0, 0),
    fun(item) = (item.usd, 1),
    fun((s1,s2),(c1,c2)) = (s1+s2,c1+c2),
    fun((s,c)) = s/c)
  .sink(…).run(…);
```

Prove Commutativity via Equality Saturation

```
source(…)
  .incrWindow(
    sliding(…),
    (0, 0),
    fun(item) = (item.usd, 1),
    fun((s1,s2),(c1,c2)) = (s1+s2,c1+c2),
    fun((s,c)) = s/c,
    commutative=true)
  .sink(…).run(…);
```

AquaLang: Code Generation



Experiments

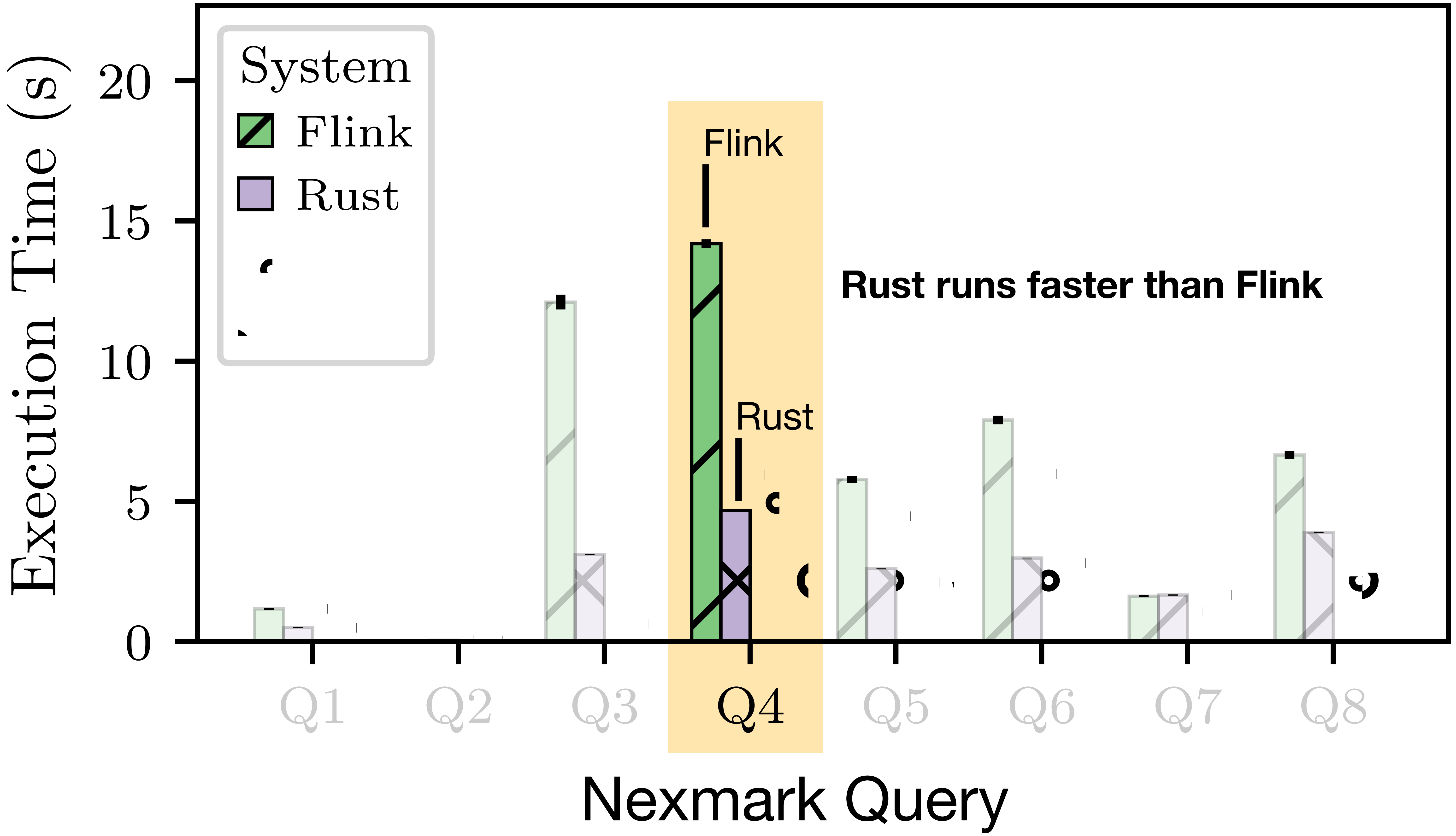
- **Nexmark Benchmark**
- **Evaluation**:
  - **Backends**: **Rust** vs. **Java-Flink**
  - **Optimisations**: **Enabled** vs. **Disabled**
- **Setup:** Single node, single thread, synthetic data, 5M events
- The results we show is after removing the I/O cost

Experiment 1: Nexmark (Q1-Q8)

Evaluation of Functional and Relational Optimisations

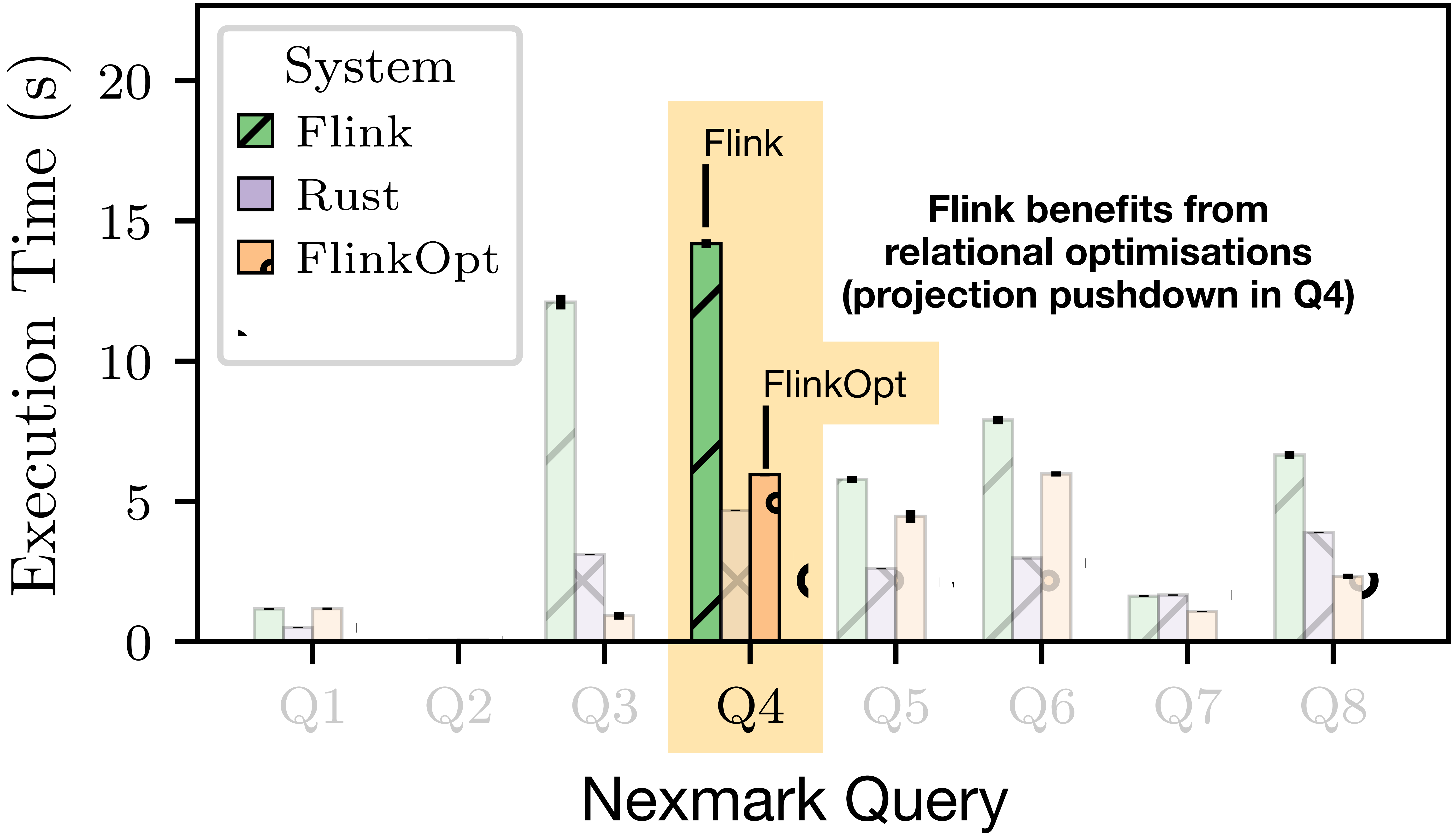
Experiment 1: Nexmark (Q1-Q8)

Evaluation of Functional and Relational Optimisations



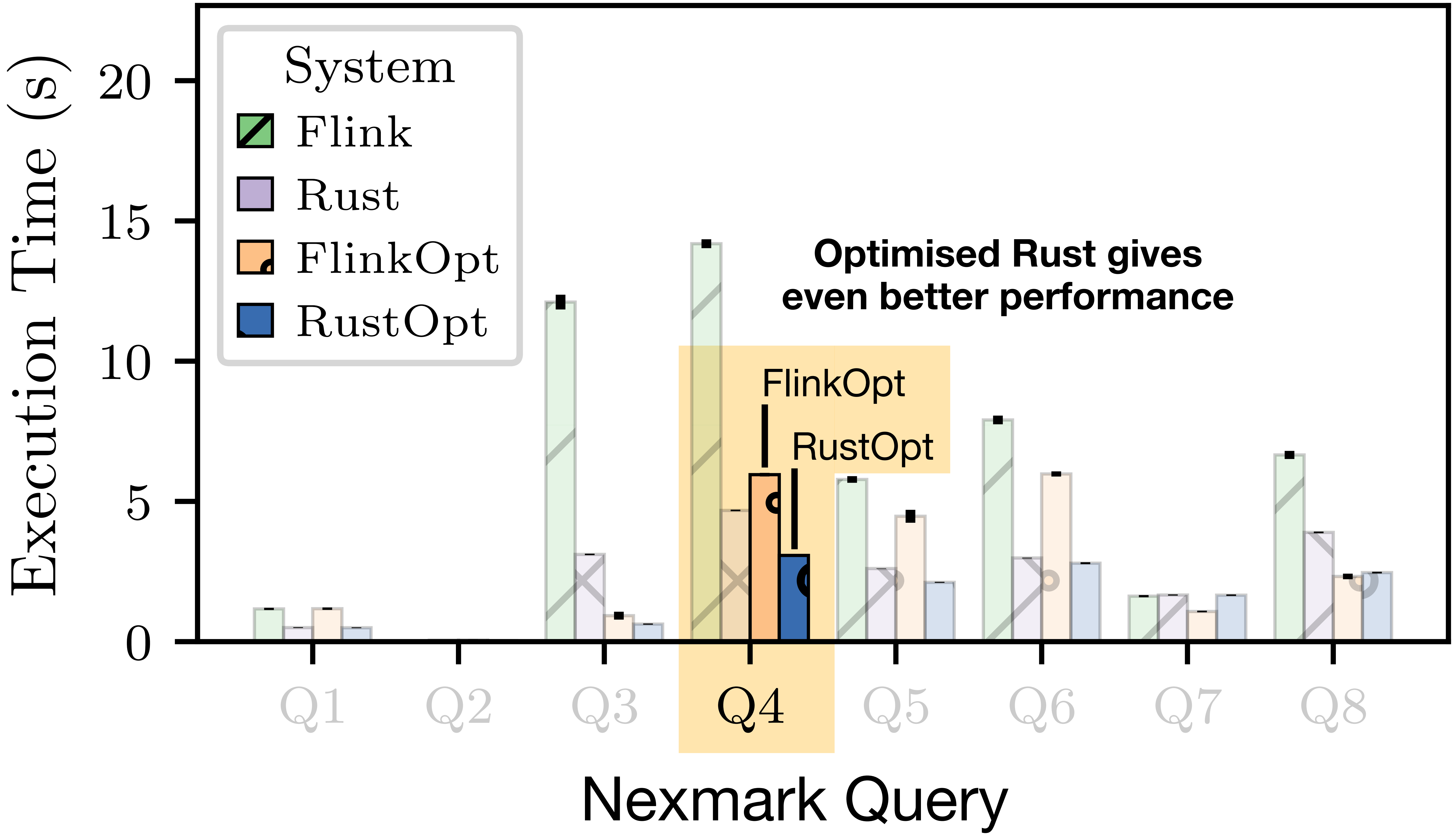
Experiment 1: Nexmark (Q1-Q8)

Evaluation of Functional and Relational Optimisations



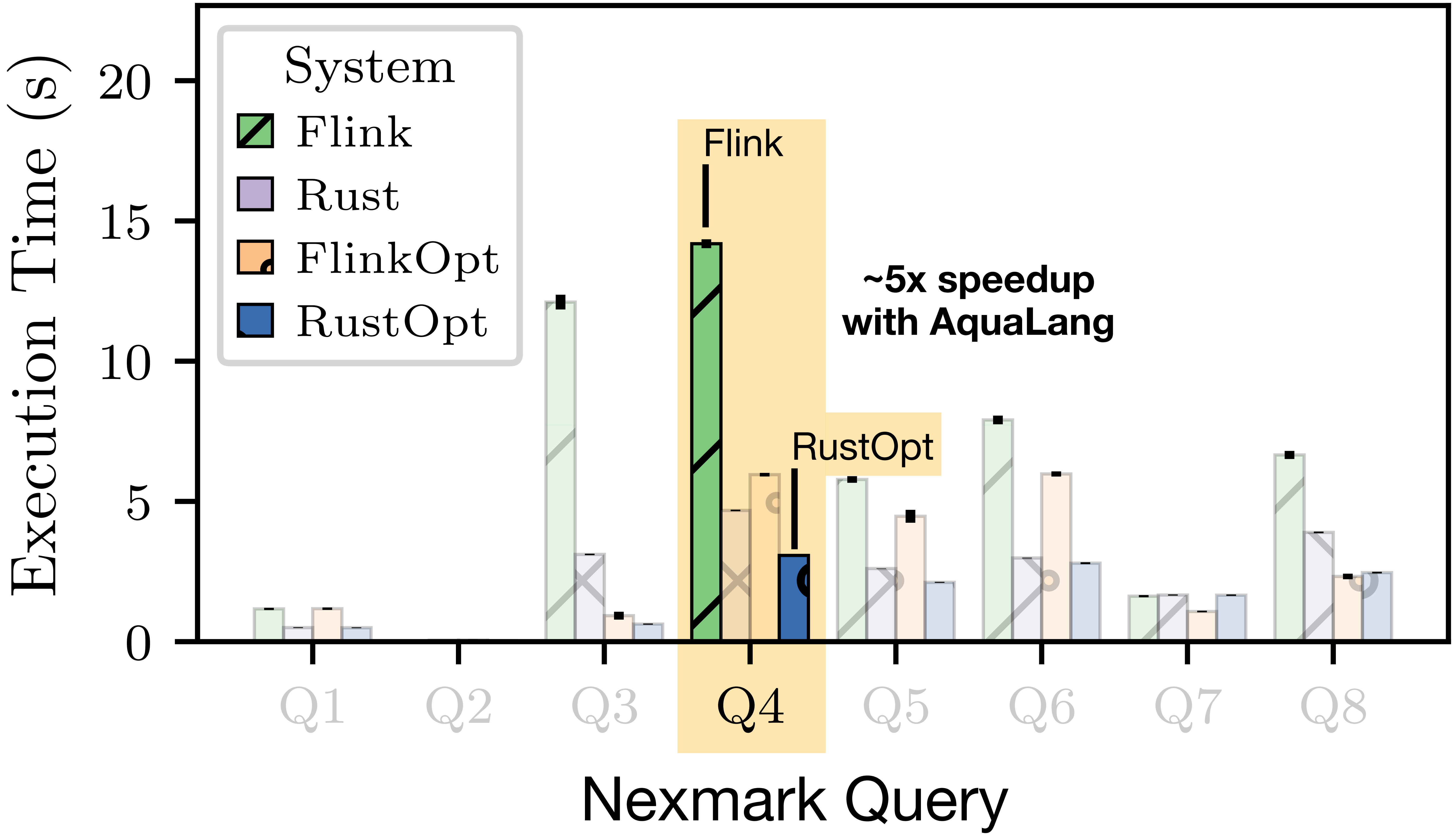
Experiment 1: Nexmark (Q1-Q8)

Evaluation of Functional and Relational Optimisations



Experiment 1: Nexmark (Q1-Q8)

Evaluation of Functional and Relational Optimisations

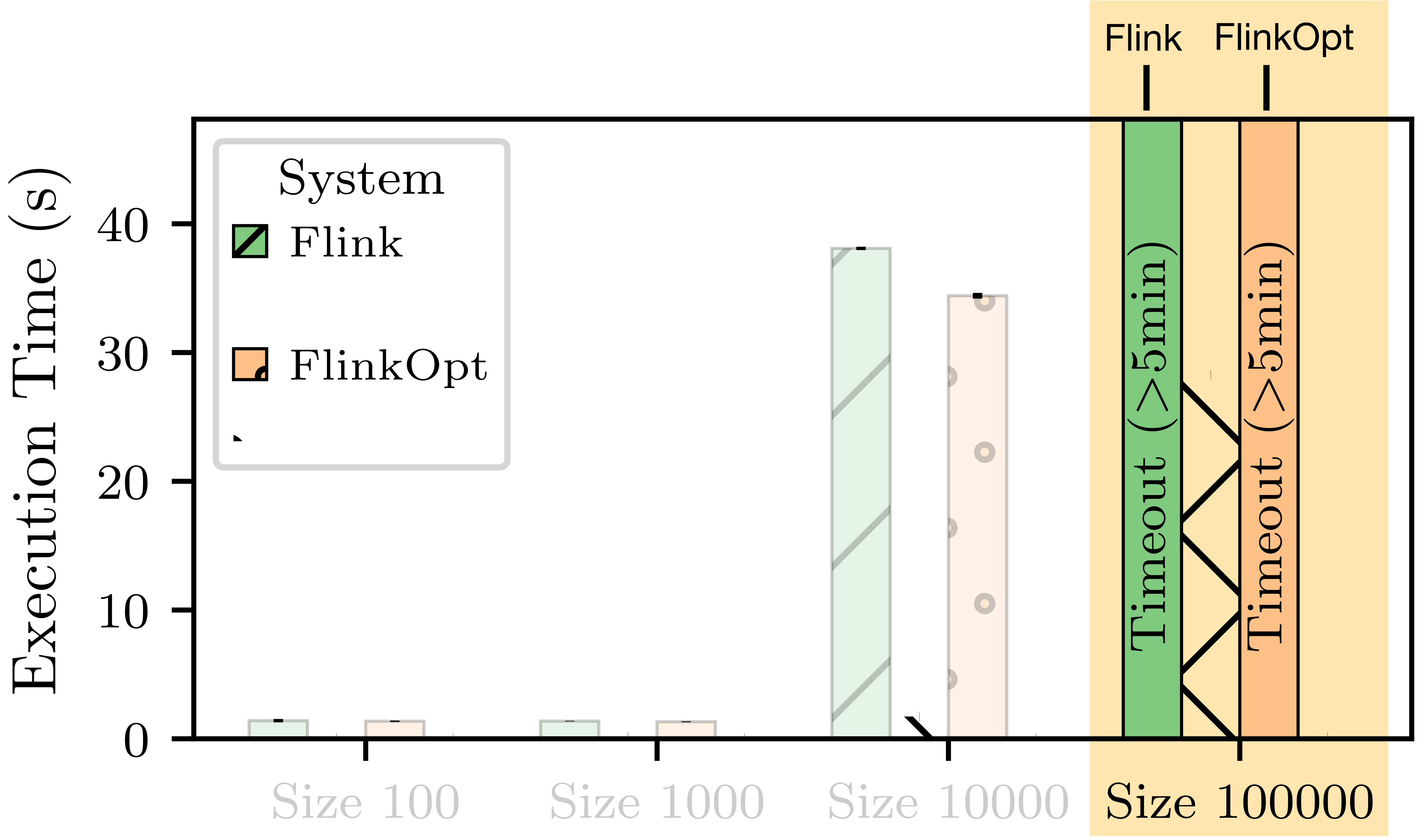


Experiment 2: Window Aggregation (Sliding, Count-Based)

Evaluation of Windows vs. Incremental Windows (stddev aggregator)

Experiment 2: Window Aggregation (Sliding, Count-Based)

Evaluation of Windows vs. Incremental Windows (stddev aggregator)

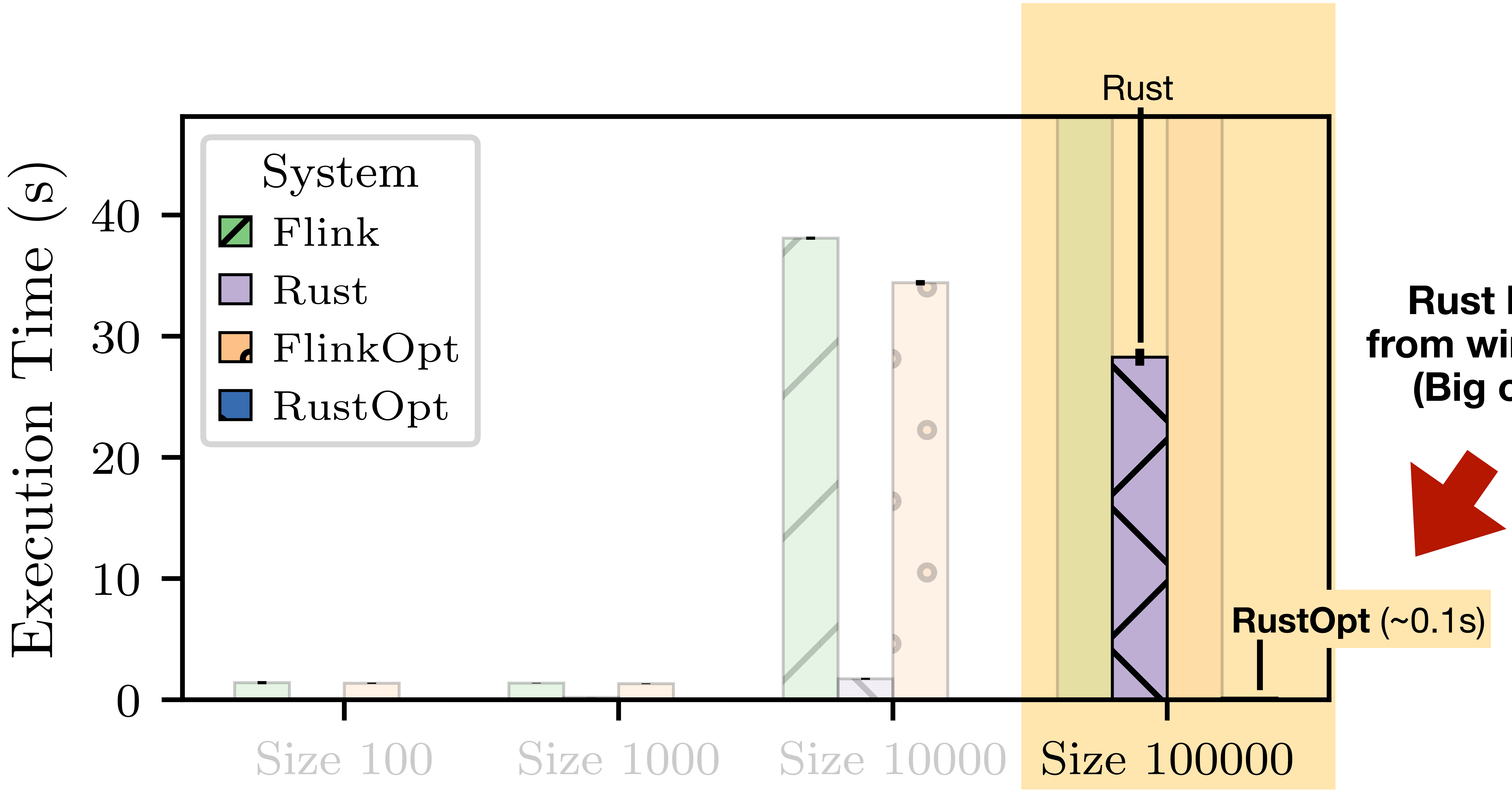


Flink does not benefit much from windowing optimisations

Window size (with constant slide of 100 events)

Experiment 2: Window Aggregation (Sliding, Count-based)

Evaluation of Windows vs. Incremental Windows (stddev aggregator)



Rust benefits significantly from windowing optimisations (Big orders of magnitude)

Window size (with constant slide of 100 events)

Summary

Summary

AquaLang Features:

Summary

AquaLang Features:

- **General Purpose Programming:**
  - Mutation, loops, I/O, etc.
  - UDTs (User-Defined Types)
  - UDFs (User-Defined Functions)

Summary

AquaLang Features:

- **General Purpose Programming:**
  - Mutation, loops, I/O, etc.
  - UDTs (User-Defined Types)
  - UDFs (User-Defined Functions)

Summary

AquaLang Features:

- **General Purpose Programming:**
  - Mutation, loops, I/O, etc.
  - UDTs (User-Defined Types)
  - UDFs (User-Defined Functions)

Query Language Problems

- **Flexibility Problems:**
  - Purely declarative code
  - Purely relational data
  - Limited extensibility

Summary

AquaLang Features:

- **General Purpose Programming:**
  - Mutation, loops, I/O, etc.
  - UDTs (User-Defined Types)
  - UDFs (User-Defined Functions)
- **Dataflow Programming (Streams)**

Query Language Problems

- **Flexibility Problems:**
  - Purely declarative code
  - Purely relational data
  - Limited extensibility

Summary

AquaLang Features:

- **General Purpose Programming:**
  - Mutation, loops, I/O, etc. → Purely declarative code
  - UDTs (User-Defined Types) → Purely relational data
  - UDFs (User-Defined Functions) → Limited extensibility
- **Dataflow Programming (Streams)**
- **Safety Features:**
  - Strongly Typed Relational Syntax
  - Effect System
  - Sandboxing (Described in paper)

Query Language Problems

- **Flexibility Problems:**
  - Purely declarative code
  - Purely relational data
  - Limited extensibility

Summary

AquaLang Features:

- **General Purpose Programming:**
  - Mutation, loops, I/O, etc. → Purely declarative code
  - UDTs (User-Defined Types) → Purely relational data
  - UDFs (User-Defined Functions) → Limited extensibility
- **Dataflow Programming (Streams)**
- **Safety Features:**
  - Strongly Typed Relational Syntax
  - Effect System
  - Sandboxing (Described in paper)

Query Language Problems

- **Flexibility Problems:**
  - Purely declarative code
  - Purely relational data
  - Limited extensibility

Summary

AquaLang Features:

- **General Purpose Programming:**
  - Mutation, loops, I/O, etc. → Purely declarative code
  - UDTs (User-Defined Types) → Purely relational data
  - UDFs (User-Defined Functions) → Limited extensibility
- **Dataflow Programming (Streams)**
- **Safety Features:**
  - Strongly Typed Relational Syntax → Weak typing
  - Effect System → Undefined behaviour
  - Sandboxing (Described in paper) → Untrusted code

Query Language Problems

- **Flexibility Problems:**
  - Purely declarative code
  - Purely relational data
  - Limited extensibility

Dataflow Library Problems:

- **Safety Problems:**
  - Weak typing
  - Undefined behaviour
  - Untrusted code

Summary

AquaLang Features:

- **General Purpose Programming:**
  - Mutation, loops, I/O, etc. → Purely declarative code
  - UDTs (User-Defined Types) → Purely relational data
  - UDFs (User-Defined Functions) → Limited extensibility
- **Dataflow Programming (Streams)**
- **Safety Features:**
  - Strongly Typed Relational Syntax → Weak typing
  - Effect System → Undefined behaviour
  - Sandboxing (Described in paper) → Untrusted code
- **Performance Features:**
  - Relational Optimisations (Described in paper)
  - Equality Saturation-based Optimisation
  - Code Generation

Query Language Problems

- **Flexibility Problems:**
  - Purely declarative code
  - Purely relational data
  - Limited extensibility

Dataflow Library Problems:

- **Safety Problems:**
  - Weak typing
  - Undefined behaviour
  - Untrusted code

Summary

AquaLang Features:

- **General Purpose Programming:**
  - Mutation, loops, I/O, etc.
  - UDTs (User-Defined Types)
  - UDFs (User-Defined Functions)
- **Dataflow Programming (Streams)**
- **Safety Features:**
  - Strongly Typed Relational Syntax
  - Effect System
  - Sandboxing (Described in paper)
- **Performance Features:**
  - Relational Optimisations (Described in paper)
  - Equality Saturation-based Optimisation
  - Code Generation

Query Language Problems

- **Flexibility Problems:**
  - Purely declarative code
  - Purely relational data
  - Limited extensibility

Dataflow Library Problems:

- **Safety Problems:**
  - Weak typing
  - Undefined behaviour
  - Untrusted code

Summary

AquaLang Features:

- **General Purpose Programming:**
  - Mutation, loops, I/O, etc.
  - UDTs (User-Defined Types)
  - UDFs (User-Defined Functions)
- **Dataflow Programming (Streams)**
- **Safety Features:**
  - Strongly Typed Relational Syntax
  - Effect System
  - Sandboxing (Described in paper)
- **Performance Features:**
  - Relational Optimisations (Described in paper)
  - Equality Saturation-based Optimisation
  - Code Generation

Query Language Problems

- **Flexibility Problems:**
  - Purely declarative code
  - Purely relational data
  - Limited extensibility

Dataflow Library Problems:

- **Safety Problems:**
  - Weak typing
  - Undefined behaviour
  - Untrusted code
- **Performance Problems:**
  - Unoptimisable code
  - Performance hacks
  - Close coupling

Conclusion and Future Work

Conclusion and Future Work

- **Conclusions:**
  - **AquaLang** is a **dataflow programming language** that targets **streaming dataflow systems**
  - **AquaLang** is designed to **hide system complexity** and be **easy to use**
  - **AquaLang** aims to **widen the user-base of existing streaming dataflow systems**
  - We are **looking for collaborators** to get AquaLang production-ready
  - **Website:** https://github.io/aqua-language/aqua

Conclusion and Future Work

- **Conclusions:**
  - **AquaLang** is a **dataflow programming language** that targets **streaming dataflow systems**
  - **AquaLang** is designed to **hide system complexity** and be **easy to use**
  - **AquaLang** aims to **widen the user-base of existing streaming dataflow systems**
  - We are **looking for collaborators** to get AquaLang production-ready
  - **Website:** https://github.io/aqua-language/aqua
- **Future work:**
  - Explore new programming models, backends and architectures

Thank you

(More to be shown in the poster session.)