Enabling Operator Reordering in Data Flow Programs Through Static Code Analysis

XLDI 2012

Fabian Hueske, **Aljoscha Krettek**, Kostas Tzoumas

Database Systems and Information Management
Technische Universität Berlin

aljoscha.krettek@campus.tu-berlin.de

September 9th 2012

Agenda

Motivation

Operator Reordering

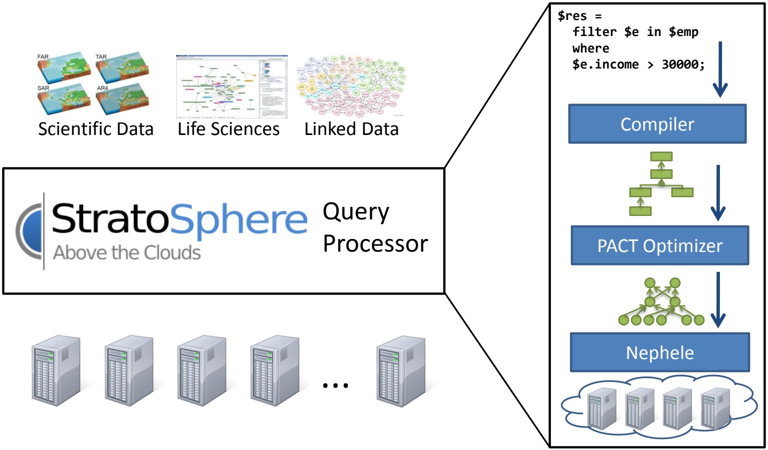
Static Code Analysis

Conclusion

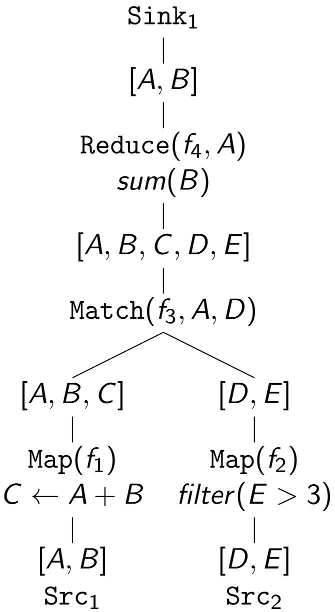
Motivation: Big Data Analytics

- "Big Data" revolution
  - Huge amounts of machine- and human- generated data, often semi-structured
  - Need for "deep" analytics beyond simple BI queries
- Breed of new parallel data management systems
  - Hadoop, Stratosphere, Asterix, SCOPE, etc.
- Common themes in programming models
  - Data flows composed (in part) of functions written in arbitrary imperative code
  - Also seen in modern MPP SQL systems (Greenplum, Aster)
  - Allows more powerful analytics on diverse data sets

Stratosphere

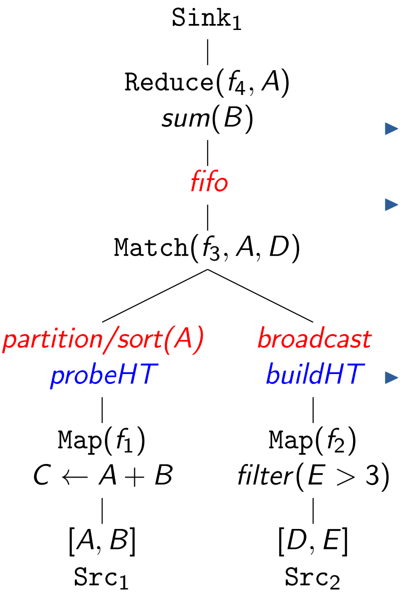


The PACT Programming Model



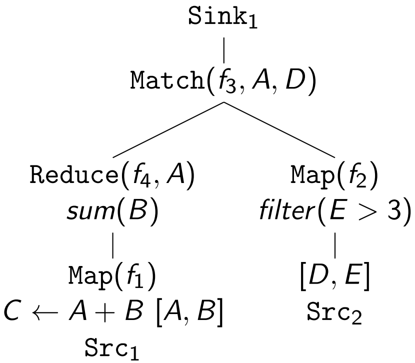
- Generalization of MapReduce
- Data flow consisting of data sources, sinks, and operators
- Operators consist of
  - *Second-order function* signature from a fixed set of system-defined SOFs - *PArallelization ConTracts*
  - *First-order function* written by programmer in Java
- Intermediate representation, but also exposed to the user
  - E.g., to implement functionality not supported by query language

Automatic Parallelization



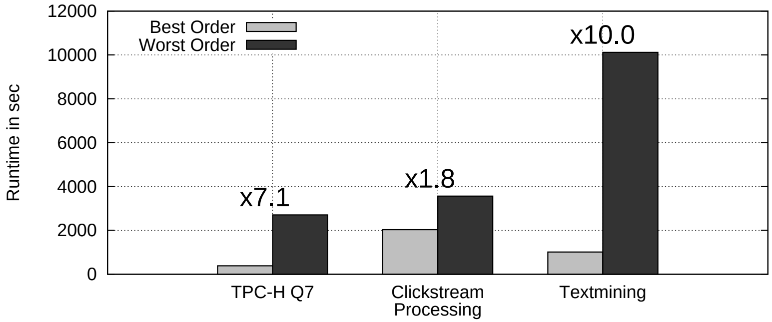
- Knowledge of PACT signature permits automatic parallelization
- E.g., for Match operator
  - Choice of broadcast, partition, SFR, etc
  - Sort-merge or hash-based physical implementation
- Cascades-style optimizer
  - Partitioning strategies propagated top-down as interesting properties

Need for Operator Reordering



- Operator reordering may reduce amount of intermediate data sets
- May introduce new opportunities for parallelization strategies
- For optimal execution, need to consider operator order, parallelization strategies, and physical execution in one step
- SOF signature not enough - need to look inside FOF

Experimental Results



Reordering Conditions

We can reorder operators when we know some specific properties of the user defined code.[1]

Define:

- Read set: Attributes that might influence FOFs output
- Write set: Attributes that might have different value after application of FOF

Example, Map-Map reordering:

- Two Map operators can be reordered if the FOFs operate on distinct values or have only read-read conflicts

Too cumbersome to ask programmer to specify read and write sets, therefore we want to estimate them using static code analysis on generic FOFs

[1] Opening the Black Boxes in Data Flow Optimization (VLDB 2012)

Example FOF

```
void match(Record left,
           Record right,
           Collector col) {
    Record out = copy(left);
    if (right.get(F) > 3) {
        out.set(D, right.get(D));
    } else {
        out.setNull(A);
    }
    out.set(E, right.get(E));
    out.set(F, 42);
    col.emit(out);
}
```

Fixed API for dealing with records: `create`, `copy`, `get`, `set`, `setNull`, and `union`.

Read set is easily determined by looking at all get statements. Write set depends on the schema of the data:

- Determine four other sets: origin, write, copy, projection
- Generate final write set from these and schema information

Example FOF (cont.)


```
void match(Record left,
           Record right,
           Collector col) {
    Record out = copy(left);
    if (right.get(F) > 3) {
        out.set(D, right.get(D));
    } else {
        out.setNull(A);
    }
    out.set(E, right.get(E));
    out.set(F, 42);
    col.emit(out);
}
```

Schema:
Left [A,B,C], Right [D,E,F]

Origin: $\{1\}$
Explicit projection$_l$: $\{A\}$
Explicit copy$_r$: $\{E\}$
Explicit write$_l$: $\{F\}$
Explicit write$_r$: $\{\}$

Final write set$_l$: $\{A, F\}$
Final write set$_r$: $\{D, F\}$

Example FOF (cont.)

```
void match(Record left,
           Record right,
           Collector col) {
    Record out = copy(left);
    if (right.get(F) > 3) {
        out.set(D, right.get(D));
    } else {
        out.setNull(A);
    }
    out.set(E, right.get(E));
    out.set(F, 42);
    col.emit(out);
}
```

Schema:
Left [A,B,C], Right [D,E,F]

Origin: $\{1\}$
Explicit projection$_l$: $\{A\}$
Explicit copy$_r$: $\{E\}$
Explicit write$_l$: $\{F\}$
Explicit write$_r$: $\{\}$

Final write set$_l$: $\{A, F\}$
Final write set$_r$: $\{D, F\}$

Example FOF (cont.)

```
void match(Record left,
           Record right,
           Collector col) {
    Record out = copy(left);
    if (right.get(F) > 3) {
        out.set(D, right.get(D));
    } else {
        out.setNull(A);
    }
    out.set(E, right.get(E));
    out.set(F, 42);
    col.emit(out);
}
```

Schema:
Left [A,B,C], Right [D,E,F]

Origin: $\{1\}$
Explicit projection$_l$: $\{A\}$
Explicit copy$_r$: $\{E\}$
Explicit write$_l$: $\{F\}$
Explicit write$_r$: $\{\}$

Final write set$_l$: $\{A, F\}$
Final write set$_r$: $\{D, F\}$

Example FOF (cont.)


```
void match(Record left,
           Record right,
           Collector col) {
    Record out = copy(left);
    if (right.get(F) > 3) {
        out.set(D, right.get(D));
    } else {
        out.setNull(A);
    }
    out.set(E, right.get(E));
    out.set(F, 42);
    col.emit(out);
}
```

Schema:
Left [A,B,C], Right [D,E,F]

Origin: $\{1\}$
Explicit projection$_l$: $\{A\}$
Explicit copy$_r$: $\{E\}$
Explicit write$_l$: $\{F\}$
Explicit write$_r$: $\{\}$

Final write set$_l$: $\{A, F\}$
Final write set$_r$: $\{D, F\}$

Example FOF (cont.)

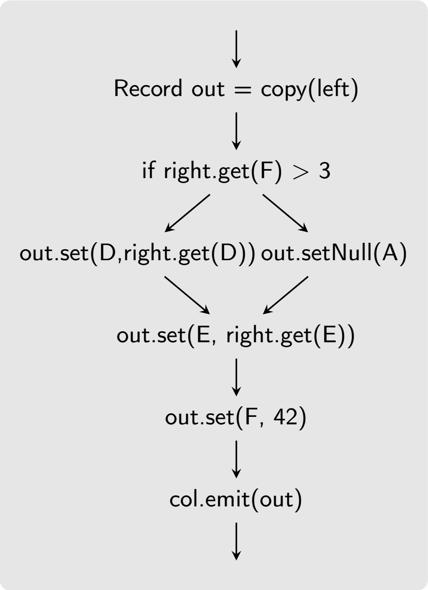

```
void match(Record left,
           Record right,
           Collector col) {
    Record out = copy(left);
    if (right.get(F) > 3) {
        out.set(D, right.get(D));
    } else {
        out.setNull(A);
    }
    out.set(E, right.get(E));
    out.set(F, 42);
    col.emit(out);
}
```

Schema:
Left [A,B,C], Right [D,E,F]

Origin: $\{1\}$
Explicit projection$_l$: $\{A\}$
Explicit copy$_r$: $\{E\}$
Explicit write$_l$: $\{F\}$
Explicit write$_r$: $\{\}$

Final write set$_l$: $\{A, F\}$
Final write set$_r$: $\{D, F\}$

Code Analysis

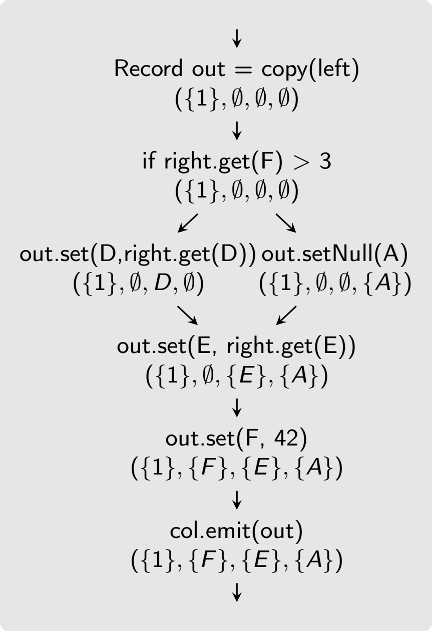


Difficult part is determining the origin, write, copy and projection sets for a user defined FOF from the control flow graph (CFG).

Solution is a recursive algorithm that builds the four sets:

- Start from the emit statements and traverse the CFG upwards
- The sets at one node in the CFG depend on the sets of the predecessors and the nature of the statement.

Code Analysis (cont.)



Final recursion cases:

`$or = create()`

$\rightarrow (\emptyset, \emptyset, \emptyset, \emptyset)$

`$or = copy($ir)`

$\rightarrow$ (*IN*(`$ir`), ∅, ∅, ∅)

For other statements Merge sets of predecessors and then modify depending on type of statement:

`$or.set(n,$ir.get(n))`

$\rightarrow$ add *n* to copy set

`$or.set(n, x)`

$\rightarrow$ add *n* to write set

`$or.setNull(n)`

$\rightarrow$ add *n* to projection set

Conclusion

- Reordering leads to potentially significant benefits
  - Up to 10x for relational and non relational tasks in our experiments
- Our static code analysis algorithm can automatically derive reordering properties of generic user-written Java code
- Difficulties arise in non-linear CFGs (if, loops) and also because the schema of input records changes with reordering
- Safety achieved through conservatism
- Related work: Manimal [2]
  - Techniques are complementary

[2] Eaman Jahani, Michael J. Cafarella, Christopher Ré: Automatic Optimization for MapReduce Programs. PVLDB 4(6): 385-396 (2011)

Thank you!

`www.stratosphere.eu`
(New open source release available)

Full SCA algorithm

function Compute-Write-Set(f, O_f, E_f, C_f, P_f)
  $W_f = E_f \cup P_f$
  for $i \in$ Inputs(f) **do**
    if $i \notin O_f$ **then** $W_f = W_f \cup (\text{Input-Fields}(f, i) \setminus C_f)$
  return W_f
function Visit-UDF(f)
  $R_f = \emptyset$
  G = all statements of the form `g:$t=getField($ir,`n`)`
  for g in G **do**
    if Def-Use(g, `$t`)$\neq \emptyset$ **then** $R_f = R_f \cup \{n\}$
  E = all statements of the form `e:emit($or)`
  (O_f, E_f, C_f, P_f) = Visit-Stmt(Any(E), `$or`)
  for e in E **do**
    (O_e, E_e, C_e, P_e) = Visit-Stmt(e, `$or`)
    (O_f, E_f, C_f, P_f) = Merge($(O_f, E_f, C_f, P_f), (O_e, E_e, C_e, P_e)$)
  return $(R_f, O_f, E_f, C_f, P_f)$
function Merge((O_1, E_1, C_1, P_1), (O_2, E_2, C_2, P_2))
  $C = (C_1 \cap C_2) \cup \{x | x \in C_1, \text{Input-Id}(x) \in O_2\}$
    $\cup \{x | x \in C_2, \text{Input-Id}(x) \in O_1\}$
  return $(O_1 \cap O_2, E_1 \cup E_2, C, P_1 \cup P_2)$

Full SCA algorithm (cont.)

```
function Visit-Stmt(s, $or)
    if visited(s, $or) then
        return Memo-Sets(s, $or)
    Visited(s, $or) = true
    if s of the form $or = create() then  return (∅, ∅, ∅, ∅)
    if s of the form $or = copy($ir) then
        return (Input-Id($ir), ∅, ∅, ∅)
    P_s = Preds(s)
    (O_s, E_s, C_s, P_s) = Visit-Stmt(Any(P_s), $or)
    for p in P_s do
        (O_p, E_p, C_p, P_p) = Visit-Stmt(p, $or)
        (O_s, E_s, C_s, P_s) = Merge((O_s, E_s, C_s, P_s), (O_p, E_p, C_p, P_p))
    if s of the form union($or, $ir) then
        return (O_s ∪ Input-Id($ir), E_s, C_s, P_s)
    if s of the form setField($or, n, $t) then
        T =Use-Def(s, $t)
        if all t ∈ T of the form $t=getField($ir,n) then
            return (O_s, E_s, C_s ∪ {n}, P_s)
        else
            return (O_s, E_s ∪ {n}, C_s, P_s)
    if s of the form setField($or, n, null) then
        return (O_s, E_s, C_s, P_s ∪ {n})
```