Towards Scalable UDTF's in Noria

Justus Adam
Technische Universität Dresden
Chair for Compiler Construction
Dresden, Germany
me@justus.science

ABSTRACT

User Defined Functions are an important and powerful extension point for database queries. Systems using incremental materialized views largely do not support UDF's because they cannot easily be incrementalized.

In this work we design single-tuple UDF and UDA interfaces for Noria, a state-of-the art dataflow system with incremental materialized views. We also add limited support for User Defined Table Functions (UDTF), by compiling them to a query fragment with single-tuple and UDA operators. We show our UDTF's scale using one previously criticized in [6] for performing badly in SQL.

ACM Reference Format:
Justus Adam. 2019. Towards Scalable UDTF's in Noria. In *Proceedings of ACM Conference (Conference'17)*. ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Noria [7] is a novel database optimized for web applications. These heavily favor reads over writes. Noria pre-computes results for queries to speed up reads and applies writes incrementally. This technique is called a *materialized view* [9, 14], speeding up reads significantly at the expense of immediate consistency for writes.

Noria achieves further performance using parallelism. It is capable of leveraging multiple cores on a single machine, as well as using a cluster of machines. This is well suited for web applications, where requests are often distributed among several machines.

Views in the Noria database are currently defined using a SQL dialect. There is no support for UDF's yet. This limits which kinds of processing tasks can be done, as they have to be covered, in full, by SQL. Common tasks such as decoding or computing the distribution over a series of values can thus not be expressed.

In this work we try to remedy this situation. We add facilities for two basic kinds of UDF. Single-tuple UDF's and User Defined Aggregations (UDA). Single tuple UDF's can, for instance, express decoding tasks or unit conversions. UDA's cover, for instance, a distribution or variance calculation. In this work we propose a scheme for defining incremental single-tuple UDF's and UDA's. This is necessary for use in a materialized view, as results need to be continuously updated. We further integrate the state needed for a UDA into Norias materialization. This allows the engine to distribute the state along with the computations across cores and machines.

```
fn main(clicks: RowStream<i32, i23, i32>)
        -> GroupedRows<i32, i32> {
    let click_streams = group_by(0, clicks);
    for (uid, group_stream) in click_streams {
        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_average()
    }
}
```

Figure 1: Clickstream analysis UDTF in Ohua

We also go one step further and add support for a limited form of User Defined Table Functions (UDTF). UDTF's pose a significant risk to performance. To the engine they are black boxes, preventing optimization and parallelization. We apply insights from the domain of implicit parallel programming [3–5]. We compile the UDTA to a query fragment containing native database operators as well as automatically generated UDF's and UDA's. These fragments give the engine a more fine grained view of the UDTF. Using this we implement a *clickstream analysis* UDTF (Figure 1) adapted from [6]. Friedman et al. showed this query is difficult to write in pure SQL and performs poorly. Our UDTF version is simpler to read and write and the experiments show the engine is able to parallelize it.

2 RELATED WORK

Traditional materialized views [9, 12] recompute the entire query, which requires no changes to the UDF integration. Incrementally maintained materialized views [10, 14, 15] are more difficult because UDF's also need to be incremental.

DBToaster [1] does support UDF's, but only single-tuple ones, which are trivial to incrementalize. Mohapatra and Genesereth [13] support User Defined Aggregates only in so far as they have to also be defined in the query language Datalog, offering no effective integration for foreign code. Oracle [11] developed interfaces for incremental UDA's supporting foreign code. Our approach builds on a similar interface, but also automates part of incrementalizing the UDA.

To the best of our knowledge there is no proposal yet for UDTF's in incremental materialized views, especially scalable ones.

There are separate approaches to scalable UDTF's using MapReduce [6] and annotations [8]. Our approach uses a more general language than MapReduce [2] and is orthogonal to [8].

3 APPROACH

We incrementalize the UDF's according to their type. In general incremental operators compute over deltas. A delta pairs a value with a *sign* i.e. $(x, +)$ is an insert of value x, $(x, -)$ a delete.

Single tuple UDF's are incrementalized by propagating the sign. Let the UDF be f, then the incremental operator f' behaves as follows:

$$f'((x, +)) = (f(x), +)$$
$$f'((x, -)) = (f(x), -)$$

UDA's always have a state keeping track of the aggregate. To incrementalize we only need to incrementalize the state. To do this we require that all modifications to this state are reversible. The input value is used to compute a set of modifications to trigger. If the sign was positive the modifications are applied, otherwise we revert them instead. Finally the result is recomputed and an update issued downstream. With this not the entire UDA must be incremental, only the state.

UDTF's . We are most interested in stateful, procedural computations because SQL cannot express these. We generate single-tuple UDF's and UDA's for parts of the program and tie them together with SQL operators to an query fragment.

We need to faithfully recreate the procedural semantics in the query fragment. For this we use the parallelizing language and compiler Ohua [4]. Calls to external functions compile to UDF and UDA operators. A dataflow graph of the UDTF is created, the nodes of which are the generated operators. This Ohua specific graph is translated into MIR an intermediate representation for queries in Noria. When the UDTF is called, this MIR fragment is spliced into the query. Figure 2 shows an overview of the compile pipeline.

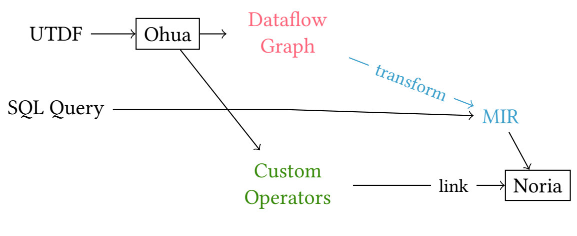

Figure 2: Compiler Pipeline

The two most important challenges for the compilation process are the representation of state and control flow.

State. is an important feature of our UDTF's, as it is absent in SQL and necessary for certain efficient algorithms [6]. Some of the operators that compute query results have private state, but there is no state sharing between operators. This is deliberate, because it requires synchronization and reduces parallelism. We make all state local by fusing parts of the UDTF such that we can generate a single operator per mutable state. This way the state is private to the operator and mutation is safe. The restrictions this imposes on how state can be shared are omitted here for brevity. We pair UDF state directly with the materialization, allowing the engine to distribute it. It also enables it to garbage-collect unused state, which Noria dose to reduce memory pressure.

Control flow. is supported in our source language as iteration (`for`) and conditional (`if`). Conditional execution can be represented as Noria dataflow using only the native operators `Filter` and `Join` and is omitted here for brevity.

Iteration is different, because the type of data flowing between backend operators is restricted to database tuples [1]. We therefore stream sequences like arrays and lists element-wise. Each item is tagged with a *scope key* that describes its position in the sequence. For instance the `group_by(grp_cols, table)` function tags each row with the current value of its `grp_cols`. Here the tag signifies the group it belongs to.

We use these *scope keys* to implement state scoping. This occurs when a state is instantiated in a `for` loop (Figure 1). The operator using the state must instantiate a new state for each iteration of the loop to satisfy the scope. When it processes a tuple, it selects the corresponding state from an internal map based on the *scope key* of the tuple. This lets us recreate procedural semantics and scope using runtime tagging.

4 RESULTS AND CONCLUSION

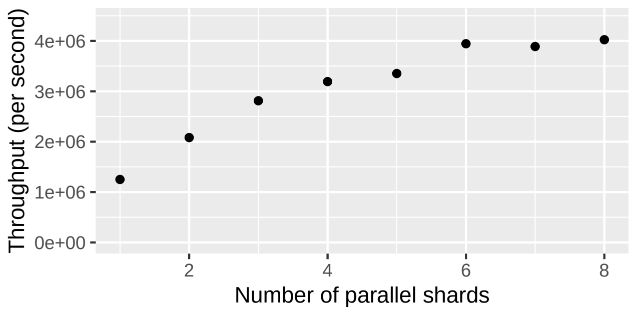

Figure 3: Scaling of the clickstream analysis UDTF over sharded data

We implement the efficient version of a clickstream analysis query from [6]. We show that the implementation effort is low using our approach. The generated query fragment can be called from SQL like typical UDF's. It parallelizes with Norias sharding capabilities, shown in Figure 3.

Our approach can express procedural UDF's as database query fragments. It allows for the use of state for efficient queries and respects control flow scope. These UDTF's are capable of leveraging parallelizing optimizations. Our approach could be supplemented with the ideas put forward in [8], further expanding the optimization potential.

[1]Heterogeneous arrays of SQL base types.

REFERENCES

[1] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. 2012. DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views. *Proc. VLDB Endow.* 5, 10 (June 2012), 968–979. https://doi.org/10.14778/2336664.2336670

[2] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (jan 2008), 107–113. https://doi.org/10.1145/1327452.1327492

[3] Sebastian Ertel, Justus Adam, Norman A. Rink, Andrés Goens, and Jeronimo Castrillon. 2019. STCLang: State Thread Composition As a Foundation for Monadic Dataflow Parallelism. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell (Haskell 2019)*. ACM, New York, NY, USA, 146–161. https://doi.org/10.1145/3331545.3342600

[4] Sebastian Ertel, Christof Fetzer, and Pascal Felber. 2015. Ohua: Implicit Dataflow Programming for Concurrent Systems. In *Proceedings of the Principles and Practices of Programming on The Java Platform (PPPJ '15)*. ACM, New York, NY, USA, 51–64. https://doi.org/10.1145/2807426.2807431

[5] Sebastian Ertel, Andrés Goens, Justus Adam, and Jeronimo Castrillon. 2018. Compiling for Concise Code and Efficient I/O. In *Proceedings of the 27th International Conference on Compiler Construction (CC 2018)*. ACM, New York, NY, USA, 104–115. https://doi.org/10.1145/3178372.3179505

[6] Eric Friedman, Peter Pawlowski, and John Cieslewicz. 2009. SQL/MapReduce: A Practical Approach to Self-describing, Polymorphic, and Parallelizable User-defined Functions. *Proc. VLDB Endow.* 2, 2 (aug 2009), 1402–1413. https://doi.org/10.14778/1687553.1687567

[7] Jon Gjengset, Malte Schwarzkopf, Jonathan Behrens, Lara Timbó Araújo, Martin Ek, Eddie Kohler, M. Frans Kaashoek, and Robert Morris. 2018. Noria: dynamic, partially-stateful data-flow for high-performance web applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 213–231. https://www.usenix.org/conference/osdi18/presentation/gjengset

[8] Philipp Große, Norman May, and Wolfgang Lehner. 2014. 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)*. ACM, New York, NY, USA, Article 36, 4 pages. https://doi.org/10.1145/2618243.2618274

[9] H. Gupta and I. S. Mumick. 2005. Selection of views to materialize in a data warehouse. *IEEE Transactions on Knowledge and Data Engineering* 17, 1 (Jan 2005), 24–43. https://doi.org/10.1109/TKDE.2005.16

[10] Himanshu Gupta and Inderpal Singh Mumick. 2006. Incremental maintenance of aggregate and outerjoin expressions. *Information Systems* 31, 6 (2006), 435 – 464. https://doi.org/10.1016/j.is.2004.11.011

[11] Ying Hu, Seema Sundara, and Jagannathan Srinivasan. 2010. Materialized views with user-defined aggregates.

[12] Ki Yong Lee and Myoung Ho Kim. 2005. Optimizing the Incremental Maintenance of Multiple Join Views. In *Proceedings of the 8th ACM International Workshop on Data Warehousing and OLAP (DOLAP '05)*. ACM, New York, NY, USA, 107–113. https://doi.org/10.1145/1097002.1097021

[13] Abhijeet Mohapatra and Michael Genesereth. 2014. Incremental maintenance of aggregate views. In *International Symposium on Foundations of Information and Knowledge Systems*. Springer, 399–414.

[14] Jingren Zhou, Per-Ake Larson, and Hicham G. Elmongui. 2007. Lazy Maintenance of Materialized Views. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB '07)*. VLDB Endowment, 231–242. http://dl.acm.org/citation.cfm?id=1325851.1325881

[15] Yue Zhuge, Héctor García-Molina, Joachim Hammer, and Jennifer Widom. 1995. View Maintenance in a Warehousing Environment. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data (SIGMOD '95)*. ACM, New York, NY, USA, 316–327. https://doi.org/10.1145/223784.223848