Ohua-Powered, Semi-Transparent UDF's in the Noria Database

By Justus Adam

Supervisor: Sebastian Ertel, Dirk Habich, Malte Schwarzkopf and Jerónimo Castrillón-Mazo

A query to start with

Query: How many clicks, on average, does it take for a user to get from the start page to a purchase

uid	Category	Timestamp
1	1	001
1	0	005
1	2	010

Table layout

```
SELECT avg(pageview_count)
FROM
( SELECT
    c.user_id, matching_paths.ts1,
    count(*) - 2 as pageview_count
  FROM
    clicks c,
    ( SELECT user_id, max(ts1) as ts1, ts2
      FROM
      ( SELECT DISTINCT ON (c1.user_id, ts1)
          c1.user_id,
          c1.ts as ts1,
          c2.ts as ts2
        FROM clicks c1, clicks c2
        WHERE
          c1.user_id = c2.user_id AND
          c1.ts < c2.ts AND
          c1.category = 1 AND
          c2.category = 2
        ORDER BY
          c1.user_id, c1.ts, c2.ts
      ) candidate_paths
      GROUP BY user_id, ts2
    ) matching_paths
  WHERE
    c.user_id = matching_paths.user_id AND
    c.ts >= matching_paths.ts1 AND
    c.ts <= matching_paths.ts2
  GROUP BY
    c.user_id, matching_paths.ts1
) pageview_counts;
```

5. Average of the count, per user

1. The table, but more than once

3. Only the non-overlapping ones

2. Delimiters for an ordered sequence, if user is the same

4. The actual clicks in between the sequence, if user is the same

1. Eric Friedman, Peter Pawlowski, and John Cieslewicz. 2009. SQL/MapReduce: a practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *Proc. VLDB Endow.* 2, 2 (August 2009), 1402-1413.

Coding it up

Query: How many clicks, on average, does it take for a user to get from the start page to a purchase

uid	Category	Timestamp
1	1	001
1	0	005
1	2	010

Table layout

```
SELECT avg(pageview_count)
FROM
( SELECT
    c.user_id, matching_paths.ts1,
      count(*) - 2 as pageview_count
  FROM
    clicks c,
    ( SELECT user_id, max(ts1) as ts1, ts2
      FROM
      ( SELECT DISTINCT ON (c1.u
          c1.user_id,
          c1.ts as ts1,
          c2.ts as ts2
        FROM clicks c1, cli
        WHERE
          c1.user_id = c2.user_id AND
          c1.ts < c
          c1.catego
          c2.catego
        ORDER BY
          c1.user_id
      ) candidate_path
      GROUP BY user_id
    ) matching_paths
  WHERE
    c.user_id = matchi
    c.ts >= matching_paths.ts1 AND
    c.ts <= matching_paths.ts2
  GROUP BY
    c.user_id, matching_paths.ts1
) pageview_counts;
```

Easier[1,2]:

```
fn click_ana(clicks: RowStream<i32, i32, i64>)
        -> GroupedRows<i32, i32> {
    for (uid, group_stream) in group_by(0, clicks) {
      let sequences = IntervalSequence::new();
      for (_, cat, time)
        in sort_on(2, group_stream) {
          if *cat == 1 {
              sequences.open(*time)
          } else if *cat == 2 {
              sequences.close(*time)
          } else {
              sequences.insert(*time)
          }
      };
      (uid,
       sequences.iter()
              .filter(Interval::is_bounded)
              .map(Interval::len)
              .average())
    }
}
```

1. Per user

2. In sorted order

3. Begin/end sequence or count event

4. Average for length of closed intervals

1. Rakesh Agrawal et al. 2008. The Claremont Report on Database Research. In: SIGMOD Rec. 37.3, 9–19.
2. Charles Welty and David W. Stemple. 1981. Human Factors Comparison of a Procedural and a Nonprocedural Query Language. In: ACM Trans. Database Syst. 626– 649

CHAIR FOR COMPILER CONSTRUCTION

Coding it up

Qu
clic
it t
fro
a p

uid
1
1
1

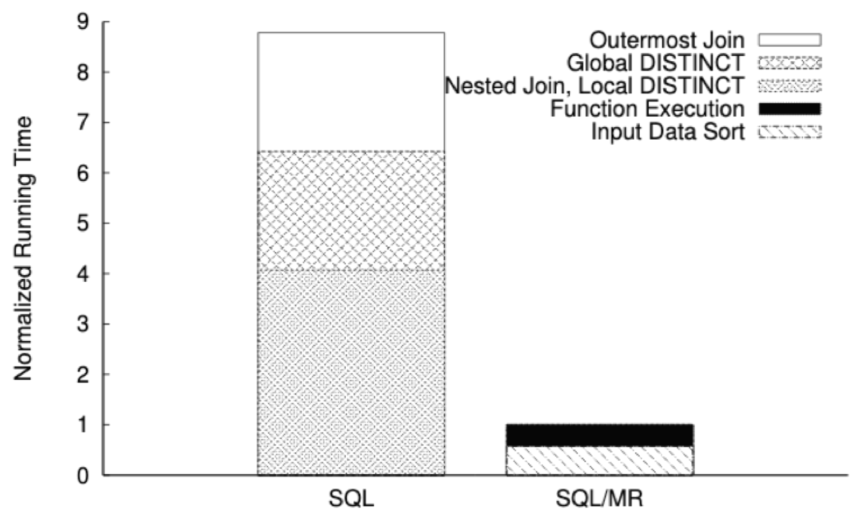


Figure 12: A comparison of the runtime breakdown of SQL and SQL/MR clickstream analysis queries.

```
icks: RowStream<i32, i32, i64>)
GroupedRows<i32, i32> {
roup_stream) in group_by(0, clicks) {
uences = IntervalSequence::new();
 cat, time)

ces.iter()
    .filter(Interval::is_bounded)
    .map(Interval::len)
    .average())
```

Imperative is more efficient because of the many joins in SQL

1. Eric Friedman, Peter Pawlowski, and John Cieslewicz. 2009. SQL/MapReduce: a practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *Proc. VLDB Endow.* 2, 2 (August 2009), 1402-1413.

CHAIR FOR COMPILER CONSTRUCTION

Coding it up

Query: How many clicks, on average, does it take for a user to get from the start page to a purchase

uid	Category	Timestamp
1	1	001
1	0	005
1	2	010

Table layout

```
SELECT avg(pageview_count)
FROM
( SELECT
    c.user_id, matching_paths.ts1,
      count(*) - 2 as pageview_count
  FROM
    clicks c,
    ( SELECT user_id, max(ts1) as ts1, ts2
      FROM
      ( SELECT DISTINCT ON (c1.user_id, ts1)
```

```
      ) candidate_paths
      GROUP BY user_id, ts2
    ) matching_paths
  WHERE
    c.user_id = matching_paths.user_id AND
    c.ts >= matching_paths.ts1 AND
    c.ts <= matching_paths.ts2
  GROUP BY
    c.user_id, matching_paths.ts1
) pageview_counts;
```

Easier:

```
fn click_ana(clicks: RowStream<i32, i32, i64>)
    -> GroupedRows<i32, i32> {
    ...stream) in group_by(0, clicks) {
```

```
        }
    };
    (uid,
     sequences.iter()
        .filter(Interval::is_bounded)
        .map(Interval::len)
        .average())
  }
}
```

Dataflow!

Noria[2]

Dataflow system. Uses materialization (state) to improve read performance

Ohua[1]

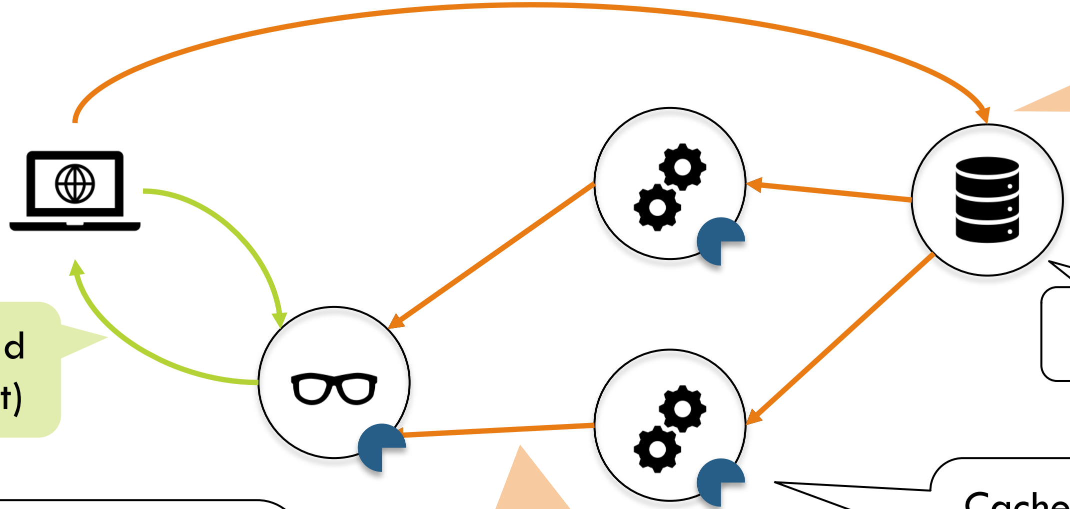
Parallelizable language with a stateful dataflow backend

1. Sebastian Ertel, Christof Fetzer, and Pascal Felber. Ohua: Implicit Dataflow Programming for Concurrent Systems. 2015. PPPJ ’15. 51–64
2. Jon Gjengset et al. 2018. Noria: dynamic, partially-stateful data-flow for high-performance web applications. In *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation* (OSDI'18). USENIX Association, Berkeley, CA, USA, 213-231.

CHAIR FOR COMPILER CONSTRUCTION

Noria

- Multicore
- Distributed
- No UDF Support



Long write path (slow)

Short read path (fast)

Base table

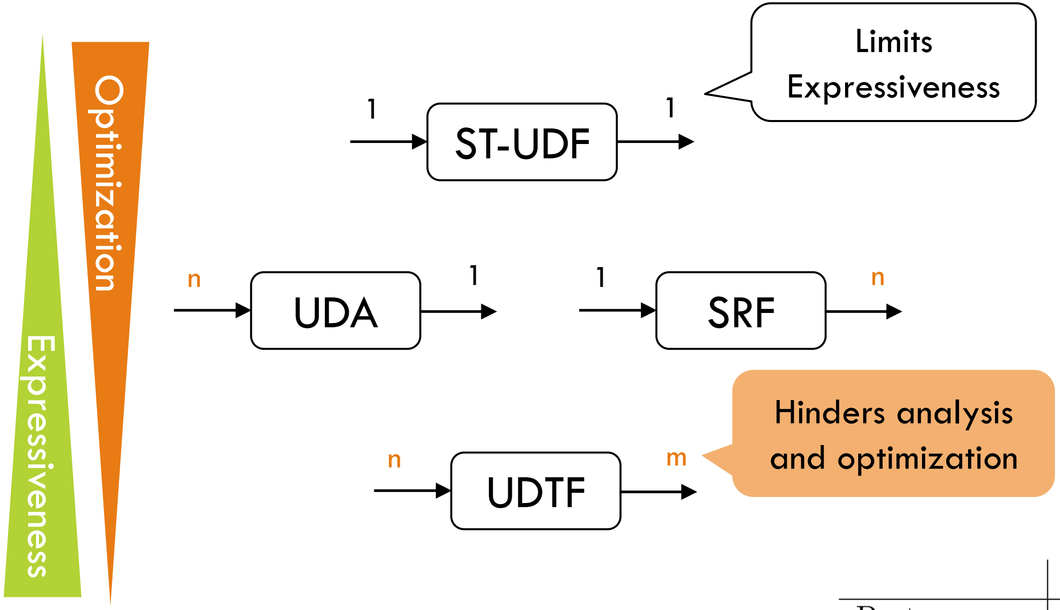
Cached computations (Incremental Materialization)

Update propagation (eventual consistency)

- Simple Materialization recomputes everything on change
- Incremental Materialization only recomputes affected rows. Operators must work with changes (Deltas)

CHAIR FOR COMPILER CONSTRUCTION

Hierarchy of UDF’s



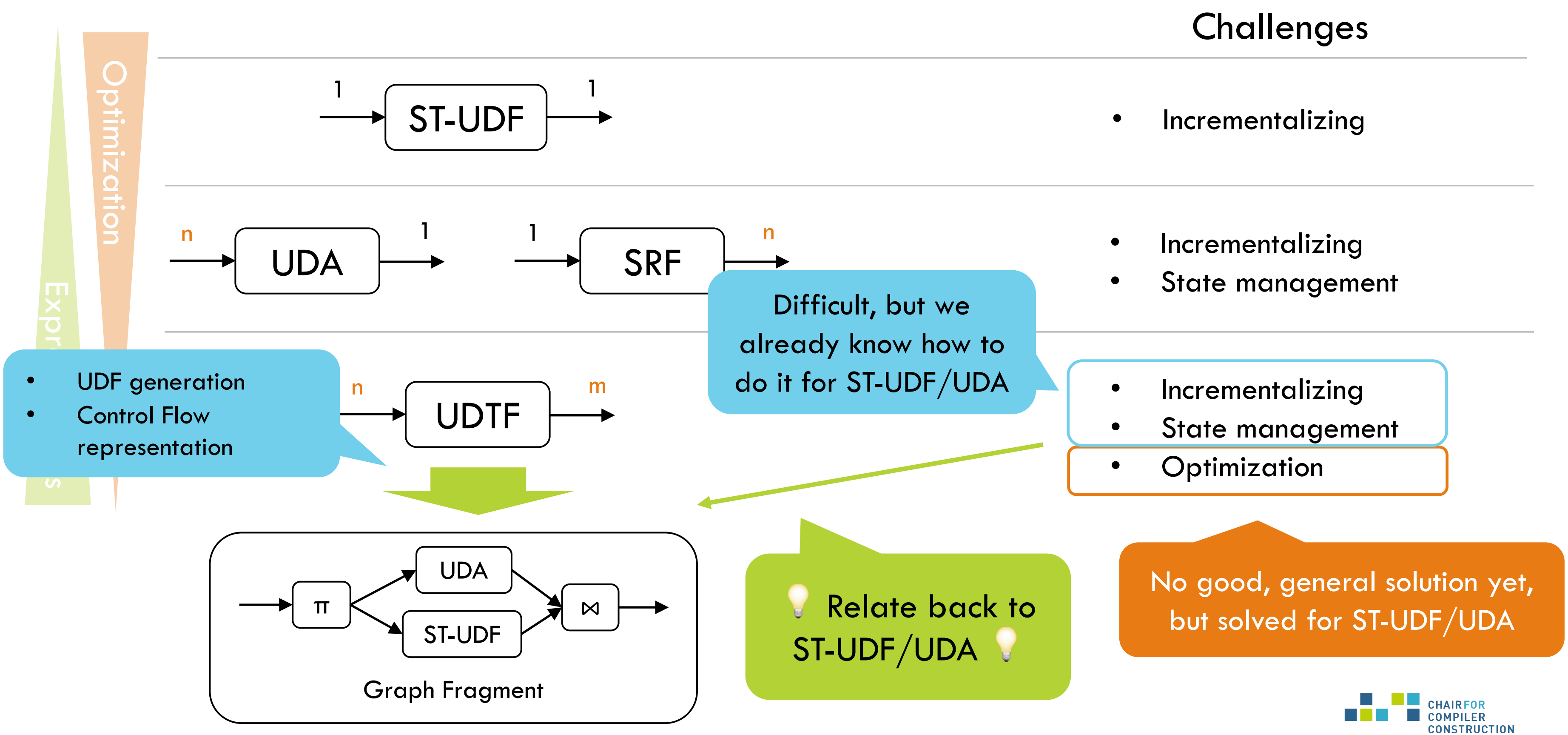
Partial order: Any UDA can be expressed as a UDTF but not vice versa

	UDF	SRF	UDA	UDTF
Postgres	×	×	×	
Apache Hive	×	×	×	×
SQLite	×	×		
MySQL	×		×	
Noria	○		○	○

Our target

UDF support in different Databases

Hierarchy of UDF's



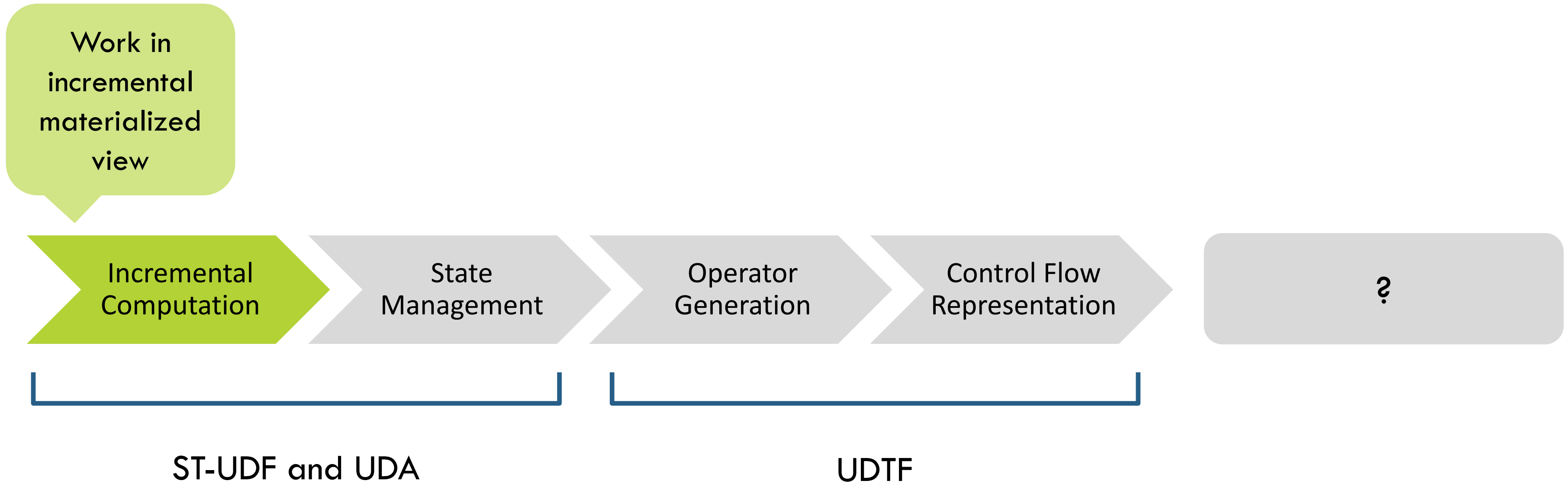
Roadmap

Incremental Computation → State Management → Operator Generation → Control Flow Representation → ?

ST-UDF and UDA

UDTF

Roadmap



Work in incremental materialized view
Incremental Computation
State Management
Operator Generation
Control Flow Representation
?
ST-UDF and UDA
UDTF

Incremental computation

Simple Mat.

- Complete Recompute
- Easy to build
- Inefficient

Incremental Mat.

- Changes recompute
- Efficient
- Difficult to build
- Represented with inserts and deletes

Operators must recompute all affected previous results (requires tracking state) and issue updates downstream.

Only state needs to be incremental

ST-UDF

Relatively easy, propagate whether input was update or delete to the output.

(Same for SRF)

UDA

- Only one, known affected previous result
- State determines new value
- Must reverse changes to state

```
trait State {
    type Action;
    type Output;
    fn apply(&mut self,
             action: Self::Action);
    fn reverse(&mut self,
               action: Self::Action);
    fn compute(&self) -> Self::Output;
}
```


CHAIR FOR COMPILER CONSTRUCTION

Roadmap

Work in incremental materialized view

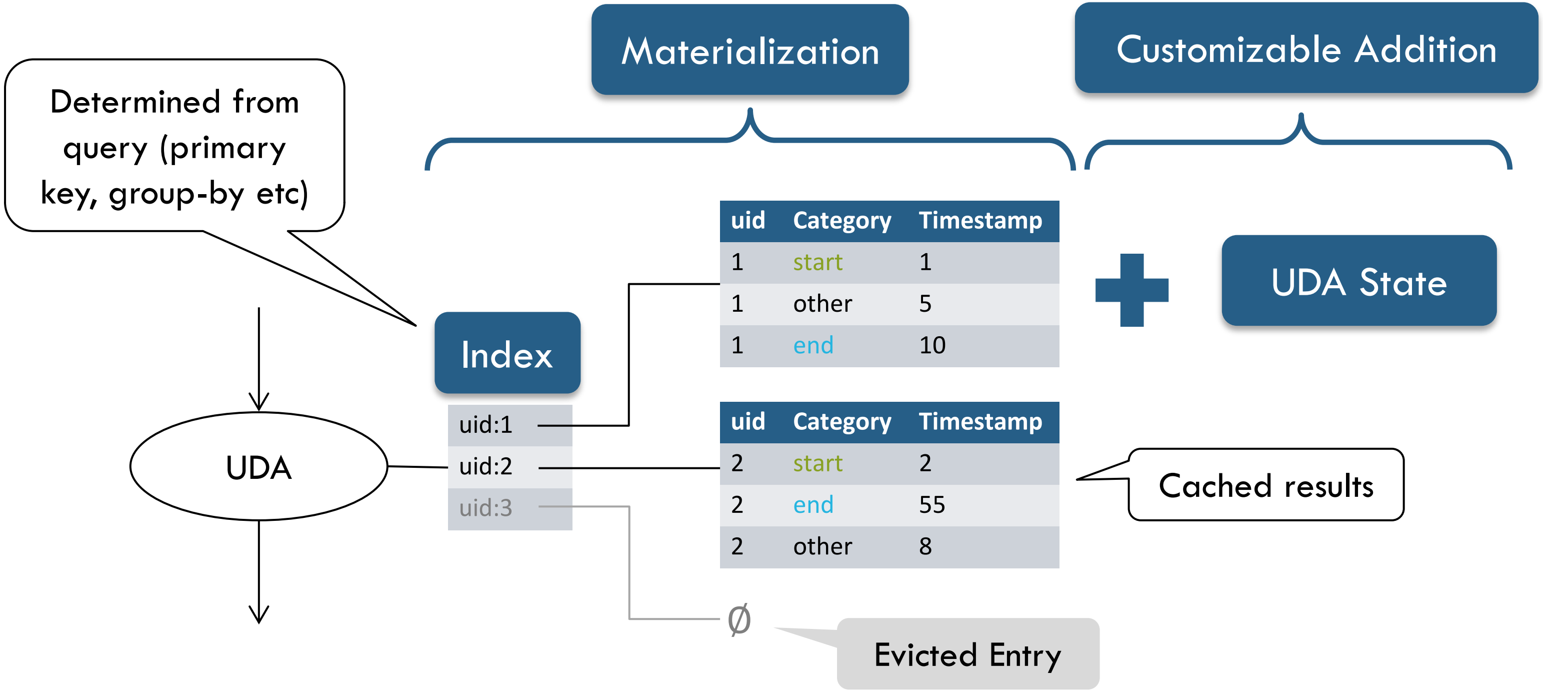
- Selection by index
- Eviction
- Sharding

Incremental Computation → State Management → Operator Generation → Control Flow Representation → ?

ST-UDF and UDA: Incremental Computation, State Management

UDTF: Operator Generation, Control Flow Representation

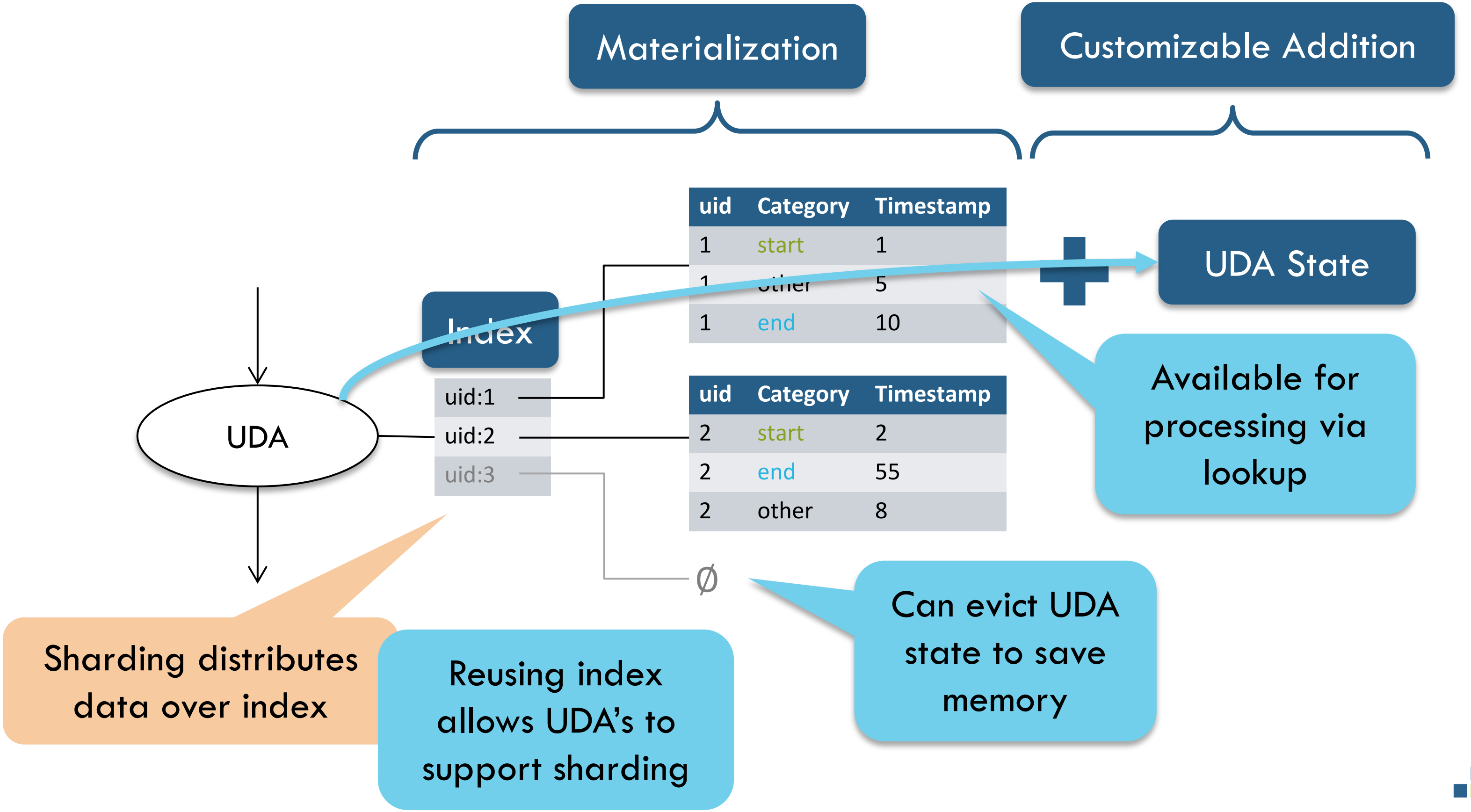
UDA State Management



uid	Category	Timestamp
1	start	1
1	other	5
1	end	10

uid	Category	Timestamp
2	start	2
2	end	55
2	other	8

UDA State Management



uid	Category	Timestamp
1	start	1
1	other	5
1	end	10

uid	Category	Timestamp
2	start	2
2	end	55
2	other	8

Roadmap

Work in incremental materialized view

- Selection by index
- Eviction
- Sharding

- Find suitable operators
- Handle shared state
- Generate boilerplate

Incremental Computation → State Management → Operator Generation → Control Flow Representation → ?

ST-UDF and UDA (Incremental Computation, State Management)

UDTF (Operator Generation, Control Flow Representation)

Operator Generation

- Shared state means synchronization
- Complicates or prevents parallelism
- Not supported in Noria

💡 <u>Make minimal operator with local state</u>

1. Select init expression

2. Select all state uses

3. Recursively select dependencies

4. Bundle into operator

5. Add boilerplate appropriate for type of generated UDF (not shown)

Only operator local state left

```
fn click_ana(clicks: RowStream<i32, i32, i64>)
          -> GroupedRows<i32, i32> {
    for (uid, group_stream) in group_by(0, clicks) {
        let sequences = IntervalSequence::new();
        for (_, cat, time)
          in sort_on(2, group_stream) {
            if *cat == 1 {
                sequences.open(*time)
            } else if *cat == 2 {
                sequences.close(*time)
            } else {
                sequences.insert(*time)
            }
        };
        (uid,
         sequences.iter()
                  .filter(Interval::is_bounded)
                  .map(Interval::len)
                  .average())
    }
}
```

```
impl Op0 {
    fn init() -> Self {
        Op0(IntervalSequence::new())
    }
    fn run(&mut self, rows: Rows<(i32, i32, i64)>)
          -> f64 {
        for (_, cat, time) in rows { ... }
        self.0.iter()
            .filter(Interval::is_bounded)
            .map(Interval::len)
            .average()
    }
}
```

Operator Core (Rust)

Rest of program

```
fn click_ana(clicks: RowStream<i32, i32, i64>)
          -> GroupedRows<i32, i32> {
    for (uid, group_stream) in group_by(0, clicks) {
        let op0 = Op0::init();
        let op0_res = op0.run(sort_on(2, group_stream));
        (uid, op0_res)
    }
}
```

UDTF (Ohua)

Roadmap

Work in incremental materialized view

- Selection by index
- Eviction
- Sharding

- Find suitable operators
- Handle shared state
- Generate boilerplate

- Representation as query
- State scoping
- Multi-arity functions

Incremental Computation → State Management → Operator Generation → Control Flow Representation → ?

ST-UDF and UDA

UDTF

Iteration Representation

Operators always work on batches for efficiency
- ➢ No special iteration operator needed

State must respect scope
State value only valid for one iteration

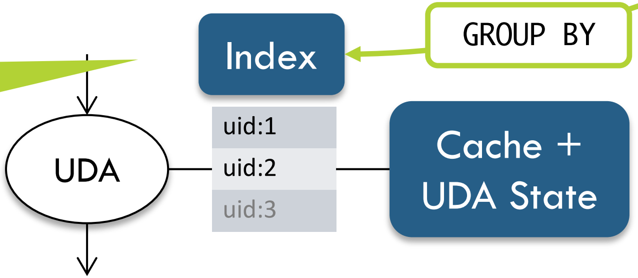
```rust
fn click_ana(clicks: RowStream<i32, i32, i64>)
        -> GroupedRows<i32, i32> {
    for (uid, group_stream) in group_by(0, clicks) {
        let op0 = Op0::init();
        let op0_res = op0.run(sort_on(2, group_stream));
        (uid, op0_res)
    }
}
```

- Sequence never created
- Source streams items
- Each row tagged with index

Number of iterations not known.
Incremental execution revisits state.
→ Cannot just duplicate operator
→ State index & dispatch needed

Sequence source provides index
Found by analysing control flow context

UDA State already indexed
12. Slide



Nesting achieved via compound indices

CHAIR FOR COMPILER CONSTRUCTION

Multi argument functions

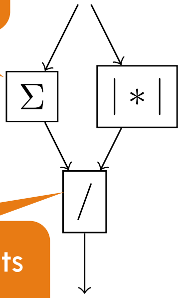
```
fn average(      : RowStream<...>) {

        div(sum(elems), count(elems))

}
```

Scope key from before also associates iterations

GROUP BY

Order of output tuples cannot be guaranteed

Only interesting for multiple outputs i.e. iteration

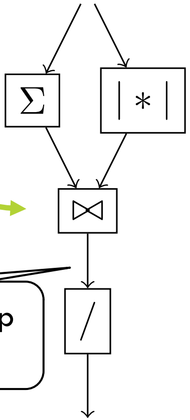


Needs to line up inputs from same iteration

There already exists an operator that does this

⋈ (join)

Needs an key to join iterations on



All inputs packaged up nicely in single row

Also works correctly for variables from outside the for-loop

CHAIR FOR COMPILER CONSTRUCTION

Roadmap

Work in incremental materialized view

- Selection by index
- Eviction
- Sharding

- Find suitable operators
- Handle shared state
- Generate boilerplate

- Representation as query
- State scoping
- Multi-arity functions

Incremental Computation → State Management → Operator Generation → Control Flow Representation → ?

ST-UDF and UDA

UDTF

<u>Evaluation</u>

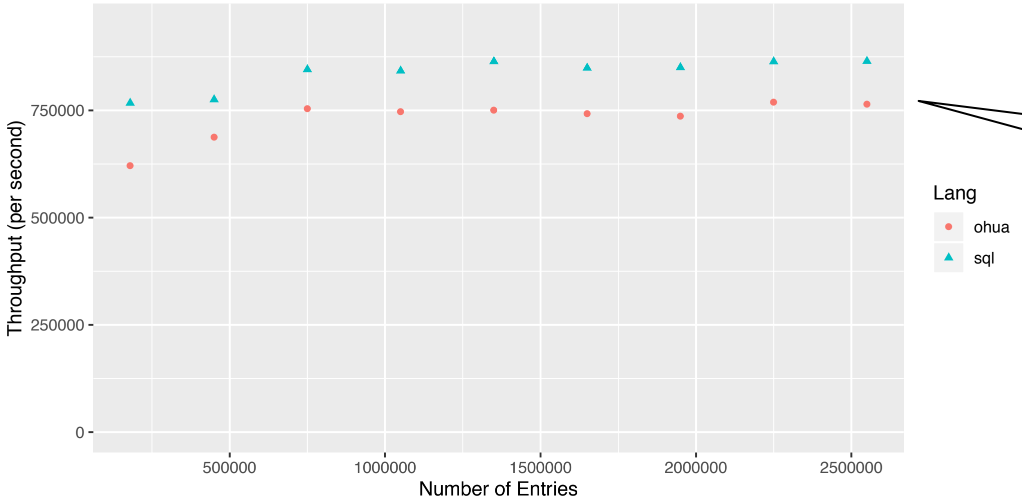
- Interoperability with SQL
- Composition/Control Flow
- Optimization (Parallelization)

CHAIR FOR COMPILER CONSTRUCTION

Evaluation – Overhead & Expressiveness

```
fn average(table: RowStream<...>) {
    for (_, elems) in group_by(0, table) {
        div(sum(elems), count(elems))
    }
}
```

Multi-argument functions and inner-joins naturally correspond

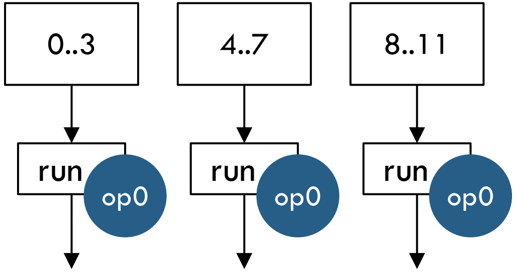


Performance difference in query due to extra operators inserted by compiler

Separate performance comparison of generated *sum* and SQL *sum* operators shows no difference

Performance of Ohua-compiled *average* query in comparison to SQL

Evaluation - Parallelism



Data sharded by range of hashes of index values

Operator state split into distinct pieces

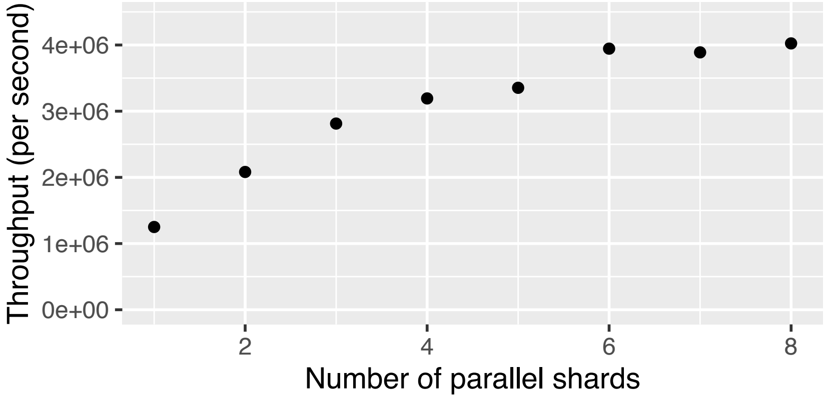
```
fn click_ana(clicks: RowStream<i32, i32, i64>)
        -> GroupedRows<i32, i32> {
    for (uid, group_stream) in group_by(0, clicks) {
        let op0 = Op0::init();
        let op0_res = op0.run(sort_on(2, group_stream));
        (uid, op0_res)
```

Iteration local state allows splitting

Leveraging the parallelism is simply setting a runtime parameter

```
let sharding_factor = 8;
let mut b = Builder::default();
b.set_sharding(Some(sharding_factor));
```

Parallel processing possible without explicit parallel contructs



Throughput of clickstream analysis with increasing sharding factor

Roadmap

Work in incremental materialized view

- Selection by index
- Eviction
- Sharding

- Find suitable operators
- Handle shared state
- Generate boilerplate

- Representation as query
- State scoping
- Multi-arity functions

Incremental Computation → State Management → Operator Generation → Control Flow Representation → **Imperative Query**

ST-UDF and UDA

UDTF

- Imperative-only query
- Embedding SQL in procedural
- Recursion

Evaluation

- Interoperability with SQL
- Composition/Control Flow
- Optimization (Parallelization)

Outlook – Embedding SQL

```
SELECT udf(x,y)
FROM tab
WHERE udf2(r,q)
```

Initial goal: Embedding Imperative in SQL

With common dataflow base we can also embed SQL in imperative program

```
let dat = run_query("SELECT...", x);
for i in dat {
    run_query("INSERT...", i);
}
run_query("DELETE...", x, y ,z);
```

SQL

Imperative

Created query dataflow representation for procedural programs

SQL compiles to dataflow

Dataflow

With Ohua, dataflow becomes common base

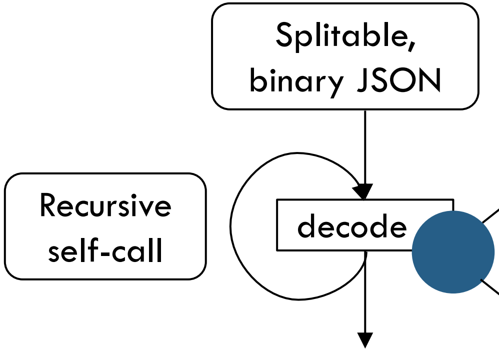
SQL involvement not necessary: Procedural-only query is possible

CHAIR FOR COMPILER CONSTRUCTION

Outlook - Recursion

```
{
    "a": ["v1", "v2"],
    "b": {
        "a_number": 1389,
        "Null": null
    }
}
```

Arbitrary nesting needs recursive decoding of inner structure



Materialization builds a map of resolved object keys [1]

Database key	Value
$.a.0	String("v1")
$.a.1	String("v2")
$.b.a_number	Number(1389)
$.b.Null	Null

1. Zhen Hua Liu et al. Closing the Functional and Performance Gap Between SQL and NoSQL. 2016. SIGMOD '16. 227–238.

Roadmap

Work in incremental materialized view

- Selection by index
- Eviction
- Sharding

- Find suitable operators
- Handle shared state
- Generate boilerplate

- Representation as query
- State scoping
- Multi-arity functions

Incremental Computation → State Management → Operator Generation → Control Flow Representation

Imperative Query

ST-UDF and UDA

UDTF

- Imperative-only query
- Embedding SQL in procedural
- Recursion

<u>Evaluation</u>

- Interoperability with SQL
- Composition/Control Flow
- Optimization (Parallelization)

CHAIR FOR COMPILER CONSTRUCTION

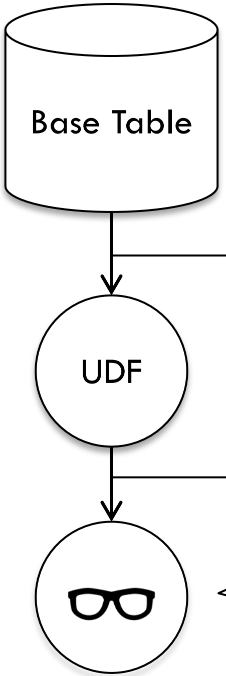
Simple Materialization

Materialization

	uid	Category	Timestamp
	1	start	1
inserted	1	other	5
	1	end	10
deleted	1	other	11

Does no processing, hence same materialization as upstream

uid	Click Distance
1	2



Data Transferred

Entire Table transferred and processed. Inefficient and with high latency

uid	Category	Timestamp
1	start	1
1	other	5
1	end	10

uid	Click Distance
1	2

Fast reads by serving from lookup table (materialization)

Incremental Materialization

Materialization

uid	Category	Timestamp
1	start	1
1	other	5
1	end	10
1	other	11

Private materialization as lookup table for downstream operators

uid	Click Distance
1	2

uid	Click Distance
1	2

Base Table

UDF

Data Transferred

Sign added to each row

Only deltas transferred and processed

sign	uid	Category	Timestamp
+	1	other	5
-	1	other	11

Operator must be able to adjust the result on delete

sign	uid	Click Distance
-	1	1
+	1	2

Output are deltas and delete outdated results

Incremental ST-UDF and UDA

ST-UDF

sign	x
+	1
-	1

$\longrightarrow \boxed{x+3} \longrightarrow$

sign	x
+	4
-	4

Revoke same result on delete

For a 1:1 function $f(x)$ the incremental function f' is:

$$f'(+,x) = \left(+, f(x)\right)$$
$$f'(-,x) = \left(-, f(x)\right)$$

UDA

sign	x
+	1

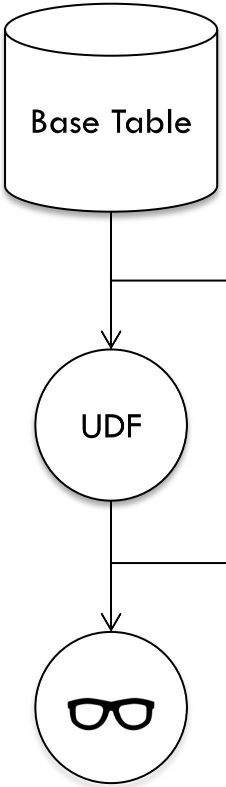
sign	x
-	1

$\longrightarrow \boxed{count} \longrightarrow$

sign	x
+	4
-	4

Simple materialization

uid	Category	Timestamp
1	start	1
1	other	5
1	end	10
1	other	3



sign	uid	Category	Timestamp
+	1	other	3

sign	uid	Click Distance
-	1	2
+	1	3

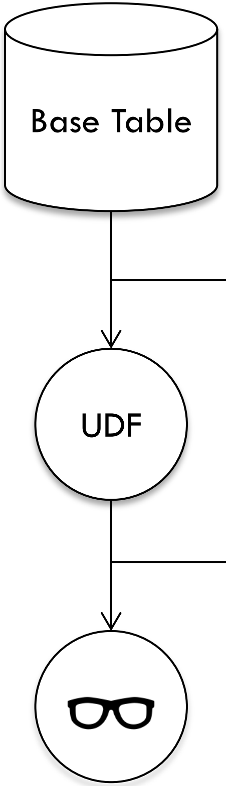
Update Path (Insert)

```
INSERT (1, other, 3)
INTO 'Base Table';
```

uid	Click Distance
1	3

Noria Execution Model

uid	Category	Timestamp
1	start	1
1	other	5
1	end	10
1	other	3



sign	uid	Category	Timestamp
-	1	other	5

sign	uid	Click Distance
-	1	3
+	1	2

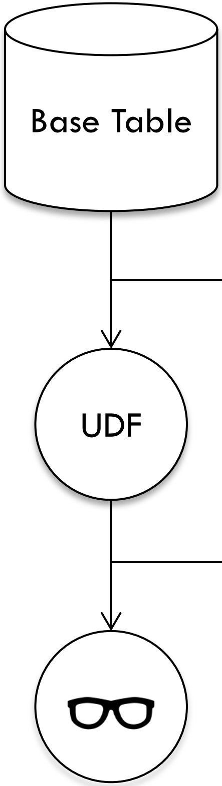
uid	Click Distance
1	2

Update Path (Delete)

```
DELETE (1, other, 3)
FROM ‘Base Table’;
```

Noria Execution Model

uid	Category	Timestamp
1	start	1
1	other	5
1	end	10
1	other	3



sign	uid	Category	Timestamp
-	1	other	5

sign	uid	Click Distance
-	1	3
+	1	2

uid	Click Distance
1	2

- On-line inserts
- On-line deletes
- Order is random

- Commutative
- Incremental
- Reversible

  Operations

UDF State Design

```
let state = iseq::Seq::new();

for (_, cat, time) in stream {

    if cat == start_cat {
        state.start(time)
    } else if cat == end_cat {
        state.end(time)
    } else {
        state.record(time)
    }

}
```

State S

Defines actions
A: {Start, End, Record}

Projection
$f : input \rightarrow A$

Affected by sign

Not affected by sign

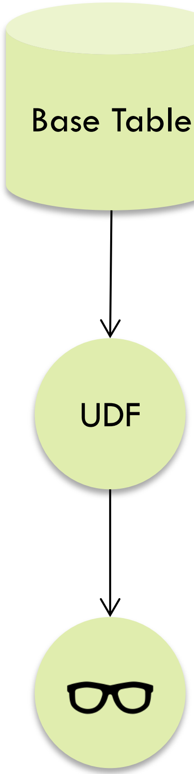
Successively apply all actions <u>and sign</u> to state
$app : \pm A \times S \rightarrow S$

```
state
    .complete_intervals()
    .map(Interval::len)
    .average()
```

Computation
$comp: S \rightarrow output$

$$UDF: [\pm input] \rightarrow output$$

Interval Sequence as State



uid	Category	Timestamp
1	start	1
1	other	5
1	end	10
1	other	3

`[[5,3,10].length()].average() == 3`

uid	Click Distance
1	3

$$s\colon \underbrace{[\,[l_0,u_0),[l_1,u_1),[l_2,u_2)]}$$

$t \in T$ such that

- $t \geq \begin{cases} l_1 & \text{if } l_1 \text{ exists} \\ u_0 & \text{otherwise} \end{cases}$
- $t < \begin{cases} u_1 & \text{if } u_1 \text{ exists} \\ l_2 & \text{otherwise} \end{cases}$

Invariants

- l_1 or u_0 must exist
- u_1 or l_2 must exist

Merge intervals to maintain

Conclusions

Must be
- Reversible
- Commutative

UDF State Design → Integrate into Partial State → Ohua Compiled UDF → Query Elements in the UDF → **Pure Ohua Query**

CHAIR FOR COMPILER CONSTRUCTION

Conclusions

UDF State Design

Integrate into Partial State

Ohua Compiled UDF

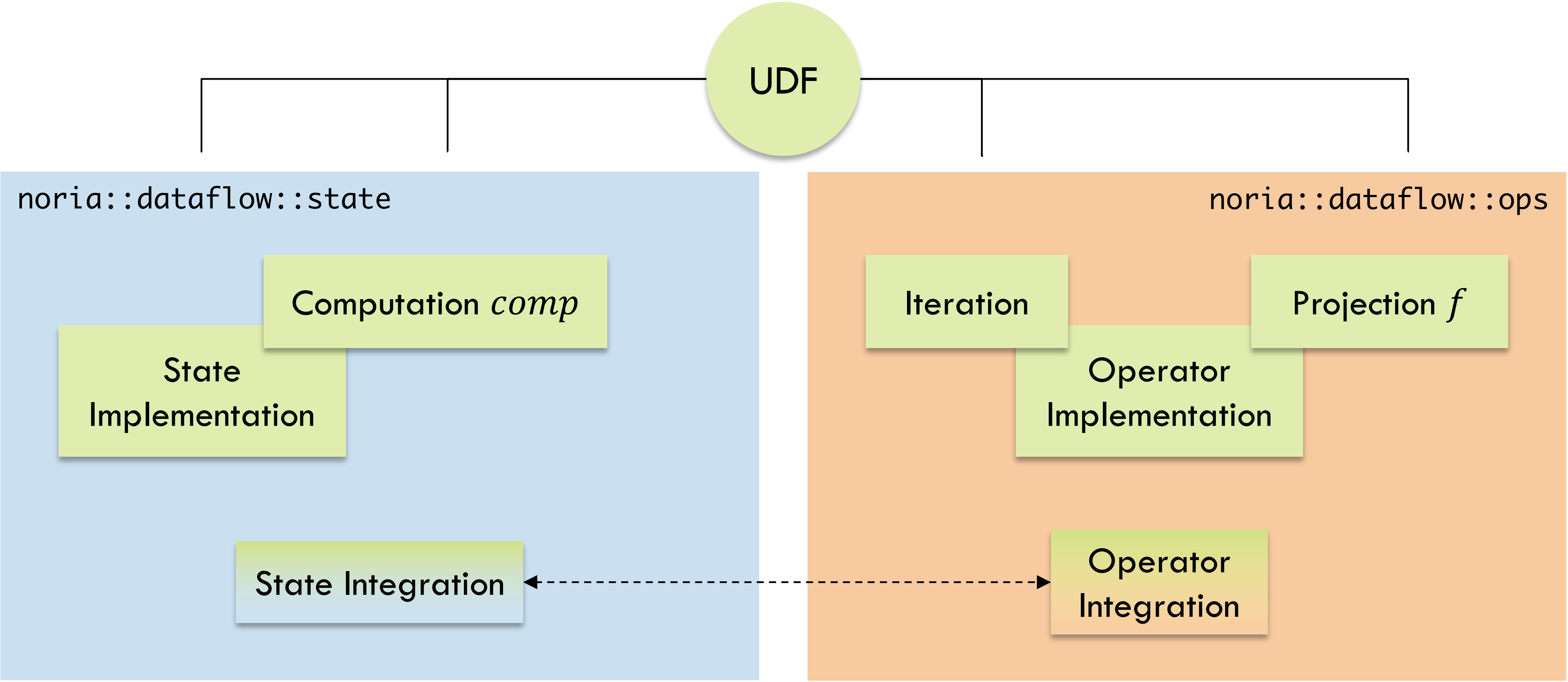
Query Elements in the UDF

Pure Ohua Query

To Support
- Eviction
- Lookups

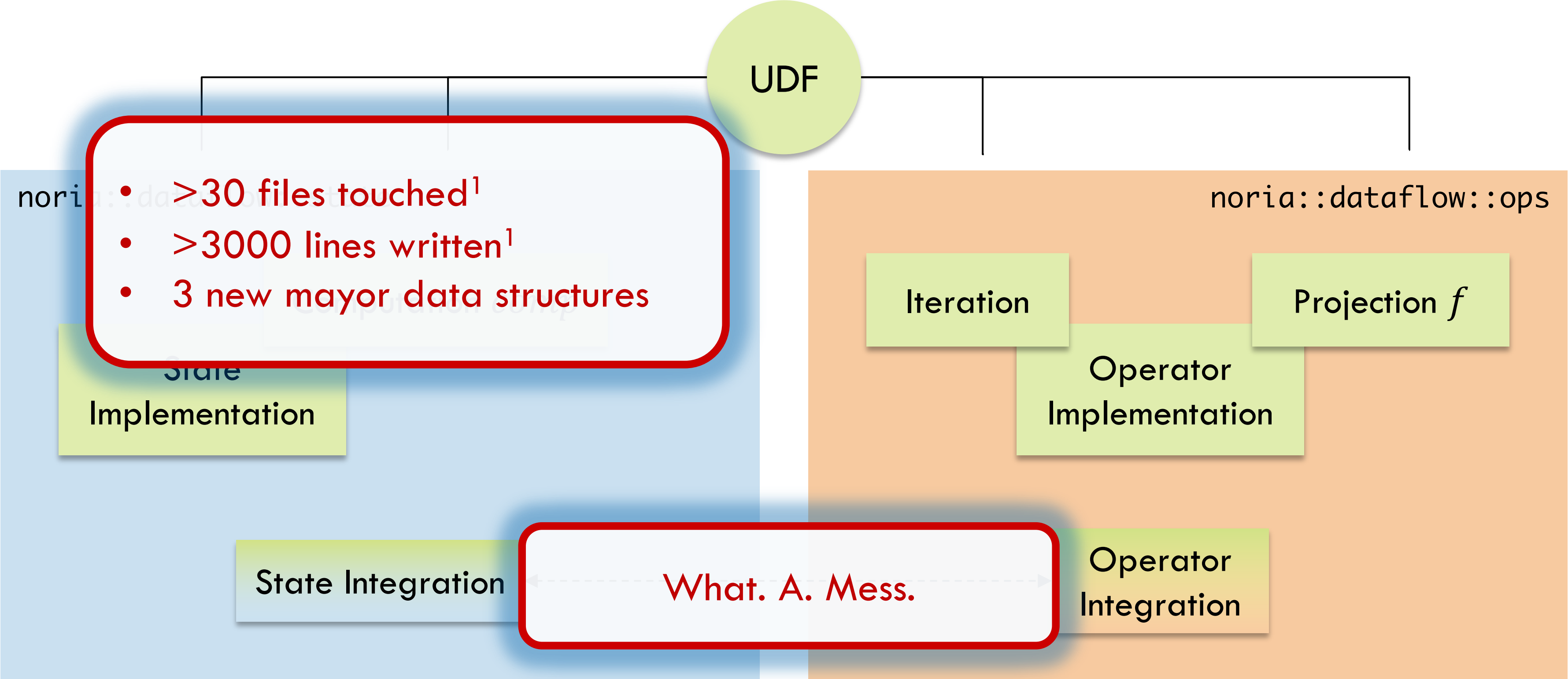
CHAIR FOR COMPILER CONSTRUCTION

Manual Implementation



CHAIR FOR COMPILER CONSTRUCTION

Manual Implementation



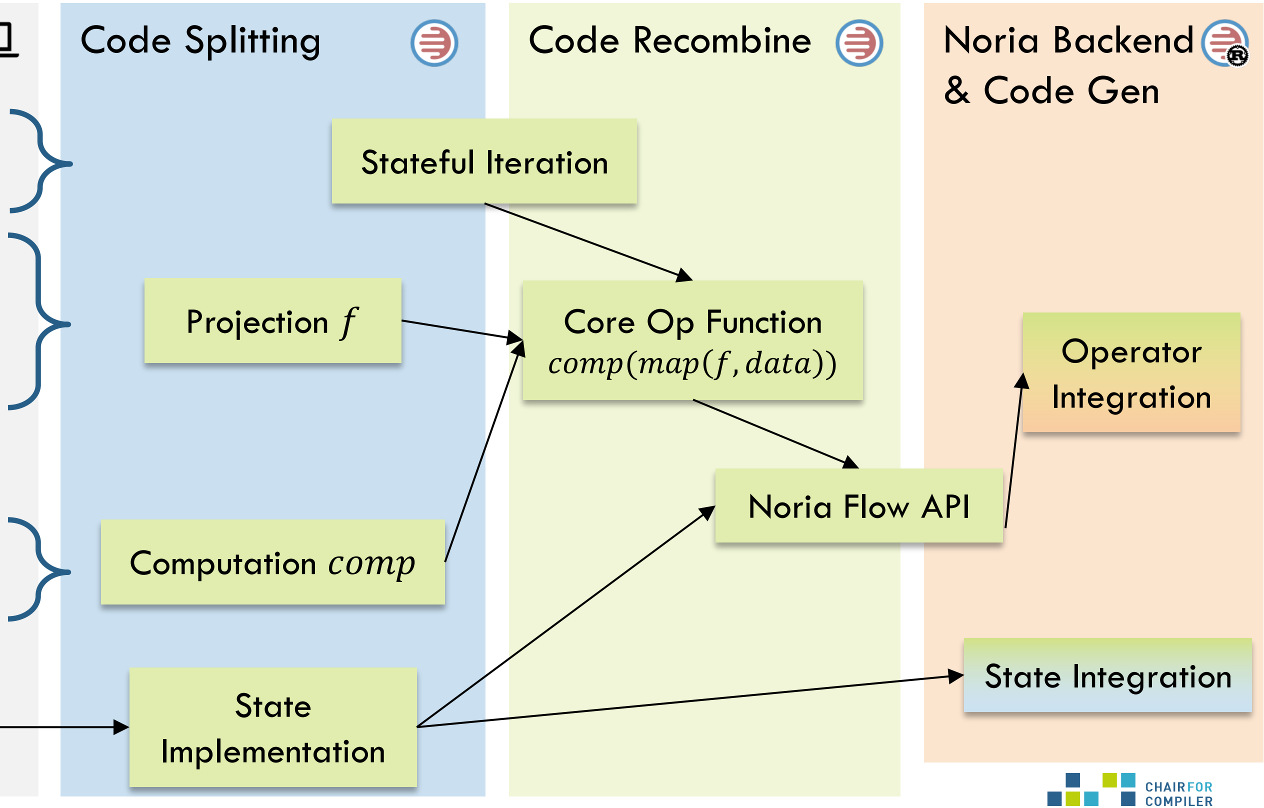
1. For whole implementation including intermediate prototypes and test code. Approximately 50% used exclusively for UDF.

Operator Compilation

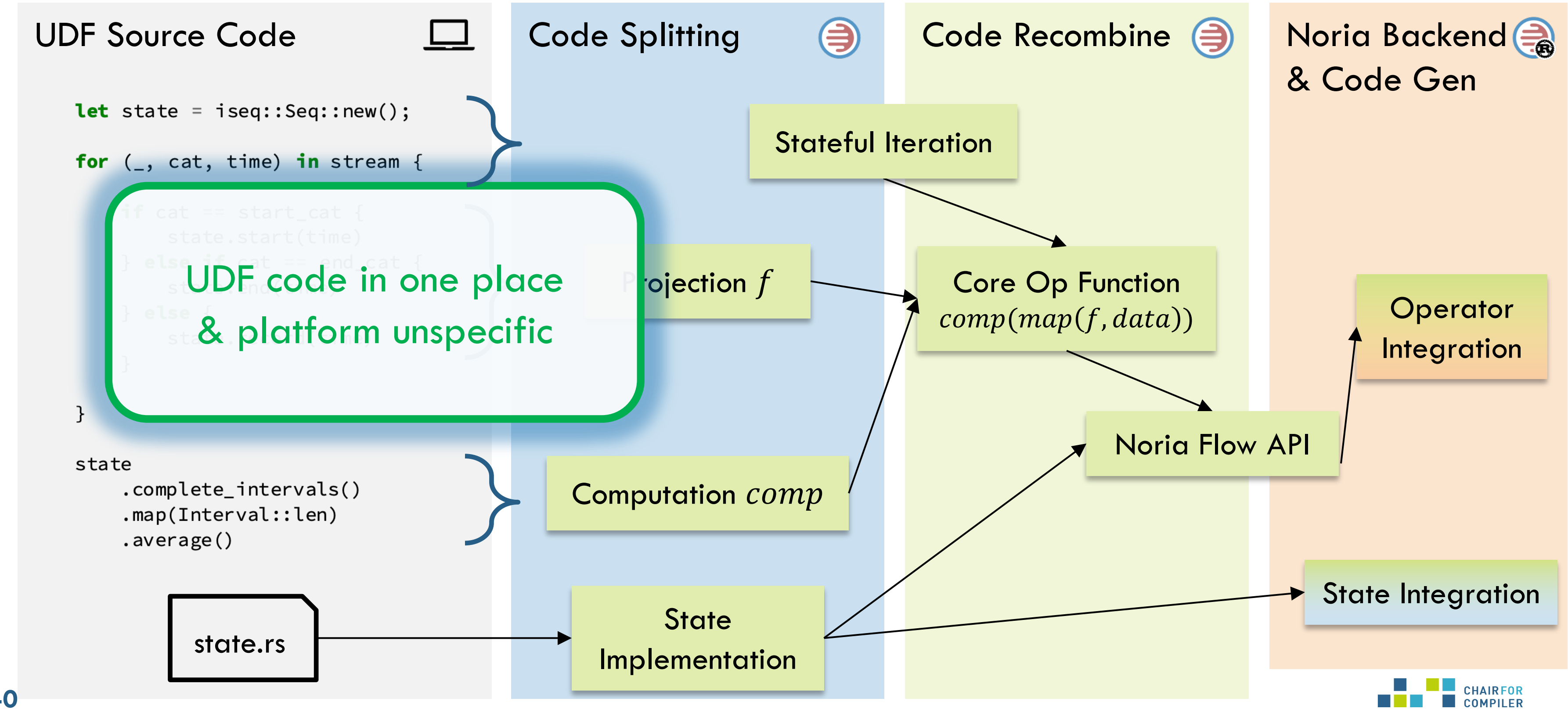
UDF Source Code

```rust
let state = iseq::Seq::new();

for (_, cat, time) in stream {

    if cat == start_cat {
        state.start(time)
    } else if cat == end_cat {
        state.end(time)
    } else {
        state.record(time)
    }

}

state
    .complete_intervals()
    .map(Interval::len)
    .average()
```

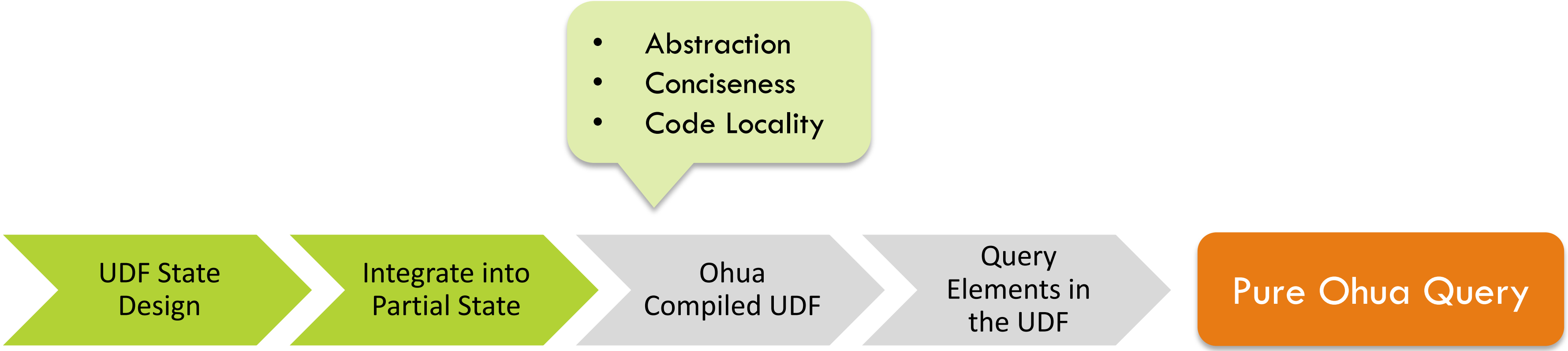
state.rs



Operator Compilation



Conclusions



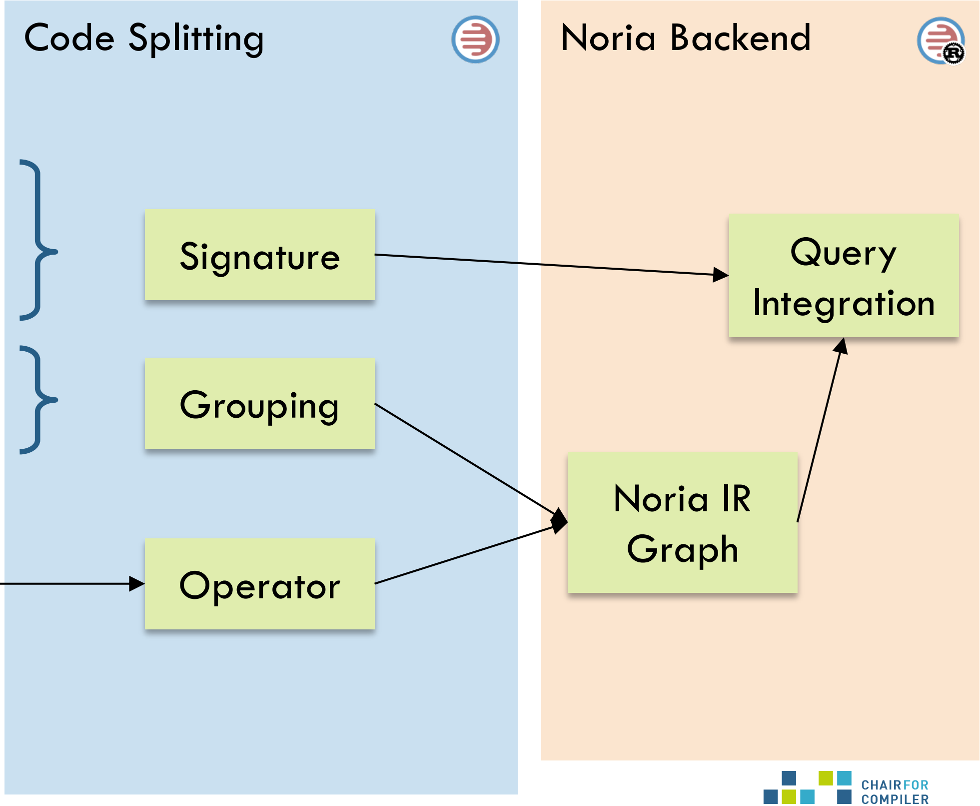
CHAIR FOR COMPILER CONSTRUCTION

UDF Compilation

UDF Source Coce

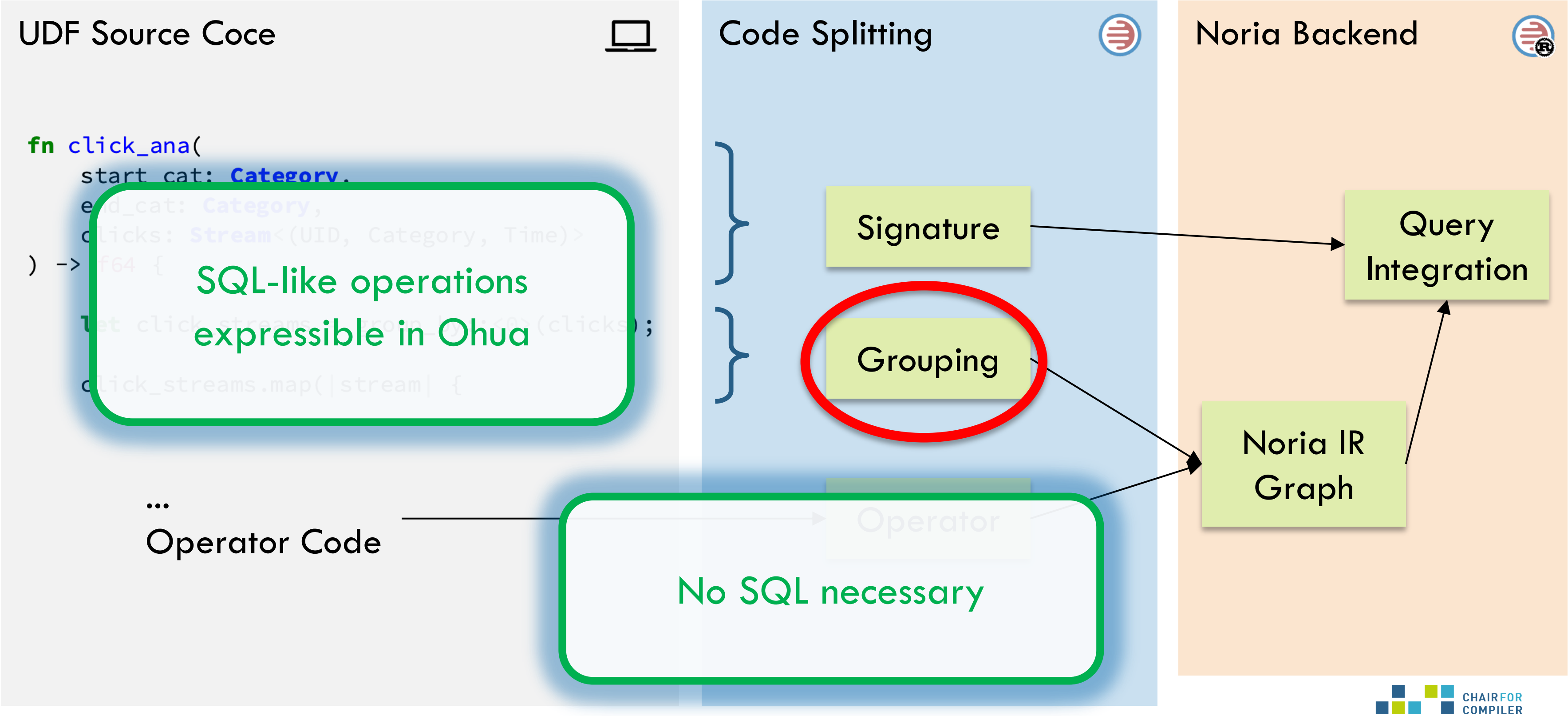
```rust
fn click_ana(
    start_cat: Category,
    end_cat: Category,
    clicks: Stream<(UID, Category, Time)>
) -> f64 {

    let click_streams = group_by::<0>(clicks);

    click_streams.map(|stream| {
```

…
Operator Code



CHAIR FOR COMPILER CONSTRUCTION

UDF Compilation

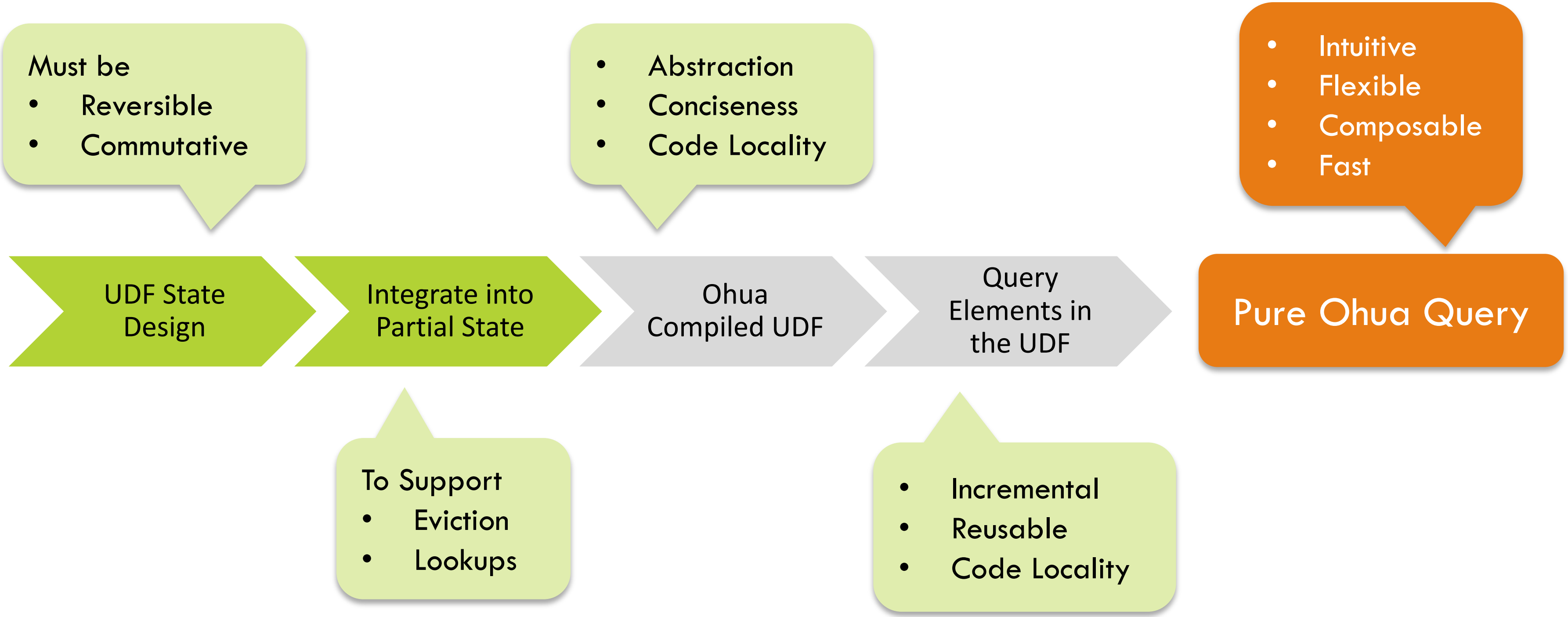


Conclusions

UDF State Design → Integrate into Partial State → Ohua Compiled UDF → Query Elements in the UDF → **Pure Ohua Query**

- Incremental
- Reusable
- Code Locality

Conclusions



CHAIR FOR COMPILER CONSTRUCTION

Conclusions

Must be
- Reversible
- Commutative

To Support
- Eviction
- Lookups

- Abstraction
- Conciseness
- Code Locality

- Incremental
- Reusable
- Code Locality

UDF State Design → Integrate into Partial State → Ohua Compiled UDF → Query Elements in the UDF → Pure Ohua Query

- Intuitive
- Flexible
- Composable
- Fast

- Additional State Patterns
- State builder Toolkit

- More Query Elements in Ohua
- Multi-State UDF's
- Non-SQL Datatypes