TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Bereich Ingenieurwesen, Institut für Technische Informatik, Lehrstuhl für Compilerbau

Ohua-powered, Semi-transparent UDF's in the Noria Database

Justus Adam

Born on: November 17, 1992 in Dresden

Master Thesis

to achieve the academic degree

Master of Science (M. Sc)

First referee

Prof. Dr.-Ing. Jeronimo Castrillon

Second referee

PD Dr.-Ing. habil. Dirk Habich

Supervisor

Dr.-Ing. Sebastian Ertel

Submitted on: November 14, 2019
Defended on: December 12, 2019

The **S**tructured **Q**uery **L**anguage (SQL) is the de facto standard for database programming. SQL is a declarative language, letting programmers write short queries easily and allowing for aggressive performance optimizations.

However this style diverges greatly from the widely used sequential programming style. This makes complex queries hard to write and understand [21].

SQL also does not allow for the use of state. Algorithms that use state to achieve efficiency are impossible to implement [12]. State can only be used in so called User Defined Functions (UDF) which come with a significant performance penalty, because the optimizations are no longer available.

In this work we improve upon the integration of imperative User Defined Functions. We use a the parallelizing compiler Ohua to generate query fragments from the UDF's. These fragments run like native SQL queries and can benefit from database optimizations, such as parallelizing.

The fragments contain operators generated by the compiler, which are state enabled. This allows us to express efficient stateful queries, as in [12].

Contents

1 Introduction

The adoption of database technology is still increasing, predominantly in areas such as, for instance, DBMS backed websites and Big Data. With the increase in adoption comes a more diverse set of tasks that databases are used for and new domains they are used in [1]. This also increases the demand for developers that program these databases. Relational database management systems (RDBMS) have largely adopted the SQL language to express the users queries. SQL is designed to concisely express relationships of tables, rows and columns, i.e. the structure of a relational database. It is also declarative rather than procedural, specifying the *result* of the query, but not precisely *how* to achieve it. This design lets the database engine choose a concrete processing strategy. This strategy is also called a *query plan*. The generation of the plan is informed by the engines knowledge on the primitive operations such as *join* and *filter* and the schema of the stored data. The engine can use this to design a plan that avoids redundant computation, shares partial results, distributes data and computation on multiple cores and so on. All this is done automatically and invisibly to the programmer.

There is an ongoing effort to move more data processing tasks from applications, such as webservers, into the database. The goal is to take advantage of the unique benefits mentioned before as well as reduce communication and serialization overhead between application and database.

Most of these tasks are written in procedural languages. Programmers tend to cope easier with procedural languages [27] than declarative ones. Complex SQL queries in particular are difficult to write and even harder to read [21]. These reasons make it easy to empathise with the fact that many developers have difficulty [1] or are unwilling [3] to switch to SQL.

In addition the problems with the language many common forms of data do not fit well into the SQL data model. Relational databases expect a structure with tables which are a set of rows, that are divided into columns. Familiar data constructs, such as arrays, nested structs or unions are not available. These have to be explicitly modeled using foreign key relationships. Writing these explicit conversions is time consuming and resulting schemata are difficult to migrate when formats change. With the rapidly growing number of data sources and their ever changing formats explicit conversions are unmanageable. Vendors and research alike are expanding the capabilities of databases with respect to handling exotic data formats [20, 5]. "The Claremont Report on Database Research" [3] identifies this as a key challenge for database research. In a similar vein there is a demand for support for processing tasks that do not fit in the SQL model like MapReduce [12, 22, 8], high-performance array processing [6] and machine learning [19, 15].

These efforts will always lag behind the most recent needs. This leads to inconsistencies between different systems and increases the amount of different processing strategies and patterns developers need to be aware of. The fallback mechanism available to almost all databases are *user defined functions (UDFs)*. These allow developers to define new, custom processing

primitives in familiar, procedural languages. This offer a much faster path to integrating custom processing tasks and data formats. The UDF interfaces are also more portable than the plethora of vendor extensions. The procedural, object-oriented and functional interfaces of the languages used to define UDF's are much more familiar to most programmers. This eases the barrier to entry for programming databases. This is an important consideration due to the increasing demand for database programmers [3, 1] Lastly certain processing tasks can be implemented much more efficiently with the use of state, which is not possible in SQL [12].

Currently UDF use comes at a significant cost. The DBMS must treat a UDF like a black box, as it lacks the in-depth knowledge on invariants and relationships it possesses for the other basic query operators. Depending on the type of the UDF, this prohibits many or all of the usual query optimizations. Today the amounts of data collected is increasing rapidly and ever more complex types of analysis tasks are employed. This causes a significant need for performance, and makes this cost unacceptable.

Große et al. were addressing these problems by annotating UDF's [14]. This gives the query engine more knowledge and enabling more efficient, parallel schedules. This is a promising approach but still leaves the UDF opaque. Friedman et al. made UDF's more transparent. They support special UDF's with a MapReduce [8] pattern [12]. They are able to achieve efficient parallel execution with a controlled form of state via MapReduce. They were able to show that this approach can outperform SQL significantly in complex multi-join processing tasks. In the cases they outline state is used to express the processing tasks more concisely and simply. This then also translates to better performance.

We want to expand upon the ideas of Friedman et al. and lift the restriction to MapReduce patterns. We instead use a parallelizable language called Ohua [9]. This will enable a broader range of UDF's. The benefit of MapReduce was that the UDF is split into two smaller parts and the *map* part is duplicated and parallelized. These smaller computations can be scheduled individually, allowing more scheduling control. Our approach achieves this for more general programs by compiling the UDF to a dataflow graph. This graph can merge directly with SQL query graphs. This gives the query engine a fine grained view of the inner workings of the UDF. The UDF is no longer a black box to the query engine. This enables previously inaccessible optimizations, such as parallelization.

A central example in the work of Friedman et al. is a *clickstream analysis query*. The purpose of this query is to calculate the average number of clicks it takes a certain user to navigate from a page of a specific start category to pages of some end category. For instance how make clicks does it take on average for the customer to go from the home page to confirming a purchase. An implementation in plain Rust is straight forward and quite easy to understand, see Figure 1.1a with 0 and 1 being the start and end category respectively. The equivalent SQL query has significantly higher cognitive complexity (Figure 1.1b). Unfortunately their paper does not provide the MapReduce source for their version of a clickstream analysis so we cannot compare it here. Ohua can express MapReduce patterns as well, via stateful `for` loops. Which allows us to implement the algorithms from [12]. In Figure 1.2 we show how our prototype implements the *clickstream analysis query*, which will be a central example in this work.

Ohua also supports modularity. Common functionality can be defined as functions and libraries which can be used in more than one query. Our prototype also provides database specific functionality, such as partitioning with `GROUP BY` in the UDF itself. These integrate naturally in the form of builtin functions, reducing the need to reimplement such functionality manually, but without needing additional language constructs.

Our prototype targets Noria, an experimental database developed at MIT [13]. Noria uses materialization to speed up read performance. A central feature of our UDF's is that they are able to use state. This enables more efficient queries, like in [12]. By targeting Noria we can leverage the benefits from the integrated UDF's and Norias efficient reads at the same time.

Implementing transparent UDF's is a large and complicated task. As an additional effort the

Noria database, which this work is implemented in, does not support any notion of UDF's in its current form. We chose Noria, because the materialization is also a form of state, giving us a framework to work in. The lack of existing UDF's in Noria means we first must develop interfaces for custom operators in Noria.

For this work we constrain the space of input UDF's for the compilation process. This costs generality, but means the effort for development of the theory and implementation manageable. The restrictions are that the operators we generate either operate on single tuples or are aggregations. The input UDF must be expressible in terms of these operators and standard database operators alone. We discuss future work towards generality in the Outlook Section (7).

The concrete contributions of this work are:

- An implementation of single-tuple UDF's and stateful User Defined Aggregations (UDA) in Noria.
- Development of a *scope* concept which emulates nested control flow Noria.
- A compilation from Ohua programs into parameterized Noria queries.
- A set of experiments comparing our UDF's to native operators and queries and testing whether our compilation process enables use of database optimizations, in this case parallelization.

The following section introduces User Defined Functions in more detail. We briefly describe the design of the Noria databaes to then illustrate the particular difficulties with UDF's in Noria. Following this we describe the abstract concepts we use in the implementation in Section 4. We focus on how to correctly integrate state in Noria operators and how to represent imperative programs in a database. Section 5 applies these concepts in a concrete implementation. We describe how we generate Noria operators, how we generate queries and how they are linked into the database. We test this implementation in comparison to native SQL queries in Section 6. We also implement the clickstream analysis query with out prototype and test whether it is able to benefit from database optimizations. We then zoom out from this particular work and take a look at how this approach could apply more generally in Section 7. Finally we place this work in the context of current UDF and database research (Section 8) and conclude in Section 9.

```rust
fn click_ana(
    clicks: &mut Vec<(UID, Category, Time)>
) -> i32 {
    let m = HashMap::new();
    let counts = Vec::new();
    for (uid, cat, _) in clicks {
        let prev = m.get(uid);
        if cat == 0 {
            m.insert(uid, 0);
        } else if prev != -1 {
            if cat == 1 {
                m.insert(uid, -1);
                counts.push(prev);
            } else {
                m.insert(uid, prev + 1);
            }
        }
    }
    let len = counts.len();
    counts.into_iter().sum() / len
}
```

(a) Clickstream Analysis with State in Rust

```sql
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
          pagetype
          (c1.page_id) = 0 AND
          pagetype
          (c2.page_id) = 1
        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;
```

(b) Clickstream Analysis in SQL, as taken from [12]

```
fn op_core(RowStream<(UID, Category, Timestamp)>)
        -> RowStream<i32> {
    let sequences = IntervalSequence::new();
    for (_, category, timestamp) in group_stream {
        let time = deref(timestamp);
        let cat = deref(category);
        if eq(cat, 1) {
            sequences.open(time)
        } else if eq(cat, 2) {
                sequences.close(time)
        } else {
            sequences.insert(time)
        }
    };
    sequences.compute_new_value()
}

fn main(clicks: RowStream<(UID, Category, Timestamp)>)
        -> GroupedRows<UID, i32> {
    let click_streams = group_by(0, clicks);
    for (uid, group_stream) in click_streams {
        op_core(group_stream)
    }
}
```

Figure 1.2: Clickstream query in Ohua

2 User Defined Functions in Noria

In its current form Noria does not support UDF's. This means there is no existing way to use custom code, as operator or otherwise, in Noria queries. Because our compiler will generate custom operator, we first require at least rudimentary support for those in Noria.

In this section we will first describe the concept of a User Defined Function. We will then introduce the Noria database and see how it differs from other database systems.

After that we specifically take a look at the features UDF's in Noria must support. In Section 2.3 we discuss incremental computation, followed by partial state in Section 2.4.

2.1 User Defined Functions

A user-defined function is a way for database programmers to implement their own processing primitives. This is often used for converting units, serialization and other functionality from third party libraries. A user-defined function is written in a non SQL, often procedural language. There is also a procedural dialect of SQL that allows the definition and manipulation of variables, as well as pointers into tables (called *cursor*) to implement stateful processing tasks. Most databases allow linking of any external code, so long as it fulfills a specific interface. Often this is facilitated via a C API.

UDF's can be broadly separated into four categories, listed in Table 2.1[1]. The nomenclature for functions which emit multiple rows is adopted from Postgres. The difference between these types of UDF is whether they process single tuples or multiple tuples, listed in the table as *multiplicity*. Some of these types of UDF inherently require state for processing. Set Returning Functions (SRFs) for instance use the state as a generator. The input tuple is used to initialize the state which is advanced in subsequent calls to produce a new tuple on each call. Table 2.1 also lists which types of UDF need state and which do not [25, 24, 16]. The support for each of these types differs across various database systems, a selected list of UDF support in database systems is shown in Table 2.2. At the time of this work there is no support for any kind of User Defined Function in Noria yet.

Additionally, Postgres [26] supports the notion of *user defined operators*. These are similar to UDF's but convey additional information to the query planner to better schedule the operator.

[1] The table lists Postgres as not supporting UDTF's. This is because it only allows SQL for UDTF's and no procedural languages (like procedural SQL). SQL-only UDTF's do not add new functionality, as a procedural UDTF would.

Type	Input:Output multiplicity	State necessary
UDF (single datum/tuple)	$1:1$	no
Set **R**eturning **F**unction	$1:n$	yes
User **D**efined **A**ggregate	$n:1$	yes
User **D**efined **T**able **F**unctions	$n:m$	yes

Table 2.1: Types of UDF's

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

Table 2.2: Support for UDF by type

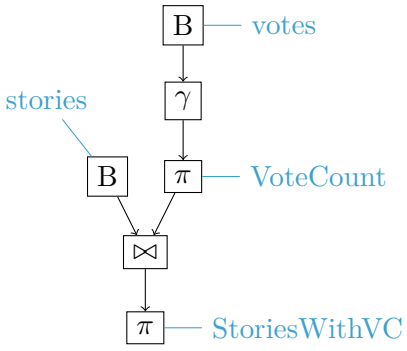
2.2 Noria

Noria[13] is a novel approach to delivering high performance for read heavy applications, predominantly targeting web sites and services. It uses a technique called *materialized view*s whereby the results of a query are saved in memory by the database to deliver subsequent results instantly. Whereas materialized views were known before, the novelty of Noria lies in that not the entire result set of a query is kept around but only relevant, partial results. This is called a *partial materialized view*. The system is designed for applications with non-uniform access patterns on the stored data, such as websites, where a small subset of data is accessed very frequently. Through a combination of demand-driven query computation and random eviction the system eventually converges on the set of most frequently used data and query results.

During execution a query in Noria is an acyclic graph of relational computing operators. The data tables are roots of this graph and the leaves are *reader* nodes from which the query results can be read. An example of a query (taken from [13]) and its corresponding runtime graph can be seen in Figure 2.1b. The rendered graph uses a node labeling shorthand explained in Table 2.3. The same shorthand is used throughout this thesis in various Noria graph renderings.

```
CREATE TABLE stories
(id int, author int, title text, url text);
CREATE TABLE votes (user int, story_id int);
CREATE INTERNAL VIEW VoteCount AS
  SELECT story_id, COUNT(*) AS vcount
    FROM votes GROUP BY story_id;
CREATE VIEW StoriesWithVC AS
  SELECT id, wuthor, title, url, vcount
    FROM stories
    JOIN VoteCount ON VoteCount.story_id = stories.id
   WHERE stories.id = ?;
```

(a) A typical query in the Noria SQL dialect [13]



(b) Runtime graph

The system realizes the materialization by storing computed results from expensive operators automatically in an associated state which acts like a cache. There can be multiple indexes into the cache which the system automatically chooses based on the needs of the query operators and the query parameters. For instance in Figure 2.1a the parameter `stories.id` will be used to index the materialization state of the query operators. As would the `GROUP BY` argument

B	Base table	Comment
γ	Group By	
$\|*\|$	Count	
Σ	Sum	
$\bowtie$	Join	
σ	Filter	
π	Projection	Renames columns, adds literals

Table 2.3: Shorthand for node labels

`story_id`, which happen to be the same in this case.

Reads first try to satisfy their request by looking at the state of the *reader* node. The desired result will either be present or *evicted*. Evicted entries signal that there could be a result here, which has been garbage collected to reduce memory consumption. When such an entry is encountered the system triggers an *upquery*. The upstream operator that receives an *upquery* first tries to fetch data from the caches of its ancestors to compute the requested result. If this lookup fails it also triggers an *upquery* to its ancestors. Through this recursive process the upquery cascades up the graph until the data necessary to backfill the missing entries is found. The system knows how input data relates to results via the indexes it has constructed before. In the worst case scenario an upquery can cascade all the way to the base tables. Once the upqueries terminate they set off an update downstream, computing and filling in the missing entries in the materializations along the way.

Upqueries are only one scenario that triggers computation. The other happens on changes to the base table. This causes an update to flow downstream. Any affected entries which are not evicted are recomputed, keeping the view consistent.

Upqueries, updates and reads can all take place simultaneously. There may be several upqueries and updates in-flight at any given time. This results in significantly better performance but sacrifices consistency. A client that performs an *insert* or *delete* operation on a table, may not see their change reflected on the next *read* request. This type of consistency model is called *eventual consistency*.

2.3 Incremental computation

Updates, either via upquery or change to the base table, flow down the dataflow graph as *deltas*. A *Delta* is a data tuple with a *sign* attached. The sign indicates whether the tuple was newly inserted (+) or deleted (−). An updated value is signified by a delete of the old value and an insert of the new value. Propagating the deltas through the graph is handled by the Noria engine. However recalculating results cannot be done automatically be the engine and is left to the operator. Figure 2.2 shows the simplified[2] main processing routine of a Noria operator. The handler function `on_input` both consumes and produces deltas. The *sign* in this case is modeled using a boolean.

2.3.1 Incremental UDF's

UDF operators must also be incremental. Handling deltas correctly means retroactively updating all output values the operator has produced which are affected by incorporating this delta.

Every operator therefore must be able to answer the following questions:

1. What set of values it has output before are affected by a given delta?
2. How does each of these values change?

[2]Some arguments are omitted, and `Records` has been renamed to `Vec<Delta>` for easier understanding.

```
type Row = Vec<DataType>;
type Delta = (bool, Row);

struct ProcessingResult {
    results: Vec<Delta>,
    misses: Vec<Miss>,
}
```

```
trait Ingredient {
    fn on_input(
        &mut self,
        from: LocalNodeIndex,
        data: Vec<Delta>,
        states: &StateMap,
    ) -> ProcessingResult;
}
```

Figure 2.2: Signature of the main operator processing routine and the result structure

Because UDF's run as the processing core of an operator it may need to be involved in answering these questions. Depending on the type of UDF answering these questions can be partially automated. Reasoning about deltas is hard, so our goal here is to automate as much as possible to reduce the burden on the programmer.

```
pub enum Action {
    Add(i32)
}

pub struct Sum(i32);

impl Sum {
    pub fn apply(&mut self,
                 action: Action,
                 is_positive: bool) {
        match action {
            Action::Add(i) =>
                if is_positive {
                    self.0 += i;
                } else {
                    self.0 -= i;
                }
        }
    }

    pub fn get_sum(&mut self) -> i32 {
        self.0
    }
}
```

Figure 2.3: Implementation of the state for `sum`

Pure, Single Tuple Functions (or Simple 1-tuple UDF) operate on a single tuple at a time and do not modify any state. Examples are value conversions and classic arithmetic functions. Both questions are easy to answer in this case. The set of affected values is simply the singular output value produced by running the function.

The change in the value is similarly straight forward. If the input value is to be deleted, the output value should also be deleted. If it is inserted, the output should be inserted. The output of the operator is therefore the sign of the input and the calculated value.

Stateful Aggregations (UDA) Aggregations operate on a sequence of tuples, producing a single output value. The classic examples are `count` and `sum`, but more complex aggregations

are also conceivable, such as average, variance etc. Aggregations are always in the presence of some *grouping criterion* describing the set of values to aggregate together. Each set of values is aggregated separately. Because any of those aggregation results might need to be updated later, the aggregation needs to store a state for each result. Separating the state according to the corresponding group can be automated and does not concern the UDA.

Output of the aggregation is either the state itself, such is the case for `sum`, or a value derived from it, such as would be the case for an `average`, which needs to maintain both a count and a sum of the elements as state. Tuples affected by a given delta is the single output value for the group the delta belongs to. This means an aggregation UDF operator simply returns an update, that deletes the old aggregation result and inserts the new one.

Calculating what the new result is cannot be so easily automated. Modifications to the state, which were caused by the previous insert of the value, have to be reversed. How this is achieved depends on the concrete aggregation. As an example Figure 2.3 shows an implementation for a `Sum` state that is capable of handling deltas by reversing previous modifications. The `is_positive` boolean flag is used to indicate whether the `Add` occurred for a positive or negative delta. In case of a negative delta (delete) the state subtracts the value instead, reverting its previous insertion.

User Defined Table Functions operate on sequences of tuples, maintaining state and produce an arbitrary number of output tuples. Answering either question in this context is not possible without knowledge about the state and the processing function. This makes it very difficult to automatically provide an incremental operator for a UDTF.

Which output values are affected depends on their association with the input as well as the association between the values in the state. An interface for this type of function would necessarily be almost the same as the interface for `Ingredient` in Figure 2.2. More precisely it would consume a delta and a state and produce an updated state and one or more output deltas. It would be required of the function to do bookkeeping internally to ensure all affected results are updated.

As a simple example consider an `enumerate` function which assigns every row an increasing, unique, continuous index. If the row with index 4 is deleted, the index of every subsequent row would have to be shifted up by 1 to keep the index continuous. To know *which* where the subsequent rows, we would have to store them somehow, causing significant overhead. Any kind of windowed computation, which are very common, would face a similar situation.

For simplicity we only consider single-tuple UDF's and UDA's for this prototype. In Section 5.1.4 we describe how the two types of functions are embedded into the dataflow by fitting an appropriate shell[3] that satisfies the interface of a Noria operator.

2.4 Partial State

Most Noria operators have an associated state. It is used for materializing the computation results ans sometimes to facilitate the computation itself, as is the case for `join`. This state is a map from indices to previously computed results. Indices can be primary keys, groups, query parameters etc. and are automatically chosen by the system based on the users query. A unique feature of the Noria model is that not all results are kept. If the memory consumption of the query is too high the system *evicts* entries from the state. The materialization is *partial*.

User-defined functions that possess state should follow the same eviction orders or they will inadvertently leak memory. As discussed in the previous section, UDF state in Noria is more

[3]The nomenclature is adopted from [17]

complex due to the fact that it needs to do bookkeeping to reverse computations safely. Depending on the computation this may mean keeping several input values stored in the state. For example the state of our clickstream analysis query (see Figure 6.7) must keep all timestamps stored in its state to recalculate interval contents if the bounds change. UDF states can therefore become large quickly. If they were disconnected from the partial state structure Noria would be unable to reclaim this memory, potentially leaving large chunks of memory occupied by state for unused calculations.

Single tuple UDF's are stateless and thus not affected by this. Aggregations and Table Functions must keep state and therefore also support reclaiming of the memory used by its state. UDTF's are complicated to integrate properly. They are too general and it is not obvious what key their state should be indexed by and out of scope for this work. UDA's are more straight forward. Aggregations in Noria are always accompanied by a grouping criterion that identifies the set of rows to aggregate over. Usually this will be specified by the user in a `GROUP BY` clause. The materialization for an aggregation uses this criterion as an index. Data stored in the materialization is currently restricted to `Rows` the operator produced as results. This is sufficient for simple aggregations such as `count` and `sum`, where the return value is the same as the aggregation state. This is not true for all aggregations, `average` and `variance` for instance need state which is different from the computed value. We can use the same as the simple aggregations but the state must be extended.

2.5 Extended partial state

The solution we came up with is to augment the state structure used in Noria. State in Noria is a map[4] of maps. The outer map is indexed by the *kind* of key. The inner map is indexed by the *key* values. This structure is reminiscent of a two-level tree. We integrate the UDF state by making this tree generic over the type of leaf nodes. All leaf manipulations that were done previously are recorded into an interface, see Figure 2.4. We then provide implementations for both the old state, which is a vector of rows (`Vec<Row>`) and the new state that additionally carries a UDF state element `T`[5]. This allows the state to be handled generically by Noria when inserting values and retrieving results. Most importantly when state values are dropped the user state is also dropped and collected guaranteeing smooth integration with the existing memory management structure.

During setup the UDF operator can override the creation of its state, thus injecting the extended partial state. During processing Noria hands the operator a generic state. The operator then safely coerces this generic state to the extended state it injected previously. This gives it access to the UDF state values within which the UDF can then modify during processing.

The same indices used for the state are also used for *sharding*. Noria can distribute computations across multiple cores or even machines by distributing the data using hash-partitioning on the same indices used for storing state. By integrating directly with the preexisting state and reusing the same keys, *sharding* comes for free. In Section 6.3 of the evaluation we show how the clickstream analysis is trivially able to leverage this fact. This kind of effortless parallelism is theoretically available to and UDF that integrates with this extended state mechanism.

[4] Also called dictionary or associative array

[5] The `Memoization` state leaf only stores single rows for the time being, as it is only used for grouping UDF's, which only ever have one result.

```
enum Memoization<T> {
    Empty,
    Store(Row),
    Computing {
        computer: T,
        memoization: Option<Row>,
    }
}
```

```
pub trait Leaf : Default {
    fn push(&mut self, item: Row);
    fn remove(&mut self, row: &[DataType])
              -> Option<Row>;
    fn row_slice(&self) -> &[Row];
    fn singleton(item: Row) -> Self;
}

impl Leaf for Vec<Row> { ... }
impl<T> Leaf for Memoization<T> { ... }
```

Figure 2.4: Leaf interface

3 The Ohua compiler framework

The aim of Ohua [9] is to make parallel programming intuitive. To achieve this, Ohua does not use any special parallel constructs such as threads, locks or channels.

Ohua programs are written in a familiar, high level language which resembles Rust[1]. This high level program is called an *algorithm*. The compiler translates the algorithm into a dataflow graph. A runtime interprets the graph and executes it. Parallelism is derived by scheduling independent nodes of this graph in parallel (task-level-parallelism) or running multiple invocations of dependent nodes in a pipeline-parallel fashion.

The runtime itself is implemented in a *host language*, such as Rust. Algorithms can call into the host language by simply importing and calling functions from the host language. This allows algorithms to abstract away from the host language. An algorithm can run on any host language, provided the same named host language functions the user calls out to are present.

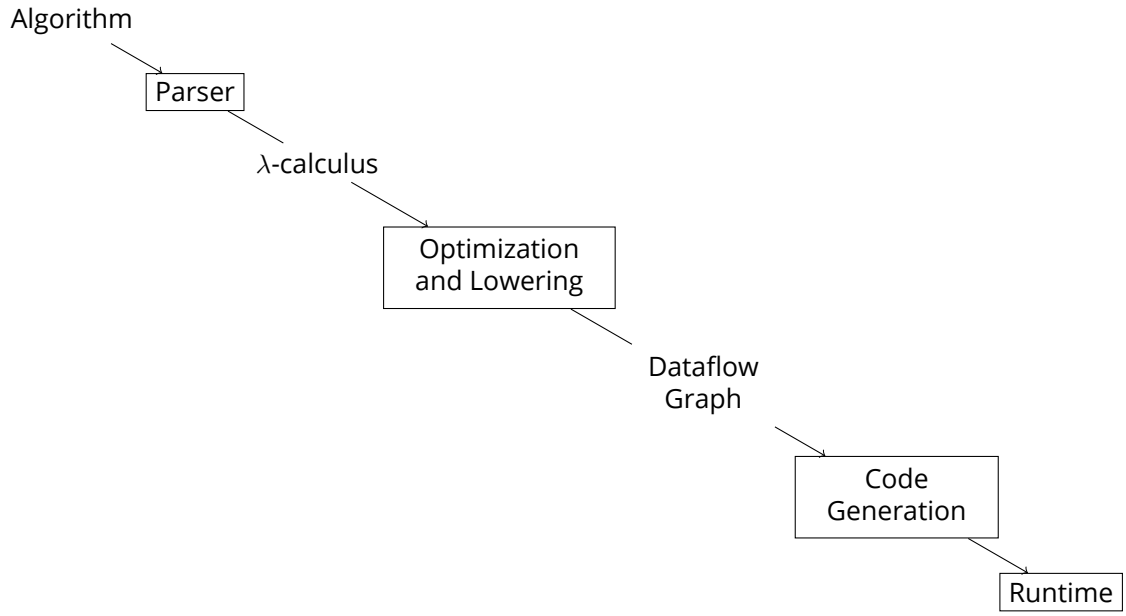


Figure 3.1: Ohua compiler flow

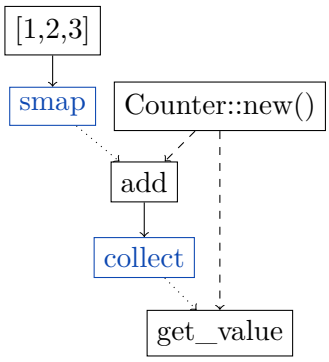
To prevent race conditions Ohua does not allow shared data. Data transfer in algorithms must always be explicitly done using arguments and return values. In the case of the runtime

[1]There are three frontends available: Lisp, Ocaml-like and Rust-like. They are equivalent and compile to the same intermediate language. in this work we chose to set the code examples in the Rust-like frontend for consistency

implemented in Rust this is enforced via the borrow checker. This also prohibits the use of global, mutable values and thus state in algorithms. Ohua then adds a limited form of state, where individual host functions may be associated with state values which it is allowed to modify. The function becomes a method call on the state. Ohua calls these method like functions *stateful functions* [11]. An example of a simple algorithm which uses state, see Figure 3.2.

```
let counter = Counter::new();
for i in [1,2,3] {
    counter.add(i);
};
i.get_value()
```

(a) Algorithm source



(b) Algorithm dataflow graph. Control flow dependencies are dotted, state dependencies dashed edges.

Figure 3.2: Example of an Ohua algorithm

The Ohua compiler creates a dataflow representation of the program, seen in Figure 3.2b. Nodes in this graph are either builtins or host functions the user calls, potentially using state. The runtime executes this graph in a task parallel fashion by running independent nodes in parallel.

In addition Ohua provides a builtin `smap` function which applies an algorithm to a sequence of items, similar to a functional `map`. The Rust-like frontend provides a `for` construct as syntactic sugar for `smap` for a more familiar programming experience. We make liberal use of this construct in the examples in this thesis for better readability. Unlike the functional `map` there is an additional dimension to `smap` in that the state which nodes use within the applied algorithm persists, like it would in a `for` loop. This makes reductions like the one in Figure 3.2 possible.

This allows for another kind of parallelism as several runs of the algorithm that `smap` applies to the sequence items are executed in parallel. If the algorithm does not use state, this leads to data parallelism, otherwise pipeline parallelism [11].

4 Concept and System Design

In order to run imperative programs as queries we need to be able to represent the features of the source language as operators and a query graph. Since we use the Ohua compiler for the UDF's this means building a new backend for Ohua that targets Noria queries. The backend targets a query IR in Noria, called MIR and also generates custom operator. The operator generation is necessary, because it is currently the simplest way to incorporate user state. We chose MIR as target, because it represents a dataflow graph, like the output of Ohua making it easier to generate than SQL. In the broadest sense the features we need to implement are *stateful functions* and *control flow*. These divide further into a handful of smaller tasks.

In Noria operators already have associated state in the form of their materialization. Because this state is already managed by Noria we want to integrate the user state here. This means less integration effort and also has some additional benefits, such as the database automatic reclaiming the memory. This gives us one state per operator to work with, i.e. shared state between operators is not possible. Therefore the first task is to rewrite the program and fuse the parts sharing state into a single operator, in Section 4.1. Then we need to ensure our state can work in a materialized view. For this we make state modifications reversible in Section 4.2.

Then we turn onto *control flow*. The most obvious challenge to solve here is how we represent control flow such as loops and conditionals in a dataflow graph. In case of a query dataflow we cannot simply employ the standard techniques. The reason is that we cannot send arbitrary data between operators. We would also like to reuse as much of the existing relational operators as possible, which will make it easier for the database to optimize the query. Section 4.5 goes into detail on this, describing the constraints in the query flow and how we model control flow here.

Because our language allows for state there is an additional dimension to the representation of control flow. What needs to be considered is how does state work properly if it is not global. A naive implementation will initialize the state when the operator is created, but this means the state is global in the query. If the algorithm the user wrote initialized state somewhere else, in a `for` loop for instance, the algorithm would no longer behave as expected. An example of this can be seen in our clickstream query (Figure 1.2), where a new `IntervalSequence` state is initialized for every user. In Section 4.3 we develop a theory for how we can represent this `scope` of a state at runtime and restore the expected behaviour. This theory also gives rise to a natural way for integrating user state with the materialization, which we do in Section 4.4.

4.1 Shared state representation

Our prototype allows for a restricted form of shared state. We allow shared state *within* a single operator, but we do not allow for shared state *between* operators. Within an operator state

sharing is safe, because the operator executes sequentially. Sharing state between operators would need additional synchronization. This is difficult and often costly which is why Noria deliberately avoids synchronizing between operators.

Use of shared state in our *source language* therefore has to compile to a single operator. We compile one operator per user state. Dependencies between user states are allowed, but have to be acyclic. As an example, take the code from Figure 4.1. In the first snippet the `sum1` state depends on the `sum2` state. This is an acyclic dependency. In the similar second snippet the line `sum2.add(sum1.value())` has been added. This means there is now an additional dependency from `sum2` to `sum1`, forming a cycle. Therefore snippet one would be allows, snippet two not.

```
let sum1 = Sum::new();
let sum2 = Sum::new();

sum1.add(sum2.value());
sum2.add(1);
return (sum1.value(),
        sum2.value());
```

(a) Algorithm with acyclic state dependencies

```
let sum1 = Sum::new();
let sum2 = Sum::new();

sum1.add(sum2.value());
sum2.add(sum1.value());
return (sum1.value(),
        sum2.value());
```

(b) Algorithm with cyclic state dependencies

Figure 4.1: State interleaving

```
struct DoubleSum {
    sum1: Sum,
    sum2: Sum,
}
impl DoubleSum {
    fn new() -> Self {
        DoubleSum {
            sum1: Sum::new(),
            sum2: Sum::new(),
        }
    }
}
```

(a) Definition of the combined state of two sums

```
let ds = DoubleSum::new();

ds.sum1.add(ds.sum2.value());
ds.sum2.add(ds.sum1.value());
return (ds.sum1.value(),
        ds.sum2.value());
```

(b) Combined algorithm

Figure 4.2: Interlaved state through combining

If this kind of interleaving is necessary the user can achieve it by explicitly creating a combined state to work on, as shown in Figure 4.2. In future this could also be done automatically by the compiler.

In practice, even this limited form of state sharing enables some common patterns to be expressed. For instance the Ohua version of the clickstream analysis query (Figure 1.2) uses an imperative version of a map-reduce pattern, where a state is manipulated in a loop, effectively performing part of the reduction as well and after the loop a return value is calculated from the state. This means our limited form of state sharing is at least as general as map-reduce.

4.2 Reversibility

Reversing modifications on a state is a mechanism for coping with the incremental nature of computations Noria. Incremental means we have to *undo* previously computed results. As detailed in Section 2.3.1 this is pretty straight forward to do for single-tuple UDF's, because they do not have state.

To undo the output of a stateless function we simply recalculate the result and issue a delete for it. UDA's, which we also want to support, however use state. Since the output there depends on the state it must be taken into account for handling deletes. The idea is simply to *undo* whatever modifications a value did to a state. Then we can use this state to recalculate the output. In addition the state also needs to allow incremental *insertion* of data and out of order computation, due to how Noria queries execute.

In the following short sections we take a look at how we can guarantee these properties for our state implementation using a *Group*. Then we introduce a more general class of states, which cover a wider application domain. We relate this class back to groups guaranteeing the incremental properties Noria requires.

4.2.1 Groups

Groups inherently posses the properties we need from our state. The binary operation of the group gives us a way to *insert* elements, and associativity lets us handle out-of-order update. Deletes can be handled by simply *inserting* the inverse.

A stateful reduction function based on a group is for instance the `sum` or `product` aggregator.

Groups cover the requirements for our state, but are inflexible for two reasons. 1. Requiring states be groups greatly restricts which states are possible. 2. Because state, input and output value all have the same type and this type is constrained by the system to have an SQL base type, only SQL base types are allowed as state.

4.2.2 Projected group state

By projecting input values into the state group type and projecting the output from the state group we can remove the restriction on SQL types only. If we understand the aggregation itself to be a function $r : [I] \to O$ from a list of inputs $[I]$ to a single output O, then its projection implementation is $a(xs) = g(\text{concat}(\text{map}(f, xs)))$ with $f : I \to G$ being a projection from the input onto the state group G and $g : G \to O$ being a projection from the state group to the output. **map** in this case is a higher-order function which applies its first argument, a function, to all elements of the second argument, a list. **concat** is a reduction of a sequence of values of a group using the binary operation of that group. The two projections functions f and g can be freely chosen by the user, but must be functions in the mathematical sense, i.e. *pure*, where if $f(x) = y$ and $f(x) = g$ then $y = g$, i.e. calling the function multiple times does not change the result.

Deltas can be incorporated easily in this scheme. The reduction processing deltas $r^{\delta} : [i \times (\pm)] \to O$ consumes a list of deltas, i.e. a tuple of a value and a sign. Whereas an individual input value x does not have an inverse, and thus we cannot just combine the sign and the value, every projected value $f(x) : G$ does, because G is a group. We thus incorporate the *insert* delta $(x, +)$ as the projection $f(x)$ and the *delete* delta $(x, -)$ as the canonical inverse of its projection $-f(x)$. Using this form lifts the restriction on the type of state. However due to the group at its heart it retains the same group characteristics.

Projected group can express more complex and interesting reductions. For two quick examples, here is the definition of *average* and *variance* based on this scheme. A *count* reduction falls into the same scheme but is less interesting.

Average

$$G = I \times I \tag{4.1}$$
$$(s_1, c_1) \cdot (s_2, c_2) = (s_1 + s_2, c_1 + c_2) \tag{4.2}$$
$$-(s, c) = (-s, -c) \tag{4.3}$$
$$f(x) = (x, 1) \tag{4.4}$$
$$g((s, c)) = \tfrac{s}{c} \tag{4.5}$$

In the *average* reduction the state is a combined group of two addition groups. The projection function f facilitates that the first field of the tuple records the value itself, and the second field always records 1 i.e. a count.

Variance

The variance of a set of equally likely events is expressed with the following formula.

$$Var(X) = \tfrac{1}{n}((\sum_{i=1}^{n} x_i^2) - \mu^2)$$

This can be expressed as a combined group of three sums that individually track parts of the equation. v tracks the sum of the squares of x, c and s track count and sum of the elements to compute the mean μ as well as providing a count for n.

$$G = I \times I \times I \tag{4.6}$$
$$(v_1, s_1, c_1) \cdot (v_2, s_2, c_2) = (v_1 + v_2, s_1 + s_2, c_1 + c_2) \tag{4.7}$$
$$-(v, s, c) = (-v, -s, -c) \tag{4.8}$$
$$f(x) = (x^2, x, 1) \tag{4.9}$$
$$g((v, s, c)) = \frac{v - (\frac{s}{c})^2}{c} \tag{4.10}$$

4.2.3 Projected free Action groups

We can further reduce the constraints on our state by separating the modifications from the state itself. In the projected group the input projection f produces a state value that is later merged. To relax this constraint we introduce a new set A called an *action group*. The elements of this group describe modifications to some state $s : S$. That means there is some way to apply an element of A to a state s and obtain a new state $\text{apply}(s, a) = s'$. To recover the properties from the groups we must require that there is an empty state and that doing nothing preserves this state, i.e. $s_e = \text{apply}(s_e, a_e)$. Also we must be able to reconstruct every state. If a state s_i exists, then there must be a sequence of actions $a_1, a_2, a_3, \ldots$ such that $s_i = \text{apply}(\ldots, \text{apply}(a_1, \text{apply}(a_2, \text{apply}(a_1, s_e))))$. Because A is a group there must also exist an a_i, which is the concatenation of the sequence such that $s_i = \text{apply}(a_i, s_e)$.

Given these laws it follows that for this case $A = S$ and S is a group, thus preserving the same properties from before. This alone does not afford us any benefits, however we can relax our constraints on A. Instead of requiring a group, we only require a set A' .

To restore the properties we desired we need to lift the set A' back into a group. We can do so by using *free groups*. A *free group* is a structure which adds group capabilities to any set by encoding the group constraints. The idea is inpired by *free monads* [4] which do the same but for the mathematical structure of a monad. Whereas a *free monad* lifts an arbitrary functor to a monad, a *free group* lifts an arbitrary set to a group. Another example of a *free*

structure is `Maybe` or `Option`, which lifts an arbitrary semigroup to a monoid by providing the unit element [23].

The higher order free group structure $G_{free}(S) = [(S \times \pm)]$ is an unordered sequence of elements of S, with a sign attached. The group operations are defined as follows. The sequence is written with angle brackets. it can be empty, i.e. [] and supports concatenation $g_1 \oplus g_2$, the result of which contains all elements from g_1 and g_2 including duplicates in an unspecified order. We also define an operation lift : $S \to G_{free}(S)$ which takes an element of S and lifts it into the group.

$$g_e = [] \tag{4.11}$$

$$g_1 \cdot g_2 = g_1 \oplus g_2 \tag{4.12}$$

$$\text{lift}(s) = [(s, +)] \tag{4.13}$$

$$-g = \left[\left(s_i, \begin{cases} + & \text{if } sign \text{ is} - \\ - & \text{otherwise} \end{cases} \right) \mid (s_i, sign) \in g) \right] \tag{4.14}$$

This allows us to instantiate $A = G_{free}(A')$. From which follow the rules for our state S.

- There is an empty state s_e, which corresponds to doing nothing, i.e. $g_e = []$.
- There is a application function $\text{apply}(s, \pm x) = s'$.
- Every state can be reconstructed from a sequence of actions, i.e. there is an equivalence between sequences of actions and state $s_i \equiv [\pm a_1, \pm a_2, \ldots]$

We can conclude that a state following these rules will behave like a group. We can further generalize our input projections function f from single action values to sequences of actions $f' : I \to [A]$.

This setup allows us to express more complex UDF's. In particular it allows for an efficient encoding of state where the merge semantics are complicated but modification is easy. The clickstream analysis state in Section 6.2.1 is one such example.

The astute reader may have already noticed that the signature of our actions, a sequence of deltas $[A \times \pm]$, is almost same as the signature for the processing routine from Figure 13 in Section 2.3. In this section we set out to conform to the semantics implied by this signature and we ended up with the signature itself again. Why did we do this detour? By using the equivalence of the free group with action and the corresponding state we can show that our state behaves as though it were a group. Taking the detour afforded us group properties, even for structures which may not themselves be merged or reversed easily.

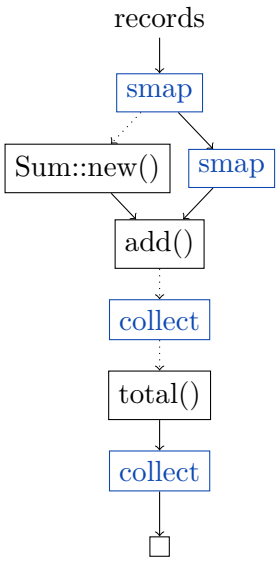
4.3 Scope

In high level programming languages the *scope* of a binding, or variable, defines the part of the program in which the binding is defined and accessible. For instance, function arguments in scope for the body of the function, other variables are valid after their declaration until the end of the immediately surrounding block.

Consider a simple example where of a stateful aggregation in a loop in Figure 4.3a. Due to the scope of `sum` the expected semantic of this program would be to sum the total purchases of each customer *individually*.

```
for (customer, purchases) in records {
    let sum = Sum::new();
    for purchase in purchases {
        sum.add(purchase.total);
    };
    sum.total()
}
```

(a) Simple stateful sum



(b) Dataflow graph for the stateful sum

In a flat dataflow scope does not exist. A naive translation of this program might compile the inner loop to a single aggregation operator in order to avoid state sharing, see also Section 4.1, and instantiate the operator with a single, fresh `Sum::new()`. However in this case a single state value will not produce the desired result and instead sum over *all* purchases, not *per user*. Multiple states are required, one for each `customer` in order to track the total individually. In a traditional query engine we could do this simply by ordering the purchases by customer and reinstantiating the state every time the customer changes. However in Noria records can always arrive out-of-order and late updates to the base table can require us to alter the computation result.

What we require is a mechanism for capturing the information about *which* state value a given row/value belongs to, or to put it the other way around, for a given row, which state value is *in scope*.

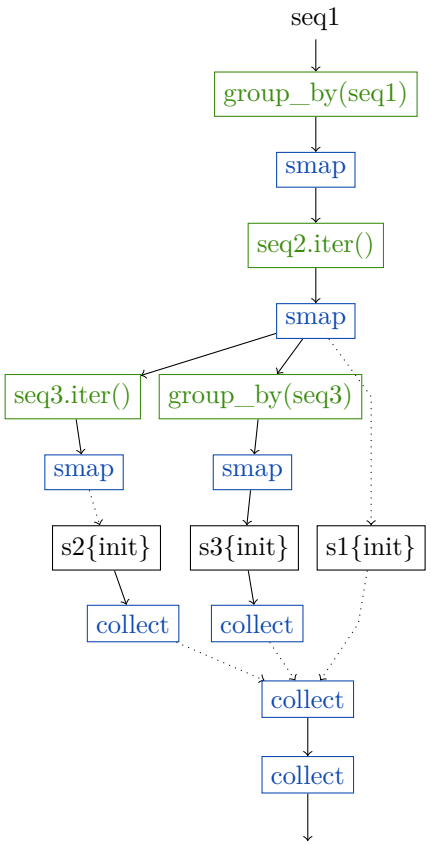
The only language constructs capable of influencing *scope*, i.e. which values a given state is valid for, are control flow constructs. Depending on these control flow constructs a state might be instantiated with or modified with different values.

Control flow in dataflow is implemented by the ratio of input to output tuples of an operator. As an example the `smap` operator in Figure 4.3b gates the *begin* of an iteration control flow context. Its input-output ratio is $1 : n$ with n being the number of items in the input sequence. Conversely the `collect` operator gates the *end* of an iteration context with an input-output ratio of $n : 1$, with n being the number of item in the sequence. Similarly a `Filter` node in Noria works similar to an `if` in that subsequent processing steps are conditional on certain properties of the tuple, our theoretical implementation of conditionals exploits this very property (Section 4.5.2).

In the Ohua source language, control flow is structural i.e. syntactic constructs, such as `for` and `if`, denote control flow. In the Ohua intermediate representation these are represented as λ-abstractions which are input to higher order functions. Through the transformations our compiler performs the scope in and around these constructs is carefully maintained. Control flow lowers exclusively to builtin operators known to the compiler and follows certain rules. For instance edges influenced by control flow do not *escape*, that is all paths leading through the entry node of a control flow structure eventually lead through the exit node of the same control flow structure. We derive the scope as an ambient property from the Ohua dataflow graph. Scope can be calculated for any node, not just state carrying ones, which we will make use of later to represent Ohua calling conventions in the Noria dataflow (Section 5.2.3).

In the source language control flow can be nested. The scope of a node can thus be influenced by more than one control flow construct or none. Therefore we capture *scope* as a *stack* of

```
for seq2 in group_by(0, seq1) {
    for seq3 in seq2.iter() {
        let s1 = State::new();
        for _ in seq3.iter() {
            let s2 = State::new();
        }
        for _ in group_by(1, seq3) {
            let s3 = State::new();
        }
    }
}
```



Scope Stack	Scope Key Example
`group_by(0, seq1)`	g_1
`iter(seq2)`	1

(a) Scope stack and key example for `s1`

Scope Stack	Scope Key Example
`group_by(0, seq1)`	g_2
`iter(seq2)`	3
`iter(seq3)`	4

(b) Scope stack and key example for `s2`

Scope Stack	Scope Key Example
`group_by(0, seq1)`	g_2
`iter(seq2)`	3
`group_by(seq3)`	g_2

(c) Scope stack and key example for `s3`

Figure 4.4: Scope Examples

individual control flow contexts that influence it. These scopes can be ordered, where the scope of a more deeply nested node is larger than the scope of a less deeply nested node. This order is partial, because scopes of nodes in disjoined control flow contexts are not comparable. If two scopes are comparable, the smaller is always contained in its entirety in the larger one. To illustrate this the code of Figure 4.4 contains three different state initializations and the resulting scope stacks. The scope of `s1` and `s2` are comparable, with the scope of `s2` being "larger". Similarly the scope for `s1` is smaller than `s3`. Scopes for `s2` and `s3` however cannot be compared because they differ in the third entry in the stack.

To calculate the *scope* of a node we take as a *derived scope* the maximum of the scopes of its ancestor nodes. Later, in Section 5.2.3, the graph will be rewritten such that this property always holds. Control flow nodes, such as the aforementioned `smap` and `collect` further change the scope. In the case of a *scope introducing* operator, such as `smap` a new control flow context for *iteration* is pushed on top of the *derived scope* stack. In case of a *scope reducing* operator, such as `collect` the topmost control flow context is popped of the stack. For `collect` that should be an *iteration* context. A visual interpretation of this would be if we want to know the scope of a node given the dataflow graph we trace the path up the graph which goes through the most scope influencing operators.

At runtime we need a *scope key*, which is a compound (SQL) value that characterizes the particular instantiation of a scope. Each data tuple must carry its relevant scope key, which we can use to select the state we need to process this tuple. We obtain the full scope key by composing individual control flow context keys, or to put it another way the compound scope key contains one entry for every entry in the scope stack.

For the purposes of this prototype we only consider iteration control flow constructs. Here the actual value for the scope entry must be provided by the creator of the sequence. The reason for this is that a sequence cannot be encoded as a single row, the fundamental data unit in Noria, and therefore must be represented virtually using tagging.

In the case of `group_by` this is done simply through selection of one or more existing columns, in the case of general iteration the function producing the sequence must add also provide an index on each created row. General iteration is not implemented, but could be added in the form of SRF's (see Section 7.2). Code generation for the SRF would prepend these indices automatically then. We also discuss the possibility of extending to other kinds of control flow in Section 7.3.2.

The scope key is carried along with the data flowing between operators in *Row*s. Each emitted *Row* has an $n+m$ schema with the first n items for the *scope keys*, followed by m items payload. The *scope key* is a concatenation of the keys of the individual contexts in the scope. As currently only iteration is implemented the keys are indices of the sequence being iterated. The compound *scope key* is used as an index for the state values. This enables the state carrying operator to handle several instances of the same state, but in different scopes. This capability is crucial, because at compile time the amount of parallel instances of the same state is not known. In the evaluation section (6) we show how scopes can be exploited at runtime to duplicate nodes and derive data parallelism via sharding.

4.4 State integration

Operator state in Noria is always bound to an index. Downstream operators can retrieve past results with keys into this index and the system can reduce memory pressure by deleting entries from the index. From the earlier scope analysis we obtained an ambient *scope key* for each state in the program, with the potential to partition the state along this key. This *scope key* turns out to logically correspond to the index key used in Noria states, as these fulfill the same function.

By using the *scope key* to index the user state we solve several representation problems, as well as obtain desirable efficiency properties.

- Keeping several, scope indexed states allows us to emulate the hierarchical, scoped execution semantics of the Ohua source language. During processing states are selected according to the scope key, meaning each particular state value is modified only by the values that would be present for the duration of its scope in a sequential execution.

- State partitioning enables easy sharding. As mentioned before partitioning the state opens up the possibility for parallel processing, as states in different scopes are guaranteed not to depend on one another.
  By using the existing indices in Noria we have already enabled sharding for our operators, without additional effort. See our experiment in Section 6.3.

- Binding state to the preexisting state structure enables the system to reason about this memory. Noria tries to keep memory consumption low by reclaiming parts of an operators state, if it is unused. By fusing user created operator state to the preexisting state system Noria is made aware of the additional memory consumption of these operators. More importantly when Noria reclaims parts of an operators state it also reclaims the users custom state, which would otherwise leak.

4.5 Control Flow

In this Section we will take a look at how the control flow in our source program can be represented in a query. The Ohua language permits three fundamental control flow constructs.

smap, a loop or more precisely a mapping operation which applies a function to every element of a sequence. `if` is a classic if-then-else construct which executes one of two branches depending on a condition and lastly tail recursion.

4.5.1 Smap and `for` loops

smap is somewhere between a loop and a pure `map` operation. Whereas only one element of the sequence is accessible at a time, such as is the case in `map`, the operator state persists across iterations, as it does in `for` loops. This allows operations such as summation to be expressed. smap is represented internally as a primitive higher order function (Figure 4.5b) and syntactic sugar, in the form of a `for` loop, is provided for more intuitive development (Figure 4.5a).

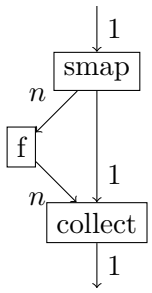
The typical runtime representation is done by two operators, `smap` and `collect`. Figure 4.5c shows the runtime graph for a simple example including the multiplicities of items in the arcs between the operators. The entry operator `smap` recieves as input a collection and outputs each element individually and, once, the total count of items. The complementary exit operator `collect` recieves the count of items from `smap` and consumes as many input items as the count specified before emitting them as a single, reassembled collection.

```
let ys = for x in xs {
  f(x)
};
...
```

(a) In the (Rust-like) source language

```
let ys =
      ohua.lang.smap
      (λx → f x)
      xs
in ...
```

(b) In the intermediate representation



(c) Runtime graph representation

Figure 4.5: Smap representation

The difference in the Noria is that items traveling along the edges are restricted to elementary SQL data items, prohibiting collections such as lists and arrays to be sent between operators. To overcome this limitation we eliminate all `smap` and `collect` operators in one of two ways.

1. For simplicity we assume all UDF's built with Ohua operate on a sequence of tuples. This means each UDF must at least one innermost loop. This innermost loop is always fused into the generated operators. This makes them operate on sequences of tuples, which Norias operator interface expects.

2. Through emulation via *scope*. The *scope* mechanism lets each operator handle the iteration individually. Operators that use state emulate scope, stateless operator simply ignore it.

4.5.2 If and conditional execution

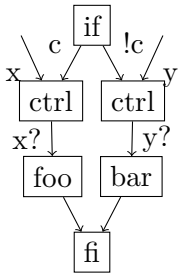
The second control flow primitive available in Ohua is `if`. The runtime typically represents this using an operator called `ohua::lang::ctrl` which guards execution of a branch by conditionally releasing its free variables (see Figure 4.6b).

In Noria the *ctrl* operator can be replaced by a `Filter` node and a join with the free variables, as depicted in Figure 4.6c.

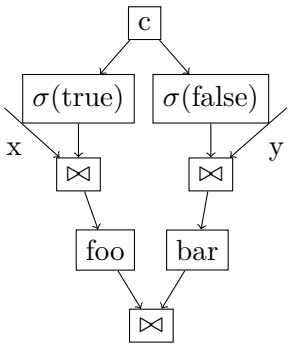
Unlike `smap`, `if` does not need a *scope key* (see also Section 23). Scope keys are necessary if an operator, and thus its state, are run in an environment which changes due to scope, as is the case with `smap` where the current iteration value changes. In the case of a conditional

```
let x = "hello";
let y = "there";
if c {
    foo(x)
} else {
    bar(y)
}
```

(a) Source code



(b) Dataflow representation in Ohua



(c) Possible representation in Noria

Figure 4.6: Choice in Ohua and Noria

one might suspect the state of the condition, whether it evaluated to true or false, would be a changing environment. However any single operator, and thus its state, only ever occurs on *one* of the branches, not both. There could be state sharing across branches, however in this case we always compile the whole expression as a single operator (see also Section 19), consequently handing over responsibility to the host language. A quick overview is provided in Figure 4.7. Figure 4.7a and 4.7b are essentially the same. The scope semantics are guaranteed because the state is only ever active on one branch. In the cases of both Figure 4.7c and 4.7d the state is shared, causing the entire expression to become a single, fused operator, and thus does not require scope.

Necessitating all such expressions to become operators limits the expressiveness of the language. Eventually we would like to support some form of actual state sharing to remove this limitation. Using scope to distinguish the conditions under which a state is executed may provide a way mimic shared state while keeping the actual management confined to a single operator.

```
let state = State::new();
if condition {
    state.method(args..);
}
```

(a) State on single branch

```
if condition {
    State::new().method(args..);
}
```

(b) Inline state on branch

```
let state = State::new();
if condition {
    state.method(args..);
} else {
    state.method(args..);
}
```

(c) State sharing across branches

```
let state = State::new();
if condition {
    state.method(args..);
};
state.method(args..);
```

(d) State sharing after conditional block

Figure 4.7: Overview of operators in conditionals

Due to time constraints we have not implemented or tested conditionals in this prototype.

4.5.3 Recursion

Ohua supports tail recursion but our prototype does not support it. At this time it is unclear how tail recursion could be done in a system such as Noria and perhaps more importantly, whether it *should* be done. Section 7.3.2 features a short discussion on this topic.

5 Compiling Algorithms to Queries

UDF integration is usually relatively straightforward. Depending on the type of UDF (see Table 2.1) a certain interface must be implemented. At runtime the database loads the code for the UDF and inserts an operator which implements the necessary internal boilerplate around the actual processing function. The operator performs necessary conversions and eventually calls out to the linked processing function. Because Noria does not yet have UDF support we must first define the wrapping operators. We constrain ourselves to simple UDF's and UDA's, the rational for this is given in Section 2.3.

However our task does not end there. We desire integration not only at the operator, but at the query level and we want both of these to be as automated as possible. As such, given an algorithm, we must make a judgment on which parts of the query we want to generate an operator for, the rest must be representable as a Noria query. To address these two concerns we hook into the Ohua compiler in two places. Figure 5.1 shows the stages of the compiler with the two additional phases added. The Operator Compilation phase hooks into the compilers middle end with a transformation on the intermediate representation ALang. The pass identifies subexpressions to compile as operators, mostly concerned with properly dealing with shared state, generates rust representations for these operators and fits appropriate boilerplate to satisfy the Noria operator interfaces. Section 5.1 describes this pass in more detail.

The Code Generation is concerned with transforming Ohua generated dataflow graphs of the program into a query which Noria is able to run and optimize. Code Generation targets the Noria intermediate representation MIR, which encodes a dataflow graph specialized for databases. Section 5.2 explains the differences between Ohua dataflow graphs and MIR, the conversion, as well as how the generated graphs are linked into the database.

5.1 Fusing Query Parts and Compiling Operators

The task of this compilation step is to identify the parts of the algorithm code written by the user that should become a Noria dataflow operator (also called `Ingredient`) in the final query and generate an `Ingredient` implementation for this function.

We chose to implement this as a pass over Ohua's central intermediate representation ALang. ALang is based on the call-by-need lambda calculus. Figure 5.2 shows possible ALang expressions. The first four are standard call-by-need lambda calculus, the last one is specific to Ohua. *State binding* is similar to method call in that it associates a function `f` with a previously `let` bound value `s`. Currently `f` needs to be a function defined in the host language and imported into the algorithm. When the bound function is called it gets passed the state value `s` as an additional argument which it may modify.

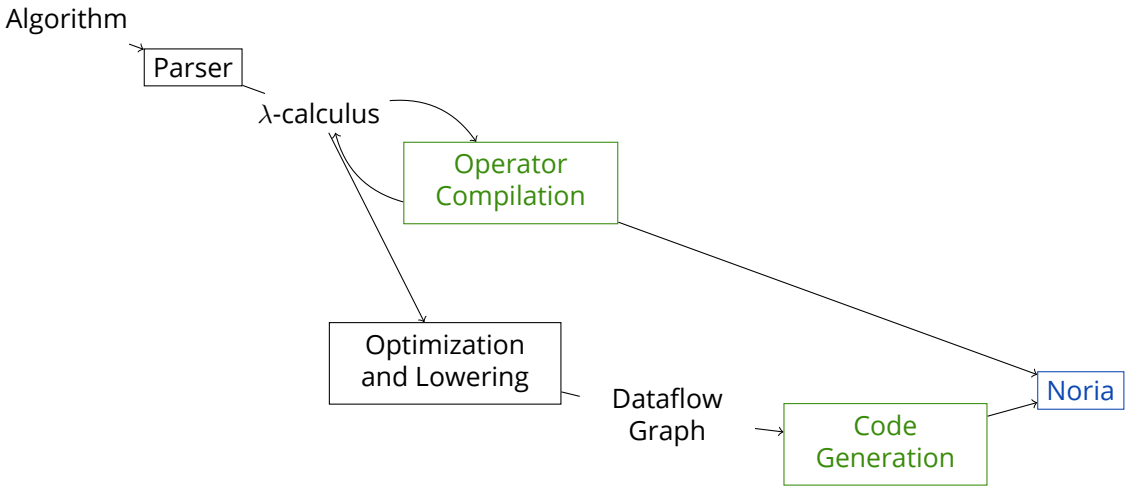


Figure 5.1: Ohua compiler flow for Noria UDF's

Named bindings	`v`
Lambda abstraction	`(λ x . E)`
Function application	`f x`
Let binding	`let x = N in M`
State binding	`s#f`

Figure 5.2: ALang as an extension to call-by-need lambda calculus

As we show in Section 5.1.3 ALang is quite straightforward to represent in Rust. This is why we chose to implement operator compilation at the level of ALang, and not on the level of dataflow graphs. In addition the compilation relies on term-level rewrites (Section 5.1.1) which are well defined on lambda calculus. We have used a similar strategy successfully in this 2018 paper [10].

The Ohua compiler allows for additional passes to be hooked into the compiler. Thus we implement the operator compilation by a pass over ALang which identifies code to be expressed as an operator, generate the operator and replace the original code in the algorithm by a call to the generated operator. The compilation can be broken down into the following four steps.

5.1.1 Identification of algorithm expressions that should become operators

There are two cases to consider. Firstly we compile all direct calls to external functions into their own operators. In this case the fragment we must consider is only the function itself and the input values.

The second case is more complicated. Our goal here is to simplify state synchronization. When we consider interactions involving state we must ensure they happen in the correct sequence so as to not break the semantics of the program the user wrote. For the purposes of this prototype we only consider a subset of possible algorithms, where the state is used in a small, local scope. We consider possible solutions to states with larger scope in Section 7.2.

For locally scoped state we chose to fuse parts of the query into a single operator. By synthesizing Rust code for a sequential version of this fragment we can ensure that it is executed with the correct sequential semantics, by deferring to Rust's sequential semantics.

A naive version which lends itself nicely to the semantics of call-by-need lambda calculus is a scope driven extraction. We may consider the scope of a state value. A state value `s` bound like `let s = N in M` is only valid in `M`. Therefore we would cover all possible uses of `s` by compiling `M` in its entirety as a single operator. However this is too coarse even for a simple example

such as in Figure 6.1. In this example the state scopes overlap after inlining the `count` and `sum` algorithms which would mean both are compiled into one operator. This poses two problems. Firstly we would need a concept of how to represent multiple states in a single operator and more importantly we loose parallelism opportunities.

We therefore employ a more sophisticated algorithm. Instead of considering only scope we traverse the scope `M`, accumulating a minimal expression which contains all interactions with `s` and computations of local dependencies. Local dependencies are all non-free values used in `M`. The earlier Ohua rewrite passes will ensure that computations in `M` all depend on `s` by floating out computations that do not.

This is predicated on the following assumptions.

- State dependencies are acyclic.

  Given two states s_1 and s_2 if a state bound call on s_2 depends on the output of a state bound call with s_1 i.e. `let x =` s_1`#f y in` s_2`#g x` the reverse must not be true, i.e. no state bound call on s_1 can depend on the output of a state bound call on s_2.

  The intuition here is that the complete expression modifying the state s_2 and its local dependencies become an operator. This operator can depend on an earlier operator modifying s_1, but this operator dependency cannot be cyclic.

- Local dependencies do not contain query operators.

  We express query operations, such as `group_by` at the level of query graphs, not the level of operators. This is not a fundamental limitation and may be added in the future, though it warrants a discussion as to whether there are any benefits to allowing this.

5.1.2 Selection of strategies for incorporating reversibility

The selection of a reversibility strategy depends also on the execution semantic of the operators and also on the implicit dependencies introduced by the state.

Generally we must lift each computation on values $f : A \to B$ to a function on deltas $f' : \pm A \to \pm B$. Similarly for other functions which consume or produce multiple values, for instance $g : [A] \to B$ must become $g' : [\pm A] \to \pm B$.

Reversibility is not trivial to implement and we would prefer to leave as much of the heavy lifting as possible to the compiler. In this work we will consider two cases in which we can implement reversibility for an entire generated operator automatically given certain preconditions. In Section 7.2.1 we further discuss possible future possibilities to expand to other, possibly more general cases.

The two cases we implemented are **Pure, Singe Tuple Function** and **Stateful Aggregations**. Section 4.2 describes the two types in more detail and shows why more general functions are difficult to implement. Section 7.2.1 discusses how we might add support for other types of functions in the future. In accordance with the processing semantics described in Section 4.2 our aggregations function follow a map-reduce like pattern facilitated with state. In this pattern the *map* part of the computation modifies the state not directly but produces `Action` values. `Action` values are defined by the user as part of the interface for the aggregation state. These values describe the modification on the state the particular input tuple *would* perform. Then for each `Action` value a handler function is invoked on the state. It consumes the `Action`, as well as the *sign* of the tuple the `Action` value originated from and modifies the state accordingly. After that a single output value is computed from the state, with no additional arguments. The core of the *Clickstream* and the SQL functions `count` and `sum` can be naturally implemented using this pattern.

Via this pattern we derive an automatic strategy for handling deltas. The observation is that the output value of the computation depends only on the state of the state after the input

values have been handled and the modifications were incorporated. Ergo the sign needs only affect the *modifications* of the state, not the computation of the result value. In fact this would also work if there were multiple states at play.

Users do not have to consider `Action` values when implementing the algorithm. Method calls on the state are rewritten to the construction of `Action` values. These values contain the same arguments provided to the method call. An example is provided in Figure 5.3. Each constructor of this type encodes an action that can be applied to the state including all arguments. This is possible due to the duality of data and code and exploits the fact that we know that these modifications all return `()`. It must also provide the handler function `handle`, which consumes actions and modifies the state accordingly. The third argument is used to communicate the sign of the delta (`true` → apply, `false` → reverse).

```
// The state type
struct IntervalSequence = { /* omitted */ }

// Permissible interactions
enum Action = {
    OpenInterval(i32),
    InsertItem(i32),
    CloseInterval(i32),
}

impl IntervalSequence {
    fn handle(&mut self, action: Action, positive: bool) {
        /* omitted */
    }
}
```

Figure 5.3: Interface for a reversible map-reduce state

Input deltas are thus completely handled by the state. This means users do not need to write an explicitly reversible algorithm. As a caveat additional effort is required, to implement a reversible state and we discuss possibilities for making this process easier in Section 7.2. However the state is not tied to a particular algorithm and can be reused.

5.1.3 Synthesis of Rust code

This last step is relatively simple. The Ohua intermediate language is an untyped call-by-need lambda calculus. All lambda calculus terms have a straight forward Rust representation illustrated in Table 5.4. The table includes three Ohua specific conversions, *State interaction*, *Conditional* and *Iteration*, which are translations of Ohua primitives.

Construct name	Ohua IR	Rust
Named bindings	`v`	`v`
Function application	`f x y z`	`f(x,y,z)`
Lambda abstraction	`(λ x . body)`	`| x | { body }`
Let Binding	`let x = e in expr`	`let x = e; expr`
State interaction	`s#f x y`	`s.f(x,y)`
Conditional	`<ITE> cond e1 e2`	`if cond { e1 } else { e2 }`
Iteration	`smap (λ x . body) xs`	`fox x in xs { body }`

Figure 5.4: Table of conversions from lambda calculus to Rust

The Ohua IR is untyped and we rely on the type inference from Rust to ensure type correctness

in the generated code. One problem which arises here is that we need to know the desired input types to correctly coerce the operator inputs. The database handles the data flowing between operators as the generic `Row` type `Vec<DataType>` which represents a sequence of primitive SQL data. Operators store which indices in these operators they care about, which are resolved before it starts running from the query schema. Our generated operators use the same index resolution schema already present in Noria. After indexing the input row we are left with one or more `DataType` values which represent arbitrary SQL data, see also Figure 5.5b. These are coerced to the input type the user provided function expects with the `Into::into` function. This can fail if the type inference fails to accurately determine the target type of the conversion. To prevent this we propagate the top level type annotations from the algorithm to guide the type inference. An example would be the annotation for `main` in Figure 1.2. Here we can determine that fields 0, 1 and 2 are of type `UID`, `Category` and `Timestamp` respectively. When these fields are accessed in `op_core` for instance we insert a type annotation in the generated operator so the `DataType` values are converted appropriately.

An issue that we ran into here is with the distinction between references and values. Ohua does not distinguish between these two, but the rust code must do so. For this prototype we decided to instead include a primitive for dereferencing. This may cause confusing errors, since values suddenly need to be dereferences *in Ohua code*, which was not necessary previously, and is not necessary in the surrounding code. It is a subtile inconsistency that we would like to get rid of in the future. Similarly we had to bake in primitives for division and equality, because these are operators in rust, but Ohua does not have operators. For a complete list refer to Table 5.5a. What this shows as well is that in order to properly generate operator code for a certain target, we may need to provide extensible facilities for specifying primitive conversions.

Ohua function	Rust operator
`primitives::deref(r)`	`r`
`primitives::eq(a, b)`	`a == b`
`primitives::div(d, n)`	`d / n`

(a) Table of baked-in primitives

```rust
pub enum DataType {
    None,
    Int(i32),
    BigInt(i64),
    Real(i64, i32),
    Text(ArcCStr),
    TinyText([u8; TINYTEXT_WIDTH]),
    Timestamp(NaiveDateTime),
}
```

(b) Arbitrary SQL data representation

5.1.4 Fitting a shell

For the code generation part we choose to generate textual Rust code which is injected into the source for the database and linked statically into the intermediate language lowering mechanism in Noria. We opted for the static version, as there is no existing facility to dynamically load code in Noria as of yet. In addition this enables the Rust type and borrow checker to verify some correctness properties for the generated code and its integration into the system. This is an obvious inconvenience when developing queries with Ohua, but does not detract from the generality of the approach and can be retrofitted with minimal performance impact. For most Ohua backends all generated operators have the same execution semantics. They consume one set of inputs at a time and produce one set/tuple of outputs at a time, potentially altering the state in the process. I.e. every operator has a $1 : 1$ execution semantic. Operators generated for Noria however can have different execution semantics. In particular the grouping operators have an $n : 1$ execution semantic. It is the task of the operator code generation to facilitate this execution semantic.

In the deployed query every operator has an $n : m$ execution semantic, with additional options

to influence the execution of the whole query by, for instance, requesting replays, or additional upstream data. Since the capabilities of our prototype are limited we do not end up needing any of the query influencing, such as replays. The task then is to wrap each identified operator core, such that its surface-level execution is $n : m$.

We use an approach similar to the one used by Chen, Hsu, and Liu in [7]. They separate the application logic, which they call *user-function*, and the system utilities, which they call *RVF shell*. Our approach is similar, in that our *user-function* comes from the operator core we synthesized and our *shell* is selected from a library of templates to fit the desired operator execution semantic. Our prototype is equipped with two templates, one for pure functions, embedding $1 : 1$ in $n : m$, and one for grouping operations, embedding $n : 1$ in $n : m$. This method leaves room for future extension and the embedding of, for instance $1 : n$ semantics, however this may also require additional theory around handling of state and triggering of replays and such, which we discuss briefly in Section 7.2.1.

Pure functions use the `Identity` operator as basis but additionally modify one of the fields, and also facilitate multi-input functions.

Grouping functions , like the state driven map-reduce pattern, use the cored implementation for aggregation functions, such as `sum`, as a basis. The map and reduce phase are spatially separated here. The *map* phase actually has no access to the state, whereas the *reduce* phase does. We are able to satisfy this in our code generation as well by using values to represent modifications, as described in Section 5.1.2. The code for the generated *map* phase emits a sequence of modifications, which are recorded, together with the sign of the respective processed record. In the reduce phase the state is retrieved with the group key, the modifications are applied and the result value is computed.

For each operator Noria stores the output in a map indexed by the group key. This map is used by child operators to try to retrieve already computed values. It is also used by the Noria system to do garbage collection by evicting unused entries from the map to keep the memory overhead low. Grouping operators, such as `sum` retrieve their state values associated with the group key they are currently processing from the map and combine it with the new updates. We piggyback our operator state on the same mechanism by extending the map values. Normally the values of the map are the *Row*s emitted by the operator. We attach an additional, optional field which contains the state value. This is possible because like the output *Row*s the operator state is scoped to the group key, see also Section 4.3. We install a coercion function in Norias `State` interface that allows us to coerce from the generic `State` to the patched version containing operator state, in a type safe way.

The integration with the existing state structure means we also integrate with the partial state mechanism described in Section 2.2. Specifically when Noria reclaims the memory for a specific state key it also deallocates the operator state used to compute that group. The operator is itself responsible for initializing a new computing state if none is present. This covers both the initialization of the states when the computation is first run as well as reinitialization upon recomputation of a previously evicted key.

Another function our shell has to provide is to specify a return type. This is not necessary for internal operations, because values in the runtime dataflow graph travel as a generic union type of SQL values. However if the user tries to read the value the type must be known to facilitate correct conversion. Currently we query the state for a description of the type. This solution was easy to implement, but it does not allow reuse of states if the computing function returns a different type. In future we would like to instead use the type annotations in the Ohua source language to communicate the desired return type.

We also added multi argument support. The builtin SQL functions on Noria operator on a single field, whereas our UDF operators may have multiple columns as inputs. See also

Section 5.2.3 where we transform queries such that multi argument functions operate on multiple columns, but with only a single ancestor node.

5.2 Query Compilation

After the operators have been generated the remaining program is compiled as normal by the Ohua compiler. This creates a dataflow representation of the program suitable for execution by an Ohua runtime.

Query compilation runs as a code generation pass of the `ohuac` standalone compiler. The objective of query compilation is to transform the output of Ohua into a MIR graph, a high-level intermediate representation in Noria. To understand how this is achieved we will first briefly introduce the graph representation in Noria and then outline how the conversion is achieved.

5.2.1 Ohua dataflow graphs

The output of Ohua is a directed dataflow graph of the program. Vertices and edges are recorded separately and connected via numeric identifiers to make the graph serializable. Vertices in the graph are named references to operators or functions to be executed and edges represent data dependencies between the nodes. Each operator may have multiple inputs and every edge is annotated with an index for the argument position it corresponds to, see also figure 3.2b. An operator may have multiple descendants, but they will all receive the same value which is duplicated if necessary.

The graph does not contain multiplicities for the operators. User implemented functions are assumed to have a $1:1$ input/output ratio. Builtin operator that are used to implement control flow can have different multiplicities. These are all known to the compiler and runtime, and hence require no annotation. Table 5.6 shows the builtin operators and multiplicities in Ohua. These operators form most of the API of an Ohua runtime.

Identifier	Multiplicity	Description
`(,)(args...)`	$1:1$	Tuple creation
`nth(idx,len,tup)`	$1:1$	Tuple indexing
`smap(seq)`	$1:n$	Iteration entry
`collect(n, item)`	$n:1$	Iteration exit
`bool(cond)`	$1:1$	`if` helper, outputs `(cond, !cond)`
`select(cond, on_true, on_false)`	$\{1,0\}:1$	`if` branch merge
`ctrl(n, val)`	$1:n$	`val` replicator
`recurse(finished, args...)`	$1:n$	Recursion helper

Figure 5.6: Ohua builtin operators

5.2.2 MIR

MIR is a the high-level intermediate representation in Noria. MIR is represented as a doubly linked, directed, acyclic dataflow graph. An SQL query is lowered into MIR after it has been parsed. During this process the various parts of the SQL query are rewritten into parameterized descriptors for runtime operators. Data dependencies between the individual computations are resolved and made explicit via links between the nodes. The system orders computations to try to reduce the number of computations necessary, for instance by pushing filters above joins where possible. Multiplicities of input to output are not recorded.

Nodes of the MIR graph also carry a description of the schema of their output rows in the form of *Column* descriptors. Child nodes use these descriptors to resolve *Column*s mentioned in the query to indices into the *Row*s received from the parent node.

When a query is being registered with the system Noria creates a MIR representation of. The MIR query is matched against the already running graph to find live nodes which can be reused. Then the new graph is merged into the existing one and made accessible to readers.

5.2.3 Rewrites

Ohua dataflow graphs and MIR are structurally similar. In both cases a network of computing nodes is connected by runtime data dependencies. Neither contains multiplicities for input and output.

The relevant differences are in how these graphs are supposed to be interpreted. In Ohua connections behave like buffered channels. A node can itself decide when it wants to pull data from which channel and may suspend its own execution until data is available on a channel it wants to pull from. The most important practical way this is used is in all nodes which implement external functions imported by the user. Such a node pulls from every input once and then calls the function with its complete set of arguments.

In Noria the system pushes data into the computations. A node is expected to perform its computation whenever the system has data available. For nodes with multiple ancestors this means either it has to buffer input until matching data from all ancestors is available, or optimistically emit results, recomputing as data from other ancestors becomes available. Section 5.2.3 addresses this problem by rewriting all multi-ancestor connections.

As a further complication the assumption that non-builtin operators have $1 : 1$ multiplicity no longer holds. Operators that were generated to represent reductions have $n : 1$ multiplicity instead. To fix this we communicate which operators were generated with which multiplicity from the operator generation pass to the code generation so that rewrites can query the multiplicity of an operator.

Relating Tuples to Columns

The only data type Ohua explicitly defines are tuples. These allow functions and algorithms to return more than one output value. Tuples are created with the `ohua::lang::(,)` operator, which is a desugaring of tuple literal syntax in the source language. Tuples in Ohua can be indexed with pattern matching, which is internally represented with the `ohua::lang::nth` operator. Often runtimes chose to fuse tuple creation and indexing with computing operators for efficiency reasons.

In Noria we are presented with an existing indexing scheme in the form of *Row*s and *Column*s. By mapping *Row*s to tuples and *Column*s to tuple fields we obtain a natural bidirectional mapping between Ohua tuples and *Row*s. This allows us to represent the tuples used internally in the Ohua program, which would not be possible using a single *Column*, as *Column*s can only contain primitive SQL data types.

The mapping itself is done by generating new *Column* descriptors from each operator id and its output indices. Then the normal column index resolution mechanism from Noria resolves these descriptors to numeric *Row* indices when setting up the graph for execution.

Using this schema we can abstract queries of the concrete fields they operate on. Instead of hardcoding field names, Ohua UDF's operate on rows as though they are tuples. An Ohua UDF is parameterized over a `RowStream`, an analogue to a table, which is parameterized with the number and types of fields these rows contain. When the UDF is called in a query the concrete field names are provided as arguments. `Project` operators are inserted before the UDF to rename the fields accordingly and link the tuple indexed UDF.

By these means UDF's become reusable through instantiation with different fields which may have completely different tables or relations backing them.

Joining

This rewrite is used to achieve the execution semantics expected for an Ohua dataflow graph. When an Ohua graph is executed all operators, except certain builtin ones like `smap`, pull from each of their inputs once, run and then produce one output value. Runtimes can often implement this simply by using buffered channels. This is not the case in Noria. At runtime dependencies between operators do not use channels. Instead an `Executor` selects which node to run and passes it an array of output rows from *one* of its ancestors. The node itself has to coordinate how it lines up inputs from multiple ancestors.

Implementing this from scratch would be a lot of work and also mean a significant amount of machinery present in each user operator. Fortunately there is already a database operation with the needed semantics. The `JOIN` operation does exactly this, if the right join key is chosen. We can easily find the right join key by looking at the *Scope* (see Section 4.3) of the two ancestors. If the two ancestors have the same scope, we join on the scope directly. If the two ancestors have different scope, one of them must be in a more deeply nested control flow context. An example of this can be seen in Figure 5.7. The function `h` has three ancestors, `f`, `g` and `q`. There the ancestor scopes differ, however in such situations the scopes will always be *comparable*, meaning they share the smallest of them as a base scope. This is true for any pair of those as well. The reason for this is that the graph is created from a lambda calculus *expression*. A `let x = E in M` expression explicitly limits the scope of the binding `x` to the nested body expression `M`. Therefore the context present at the binding site of `x` *must* also be present for any use of `x`, since those uses may only occur in `M`.

By introducing `JOIN` nodes before `h` we effectively lift the output from `g` and `f` into the same scope as `h` by duplicating the values as needed. By using `JOIN` to align inputs for the multi ancestor operators we create the execution semantic we need and simultaneously also take care of correctly handling free variables in control flow contexts.

```
let x = f(); // scope: []

// 'for' (== smap) introduces 'iter(A)'
for j in table("A") {
  // scope: [iter(A)]

  // 'for' introduces 'iter(B)'
  for i in table("B") {
    // scope: [iter(B), iter(A)]
    h(x, j, i)
  }
}
```

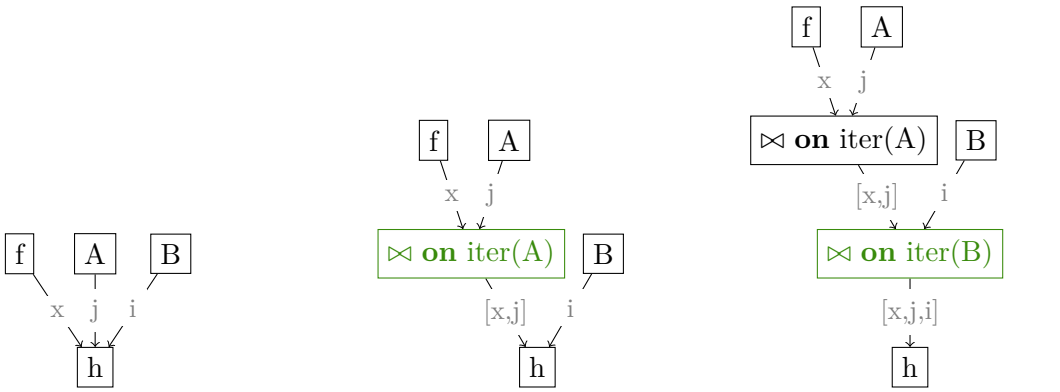
Figure 5.7: An example for scope-different join

Scope Joining example Let us consider the Algorithm from 5.7 and start with a single value for `x`.

At the next junction we begin iterating over table `A`. The goal is to end up with rows containing all free variables the functions need. This means they must contain the value of `x` as well as the current value for the only column in table `A`, `j`.

Once we enter the second loop we now need, in addition, the current value for the Column `i` from table `B`.

The `JOIN` combines two ancestor arcs by concatenating those values from their `Rows` we are interested in. In preparation we first combine all edges from the same ancestor into one, preserving input and output indices. For all nodes with multiple incoming arcs after this step we



(a) Initial dataflow graph (b) $x \bowtie y$ inserted (c) $[x, y] \bowtie z$ inserted

Figure 5.8: Joining of the `h` node from Figure 5.7

x
1

(a) x

x	A
1	j_1
1	j_2
1	j_3
1	...

(b) $x \bowtie A$

x	A	B
1	j_1	i_1
1	j_1	i_2
1	j_1	i_3
1	j_1	...
1	j_2	i_1
1	j_2	i_2
1	j_2	i_3
1	j_2	...
1	...	...

(c) $x \bowtie A \bowtie B$

Table 5.1: Result rows from joining $[x, y, z]$

then recursively insert `JOIN` nodes, combining two arcs at a time, until only a single arc with the combined record remains. Figure 5.8 illustrates this using the example from Figure 5.7.

Through this process we accumulate input and output indices in the edge annotations. From the indices we later generate column descriptors. When the graph is registered in Noria the column resolution mechanism is used to resolve the actual indices for the *Row* arrays.

By using `JOIN`s to implement multi argument functions we not only found an elegant way to attain the desired execution semantics, but more importantly we believe that this indicates that control flow is an intuitive way to represent `JOIN`. Here we only consider `for` (or `smap`) as control flow, however our prior work [2] found a similar correspondence of context/scope in dataflow and control flow in programs. This indicates to us that other control flow, such as choice/conditional (`if`) should be representable in a similar manner, which we discuss further in Section 7.3.2.

5.2.4 Scope setup

At runtime the ambient property of scope must be concretely represented in the dataflow graph. Each `Row` at runtime carries n scope columns used for joining and state scope. Each scope introducing operator, such as the `smap/group_by` combination, result in a projection operator at runtime, which concatenates the new m-item scope key onto the old n-item scope key, resulting in rows with the first $n + m$ columns being scope columns. On scope exit additional columns are implicitly discarded by the column resolution. The first out-of-scope operator works simply only selects the relevant columns, implicitly discarding the additional scope.

5.2.5 Graph integration

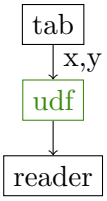
After compilation the partial query graph for the Ohua UDF is stored in a structure which contains a list of descriptions for the operators in the query. This includes the output schema for each operator, as required in MIR. Dependency edges between operators are stored as an adjacency list to make it serializable. The whole structure is written out as a Rust module and linked directly with the database during compilation. The partial query itself is linked by-name into the Noria query compilation process.

```
fn udf(inp : RowStream<i32,i32>)
       -> RowStream<i32> {
    for (field_1, field_2) in inp {
        h(f(field_1), g(field_2))
    }
}
```

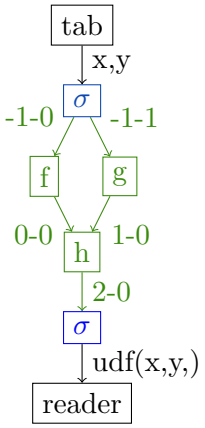
(a) UDF source code

```
CREATE TABLE tab (x int, y int);
SELECT udf(x, y)
FROM tab;
```

(b) Query source code



(c) MIR graph before splicing



(d) MIR graph with spliced UDF and projection at the boundaries

Figure 5.9: Graph splicing example

Initially a UDF is represented as a single operators in MIR, see the green `udf` node in Figure 5.9c.

A dedicated traversal over the MIR graph expands each UDF node by splicing in the partial graph constructed by the Ohua compiler. MIR graphs are doubly linked acyclic pointer based graphs. Therefore we first initialize all operators for the fragment and then link their pointers according to the dependencies recorded in the adjacency list. The resulting graph is depicted in Figure 5.9d.

A projection is inserted both before and after the generated graph, depicted as blue nodes in Figure 5.9d. The responsibility of these nodes is to link columns by translating their names specified in the query to the name generated by Ohua[1] and backwards. The UDF itself never specifies field names. It operates solely on destructuring of rows. This is not only a more intuitive notion in Ohua, but also means that UDF's are generic over the concrete fields they operate on, because the projections linking them are inserted when the query graph is prepared for running, not at compile time of the UDF. Similarly a UDF can produce more than one output value by returning a tuple[2]. Eventually the entry projections can also be used to provide additional literal arguments provided to the algorithm via the call in SQL.

[1] Ohua generates the column names from the Ohua internal node indices and output indices, hence the pictured format.

[2] This mechanism is not yet implemented.

The current mechanism for splicing the graph requires the node being replaced to only posses one ancestor, though it permits multiple arguments. This limitation exists out of convenience, as this capability is unnecessary for the properties we wanted to test with this prototype. It can be added straightforwardly later.

6 Evaluation

Our evaluation is structured into three sets of tests. Each part seeks to evaluate different features of our prototype, going from simple to complex. Broken down by these three parts we want to answer the following questions:

1. Basic
   - Can our source language express processing tasks in such a way that they feel natural in the source language?
   - Do the expected semantics in the source language hold in the generated query?
   - How large is the overhead versus an implementation with SQL and builtin operators?
2. Clickstream Analysis
   - Can using our compiler reduce the implementation effort?
3. Optimizations
   - Can queries generated by our compiler benefit from the same query optimizations available to native queries?

Because Noria implements partial materialized views there are two phases to consider for the measurements. Certain operators, such as aggregations are evaluated eagerly. They run **when the data is inserted** as opposed to when the data is retrieved. We plot *load* and *lookup* phases as applicable for the individual experiment.

All of these experiments were run on a synthetic, randomly generated dataset. We ran these experiments on 64bit Ubuntu Linux 18.04.2 with an Intel i7-6700, quad-core with hyperthreading at 3.4GHz. Unless otherwise specified each measurement is the average result over ten repetitions of the same measurement.

6.1 Basic Benchmarks

In these tests we take a look at the basic language features. We are interested in whether we achieve comparable performance to the native operators and whether Ohua queries are concise and semantically consistent. What we mean here is, can we express a query well in Ohua and does the algorithm we write run as we would expect in our imperative language.

First we implement a `sum`. The source function for this is `sum` in Figure 6.1. It is accompanied by the reversible `Sum` state from Figure 13. The implementation is straight forward. In particular the Ohua source code for the `sum` function is a typical way to implement such a function in an imperative language.

Due to the use of state in this `for`-loop our compiler will be forced to generate a single operator for this. The native `sum` function in Noria is also a single operator, which gives us an opportunity to compare generated code and native operator in isolation.

```
fn sum(rows: RowStream) {
    let acc = Sum::new();
    for (_,r) in rows {
        acc.add(deref(r));
    };
    acc.get_sum()
}

fn count(rows: RowStream) {
    let acc = Count::new();
    for r in rows {
        acc.increment();
    };
    acc.get_value()
}
```

```
fn average(rows: RowStream<i32,i32>) {

    for g in group_by(0, rows) {
        div_not_zero(count(g), sum(g))
    }
}
```

Figure 6.1: Complete algorithm for the basic benchmarks

The dataset consists of 8000 keys which we group over. A random number of entries is generated per key with the mean number of entries per key shown as the x axis. The y axis shows how many events are processed per millisecond.

The data set is regenerated when the number of events per key changes. As a result measurements which differ in the number of events per key can only be compared in broad trends. The two implementation versions operate on the same dataset however, making them comparable. What we are most interested in is how close the performance is between the two `sum` versions.

In both phases the two implementations exhibit almost the same performance, with differences being within less than 1%. There is one outlier in the first measurement for the *load* phase. This is likely due to the very low number of entries per key, which exacerbates any fluctuations during processing. The generality of our operator implementation seems not to impact performance.

In the second part of this experiment we implement an `average` function. In Ohua we do this by using the `sum` we already implemented and combine it with a `count` in Figure 6.1. `count` is also accompanied by a state implementation similar to the state for `sum`, which we omit here for brevity. Noria does not provide an `average` primitive so we implemented it in SQL as presented in Figure 6.4a.

The `average` UDF shows how a database `join` is represented in the imperative source language. The imperative version uses a regular function with two inputs. In our model this gives rise to an implicit `join`, using *scope* information as explained in Section 5.2.3. This `join` ends up the same as the `join` in the SQL query from Figure 6.4a. For the average query our imperative source language gives us a concise way to express the same computation as the corresponding SQL because it hides some of the underlying detail.

For the performance evaluation we did the same experiment as in the first part with the `sum` function, but we dropped the number of keys down to 6000. What we can see here is that for the computationally expensive *load* part the Ohua version exhibits a constant 10% overhead. This is likely due to the *entry* and *exit* operators. Figure 6.4b overlays the graphs generated by the SQL query and by Ohua with Ohua exclusive operators marked like this.

As mentioned in Section 5.2.5 each query fragment generated by Ohua gets fenced in by two projection nodes, used for column renaming. Unfortunately these projections rebuild each row, as they can also be used to filter out columns. Since our query is so small with only 4 operators in total, these two extra operators are significant. In future we should be able to use an `Identity`

instead, reducing the impact of the column renaming.

During the *lookup* phase the constant overhead cannot be observed. Instead the two implementations perform nearly identical until about 300 items per key where the performance of the Ohua implementation seems to suddenly taper off. We have not been able to find a reason for this yet and leave this investigation to future work.

6.2 Clickstream integration

In the second set of experiments we take a look at our central example, the clickstream analysis query.

For a correct clickstream UDF we first require an implementation for the state. In Figure 1.1a we showed a version that used a `HashMap` as state. While this version was certainly easy to read, its meaning was not trivial to comprehend. The `HashMap` state was used to do two things at the same time 1. Group the events by user and 2. Count the number of events.

In our UDF implementation we have access to database processing primitives. As a result we can avoid this intermingling of concerns. The first responsibility of the `HashMap` was to group events by user. We can achieve the same more concisely using the `group_by` function. This simplifies the per-user part of the algorithm, which now only needs to do the counting.

Normally a simple counter might be enough, as is the case in Figure 1.1a. In Noria data arrives out of order and algorithms need to be reversible. A simple counter is not sufficient for this. Instead we use a state that orders all timestamps of events into a sequence of intervals.

6.2.1 State Implementation

The clickstream state is the first reversible state we implemented, which is not itself a group (see also Section 4.2). It does however still fit into the map-reduce aggregation pattern described in Section 5.1.2, meaning the compiler can automatically generate the code that keeps track of which values are updated on a given change. However the incremental nature of the system means the simple implementation from Figure 8 does not work here. This is for three reasons.

1. Directly returning the interval size after `end_cat` is received assumes the timestamps arrive in order and that upon receiving a `end_cat` event, it has definitely seen all events between this event and the last `start_cat`. This assumption is violated, due to the potential out-of-order execution in Noria. This means we need to store the size until we have seen all relevant items.

2. The simple counting algorithm which does not store the bounds, means that when we receive a *delete* event for a timestamp, we do not know which interval shrunk, and cannot recalculate. This means we need to store the bounds of each interval along with the size, even across invocations of the operator.

3. Even if we do store the bounds we cannot deal with updates to bounds. If a new bound is inserted into a closed interval, the interval must split into two smaller ones, one closed, one half open. The size alone cannot tell us which events belong to which of the two new intervals. This means we also need to store the timestamps themselves, so that they can be accurately assigned in such a case.

From these three constraints we can construct the state pretty straightforwardly as an ordered sequence of intervals, which have optional lower and upper bounds, see Figure 6.7.

A few rules apply here

1. An item placed in an interval must always be within the bounds this interval has, i.e. larger or equal to the lower bound, if such a bound exists and smaller than the upper bound, if such a bound exists.

2. Two adjacent intervals may not be "opened toward each other", i.e. if the interval with the smaller elements lacks an upper bound, the interval with the larger elements must have a lower bound. In such a case the two must be merged into a closed interval.

3. An item must always be placed in the "best fitting" interval. I.e. either in the interval with the largest existing lower bound for which it is still in bounds or the lowest upper bound for which it is still in bound. Due to rule number 2 these two are equivalent.

For our clickstream algorithm `start_cat` events (or events of type 0) insert lower bounds, `end_cat` (or type 1 events) insert upper bounds and any other event becomes an item. Computing the result value is then the average number of items for all closed intervals.

6.2.2 Results

We compare three versions of the clickstream analysis. A manual implementation as a UDF, operator compilation through Ohua and finally whole-query compilation through Ohua. The complete source code is shown in Figure 1.2. For the *operator comp* numbers we only compiled the `op_core` function using Ohua, which due to state scoping, results in one Noria operator being generated (compare Section 5.1.1). We call this operator directly with the SQL snippet in Figure 6.8. In *ohua query* we then let Ohua handle the entire `main` algorithm, including graph integration (compare Section 5.2.5).

Table 6.1 shows a breakdown of the code necessary to implement `op_core` by hand as a Noria operator and link it, such that it is accessible by a query. In particular the *Operator Linking* part is tricky, as it is spread across several files throughout the database. Ohua automates the boilerplate and linking. This leaves the 400 lines from the state implementation, which are unavoidable, and the 15 lines in the body of `op_core`.

The whole-query compilation does not reduce the line count in this case. It removes the `GROUP BY` line from the SQL used to call the UDF, but adds more lines in the UDF itself that call `group_by`. However as shown in the experiments with the `average` function, this allows for more complex queries to be expressed with Ohua.

Component	LOCs	Number of files
State	400	1
Operator	240	1
State integration	48	2
Operator linking	9	5

Table 6.1: Implementation effort for Clickstream

The performance comparisons turn out very similar to the previous experiments. Figure 6.9 shows a similar constant offset for the *load* phase between the manual implementation and generated query. The *lookup* phase also exhibits a similar behavior to the previous experiment. At the higher load end the Ohua query lags behind its manual counterpart. The fact that both `average` and the clickstream analysis are affected so similarly indicates that the reason lies in a common part of the query compilation. Likely the additional operators, which are even more visible here, as the graph generated contains less operators overall than in the `average` query.

Originally we had planned to contrast this with a pure SQL implementation. Unfortunately we were unable to adapt the SQL query from Figure 1.1b to work correctly in Noria. We will therefore not be able to show performance numbers as a comparison.

We were able to create a query that was accepted by Noria. However results of running the query were incorrect. The query did not compute the average of the clickstream sequence, instead returning 0 for any key we request. It shows us that the writing of larger SQL queries comes with significant difficulty. Our imperative version on the other hand is easier to understand and requires no SQL to run.

6.3 Parallelism

The last question we want to answer is whether the queries we generate are able to benefit from optimizations. For this work we specifically take a look at data parallelism. Noria is able to parallelize queries using *Sharding*. This hash partitions the data and distributes it across multiple cores or machines. The downside of a black box UDF, as shown in Figure 1.1a is that the database does not know the data access patterns. This would prevent such a UDF from being *sharded*.

The Ohua implementation of the clickstream analysis integrates with the native Noria indexing mechanisms, such as `GROUP BY` and is able to trivially take advantage of sharding.

In fact for our experiment we did not need to perform any additional source code modifications. We simply set the *sharding* parameter in the database options during startup.

We run a scaled up version of the computation from the previous experiment. We successively increase the sharding factor, thus parallelizing the workload. Because the clickstream analysis is also an aggregation the bulk of the work is performed when the data is loaded. Figure 6.11 shows how the throughout in the *load* phase increases as more cores are used for computation. Unfortunately we were unable to measure the computation time separate from the data load. As a result we cannot make conclusions as to how exactly the computation scales, but only that it does not prevent the scaling of the data loading.

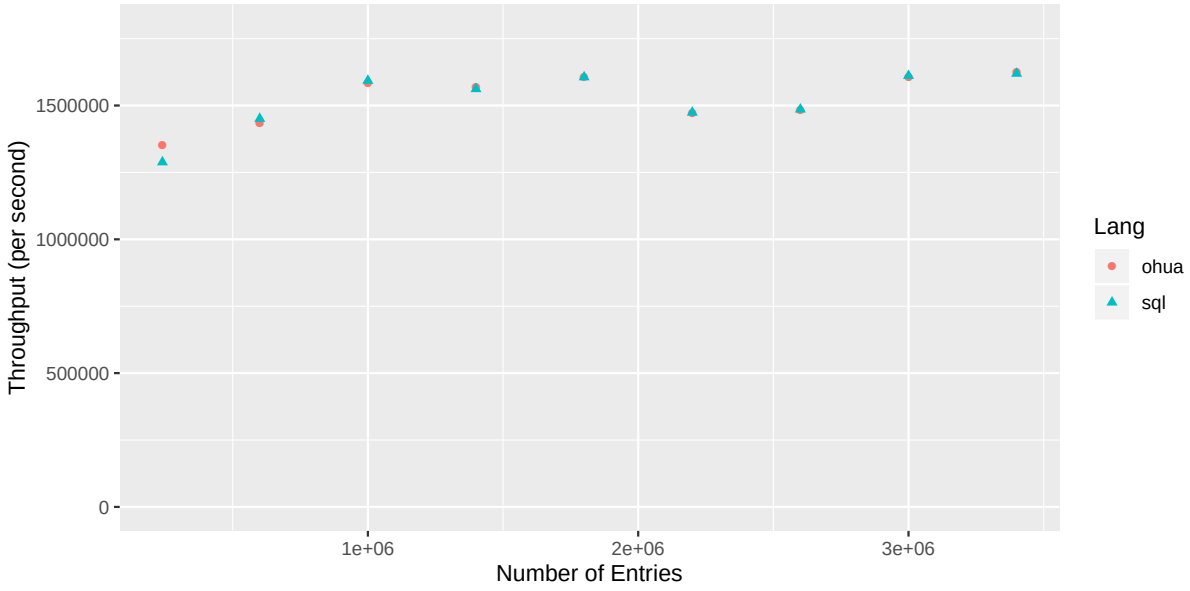


Figure 6.2: Throughput of `sum` during load

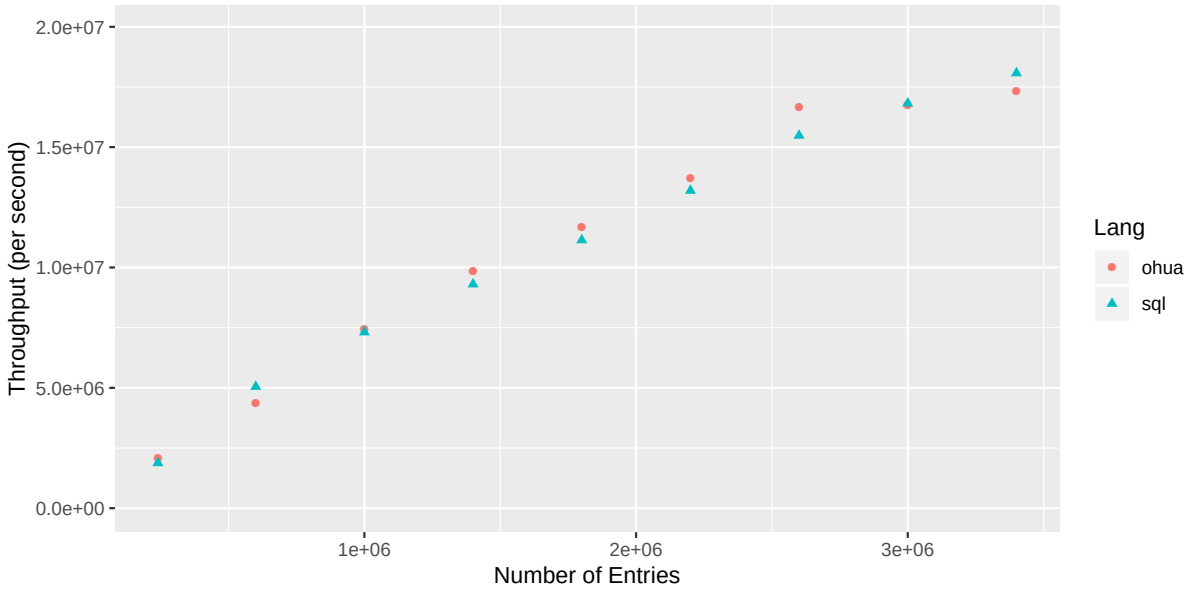
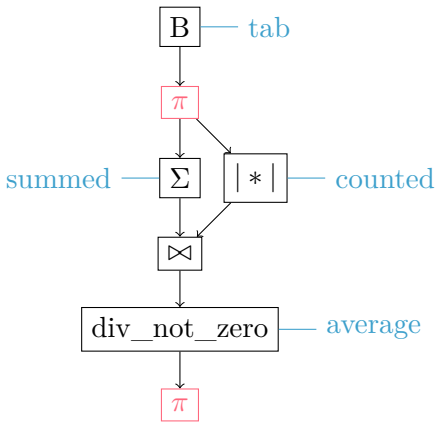


Figure 6.3: Throughput of `sum` during read

```
CREATE INTERNAL VIEW counted AS
    SELECT count(*) as c
    FROM tab
    GROUP BY grp;
CREATE INTERNAL VIEW summed AS
    SELECT sum(i) as s
    FROM tab
    GROUP BY grp;
CREATE VIEW average AS
    SELECT s.s / c.c
    FROM summed
    JOIN counted
    ON summed.grp = counted.grp;
```

(a) The SQL implementation of an averaging query



(b) Overlayed graph rendering of `average` with highlighted Ohua exclusive nodes

Figure 6.4: A query computing averages in SQL and as a Noria runtime graph.

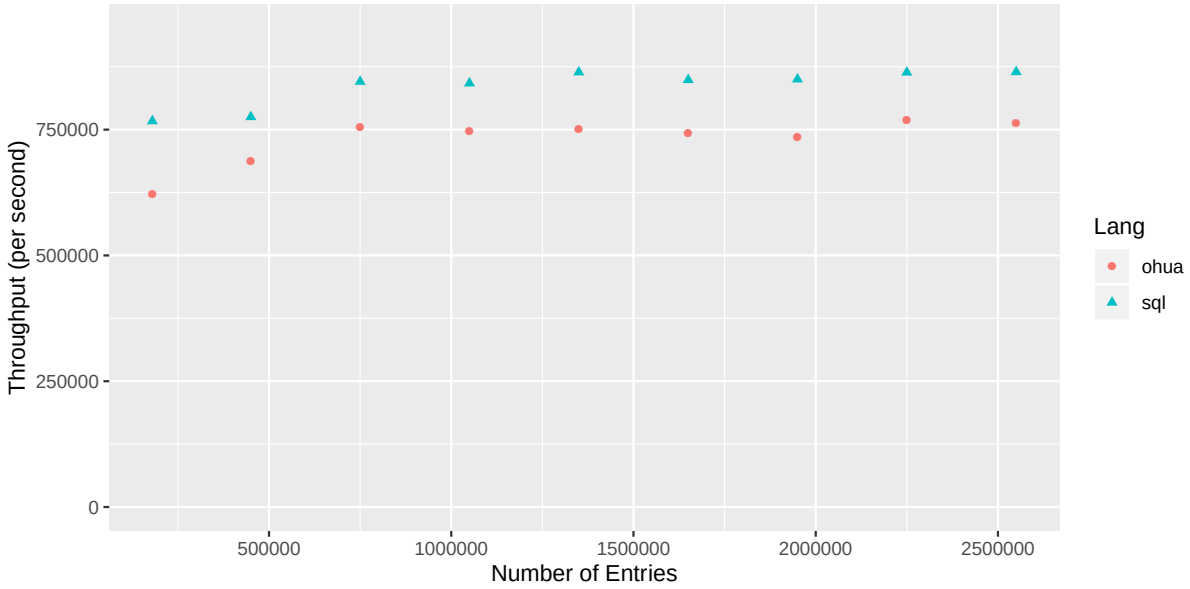


Figure 6.5: Throughput comparison of `average` queries in load phase

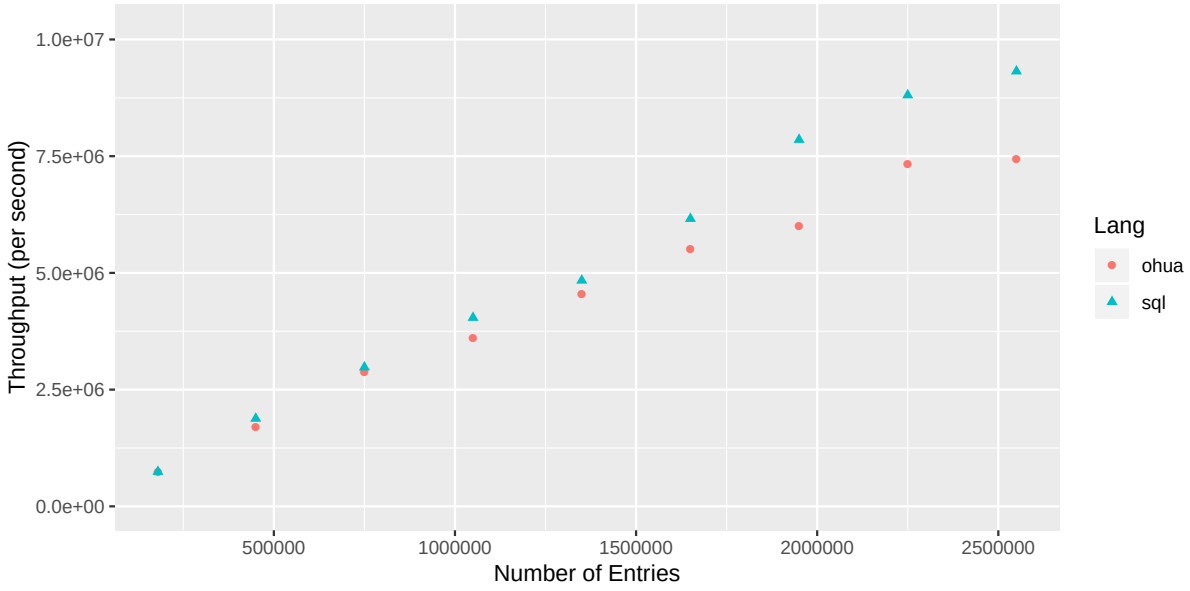


Figure 6.6: Throughput comparison of `average` queries during reads

```rust
pub struct Seq<T>(Vec<Interval<T>>);

pub struct Interval<T> {
    lower_bound: Option<T>,
    upper_bound: Option<T>,
    items: Vec<T>,
}

pub enum Action<B> {
    Open(B),
    Close(B),
    Insert(B),
}

impl<T:Ord> Seq<T> {

    pub fn apply(&mut self,
                 action: Action<T>,
                 positive: bool) { ... }

    pub fn compute_new_value(&mut self) -> f64 {
        let lens : Vec<usize> =
                self.0.iter()
                    .filter(|e| e.is_closed())
                    .map(|i| i.items.len())
                    .collect();
        let l = lens.len();
        match l {
            0 => 0.0,
            _ => {
                let sum : usize = lens.into_iter().sum();
                sum as f64 / l as f64
            }
        }
    }
}
```

Figure 6.7: Interval Sequence, shortened

```
VIEW clickstream_ana:
    SELECT user_id, op_s_sequences_0_0(pagetype,ts)
    FROM clicks
    WHERE user_id = ?
    GROUP BY user_id;
```

Figure 6.8: SQL snippet to embed the clickstream operator

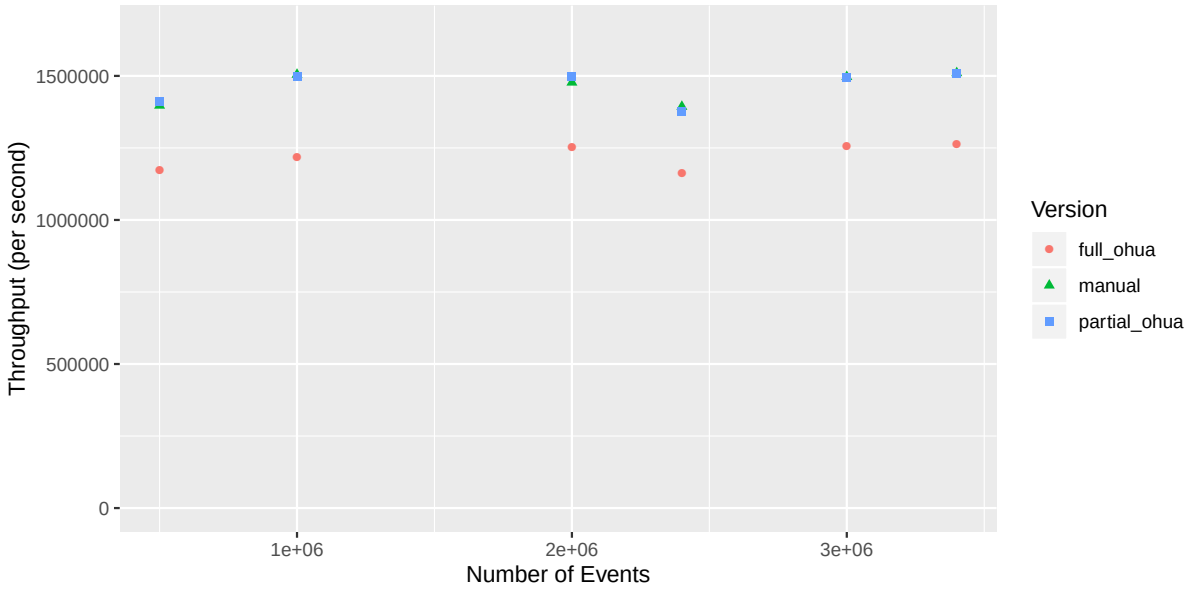


Figure 6.9: Throughput comparison of the different clickstream query versions on load

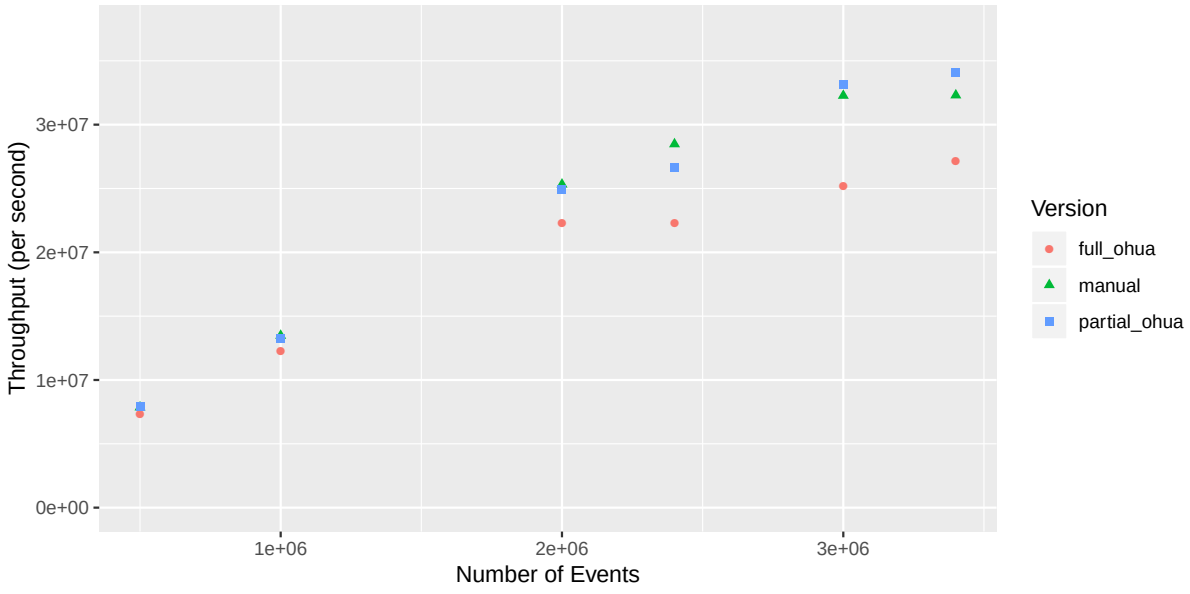


Figure 6.10: Throughput comparison of the different clickstream query versions during reads

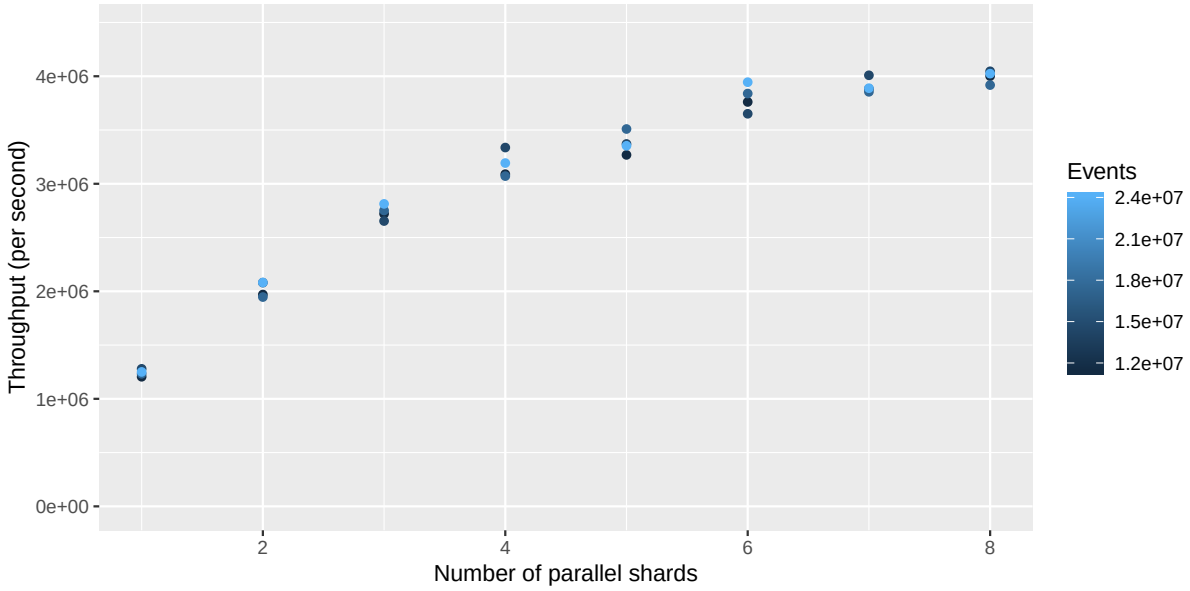


Figure 6.11: Performance of the ohua clickstream analysis with sharding

7 Outlook

In this work we tentatively delved into the general idea of a) Representing imperative, state transforming algorithms in a database dataflow and b) Express relational queries in a general purpose language. What we hope to achieve most is a cooperative interplay between the two such that efficient, stateful algorithms become available in database queries. At the same time we hope to make relational queries easier to develop incrementally, by using a language more familiar to most. Achieving a smooth integration between the two paradigms is not a trivial task. Naturally this work is by no means exhaustive, giving rise to many unanswered questions and research challenges.

In this section we try to address open questions arising from our experiments and give some intuition how they may be solved in the future. We have so far shown that programs written in a restricted, general purpose style can be represented as a database query. What remains to be investigated is how widely applicable this approach is. What, if any, are the limits for the representation of general purpose programs in a database?

There are two perspectives to consider here, first, what and how queries can be expressed in general purpose style (Section 7.1), and second what and how general purpose programs can be represented, and represented efficiently in a database, we discuss in Section 7.3.

We chose Noria as a target for our prototype, which is an environment with unusual and restrictive execution semantics. It is our belief, but remains to be shown, that this translates straightforwardly to less restrictive environments. Since the Ohua internal operators and control flow are represented entirely with standard database operators we are confident that assumption of portability should hold. The particular ways in which state is treated due to Norias restrictions, may in fact form a basis for extracting more parallelism from stateful queries. State splitting can be used to duplicate operators, allowing exploitation if Data Parallelism and out-of-order-execution at the level of operators can be used for flexible Pipeline Parallelism through scheduling.

With respect to User Defined Functions in Noria, whether or not realized as partial queries via Ohua or simply as operators, there are more open questions, particularly around reversible UDF's. For grouping operators we were able to reverse the function by reversing modifications performed on the state. We also reasoned that, in general, reversing modifications are likely not so easy, because it is difficult and costly to track the precise relationships between input and output data in the presence of state. In Section 7.2.1 we take a look at possible ways to construct reversible states.

7.1 Replacing SQL

For most programmers the languages they are most familiar with and which they understand best are imperative, procedural or object oriented. SQL on the other hand is declarative. This is a huge boon for query optimizers and for simple queries too, because a programmer does not have to specify how exactly the data is processed. SQL thus can be more terse, that is it expresses complex relationships with little code. This works incredibly well for many cases, especially relatively simple queries. However according to Lu et al. complex queries are common and difficult to understand for users, in particular when they involve nesting. But even just a *join*, a fundamental element of SQL, users find difficult to reason about [21].

By integrating UDF's as queries we improve UDF performance, but also open up the use of UDF's as a real alternative for writing queries. One surprising finding of this work is that *join*s express themselves quite naturally through loops. Unlike *joins*, loops are a concept most programmers are intimately familiar with, be that in the form of C-style `for` or functional `map`.

Other common SQL operations, similarly to the already implemented `group_by`, will be served well with builtin functions and higher order primitives, such as `filter` to express `WHERE`. These primitives largely borrow from stream processing, where compositions of primitives are used to functionally transform, potentially infinite, streams of data. The conditions in the database are similar, particularly in Noria, lending themselves nicely to stream transformers.

These constructs largely retain the semantic a programmer would expect, while allowing the system opportunities for optimzations comparable to native SQL. We believe this could be a more intuitive language for programming databases, especially for beginners.

7.2 State Theory

For the purposes of this work we chose to only consider a particular kind of stateful algorithms which follow a kind of map-reduce pattern facilitated through state. As a result we knew for any algorithm following this pattern, which updates would have to be issued upon a change in the data. As Table 2.1 shows this covers single tuple UDF's and UDA's, leaving SRF's and UDTF's open.

SRF's are an important building block for a more general notion of iteration. At the moment the only function which creates a new sequence is `group_by`. SRF's can allow the user to expend arbitrary data into sequences. Examples could be a `range` function or a string `split`. Like UDTF's these functions use state. However unlike UDF's where the relationships between input and output can be arbitrary, state for SRF's is initialized *per input value*. This means there is a clear dependency between an input and all its output, but crucially there is no dependency across *different input tuples*. As a result SRF's can be lifted to operate on deltas by copying the sign of an input value to all output values. In this sense SRF's are similar to the pure functions, but with a sequence of return values rather than only one.

UDTF's As mentioned before Table Functions are non trivial. Input-Output dependencies are highly dependent on the actual processing function. From a system perspective the simplest solution is to require the user to handle deltas themselves. At this point the developer is basically implementing low-level database operators, a wholly undesirable situation. The next section therefore explores possible avenues to implement reversible UDTF's automatically. State is central here, as it determines these relationships. If we were to obtain a handle on how to efficiently determine input-output relationships for these states, UDTF's may become feasible in Noria.

7.2.1 Non-grouping State

As an example of where that is not as trivially possible consider a very similar function called sessionize in Figure 7.1. In this function the output values are annotated with a bucket number. We could again model the state using an `IntervalSequence`, however if one of the buckets change, we must emit updates for all logically subsequent values.

This means that the presence of state may result in data dependencies between an input value and a non-canonical [1] output value. For more general algorithms we therefore need a way to determine necessary updates resulting from a sequence of state modifications.

```
type Session = i32;

fn sessionize(session_length: TimeDiff,
              stream: RowStream<(UID, Timestamp)>)
    -> RowStream<(UID, Timestamp, Session)> {
    for ustream in group_by(0, stream) {
        let mut session_num = 0;
        let mut session_begin = None;
        for (uid, ts) in ustream {
            let begin = Option::or(session_begin, ts);
            if TimeDiff::diff(ts, begin) > session_length {
                session_begin = Some(ts);
                session_num += 1;
            }
            (uid, ts, session_num)
        }
    }
}
```

Figure 7.1: The sessionize UDF

In before we chose the example to focus on in this work we considered how a general solution for properly using state in Noria might be achieved. One of the benefits of the solutions we presented here was that the user did not have to consider reversibility in their algorithm. Instead this solely fell on the state implementation. While this is desirable our preliminary theories suggest that this is difficult to achieve for a general case where completely arbitrary interactions with the state are allowed. The trouble is that once state interactions can not only modify the state but also return values, those can in turn be used to modify the state. This complicates reasoning about the precise relationship between the input values and state deltas. This leads either to high tracking overhead or conservative and costly recalculations.

In general the system would have to conservatively assume that if an input value i_i to a UDTF changes, all values emitted by the function for subsequent values i_{i+1}, i_{i+2} etc may be affected and must be recalculated. The tracking alone would have to keep the exact order of items in memory to facilitate the recalculation, not to mention the additional effort of the recalculation itself.

In future we would like to develop a principled approach to defining reversible state either by construction, through derivation strategies driven by annotations or by analyzing how the code uses the return values. There is already a body of research surrounding the idea of bidirectional programming, perhaps its findings may also be applicable in this situation.

[1] A value not directly associated

7.3 Extended language support

Our prototype currently covers only a subset of the input language. Eventually we want to be able to provide a fully featured environment for query programming, which is, as far as possible, indistinguishable from using Rust or a similar language. An important feature of this (type of) language is the ability to abstract. One such abstraction feature we already provide in the form of functions. In an Ohua UDF the user can define functions which are reusable and zero-cost both as inline lambdas as well as top-level reusable pieces of code. In the same vein other abstractions such as user defined data types, will be important to easily define complex data processing algorithms.

7.3.1 Compound types

Database runtimes, such as Noria's are specialized to the native SQL datatypes and their one dimensional composition, *Row*s. Rust is a general purpose programming language and supports more complex data types such as multi dimensional arrays, structs and arbitrary nestings of these. To support a richer and more flexible programming interface it may become necessary to support complex data types in the bridge between Rust and Noria facilitated by Ohua.

Extending the native SQL type with the user defined types is a possible way to enable complex types in queries. However the database interacts with the data items for reordering, comparison, copying and storage. This means a lot of code generation to derive the necessary implementations for each user defined data type. An efficient implementation would entail yet more patching of the database source and a generic version using vtables would be inefficient.

Flattening of complex types is a more promising approach. This applies the same ideas that underlie data representation in memory. The data structure is recursively flattened until a single, heterogenous array of base types remains. This works well for non-recursive product and sum types (structs and enums), but makes representation of arrays and other usually heap allocated types difficult.

7.3.2 Extended control flow support

For this work we restricted ourselves to grouping iteration when it comes to control flow. With support for *SRF*'s, as mentioned in Section 7.3 users would attain the ability to create their own sequences, necessitating a generalized notion of iteration. Section 4.5.2 already discussed how conditionals could be implemented, another common control flow construct, ubiquitous in general purpose languages.

General Iteration

In a database environment programs operate on a sequence of data items by default. This was particularly apparent when generating code in Section 4.3 and is one of the reasons why dataflow applies so naturally here. This stands in opposition to general purpose languages which always explicitly source data, providing an implicit ordering on data which is used in the ubiquitous iteration constructs such as `for`.

In databases, and Noria in particular, we operate on sequences of tuples. However there is no inherent ordering to these tuples. This lack of ordering is a desirable property for a database, as it allows the scheduler more decision freedom. From the theory of *scope* (Section 5.7) we already see the notion of a tagged sequence arising. In the particular case of our prototype the tag is a group-key, introduced by a `group_by` primitive. For general iteration to work as expected user defined sequences created with SRF's may need additional indices for scoping state and joining

(see also Section 4.3). Those indices could also be used for a `sort_by` primitive when the ordering is explicitly required.

Builtin SRF's can also serve as annotations. For instance a `chunk_by(low, high, sequence)` could create virtual subsequences, similar to `group_by`, with sizes ranging from `low` to `high`. Instead of actually creating these sequences, these SRF's communicate to the database how the input data could be split, enabling additional data parallelism. While `chunk_by` might not be the most useful, Große et al. found similar annotations which can be used to parallelize UDF execution [14]. Unlike annotations these partitioning functions are not syntax extensions and integrate naturally with the source language.

Recursion

Ohua supports a limited form of recursion. Tail recursive functions are supported, because these can be represented as a cyclic dataflow graph with a termination condition. General recursion would require a more complex implementation, essentially replicating a call stack. Recursion is an immensely powerful concept, even in a limited form. It can be represented in dataflow, but naturally it needs a cycle in the graph. MIR graphs, the target of our compilation, are acyclic. The same is true for the runtime operator graphs. The system uses the graph to propagate messages up and down the graph. It is not obvious how Norias messages work in a graph with cycles.

There could be substantial benefits however. Oracle recently added native JSON support to their database to deal with the increasing demand for support for unstructured data [20]. One of the central ideas in this paper is that the database stores not a JSON blob but indexes it automatically by its fields. This allows queries to efficiently operate on JSON fields natively, without requiring an explicit schema.

```
fn decoder(key: Key, inp: Vec<u8>) {
    match inp[0] {
        0 => emit((key, Value::Number(JSON::decode_number(inp[1..])))),
        1 => emit((key, Value::String(JSON::decode_string(inp[1..])))),
        2 => emit((key, Null)),
        3 => {
            for (field, val) in JONS::object_iter(inp[1..]) {
                emit(decoder(key.clone().extend(String(field)), val))
            }
        }
        4 => {
            for (idx, val) in JSON::array_iter(inp[1..]).enumerate() {
                emit(decoder(key.clone().extend(Number(idx)), val))
            }
        }
        _ => panic!("Unknown JSON value type"),
    }
}
```

Figure 7.2: JSON decoder recursive SRF

The trouble with nested, often schemaless data, such as JSON, is that the arbitrary nesting is difficult to deal with in a query graph. One possibility for dealing with this may be a recursive decoder UDF. Such a function could decode a JSON string from the outside in, recursively calling itself on inner objects. In Noria this would manifest as an operator which emits partial decoding results with increasing compound keys. Eventually the recursion terminates and the output state the Noria system maintains would contain the decoded objects indexed by their path. An example of how this may look is shown in Figure 7.3, which shows a JSON object

and the representation of the decoded values[2]. An imagined version of such a more or less tail recursive decoder can be seen in Figure 7.2.

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

(a) JSON input

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

(b) Database view of the indexed object

Figure 7.3: JSON decoder example

7.3.3 Embedding SQL and eliminating transactions

In application code SQL often occurs as a sequence of queries with data processing in the application language in between. Even worse, those sequences of queries are often in a transaction which have significant bookkeeping overhead. The database must keep logs of all executed queries, to roll back whole transactions in case of conflicting concurrent queries.

The database has to be so conservative because it does not know what data the application might request or modify next. The application, like the UDF's, is a black box. In a concurrent setting multiple transactions run in the database at the same time. Atomicity guarantees however mean that only non conflicting updates succeed, aborting competing transactions. This effectively sequentializes the competing transactions. Application performance achieved through parallelism is reduced significantly while adding significant bookkeeping overhead.

In this work we have begun to develop techniques which may allow application code to be ported and run directly in the database. A further extension could allow for embedded SQL in this code which can link with the dataflow graph Ohua generates for the application code. This hybrid graph could run directly in the database. With knowledge about which of these graphs may conflict the query engine could design non-conflicting, parallel transaction schedules. For these parallel transactions there would be no bookkeeping required.

The query engine may also partition the queries into conflicting and non conflicting sections. Non conflicting sections could be run in parallel. Conflicting sections could be sequentialized through scheduling alone. In some cases the engine might not be able to eliminate the bookkeeping. However it could still be reduced by only checking integrity in conflicting sections.

7.4 Optimizations

This work touched on optimizations only in so far as that we are able to use state to implement an otherwise costly multi-join efficiently. We also showed some additional parallelism that could be gained from scoping the state to a `group_by` and sharding the data. We believe that this direction has yet more to give, see Section 7.4.1. What we were so far not able to tackle is a tight integration with the query planner and optimizer, which we discuss further in Section 7.4.2.

7.4.1 Parallelism and Scheduling

Parallelizing the computation to remove UDF's as bottlenecks was our central aim in this work and the reason we employ the Ohua parallelizing compiler. However the exploitable parallelism

[2]Partial results are omitted in the database view.

in a program is bounded by the degree to which the program can be broken into individual tasks and by the degree to which it *should* be broken apart. This is also known as the *Granularity Problem*. Both of these come down to compiler knowledge. The more fine grained our notion of dependencies in the program is, and the fewer dependencies there are the more parallelizable tasks we are going to find. Similarly the more accurate our cost estimate of each computation, the better our scheduling decisions are going to be.

Partitioning

The restrictive programming model in Ohua, with its explicit data dependencies provides us with a baseline of a parallelizable graph. As we saw in this work we can expand upon that using node duplication and sharding, afforded by the scope knowledge. To further this quest we should consider more partitioning primitives, such as bounding, size partitioning etc. to enable the programmer to communicate a more fine grained view of the dependencies in the algorithm. Partitioning functions, such as `group_by` are natural to the programmer, require little instruction and can be added to the query later and thus pose little effort. The *scope* mechanism presented in Section 4.3 will provide the way to capture partitioning inside the compiler. It is likely that we could provide an API for users to provide their own partitioning primitives, including a description of how their scope keys keys function. This would result essentially in an overloaded *smap*.

As a contrary approach we can try to derive partitioning from the type of state that is used. If we consider our original version of the clickstream analysis in Figure 1.1a the state was a `HashMap` over the user ids. This data structure in essence expresses the same relationship as the `group_by` combinator. If we can establish a correspondence between the execution of an algorithm and the representation of its state, we can derive partitionings as a translation of the data structure used.

Splittable reducers

A potential problem with the state driven map-reduce, as used in our evaluated programs, is that the map-reduce UDF can only ever process one item at a time, because the state needs to be handled sequentially. We can see this in the state interface in Figure 5.3. Though the order does not matter, each action is applied to a single state value, preventing a parallel execution.

Mathematical *Group*s offer a solution in this respect. A group supports the notion of reversibility (through negation), but since every delta is itself a member of the group the states themselves could be combined as well. This applies for instance to both the `count` and `sum` functions from Section 6 where the state is the same as the deltas. Groups are more restrictive than necessary in this case though, because we only require reversibility for the deltas and the states need to support merging. The actual constraints would be a negation for the deltas and the states should be *Monoids*. In the case were delta and state are the same this leads back to a group, however crucially it would allow state and delta to be two different types.

The sawzall [22] language uses a similar idea. They use special, builtin reducers that support partial computation and merging to achieve a similar effect. Using monoidal states would allow us to define similar special reducers, but also potentially to develop a framework for users to define their own. This could open yet more opportunities for parallelism.

Fusion and Domains

As a contrary path to the techniques mentioned above, which expose opportunities for parallelism, we must also consider the reverse. While *knowing* about opportunities to parallelize is always beneficial, parallelization itself is not always better. Our system allows us two mechanism for reducing parallelism *Fusion* and *Domains*. *Fusion* is a compile-time technique where we can combine several small computations into a single operator. This reduces the scheduling overhead,

data copies at the beginning of the data processing phase, and lowers the memory overhead by eliding an additional Noria operator state. The second technique available is *Domains*. *Domains* describe the scheduling of parts of the graph onto processing units. By assigning two nodes to the same domain, they will run on the same processor, and not in parallel.

Domains can give us a runtime option to reduce the amount of parallelism. In particular we could use online monitoring of the execution, and if we find a communication bottleneck we can adjust the graph and reassign the offending nodes to run on the same domain.

By contrast *Fusion* is better suited for compile time. It offers greater benefits by eliding any scheduling and communication as well as the memory for state, but relies on compile time knowledge. It is perhaps best suited for optimizing known, cheap functions in an Ohua program by fusing them onto ancestors or successors. Expanding this to user defined functions would require a cost model of functions. There are ongoing efforts to use data collected in an instrumented Ohua program to guide a fusion process.

7.4.2 Query rewriting

This one is perhaps most straight forwards, we want to leverage existing query rewrites in the database. Our work here achieved that the database now has structural knowledge of the UDF, but it does not yet leverage knowledge we have, or could have, of individual operators. A typical rewrite for instance is the movement of predicates and filters before *join* nodes.

The current *Joining* rewrite is naive. It uses $n - 1$ joins for *every* $n-$argument function. In theory each free variable only needs to be lifted into the nested context once, and could then be reused by all operators. We did not implement this in this work, because it is more complicated, due to the rewrite being non-local, and only affect larger queries. It also touches on the issues discussed in Section 7.4.1 because it introduces additional data dependencies, and thus obscures parallelism opportunities.

8 Related Work

User defined functions are one side of a continuing effort to expand the expressiveness of databases. There are two fundamental directions of approach.

1. Extending the native capabilities of databases.
2. Improving the integration of foreign, often general purpose, languages.

The former approach is used for instance by Oracle to add native capabilities for representing and manipulating JSON [20] in an attempt handle semi-structured data. Similarly Asterix [5] and Sawzall [22] and sciDB [6] define new Query languages. The underlying query model is similar to SQL, with better support for certain aspects of queries. In the case of Asterix that is support for semi-structured data, in the case of Sawzall reducer performance, facilitated by providing special, reducers with mergeable state. sciDB specifically optimizes for operations on large arrays with support for special, user defined operators.

In the latter approach one starts with the flexible User Defined Function and tries to alleviate its shortcomings and expand its expressiveness. Basic UDF's operated on single tuples or as aggregations, effectively allowing users to implement their own arithmetic(-like) functions. This eventually extended to functions operating on whole tables [18].

Depending on the type of database system UDF's may be implemented in various languages. Common languages are procedural SQL dialects or high-performance low-level languages, such as C and C++. These languages are designed for expressiveness and performance, allowing little introspection and making sophisticated analysis difficult. Apart from the type of UDF (Tuple, Aggregation, Table), the DBMS has no knowledge about the functions runtime, state partitioning, inter-tuple-dependencies and so on. Several solutions have been suggested, such as multi-input functions with scalars and blocks of tuples as input [7] and relation-in, relation-out Functions [17]. These extend the types of inputs available to the UDF. Große, May, and Lehner on the other hand *constrain* the inputs of table functions by annotation which specifies access patterns, enabling parallelization [14]. Finally Friedman, Pawlowski, and Cieslewicz use a form of introspection by adding native support for UDF's implementing a MapReduce pattern [12].

To the best of our knowledge none of these approaches achieves the same full-query integration we are aiming for.

9 Conclusions

UDF's are the easiest way to perform non-traditional processing tasks in a database. The most problematic aspect of using UDF's is that they do not benefit from database optimizations. We proposed to improve the performance of UDF's in databases by compiling UDF's to dataflows that integrate with a query graph.

In this work we have shown how imperative iteration and conditionals can be represented as query dataflow. Our prototype is able to generate query graphs and operators for non-trivial UDF's with near-native performance.

We also proposed a mechanism for making single-tuple UDF's automatically incremental. In addition we show how incremental UDA's can be built from a state with reversible modifications. Both of these are able to integrate fully with Norias partial state.

Our results indicate that performance similar to SQL queries can be achieved using an imperative language. In addition they are able to effortlessly benefit from a parallelizing optimization.

We believe that future work should be able to further expand the range of imperative algorithms that can run as queries. This paves the way to move even more computations from application logic into the data store.

Bibliography

[1] Daniel Abadi et al. "The Beckman Report on Database Research". In: *SIGMOD Rec.* 43.3 (Dec. 2014), pp. 61–70. ISSN: 0163-5808. DOI: 10.1145/2694428.2694441. URL: http://doi.acm.org/10.1145/2694428.2694441.

[2] Justus Adam. "Control Flow and Side Effects support in a Framework for automatic I/O batching". MA thesis. Technische Universität Dresden, 2017.

[3] Rakesh Agrawal et al. "The Claremont Report on Database Research". In: *SIGMOD Rec.* 37.3 (Sept. 2008), pp. 9–19. ISSN: 0163-5808. DOI: 10.1145/1462571.1462573. URL: http://doi.acm.org/10.1145/1462571.1462573.

[4] Steve Awodey. *Category Theory*. Oxford University Press, 2006. DOI: 10.1093/acprof:oso/9780198568612.001.0001. URL: https://www.oxfordscholarship.com/10.1093/acprof:oso/9780198568612.001.0001/acprof-9780198568612.

[5] Alexander Behm et al. "ASTERIX: towards a scalable, semistructured data platform for evolving-world models". In: *Distributed and Parallel Databases* 29.3 (June 2011), pp. 185–216. ISSN: 1573-7578. DOI: 10.1007/s10619-011-7082-y. URL: https://doi.org/10.1007/s10619-011-7082-y.

[6] Paul G. Brown. "Overview of sciDB: Large Scale Array Storage, Processing and Analysis". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. SIGMOD '10. Indianapolis, Indiana, USA: ACM, 2010, pp. 963–968. ISBN: 978-1-4503-0032-2. DOI: 10.1145/1807167.1807271. URL: http://doi.acm.org/10.1145/1807167.1807271.

[7] Qiming Chen, Meichun Hsu, and Rui Liu. "Extend UDF Technology for Integrated Analytics". In: *Data Warehousing and Knowledge Discovery*. Ed. by Torben Bach Pedersen, Mukesh K. Mohania, and A. Min Tjoa. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 256–270. ISBN: 978-3-642-03730-6.

[8] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. URL: http://doi.acm.org/10.1145/1327452.1327492.

[9] Sebastian Ertel, Christof Fetzer, and Pascal Felber. "Ohua: Implicit Dataflow Programming for Concurrent Systems". In: *Proceedings of the Principles and Practices of Programming on The Java Platform*. PPPJ '15. Melbourne, FL, USA: ACM, 2015, pp. 51–64. ISBN: 978-1-4503-3712-0. DOI: 10.1145/2807426.2807431. URL: http://doi.acm.org/10.1145/2807426.2807431.

[10] Sebastian Ertel et al. “Compiling for Concise Code and Efficient I/O”. In: *Proceedings of the 27th International Conference on Compiler Construction*. CC 2018. Vienna, Austria: ACM, 2018, pp. 104–115. ISBN: 978-1-4503-5644-2. DOI: 10.1145/3178372.3179505. URL: http://doi.acm.org/10.1145/3178372.3179505.

[11] Sebastian Ertel et al. “STCLang: State Thread Composition As a Foundation for Monadic Dataflow Parallelism”. In: *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*. Haskell 2019. Berlin, Germany: ACM, 2019, pp. 146–161. ISBN: 978-1-4503-6813-1. DOI: 10.1145/3331545.3342600. URL: http://doi.acm.org/10.1145/3331545.3342600.

[12] Eric Friedman, Peter Pawlowski, and John Cieslewicz. “SQL/MapReduce: A Practical Approach to Self-describing, Polymorphic, and Parallelizable User-defined Functions”. In: *Proc. VLDB Endow.* 2.2 (Aug. 2009), pp. 1402–1413. ISSN: 2150-8097. DOI: 10.14778/1687553.1687567. URL: https://doi.org/10.14778/1687553.1687567.

[13] Jon Gjengset et al. “Noria: dynamic, partially-stateful data-flow for high-performance web applications”. In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 213–231. ISBN: 978-1-939133-08-3. URL: https://www.usenix.org/conference/osdi18/presentation/gjengset.

[14] Philipp Große, Norman May, and Wolfgang Lehner. “A Study of Partitioning and Parallel UDF Execution with the SAP HANA Database”. In: *Proceedings of the 26th International Conference on Scientific and Statistical Database Management*. SSDBM ’14. Aalborg, Denmark: ACM, 2014, 36:1–36:4. ISBN: 978-1-4503-2722-0. DOI: 10.1145/2618243.2618274. URL: http://doi.acm.org/10.1145/2618243.2618274.

[15] Joseph M. Hellerstein et al. “The MADlib Analytics Library: Or MAD Skills, the SQL”. In: *Proc. VLDB Endow.* 5.12 (Aug. 2012), pp. 1700–1711. ISSN: 2150-8097. DOI: 10.14778/2367502.2367510. URL: http://dx.doi.org/10.14778/2367502.2367510.

[16] *Hive language manual - UDF*. URL: https://cwiki.apache.org/confluence/display/Hive/LanguageManual+UDF.

[17] Meichun Hsu et al. “Generalized UDF for Analytics Inside Database Engine”. In: *Web-Age Information Management*. Ed. by Lei Chen et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 742–754. ISBN: 978-3-642-14246-8.

[18] Michael Jaedicke and Bernhard Mitschang. “User-Defined Table Operators: Enhancing Extensibility for ORDBMS”. In: *VLDB’99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*. Ed. by Malcolm P. Atkinson et al. Edinburgh, Scotland, UK: Morgan Kaufmann, 1999, pp. 494–505. ISBN: 1-55860-615-7. URL: http://www.vldb.org/conf/1999/P47.pdf.

[19] Xupeng Li et al. “MLog: Towards Declarative In-database Machine Learning”. In: *Proc. VLDB Endow.* 10.12 (Aug. 2017), pp. 1933–1936. ISSN: 2150-8097. DOI: 10.14778/3137765.3137812. URL: https://doi.org/10.14778/3137765.3137812.

[20] Zhen Hua Liu et al. “Closing the Functional and Performance Gap Between SQL and NoSQL”. In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD ’16. San Francisco, California, USA: ACM, 2016, pp. 227–238. ISBN: 978-1-4503-3531-7. DOI: 10.1145/2882903.2903731. URL: http://doi.acm.org/10.1145/2882903.2903731.

[21] Hongjun Lu, Hock Chuan Chan, and Kwok Kee Wei. “A Survey on Usage of SQL”. In: *SIGMOD Rec.* 22.4 (Dec. 1993), pp. 60–65. ISSN: 0163-5808. DOI: 10.1145/166635.166656. URL: http://doi.acm.org/10.1145/166635.166656.

[22] Rob Pike et al. “Interpreting the data: Parallel analysis with Sawzall”. In: *Scientific Programming* 13.4 (2005), pp. 277–298.

[23] John Rhodes and Benjamin Steinberg. *The Q-theory of Finite Semigroups*. 1st. Springer Publishing Company, Incorporated, 2008. ISBN: 0387097805, 9780387097800.

[24] *User defined functions in MySQL*. URL: https://dev.mysql.com/doc/refman/8.0/en/adding-udf.html.

[25] *User defined functions in Postgres*. URL: https://www.postgresql.org/docs/current/xfunc-c.html.

[26] *User defined operators in Postgres*. URL: https://www.postgresql.org/docs/current/xoper.html.

[27] Charles Welty and David W. Stemple. "Human Factors Comparison of a Procedural and a Nonprocedural Query Language". In: *ACM Trans. Database Syst.* 6.4 (Dec. 1981), pp. 626–649. ISSN: 0362-5915. DOI: 10.1145/319628.319656. URL: http://doi.acm.org/10.1145/319628.319656.