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
$\text{res} = \text{filter}\; \text{$e$ in $emp$ where $e.income > 30000;}$
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

Sink₁
- Reduce\((f_4, A)\)
  - \(sum(B)\)
  - \(fifo\)
- Match\((f_3, A, D)\)

\(\text{partition/sort}(A)\)
- \(probeHT\)
  - Map\((f_1)\)
    - \(C \leftarrow A + B\)
      - \([A, B]\)
        - \(Src_1\)
  - \(buildHT\)
    - Map\((f_2)\)
      - \(filter(E > 3)\)
        - \([D, E]\)
          - \(Src_2\)

- 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

- **TPC-H Q7**
  - Best Order: x7.1
  - Worst Order: x20.0

- **Clickstream Processing**
  - Best Order: x10.0

- **Textmining**
  - Worst Order: x10.0
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

¹Opening the Black Boxes in Data Flow Optimization (VLDB 2012)
Example FOF

```java
  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
```java
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.)

```java
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.)

```java
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.)

```java
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\}
1. `void match(Record left, Record right, Collector col) {`

2. `Record out = copy(left);`

3. `if (right.get(F) > 3) {
   out.set(D, right.get(D));
   } else {
   out.setNull(A);
   }`

4. `out.set(E, right.get(E));
out.set(F, 42);
oc.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** \(_r_\): {}
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.
Record out = copy(left)  
\(\{1\}, \emptyset, \emptyset, \emptyset\)

if right.get(F) > 3  
\(\{1\}, \emptyset, \emptyset, \emptyset\)

out.set(D, right.get(D)) out.setNull(A)  
\(\{1\}, \emptyset, D, \emptyset\)  
\(\{1\}, \emptyset, \emptyset, \{A\}\)

out.set(E, right.get(E))  
\(\{1\}, \emptyset, \{E\}, \{A\}\)

out.set(F, 42)  
\(\{1\}, \{F\}, \{E\}, \{A\}\)

col.emit(out)  
\(\{1\}, \{F\}, \{E\}, \{A\}\)

Final recursion cases:

$or = create()  
\rightarrow (\emptyset, \emptyset, \emptyset, \emptyset)$

$or = copy($ir)  
\rightarrow (IN($ir), \emptyset, \emptyset, \emptyset)$

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

$or.set(n, $ir.get(n))  
\rightarrow \text{add } n \text{ to copy set}$

$or.set(n, x)  
\rightarrow \text{add } n \text{ to write set}$

$or.setNull(n)  
\rightarrow \text{add } n \text{ to projection set}$
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

1: function Compute-Write-Set($f, O_f, E_f, C_f, P_f$)
2:  \[ W_f = E_f \cup P_f \]
3:  for $i \in \text{Inputs}(f)$ do
4:    if $i \notin O_f$ then $W_f = W_f \cup (\text{Input-Fields}(f, i) \setminus C_f)$
5:  return $W_f$
6: function Visit-UDF($f$)
7:  $R_f = \emptyset$
8:  $G =$ all statements of the form $g: t = \text{getField}($ir,$n$)
9:  for $g$ in $G$ do
10:    if Def-Use($g, t) \neq \emptyset$ then $R_f = R_f \cup \{n\}$
11:  $E =$ all statements of the form $e: \text{emit}($or$)$
12:  $(O_f, E_f, C_f, P_f) = \text{Visit-Stmt}(\text{Any}(E),$or$)$
13:  for $e$ in $E$ do
14:    $(O_e, E_e, C_e, P_e) = \text{Visit-Stmt}(e,$or$)$
15:  $(O_f, E_f, C_f, P_f) = \text{Merge}((O_f, E_f, C_f, P_f), (O_e, E_e, C_e, P_e))$
16:  return $(R_f, O_f, E_f, C_f, P_f)$
17: function Merge($[(O_1, E_1, C_1, P_1), (O_2, E_2, C_2, P_2)]$)
18:  $C = (C_1 \cap C_2) \cup \{x | x \in C_1, \text{Input-Id}(x) \in O_2\}$
19:    \[ \cup \{x | x \in C_2, \text{Input-Id}(x) \in O_1\} \]
20:  return $(O_1 \cap O_2, E_1 \cup E_2, C, P_1 \cup P_2)$
1: function Visit-Stmt(s, $or)
2:    if visited(s, $or) then
3:        return Memo-Sets(s, $or)
4:    Visited(s, $or) = true
5:    if $or = create() then return (∅, ∅, ∅, ∅)
6:    if $or = copy($ir) then
7:        return (Input-Id($ir), ∅, ∅, ∅)
8:    $ps = Preds(s)
9:    ($os, $es, $cs, $ps) = Visit-Stmt(Any($ps), $or)
10:   for p in $ps do
11:       ($op, $ep, $cp, $pp) = Visit-Stmt(p, $or)
12:       ($os, $es, $cs, $ps) = Merge(($os, $es, $cs, $ps), ($op, $ep, $cp, $pp))
13:   if $or = union($or, $ir) then
14:       return ($os ∪ Input-Id($ir), $es, $cs, $ps)
15:   if $or = setField($ir, n, $t) then
16:       $t = Use-Def(s, $t)
17:       if all $t ∈ $T of the form $t=getField($ir, n) then
18:          return ($os, $es, $cs ∪ {n}, $ps)
19:       else
20:          return ($os, $es ∪ {n}, $cs, $ps)
21:   if $or = setField($ir, n, null) then
22:       return ($os, $es, $cs, $ps ∪ {n})