Justus Adam

Informatik, Compiler Construction, Supervisor: Sebastian Ertel

Addressing current challenges for databases with deeply integrated UDF’s

Profilmodul Grundlagenforschung // Dresden, March 28, 2019

Landscape of Data Management [a, b]

---

[a] Daniel Abadi et al. “The Beckman Report on Database Research”. In: *SIGMOD Rec.* 43.3 (12/2014), pp. 61–70

[b] Rakesh Agrawal et al. “The Claremont Report on Database Research”. In: *SIGMOD Rec.* 37.3 (09/2008), pp. 9–19

Landscape of Data Management [a, b]

Unstructured Data

- 90% of "Big Data" is unstructured [c]
- Unsuitable for a rigid schema

---

[c] Alexander Behm et al. "ASTERIX: towards a scalable, semistructured data platform for evolving-world models". In: *Distributed and Parallel Databases* 29.3 (06/2011), pp. 185–216

Landscape of Data Management [a, b]

Unstructured Data

- 90% of “Big Data” is unstructured [c]
- Unsuitable for a rigid schema

Non SQL Languages

- SQL is large
- Queries are hard to read [d]
- Unsuitable for certain problems [e]

---

[d] Hongjun Lu, Hock Chuan Chan, and Kwok Kee Wei. “A Survey on Usage of SQL”. In: *SIGMOD Rec.* 22.4 (12/1993), pp. 60–65

[e] Charles Welty and David W. Stemple. “Human Factors Comparison of a Procedural and a Nonprocedural Query Language”. In: *ACM Trans. Database Syst.* 6.4 (12/1981), pp. 626–649

```
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) = 'X' AND
               pagetype(c2.page_id) = 'Y'
            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;
```

[f]

---

[f] 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 (08/2009), pp. 1402–1413

```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) = ’X’ AND
               pagetype(c2.page_id) = ’Y’
            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;
```

[f]

```rust
fn click_ana(start_cat: Category,
             end_cat: Category,
             clicks: &mut Vec<(UID, Category, Time)>) -> i32 {
  let stream = HashMap::new();
  let distances = for (uid, cat, _) in clicks {
    let prev = stream.get(uid);
    if cat == start_cat {
      stream.insert(uid, 0);
      None
    } else if prev != -1 {
      if cat == end_cat {
        stream.insert(uid, -1);
        Some(prev)
      } else {
        stream.insert(uid, prev + 1);
        None
      }
    }
    None
  }
  average(drop_none(distances))
}
```

---

[f] 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 (08/2009), pp. 1402–1413

Landscape of Data Management [a, b]

Unstructured Data

- 90% of “Big Data” is unstructured [c]
- Unsuitable for a rigid schema

Non SQL Languages

- SQL is large
- Queries are hard to read [d]
- Unsuitable for certain problems [e]

Landscape of Data Management ^{a,b}

Unstructured Data

Non SQL Languages

UDF's

Landscape of Data Management [a, b]

Unstructured Data

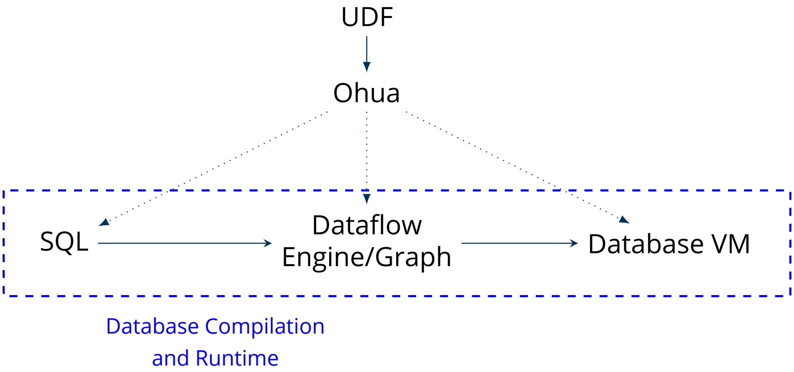
Non SQL Languages

Parallelism

UDF’s

- Black box approach
- No reordering
- No partitioning

Our Approach


UDF

Ohua

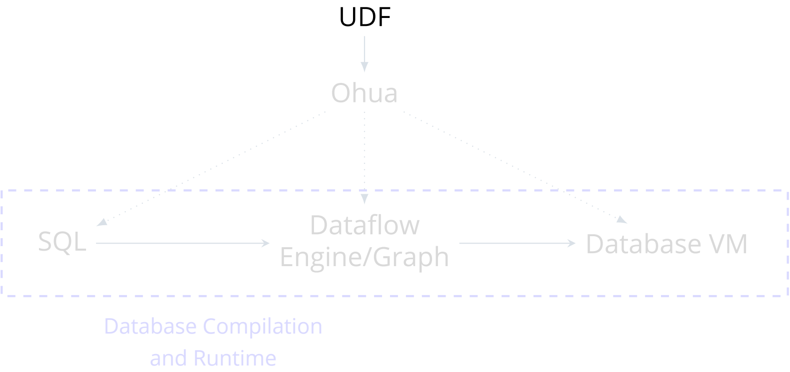
SQL

Dataflow Engine/Graph

Database VM

Database Compilation and Runtime

Our Approach



UDF

Ohua

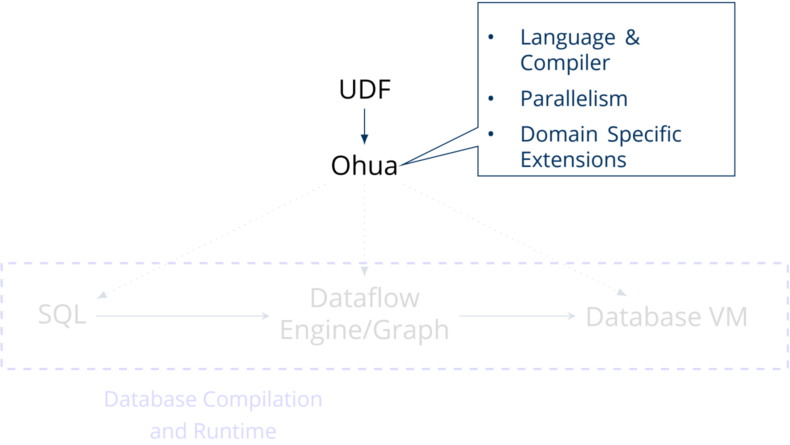
SQL

Dataflow Engine/Graph

Database VM

Database Compilation and Runtime

Our Approach



UDF

Ohua

- Language & Compiler
- Parallelism
- Domain Specific Extensions

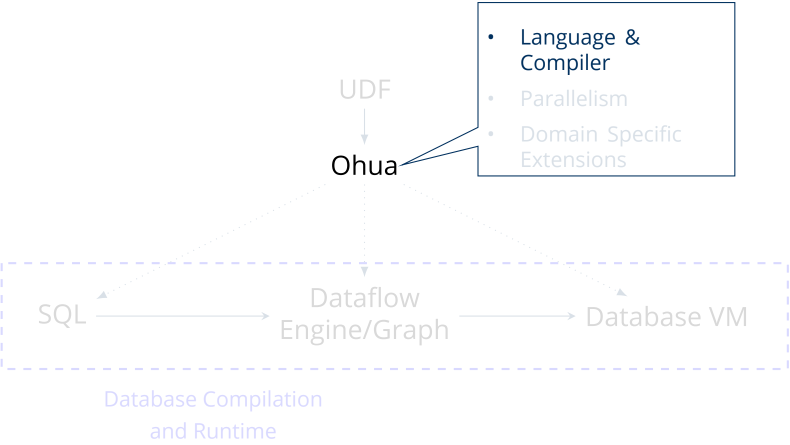
SQL

Dataflow Engine/Graph

Database VM

Database Compilation and Runtime

Our Approach



UDF

Ohua

- Language & Compiler
- Parallelism
- Domain Specific Extensions

SQL

Dataflow Engine/Graph

Database VM

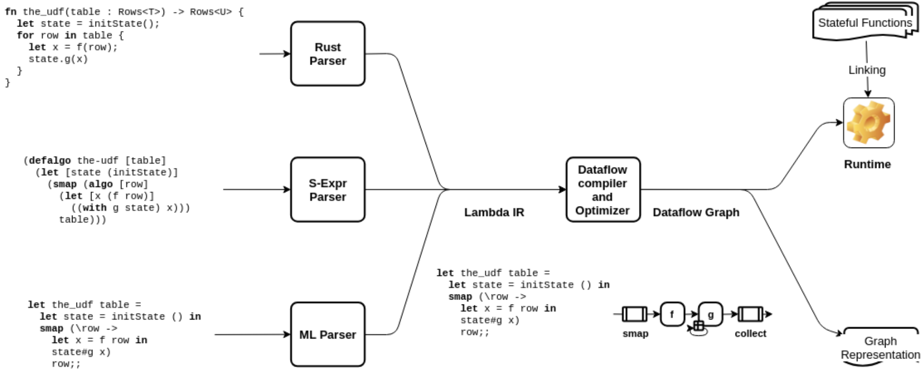
Database Compilation and Runtime

Ohua

```
fn the_udf(table : Rows<T>)
     -> Rows<U> {
  let state = initState();
  for row in table {
    let x = f(row);
    state.g(x)
  }
}
```

Figure 1: A simple Ohua program

Ohua



```
fn the_udf(table : Rows<T>) -> Rows<U> {
  let state = initState();
  for row in table {
    let x = f(row);
    state.g(x)
  }
}
```

```
(defalgo the-udf [table]
  (let [state (initState)]
    (smap (algo [row]
      (let [x (f row)]
        ((with g state) x)))
      table)))
```

```
let the_udf table =
  let state = initState () in
  smap (\row ->
    let x = f row in
    state#g x)
    row;;
```

```
let the_udf table =
  let state = initState () in
  smap (\row ->
    let x = f row in
    state#g x)
    row;;
```

Figure 2: Ohua Compiler Flow

Ohua

```
fn the_udf(table : Rows<T>)
     -> Rows<U> {
  let state = initState();
  for row in table {
    let x = f(row);
    state.g(x)
  }
}
```

Figure 1: A simple Ohua program

Ohua

```
fn the_udf(table : Rows<T>)
     -> Rows<U> {
  let state = initState();
  for row in table {
    let x = f(row);
    state.g(x)
  }
}
```

Figure 1: A simple Ohua program

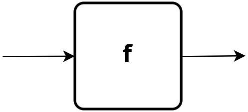


Figure 2: Dataflow graph for `the_udf`

Ohua

```
fn the_udf(table : Rows<T>)
     -> Rows<U> {
  let state = initState();
  for row in table {
    let x = f(row);
    state.g(x)
  }
}
```

Figure 1: A simple Ohua program

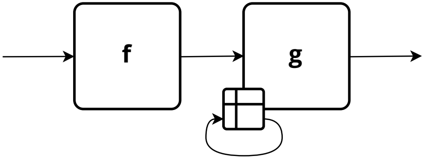


Figure 2: Dataflow graph for `the_udf`

Ohua

```
fn the_udf(table : Rows<T>)
     -> Rows<U> {
  let state = initState();
  for row in table {
    let x = f(row);
    state.g(x)
  }
}
```

Figure 1: A simple Ohua program

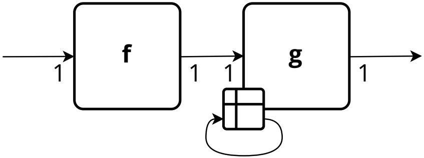


Figure 2: Dataflow graph for `the_udf`

Ohua

```
fn the_udf(table : Rows<T>)
     -> Rows<U> {
  let state = initState();
  for row in table {
    let x = f(row);
    state.g(x)
  }
}
```

Figure 1: A simple Ohua program

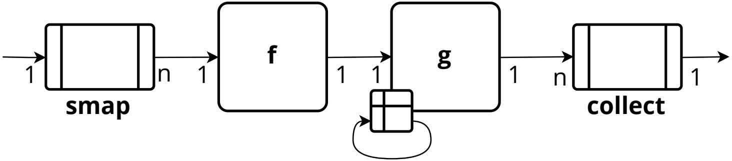
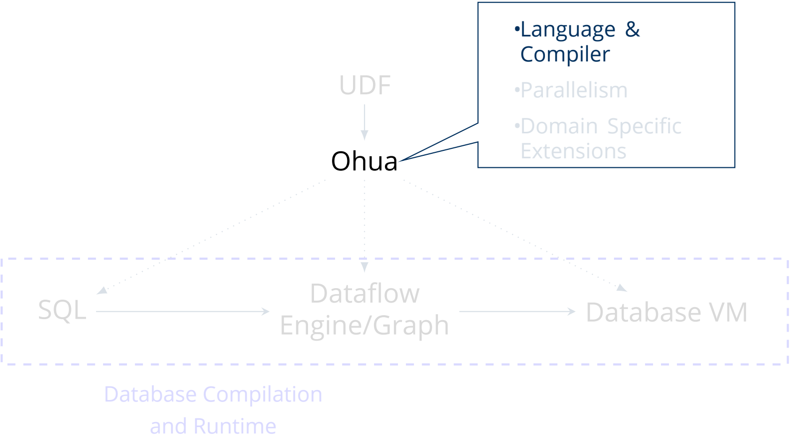


Figure 2: Dataflow graph for `the_udf`

Parallelism



- Language & Compiler
- Parallelism
- Domain Specific Extensions

UDF

Ohua

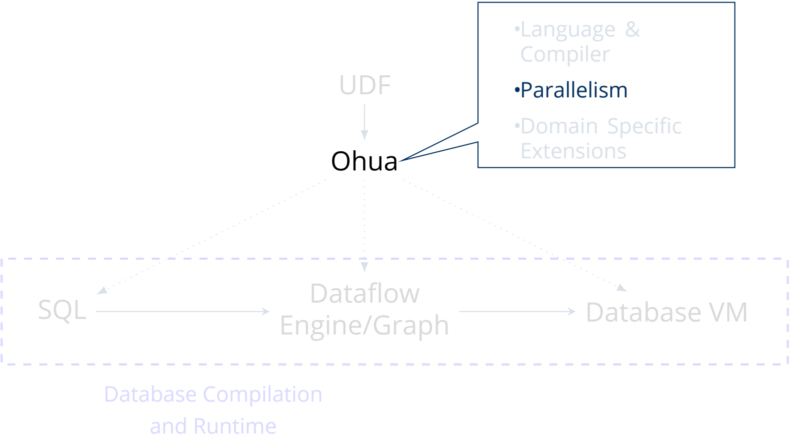
SQL

Dataflow Engine/Graph

Database VM

Database Compilation and Runtime

Parallelism



- Language & Compiler
- Parallelism
- Domain Specific Extensions

UDF

Ohua

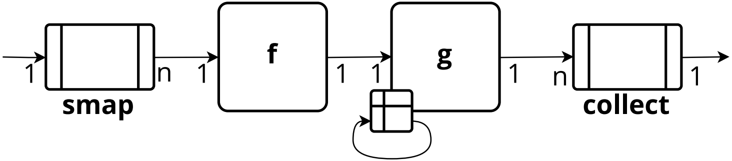
SQL

Dataflow Engine/Graph

Database VM

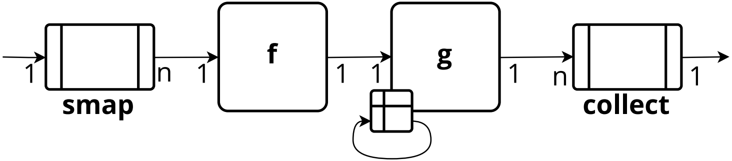
Database Compilation and Runtime

Parallelism



Parallelism

- Pipeline Parallelism
- Task level parallelism

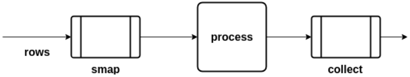


Implicit Partitioning

```
fn the_udf(table: Rows) {
  for row in table {
    process(row)
  }
}
```

Figure 3: Pure iteration

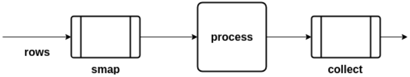
Implicit Partitioning



```
fn the_udf(table: Rows) {
  for row in table {
    process(row)
  }
}
```

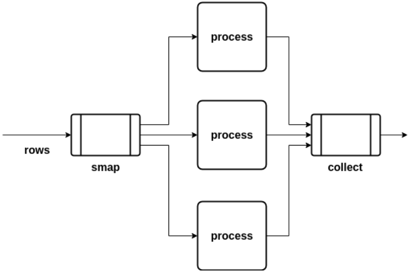
Figure 3: Pure iteration

Implicit Partitioning



```
fn the_udf(table: Rows) {
  for row in table {
    process(row)
  }
}
```

Figure 3: Pure iteration



Implicit Partitioning

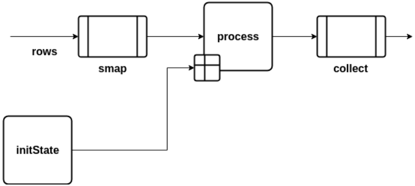
```
fn the_udf(table: Rows<T>) -> Rows<U> {
  let state = initState();
  for row in table {
    state.process(row)
  }
}
```

Figure 4: Stateful iteration

Implicit Partitioning

```
fn the_udf(table: Rows<T>) -> Rows<U> {
  let state = initState();
  for row in table {
    state.process(row)
  }
}
```

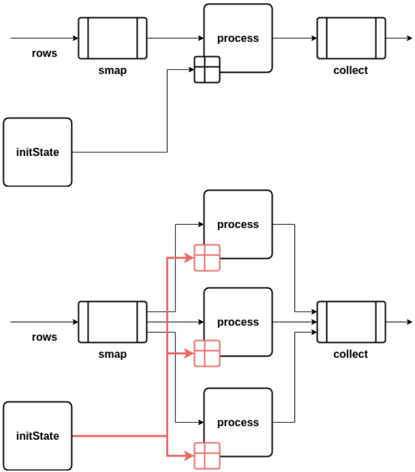
Figure 4: Stateful iteration



Implicit Partitioning

```
fn the_udf(table: Rows<T>) -> Rows<U> {
  let state = initState();
  for row in table {
    state.process(row)
  }
}
```

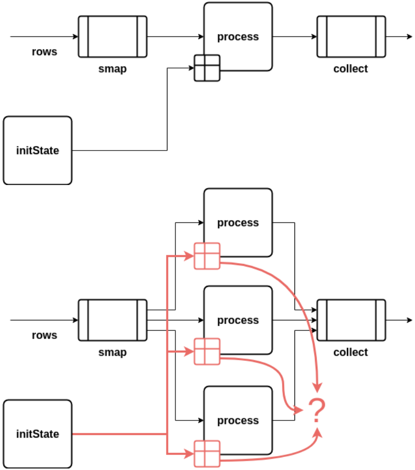
Figure 4: Stateful iteration

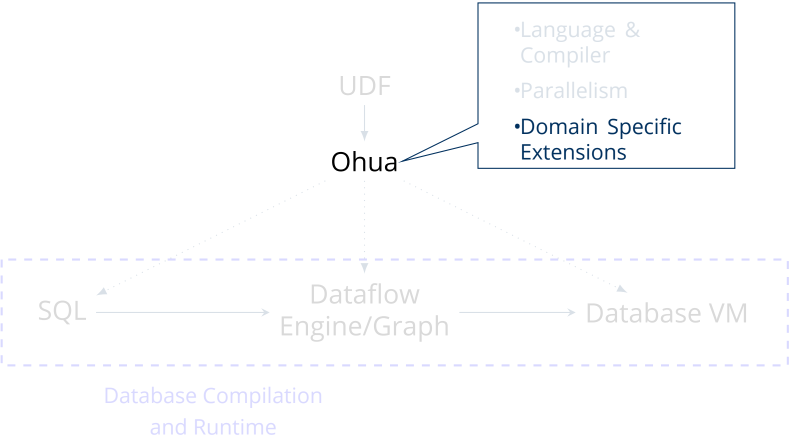


Implicit Partitioning

```
fn the_udf(table: Rows<T>) -> Rows<U> {
  let state = initState();
  for row in table {
    state.process(row)
  }
}
```

Figure 4: Stateful iteration





UDF

Ohua

- Language & Compiler
- Parallelism
- Domain Specific Extensions

SQL

Dataflow Engine/Graph

Database VM

Database Compilation and Runtime

Simple Explicit Partitioning

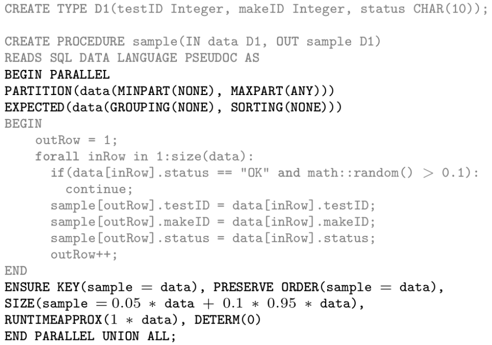
```
ohua::sql::partition(lowerBound: u32,
                     upperBound: u32,
                     rows: Rows<T>) -> Iter<Rows<T>>
```

Figure 5: The `partition` primitive

```
fn the_udf(table: Rows<i32>) -> i32 {
  let results = for p in partition(100, 1000, table) {
    sum(p)
  };
  sum(results)
}
```

Figure 6: Partitioned iteration

Domain Specific Extensions



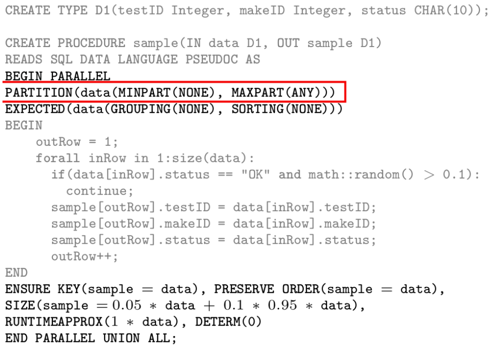
```
CREATE TYPE D1(testID Integer, makeID Integer, status CHAR(10));

CREATE PROCEDURE sample(IN data D1, OUT sample D1)
READS SQL DATA LANGUAGE PSEUDOC AS
BEGIN PARALLEL
PARTITION(data(MINPART(NONE), MAXPART(ANY)))
EXPECTED(data(GROUPING(NONE), SORTING(NONE)))
BEGIN
    outRow = 1;
    forall inRow in 1:size(data):
      if(data[inRow].status == "OK" and math::random() > 0.1):
        continue;
      sample[outRow].testID = data[inRow].testID;
      sample[outRow].makeID = data[inRow].makeID;
      sample[outRow].status = data[inRow].status;
      outRow++;
END
ENSURE KEY(sample = data), PRESERVE ORDER(sample = data),
SIZE(sample = 0.05 ∗ data + 0.1 ∗ 0.95 ∗ data),
RUNTIMEAPPROX(1 ∗ data), DETERM(0)
END PARALLEL UNION ALL;
```

Figure 7: SQLScript with annotations [1]

---

[1] (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)

Domain Specific Extensions



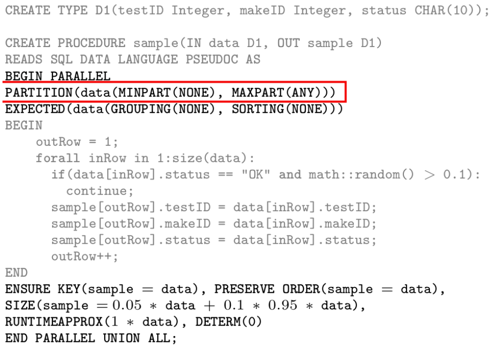
```
CREATE TYPE D1(testID Integer, makeID Integer, status CHAR(10));

CREATE PROCEDURE sample(IN data D1, OUT sample D1)
READS SQL DATA LANGUAGE PSEUDOC AS
BEGIN PARALLEL
PARTITION(data(MINPART(NONE), MAXPART(ANY)))
EXPECTED(data(GROUPING(NONE), SORTING(NONE)))
BEGIN
    outRow = 1;
    forall inRow in 1:size(data):
      if(data[inRow].status == "OK" and math::random() > 0.1):
        continue;
      sample[outRow].testID = data[inRow].testID;
      sample[outRow].makeID = data[inRow].makeID;
      sample[outRow].status = data[inRow].status;
      outRow++;
END
ENSURE KEY(sample = data), PRESERVE ORDER(sample = data),
SIZE(sample = 0.05 * data + 0.1 * 0.95 * data),
RUNTIMEAPPROX(1 * data), DETERM(0)
END PARALLEL UNION ALL;
```

Figure 7: SQLScript with annotations [1]

---

[1] (Große, May, and Lehner, “A Study of Partitioning and Parallel UDF Execution with the SAP HANA Database”)

Domain Specific Extensions



```
CREATE TYPE D1(testID Integer, makeID Integer, status CHAR(10));

CREATE PROCEDURE sample(IN data D1, OUT sample D1)
READS SQL DATA LANGUAGE PSEUDOC AS
BEGIN PARALLEL
PARTITION(data(MINPART(NONE), MAXPART(ANY)))
EXPECTED(data(GROUPING(NONE), SORTING(NONE)))
BEGIN
    outRow = 1;
    forall inRow in 1:size(data):
      if(data[inRow].status == "OK" and math::random() > 0.1):
        continue;
      sample[outRow].testID = data[inRow].testID;
      sample[outRow].makeID = data[inRow].makeID;
      sample[outRow].status = data[inRow].status;
      outRow++;
END
ENSURE KEY(sample = data), PRESERVE ORDER(sample = data),
SIZE(sample = 0.05 ∗ data + 0.1 ∗ 0.95 ∗ data),
RUNTIMEAPPROX(1 ∗ data), DETERM(0)
END PARALLEL UNION ALL;
```

Figure 7: SQLScript with annotations [1]

```
ohua::sql::partitionOn(
    min_part: Grouping,
    max_part: Grouping,
    rows: Rows<T>) -> Iter<Rows<T>>
```

Figure 8: The `partitionOn` primitive

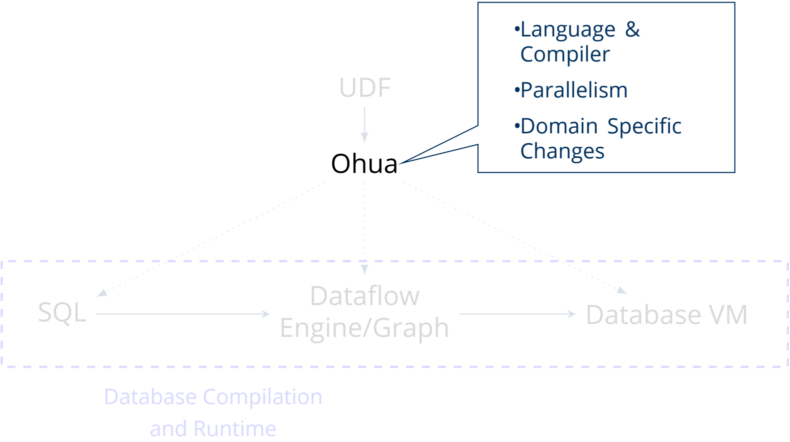
---

[1] (Große, May, and Lehner, “A Study of Partitioning and Parallel UDF Execution with the SAP HANA Database”)

Grouping Partitions

```
ohua::sql::partitionOn(min_part: Grouping,
                       max_part: Grouping,
                       rows: Rows<T>) -> Iter<Rows<T>>

fn the_udf(table: Rows<(Key,Key,i32)>) -> Rows<((Key, Key), i32)> {
  let results =
    for (group, p) in partitionOn("col1, col2", "none", rows) {
      let s = sumColumn("col3", p);
      (group, s)
    };
  concat(results)
}
```

Figure 9: Key-partitioned iteration



UDF

Ohua

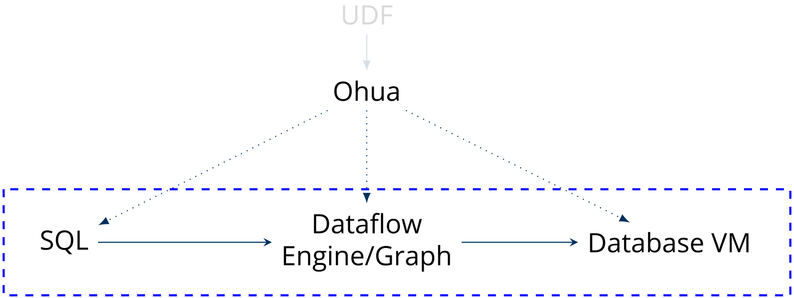
- Language & Compiler
- Parallelism
- Domain Specific Changes

SQL

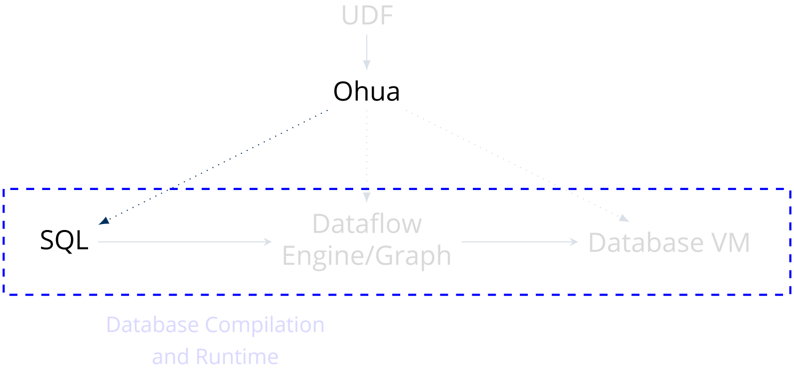
Dataflow Engine/Graph

Database VM

Database Compilation and Runtime



Database Compilation and Runtime


UDF

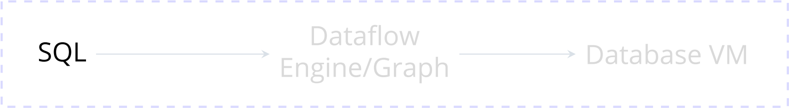
Ohua

SQL

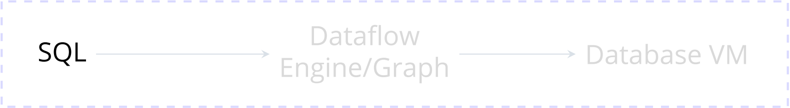
Dataflow Engine/Graph

Database VM

Database Compilation and Runtime



SQL → Dataflow Engine/Graph → Database VM

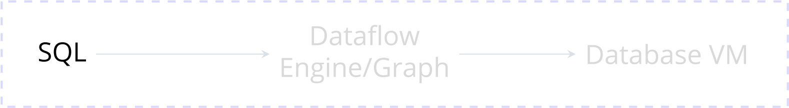

SQL

Dataflow Engine/Graph

Database VM

```
fn the_udf(table : Rows<T>) -> Rows<U> {
  for row in table {
    f(row);
  }
}
```

```
SELECT f(the_column) FROM table;
```


SQL → Dataflow Engine/Graph → Database VM

```
fn the_udf(table : Rows<T>) -> Rows<U> {
  for p in partitionOn("none"
                       "col1, col2",
                       table) {
      f(p);
  }
}
```

```
SELECT f(the_column) FROM table;

CREATE PROCEDURE f(IN data T, OUT outp U)
BEGIN PARALLEL
PARTITION(data(MINPART(NONE)
               MAXPART(col1, col2)))
BEGIN ...
```

SQL → Dataflow Engine/Graph → Database VM

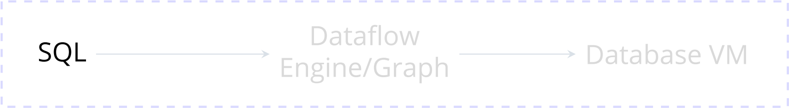
```
fn the_udf(table : Rows<T>) -> Rows<U> {
  for row in table {
    let x = f(row);
    g(x)
  }
}
```

```
SELECT g(temp_table.temp_col) FROM
  (SELECT f(the_column) as temp_col
   FROM table) temp_table;
```

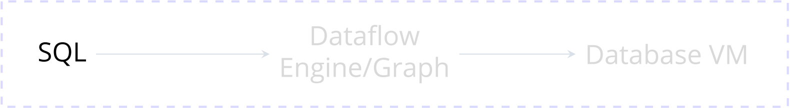
SQL → Dataflow Engine/Graph → Database VM

```
fn the_udf(table : Rows<T>) -> Rows<U> {
  for row in table {
    let x = f(row);
    g(x)
  }
}
```

```
SELECT g(temp_table.temp_col) FROM
  (SELECT f(the_column) as temp_col
   FROM table) temp_table;

SELECT g(f(the_column)) FROM table
```



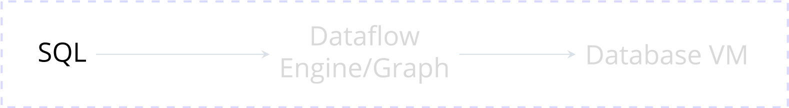
```
fn the_udf(table : Rows<T>) -> Rows<U> {
  let state = initState();
  for row in table {
    let x = f(row);
    state.g(x)
  }
}
```



SQL → Dataflow Engine/Graph → Database VM

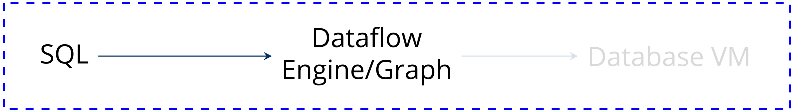
```
fn the_udf(table : Rows<T>) -> Rows<U> {
  let state = initState();
  for row in table {
    let x = f(row);
    state.g(x)
  }
}
```

?


SQL → Dataflow Engine/Graph → Database VM

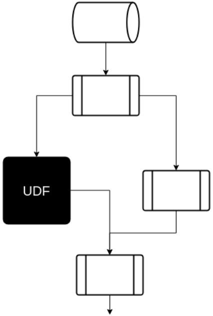
Arguments

- + Most general and portable
- + Approach to partitioning already exists
- - Feasibility and performance unclear



SQL → Dataflow Engine/Graph → Database VM

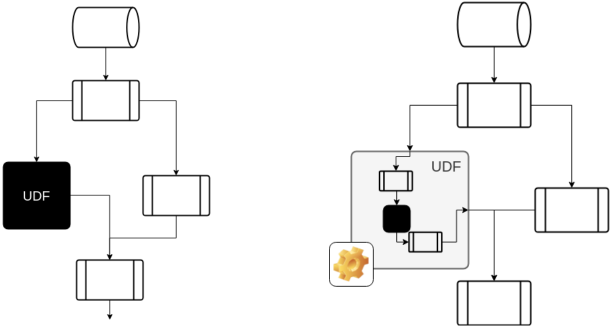
SQL → Dataflow Engine/Graph → Database VM



(a) Black Box

Figure 10: Step-wise integration

SQL → Dataflow Engine/Graph → Database VM

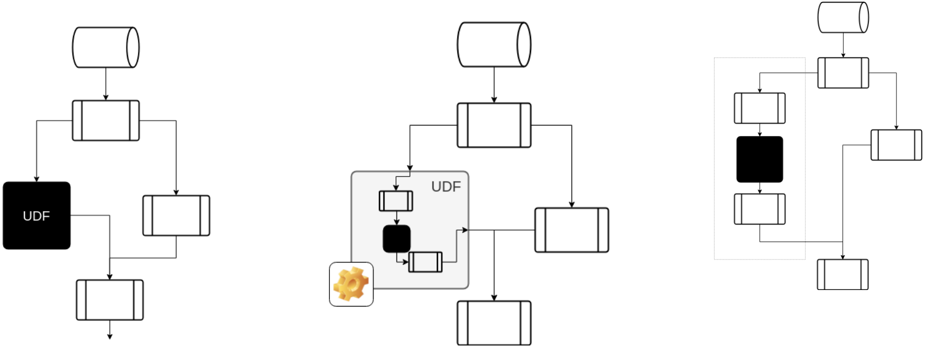


(a) Black Box

(b) Single Node

Figure 10: Step-wise integration

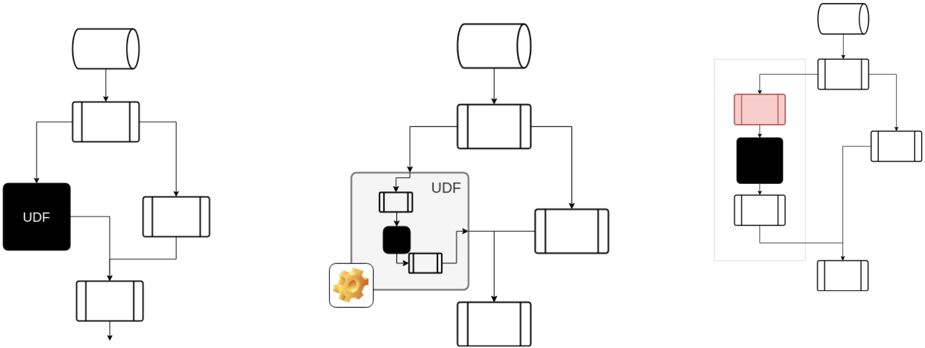
SQL → Dataflow Engine/Graph → Database VM



(a) Black Box (b) Single Node (c) Multi Node

Figure 10: Step-wise integration

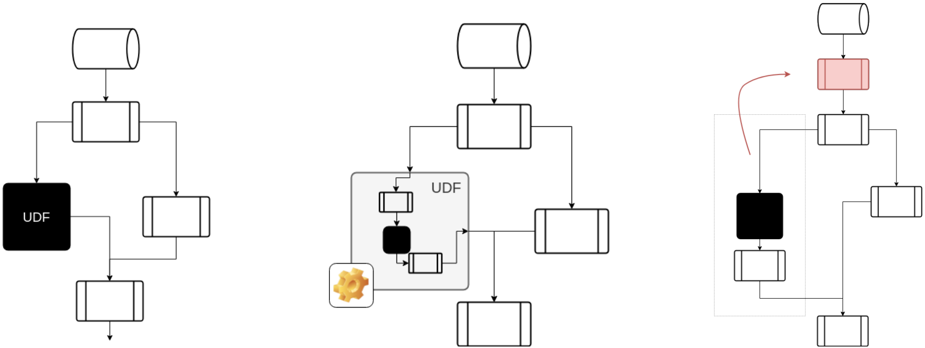
SQL → Dataflow Engine/Graph → Database VM



(a) Black Box (b) Single Node (c) Multi Node

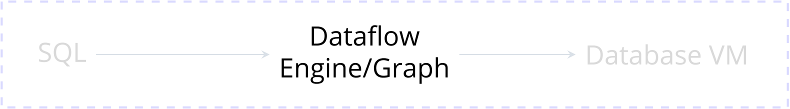
Figure 10: Step-wise integration

SQL → Dataflow Engine/Graph → Database VM



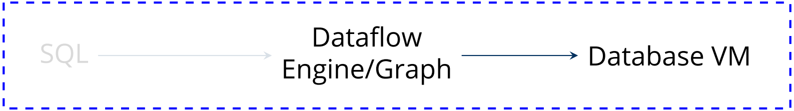
(a) Black Box (b) Single Node (c) Multi Node

Figure 10: Step-wise integration

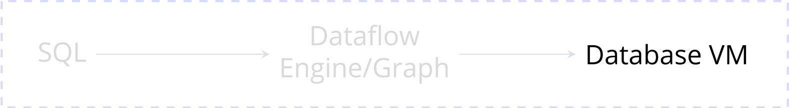

SQL → Dataflow Engine/Graph → Database VM

Arguments

- + Access to DB optimisations
- + Performance easier to estimate



SQL → Dataflow Engine/Graph → Database VM


SQL → Dataflow Engine/Graph → Database VM

Arguments

- - No portability
- - No access to query planner optimizations

Tentative Evaluation Ideas

- Map-reduce ⇒ Wordcount[1]
- Clickstream analysis[2]
- Connected component analysis[3]
  ⇒ Effectiveness of partitioning

---

[1] Feng Li et al. “Distributed Data Management Using MapReduce”. In: *ACM Comput. Surv.* 46.3 (01/2014), 31:1–31:42.

[2] Behm et al., “ASTERIX: towards a scalable, semistructured data platform for evolving-world models”.

[3] Große, May, and Lehner, “A Study of Partitioning and Parallel UDF Execution with the SAP HANA Database”.

Related Work: UDF Integration

Große, May, and Lehner

Annotate black box UDF’s with partitioning information to derive data parallelism.[4]

Friedman, Pawlowski, and Cieslewicz

Uses mapreduce paradigm for UDF’s to enable stateful data parallelism.[5]

---

[4] Große, May, and Lehner, “A Study of Partitioning and Parallel UDF Execution with the SAP HANA Database”.

[5] Friedman, Pawlowski, and Cieslewicz, “SQL/MapReduce: A Practical Approach to Self-describing, Polymorphic, and Parallelizable User-defined Functions”.

Related Work: Miscellaneous

Unstructured Data

Non SQL Languages

Parallelism

Related Work: Miscellaneous

Unstructured Data Non SQL Languages Parallelism

- JSON in Oracle [a]
- Asterix [b]

---

[a] 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

[b] Alexander Behm et al. “ASTERIX: towards a scalable, semistructured data platform for evolving-world models”. In: *Distributed and Parallel Databases* 29.3 (06/2011), pp. 185–216

Related Work: Miscellaneous

Unstructured Data

- JSON in Oracle [a]
- Asterix [b]

Non SQL Languages

- Pig Latin [c]
- Sawzall [d]

Parallelism

---

[c] Christopher Olston et al. “Pig Latin: A Not-so-foreign Language for Data Processing”. In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’08. Vancouver, Canada: ACM, 2008, pp. 1099–1110

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

Related Work: Miscellaneous

Unstructured Data

- JSON in Oracle [a]
- Asterix [b]

Non SQL Languages

- Pig Latin [c]
- Sawzall [d]

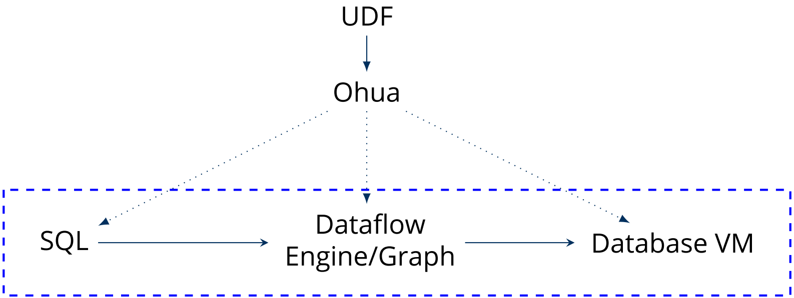
Parallelism

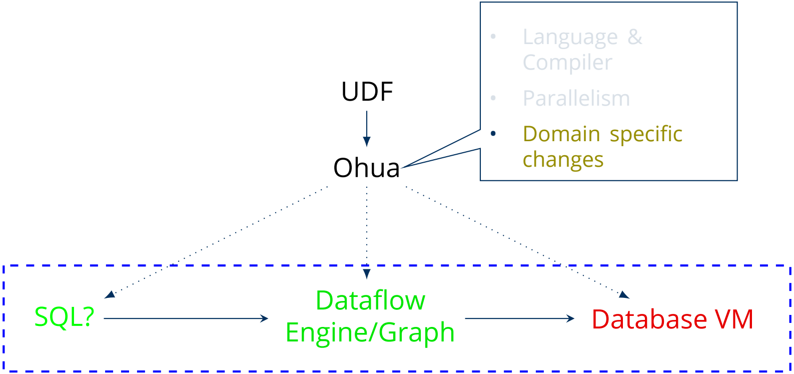
- MapReduce [e]
- Hive [f]

---

[e] Feng Li et al. “Distributed Data Management Using MapReduce”. In: *ACM Comput. Surv.* 46.3 (01/2014), 31:1–31:42

[f] Ashish Thusoo et al. “Hive: A Warehousing Solution over a Map-reduce Framework”. In: *Proc. VLDB Endow.* 2.2 (08/2009), pp. 1626–1629


UDF
Ohua
SQL
Dataflow Engine/Graph
Database VM



UDF

Ohua

- Language & Compiler
- Parallelism
- Domain specific changes

SQL?

Dataflow Engine/Graph

Database VM

Thank you for listening.