Optimizing Across Relational and Linear Algebra in Parallel Analytics Pipelines

Asterios Katsifodimos
Assistant Professor @ TU Delft

a.katsifodimos@tudelft.nl
http://asterios.katsifodimos.com
Outline

• **Introduction and Motivation**
  • Databases vs. parallel dataflow systems

• **Declarativity in Parallel-dataflow Systems**
  • The Emma language and optimizer
  • The Lara language for mixed linear- and relational-algebra programs

• **Optimizing across linear & relational Algebra**
  • BlockJoin: A join algorithm for mixed relational & linear algebra programs

• The road ahead
A timeline of data analysis tools
from a biased perspective

<table>
<thead>
<tr>
<th>Appearance of Relational Databases</th>
<th>First parallel shared-nothing architectures</th>
<th>Open Source Projects and mainstream databases</th>
<th>First columnar storage Databases. Faster!</th>
<th>Scalable, UDF-based, data analytics blooms</th>
<th>Alternative MapReduce implementations go mainstream</th>
<th>Better Abstractions for dataflow systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>70s</td>
<td>80s</td>
<td>90s</td>
<td>00s</td>
<td>2004 - 2009</td>
<td>2009 - 2016</td>
<td>2016 -</td>
</tr>
<tr>
<td>Ingres</td>
<td>Gamma</td>
<td>BerkeleyDB</td>
<td>MonetDB</td>
<td>Hadoop MapReduce</td>
<td>Spark Flink</td>
<td>Spark Dataframes SystemML Emma MRQL</td>
</tr>
<tr>
<td>Oracle</td>
<td>TRACE</td>
<td>MS Access</td>
<td>C-Store</td>
<td></td>
<td>Impala Giraph</td>
<td></td>
</tr>
<tr>
<td>System-R</td>
<td>Teradata</td>
<td>PostgreSQL</td>
<td>Vertica</td>
<td></td>
<td>Hive (SQL!)</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>MySQL</td>
<td>Aster Data</td>
<td></td>
<td>HBase Blu</td>
<td></td>
</tr>
<tr>
<td>1974: Don Chamberlin publishes SQL</td>
<td>Open Source rises</td>
<td>Open Source rises</td>
<td>MonetDB</td>
<td>Hadoop MapReduce</td>
<td>Spark Flink</td>
<td>Spark Dataframes SystemML Emma MRQL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MySQL</td>
<td></td>
<td>Impala Giraph</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>PostgreSQL</td>
<td></td>
<td>Hive (SQL!)</td>
<td>We need easier to use tools!</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MS Access</td>
<td></td>
<td>HBase Blu</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>C-Store</td>
<td></td>
<td>Blu</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Aster Data</td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MonetDB</td>
<td>Hadoop MapReduce</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>C-Store</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Vertica</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Aster Data</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>GreenPlum</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We need easier to use tools!
Relational DBs vs. Modern Big Data Processing Stack

SQL
- SQL Compiler & Optimizer
- Relational Dataflow Engine
- Transaction Manager
- Row/Column Storage
- Operating System

SQL-like abstraction
- SQL Compiler & Optimizer
- Apache Flink
- Apache Spark
- Hadoop
- HBase

Iterative M/R-like Jobs (Java, Scala, ...)
- Put/Get ops
- Files/Bytes

Hadoop Distributed File System (HDFS)
- Cluster Management (Kubernetes, YARN, ...)

Files/Bytes
Declarative Data Processing

Relational Databases

SQL

Relations

RDBMS

Second-Order Functions (Map, Reduce, etc.)

Distributed Collections

Parallel Dataflow Engines

Parallel Dataflows
People with data analysis skills

Big data analysts

Big Data systems programming experts
Quiz: guess the algorithm!

```scala
... // initialize
while (theta) {
  newCntrds = points
    .map(findNearestCntrd)
    .map((c, p) => (c, (p, 1L)))
    .reduceByKey((x, y) =>
      (x._1 + y._1, x._2 + y._2))
    .map(x => Centroid(x._1, x._2 / x._3))

  bcCntrs = sc.broadcast(newCntrds.collect())
}

... // initialize
val cntrds = centroids.iterate(theta) {
  currCntrds =>
    val newCntrds = points
      .map(findNearestCntrd).withBcSet(currCntrds, "cntrds")
      .map((c, p) => (c, (p, 1L)))
      .groupBy(0).reduce((x, y) =>
        (x._1 + y._1, x._2 + y._2))
      .map(x => Centroid(x._1, x._2 / x._3))

    currCntrds
}
```
K-Means Clustering

• **Input**
  • Dataset (set of points in 2D) — large
  • K initial centroids — small

• **Step 1:**
  • Read K-centroids
  • Assign each point to the closest centroid (wrt distance)
  • Output <centroid, point> (or, cluster)

• **Step 2:**
  • For each centroid
    • Calculate its average position wrt distance from its points
    • Output: <new centroid> (coordinates)
Partial Aggregates

... // initialize
while (theta) {
    newCntrds = points
        .map(findNearestCntrd)
        .map( (c, p) => (c, (p, 1L)) )
        .reduceByKey( (x, y) =>
            (x._1 + y._1, x._2 + y._2) )
        .map( x => Centroid(x._1, x._2 / x._3) )

    bcCntrs = sc.broadcast(newCntrds.collect())
}

... // initialize
val cntrds = centroids.iterate(theta) {
    currCntrds =
        val newCntrds = points
            .map(findNearestCntrd).withBcSet(currCntrds, "cntrds")
            .map( (c, p) => (c, (p, 1L)) )
            .groupBy(0).reduce( (x, y) =>
                (x._1, x._2 + y._2, x._3 + y._3) )
            .map( x => Centroid(x._1, x._2 / x._3) )
    currCntrds
}
... // initialize
while (theta) {
    newCntrds = points
        .map(findNearestCntrd)
        .map( (c, p) => (c, (p, 1L)) )
        .reduceByKey( (x, y) =>
            (x._1 + y._1, x._2 + y._2) )
        .map( x => Centroid(x._1, x._2 / x._3) )
    bcCntrs = sc.broadcast(newCntrds.collect())
}

... // initialize
val cntrds = centroids.iterate(theta) { currCntrds =>
    val newCntrds = points
        .map(findNearestCntrd).withBcSet(currCntrds, "cntrds")
        .map( (c, p) => (c, p, 1L) )
        .groupBy(0).reduce( (x, y) =>
            (x._1 + y._1, x._2 + y._2, x._3 + y._3) )
        .map( x => Centroid(x._1, x._2 / x._3) )
    currCntrds
}
Native Iterations

... // initialize
while (theta) {
  newCntrds = points
  .map(findNearestCntrd)
  .map( (c, p) => (c, (p, 1L)) )
  .reduceByKey( (x, y) =>
    (x._1 + y._1, x._2 + y._2) )
  .map( x => Centroid(x._1, x._2 / x._3) )
  .groupBy(0).reduce( (x, y) =>
    (x._1, x._2 + y._2, x._3 + y._3) )
  .map( x => Centroid(x._1, x._2 / x._3) )
  .withBcSet(currCntrds, "cntrds")
  bcCntrs = sc.broadcast(newCntrds.collect())
}

... // initialize
val cntrds = centroids.iterate(theta) { currCntrds =>
  val newCntrds = points
  .map(findNearestCntrd).withBcSet(currCntrds, "cntrds")
  .map( (c, p) => (c, (p, 1L)) )
  .groupBy(0).reduce( (x, y) =>
    (x._1, x._2 + y._2, x._3 + y._3) )
  .map( x => Centroid(x._1, x._2 / x._3) )
  currCntrds
}

Feedback edges in the dataflow graph have to be specified explicitly.
Beyond Datasets: End-to-end ML Pipelines

A typical Data Science Workflow

Data Management
1. Question Formulation
2. Data Source Exploration & Selection
3. Data cleaning & preparation
4. Feature Selection

Machine Learning
5. ML Algorithm Selection
6. Hyper Parameter Search
7. Model Training
8. Model Evaluation

80% of data scientists’ time
A very simple ML pipeline example

Factory 1 → Gather and clean sensor data → Principal Component Analysis → Train ML Model

Factory 2

Factory 3
Preprocessing

Naturally expressed in

- SQL
- Dataflow APIs

```
SELECT f1.v, f2.v, f3.v
FROM factory_1 AS f1,
     factory_2 AS f2,
     factory_3 AS f3
WHERE f1.sid == f2.sid AND f2.sid == f3.sid AND f1.v != NULL AND f2.v != NULL AND f3.v != NULL
```
Impedance mismatch!

**Matrix Operations/ML in SQL**

Calculating covariance matrix in SQL

```sql
SELECT
    idxs.*,
    ((idx_val2 - idx_val2_avg) * (idx_val1 - idx_val1_avg)) / total_rows AS cv
FROM
    (SELECT
        dt,
        MAX(CASE WHEN indx_nm = 'ABQI' THEN indx_nm ELSE NULL END) AS indx_nm1,
        MAX(CASE WHEN indx_nm = 'ABQI' THEN indx_val ELSE NULL END) AS indx_val1,
        MAX(CASE WHEN indx_nm = 'ACNACTR' THEN indx_nm ELSE NULL END) AS indx_nm2,
        MAX(CASE WHEN indx_nm = 'ACNACTR' THEN indx_val ELSE NULL END) AS indx_val2
    FROM table1 a
    GROUP BY dt)
    idxs
INNER JOIN
    (/* Join against a query returning the AVG() and COUNT() across all rows */
    SELECT
        'ABQI' AS indx_nm1_aname,
        AVG(CASE WHEN indx_nm = 'ABQI' THEN indx_val ELSE NULL END) AS indx_val1_avg,
        'ACNACTR' AS indx_nm2_aname,
        AVG(CASE WHEN indx_nm = 'ACNACTR' THEN indx_val ELSE NULL END) AS indx_val2_avg,
        COUNT(*) AS total_rows
    FROM table1 b
    WHERE indx_nm IN ('ABQI','ACNACTR')
    /* And it is a cartesian product */
    ) aggs ON 1=1
WHERE
    indx_nm1 IS NOT NULL
    AND indx_nm2 IS NOT NULL
ORDER BY dt
```

Machine Learning in “Native” Syntax

Naturally expressed in

- Matlab
- R
- ...

```matlab
// read factories table as matrix FacMatrix
[rows, cols] = size(FacMatrix)
colMeans = mean(FacMatrix)
NormMat = zeros(rows, cols)
for i = 1:rows
    NormMat(i, :) = FacMatrix(i, :) - colMeans;
end
CovMat = NormMat' * NormMat / (rows - 1)

// ...
```

Principal Component Analysis → Train ML Model
SQL + ML Native Syntax = Optimization Barrier

Gather and clean sensor data
Principal Component Analysis
Train ML Model

Factory 1
Factory 2
Factory 3

Optimization barrier!
Big Data Language Desiderata

• Declarativity: what vs. how
  • Enables optimizability, eases programming

• User-Defined Functions as first-class citizens
  • Important reason why MapReduce took off

• Control-flow for Machine Learning and Graph workloads
  • Iterations, if-then-else branches
  • Turing complete “driver” with parallelizable constructs

• Portability
  • Should support multiple backend engines (Flink, Spark, MapReduce, etc.)

• Deep embedding
  • e.g. à la LINQ (Language Integrated Query) in C#, Ferry in Haskel, etc.
Gather and clean data

```scala
val factories = for {
  f1 <- factory_1
  f2 <- factory_2
  if f1.sid == f2.sid &&
      f1.v != null &&
      f2.v != null &&
} yield (f1.v,..., f2.v, ...)
```

Principal Component Analysis

```scala
while(...) { 
  val FacMatrix = factories.toMatrix()
  //Calculate the mean of each of the columns
  val colMeans = FacMatrix.cols(col => col.mean() )

  val NormMat = FacMatrix - colMeans
  //Calculate covariance matrix
  val CovMat = NormMat.t **% NormMat/(NormMat.numRows - 1)
}
```

Deeply Embedded in Scala

- **Databag type**
  - Like RDDs on Spark, Datasets on Flink, etc.
  - Operations on Databags are parallelized

- **Declarative for-comprehensions**
  - Like Select-From-Where in SQL
  - Can express join, cross, filter, etc.

- **Loops: First-class citizens**

- **Databag => Matrix**
  - Enable optimization across Linear and relational algebra
  - Intermediate representation to reason across linear and relational algebras (WiP)

- **UDFs: First-class citizens**
  - Against Impedance mismatch

[SIGMOD15/16, SIGMOD Record 16, BeyondMR 2016, PVLDB 2017]
Comprehensions: generalize SQL and are available as first-class syntax in modern general purpose programming languages.

\[
\{ (x, y) \mid x \in xs, y \in ys, x = y \}
\]

**Math**

**SQL**

```
SELECT x, y FROM x AS xs, y AS ys WHERE x = y
```

**Python**

```
[ (x, y) for x in xs, y in ys if x == y ]
```

**Scala**

```
for (x <- xs; y <- ys; if x == y) yield (x, y)
```
Automatic Optimization of Programs

• **Caching** of intermediate results
  • Reused intermediate results (e.g., within a loop) are automatically cached

• **Choice of Partitioning**
  • Pre-partition datasets which are used in multiple joins

• **Unnesting**
  • e.g., converting exists() clauses into joins

• **Join order/algorithm** selection
  • Classic database optimizations

• **Fold-group Fusion**
  • Calculate multiple aggregates on one pass

• ...
A concrete optimization

BlockJoin: partitioning data for linear algebra operations through joins

[PVLDB ‘17 (VLDB 18)] "BlockJoin: Efficient Matrix Partitioning Through Joins”
Row- vs. block-partitioned operations

Certain relational operations better executed over row-partitioned tables.

Scalable Linear Algebra operations such as Matrix Multiplication better executed over block-partitioned matrices. Orders of magnitude speedups.
The contributions of this paper are summarized as follows:

1. BlockJoin is based on the observation that in contrast to the baseline approach, BlockJoin is up to 7x faster compared to the baseline implementation in a distributed dataflow system.

2. We compare the performance of BlockJoin against a baseline of partitioned manner. Instead, it performs the joining and completely avoids to materialize the join results in a row-chines hosting the corresponding blocks. Thus, BlockJoin materializes the join result by sending tuples to the main memory of the systems hosting the corresponding blocks.

3. We identify a partitioning function for each tuple, which is derived after the matching tuple-id pairs have been identified. A partitioning function derives, for each tuple, the set(s) of columns hosting the matching tuples of the two joining sides. Therefore, such workloads are typically implemented by applying the join operator which consumes row-partitioned relational data and outputs a block-partitioned matrix. The task of designing such an operator poses a number of challenges. First, in order to block-partition a matrix which results from a join, the system needs to know which keys and corresponding tuple-id pairs need to be known in advance. This stems from the fact that a matrix block is a fixed-size matrix block. Thus, we need to know which blocks are needed to perform the join and which blocks are needed to perform the join result. Therefore, we first need to join the records from these tables to obtain reviews with their corresponding products. Next, we need to transform these attributes of both relations. Therefore, we first need to join the records from these tables to obtain reviews with their corresponding products. Finally, we aggregate these vectors into a suitable representation for an ML algorithm, which means we need to apply a user defined function (UDF) that transforms the attributes into a vector representation. Finally, we aggregate these vectors into a suitable representation for an ML algorithm, which means we need to apply a user defined function (UDF) that transforms the attributes into a vector representation.

4. The matrix block-partitioning technique is best suited for tables with large numbers of columns, BlockJoin can contain columns from both input relations. Thus, the partitioning function cannot know which those columns are, and it is unable to know which keys and corresponding tuple-id pairs need to be known in advance. This stems from the fact that a matrix block is a fixed-size matrix block. Thus, we need to know which blocks are needed to perform the join and which blocks are needed to perform the join result. Therefore, we first need to join the records from these tables to obtain reviews with their corresponding products. Next, we need to transform these attributes of both relations. Therefore, we first need to join the records from these tables to obtain reviews with their corresponding products. Finally, we aggregate these vectors into a suitable representation for an ML algorithm, which means we need to apply a user defined function (UDF) that transforms the attributes into a vector representation.

5. Analogous to joins which have been proposed for columnar databases [23, 8, 1], BlockJoin builds on two main concepts:

- Analogous to joins which have been proposed for columnar databases [23, 8, 1], BlockJoin builds on two main concepts:
  - BlockJoin first identifies the matching tuple pairs by performing a join on the keys of the two relations (analogous to joins which have been proposed for columnar databases [23, 8, 1], BlockJoin builds on two main concepts: block-partitioning those results.
  - We compare the performance of BlockJoin against a baseline of partitioned manner. Instead, it performs the joining and completely avoids to materialize the join results in a row-chines hosting the corresponding blocks. Thus, BlockJoin materializes the join result by sending tuples to the main memory of the systems hosting the corresponding blocks. In contrast, BlockJoin is based on the observation that in contrast to the baseline approach, BlockJoin is up to 7x faster compared to the baseline implementation in a distributed dataflow system.

6. We propose BlockJoin, a distributed join algorithm based on Apache Spark [38] and experimentally show there are identified, a partitioning function derives, for each tuple, the set(s) of columns hosting the matching tuples of the two joining sides. Therefore, such workloads are typically implemented by applying the join operator which consumes row-partitioned relational data and outputs a block-partitioned matrix. The task of designing such an operator poses a number of challenges. First, in order to block-partition a matrix which results from a join, the system needs to know which keys and corresponding tuple-id pairs need to be known in advance. This stems from the fact that a matrix block is a fixed-size matrix block. Thus, we need to know which blocks are needed to perform the join and which blocks are needed to perform the join result. Therefore, we first need to join the records from these tables to obtain reviews with their corresponding products. Next, we need to transform these attributes of both relations. Therefore, we first need to join the records from these tables to obtain reviews with their corresponding products. Finally, we aggregate these vectors into a suitable representation for an ML algorithm, which means we need to apply a user defined function (UDF) that transforms the attributes into a vector representation. Finally, we aggregate these vectors into a suitable representation for an ML algorithm, which means we need to apply a user defined function (UDF) that transforms the attributes into a vector representation.
Common Pattern: Blocked Matrices as Join Results

e.g., join products and their reviews, to form a feature matrix. The matrix
is block partitioned to be used in linear algebra operations.
Data shuffling dominates execution time

• Challenge: can we join and block-partition results with less shuffling?

• Solution: BlockJoin, a one-step parallel join algorithm which outputs block-partitioned results

• Basic idea
  1. Find out which and how many tuples survive the join (without shuffling)
  2. Give each surviving tuple a unique, sequential ID
  3. Shuffle only surviving tuples and block-partition the joining tuples on the fly

• Results: up to >7x speedup over row-wise join followed by block-partitioning
  • Implementation can be further optimized (compression, sorting, etc.)
  • BlockJoin can be used for multi-way joins (where we expect larger speedups)
Goal: Join two tables, then block-partition them

<table>
<thead>
<tr>
<th>User</th>
<th>Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>a 1</td>
<td>a 2 3</td>
</tr>
<tr>
<td>b 1</td>
<td>a 2 3</td>
</tr>
<tr>
<td>d 1</td>
<td>a 2 3</td>
</tr>
<tr>
<td>b 2</td>
<td>b 2 3</td>
</tr>
<tr>
<td>c 2</td>
<td>b 2 3</td>
</tr>
</tbody>
</table>

Join

<table>
<thead>
<tr>
<th>Original Tables</th>
<th>Join Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>B[0,0]</td>
</tr>
<tr>
<td>a 1</td>
<td>a 1 2 3</td>
</tr>
<tr>
<td>b 1</td>
<td>a 1 2 3</td>
</tr>
<tr>
<td>d 1</td>
<td>a 1 2 3</td>
</tr>
<tr>
<td>b 2</td>
<td>b 1 2 3</td>
</tr>
<tr>
<td>c 2</td>
<td>b 1 2 3</td>
</tr>
<tr>
<td>Transaction</td>
<td>B[0,1]</td>
</tr>
<tr>
<td>a 2 3</td>
<td>a 1 2 3</td>
</tr>
<tr>
<td>a 2 3</td>
<td>a 1 2 3</td>
</tr>
<tr>
<td>a 2 3</td>
<td>a 1 2 3</td>
</tr>
<tr>
<td>b 2 3</td>
<td>b 1 2 3</td>
</tr>
<tr>
<td>b 2 3</td>
<td>b 1 2 3</td>
</tr>
<tr>
<td>c 2 3</td>
<td>b 1 2 3</td>
</tr>
</tbody>
</table>
Join two tables, then block-partition them
BlockJoin: fused join + block-partitioning operator
Saves heavy repartitioning costs.

**Step I: Local Index Join**
1. Find which tuples survive the join.
2. Prepare surviving tuple IDs and metadata.
3. Broadcast tuple IDs and metadata to all nodes.

**Step II: Block Materialization**
Exchange tuples parts in order to materialize blocks on each node participating in the join. Two materialization strategies, late & early.

**Main results:**
- 7x speedups compared to separate joining and block-partitioning
- Scales more gracefully

---

BlockJoin step 1: Find the surviving tuples with a semi-join/index-join

Driver Node: calculates scaffolds of final results (similar to index-join). The scaffold can be used for block partitioning (next step).

Collect join keys at the driver.

<table>
<thead>
<tr>
<th>U</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>a 1</td>
<td>a 2 3</td>
</tr>
<tr>
<td>a 2 3</td>
<td>b 1</td>
</tr>
<tr>
<td>a 2 3</td>
<td>b 2 3</td>
</tr>
<tr>
<td>d 1</td>
<td>c 2 3</td>
</tr>
</tbody>
</table>

U ⊙ T =

<table>
<thead>
<tr>
<th>B[0,0]</th>
<th>B[0,1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 a</td>
<td>B[0,1]</td>
</tr>
<tr>
<td>1 a</td>
<td></td>
</tr>
<tr>
<td>2 a</td>
<td></td>
</tr>
<tr>
<td>3 b</td>
<td></td>
</tr>
</tbody>
</table>

B[1,0] =

<table>
<thead>
<tr>
<th>B[1,0]</th>
<th>B[1,1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 a</td>
<td>B[1,1]</td>
</tr>
<tr>
<td>1 a</td>
<td></td>
</tr>
<tr>
<td>2 a</td>
<td></td>
</tr>
<tr>
<td>3 b</td>
<td></td>
</tr>
</tbody>
</table>

Broadcast
BlockJoin step 2: Fill-in the blanks
and shuffle the right tuples to their respective blocks

U
T

a 1
b 1
a 2 3
a 2 3

=>

B[0,0]
B[0,1]

Shuffle

Node 1

B[1,0]
B[1,1]

Node 2

U
T

d 1
a 2 3
b 2 3
a 2 3

b 2 3

=>

0 a
1 a
2 a
3 b

0 a
1 a
2 a
3 b

1 2 3
1 2 3
1 2 3
1 2 3

2 3
2 3
2 3
2 3

0 1 2 3
0 1 2 3
0 1 2 3
0 1 2 3

1
1
1
1

2 3
2 3
2 3
2 3

0 1 2 3
0 1 2 3
0 1 2 3
0 1 2 3

2 3
2 3
2 3
2 3

2 3
2 3
2 3
2 3
The devil is in the detail but details were skipped to simplify presentation

- The shown example assumed
  - Primary key on User and foreign key on Transaction
    - This can be relaxed either 1) similarly to late materialization in columnar-databases or 2) extra counters
  - Ordered relations - algorithm works with unordered relations

- The algorithm performs best when
  - Join selectivity is low (less tuples survive, huge savings vs. row-wise partitioning)
  - The number of columns is big

- Up to >7x speedup over row-wise join followed by block-partitioning
  - Implementation can be further optimized (compression, sorting, etc.)
  - BlockJoin can be used for multi-way joins (where we expect larger speedups)
The road ahead

Deep in-database analytics
Linear Algebra + Relational Algebra = ❤️
Theoretical Foundation

• Goal
  • Intermediate representation/algebra to support equivalences and transformations of mixed linear-relational algebra programs.

• Fundamental questions
  • What is the right intermediate representation?
  • Which is the set of transformations/equivalences for optimization?
  • Do current cost models work?

• Main challenge
  • Data model mismatch: unordered sets (relations) vs. ordered/indexed matrices
Programming Abstraction

//SELECT f1.v, f2.v
//FROM factory_1 as f1, factory_2 as f2
//WHERE f1.id == f2.id and f1.v != null and f2.v != null
val factories = for { //Emma’s for-comprehension for joining
  f1 <- factory_1
  f2 <- factory_2
  if f1.id == f2.id &&
    f1.v != null &&
    f2.v != null &&
  } yield (f1.v, ..., f2.v, ...)

val FacMatrix = factories.toMatrix()

//Calculate the mean of each of the columns
val colMeans = FacMatrix.cols( col => col.mean() )

val NormMat = FacMatrix - colMeans
//Calculate covariance matrix
val CovMat = NormMat.t %*% NormMat / (NormMat.numRows - 1)
// ...

// SELECT f1.v, f2.v FROM factory_1 as f1, factory_2 as f2 WHERE f1.id == f2.id and f1.v != null and f2.v != null
val factories = for {
  f1 <- factory_1
  f2 <- factory_2
  if f1.id == f2.id &&
    f1.v != null &&
    f2.v != null &&
  }
  yield (f1.v, ..., f2.v, ...)

val FacMatrix = factories.toMatrix()

// Calculate the mean of each of the columns
val colMeans = FacMatrix.cols( col => col.mean() )

val NormMat = FacMatrix - colMeans
// Calculate covariance matrix
val CovMat = NormMat.t %*% NormMat / (NormMat.numRows - 1)
// ...
Goal

- Systems natively supporting mixed linear-relational algebra programs.

Fundamental questions

- Data partitioning schemes (row- vs. column- vs. block-partitioning)?
- Relational operators operating on block-partitioned data?
- Linear algebra operators operating on row/column-partitioned data?
- Operator Fusion?

Main challenges

- Novel, non-trivial operators (e.g., BlockJoin)
- Reuse of existing operators
Thank you!

a.katsifodimos@tudelft.nl
http://asterios.katsifodimos.com
The DataBag abstraction

class DataBag[+A] {
  // Type Conversion
  def this(s: Seq[A]) // Scala Seq -> DataBag
  def fetch() // DataBag -> Scala Seq
  // Input/Output (static)
  def read[A](url: String, format: ...): DataBag[A]
  def write[A](url: String, format: ...)(in: DataBag[A])
  // Monad Operators (enable comprehension syntax)
  def map[B](f: A => B): DataBag[B]
  def flatMap[B](f: A => DataBag[B]): DataBag[B]
  def withFilter(p: A => Boolean): DataBag[A]
  // Nesting
  def groupBy[K](k: (A) => K): DataBag[Grp[K,DataBag[A]]]
  // Difference, Union, Duplicate Removal
  def minus[B >: A](subtrahend: DataBag[B]): DataBag[B]
  def plus[B >: A](addend: DataBag[B]): DataBag[B]
  def distinct(): DataBag[A]
  // Structural Recursion
  def fold[B](z: B, s: A => B, p: (B, B) => B): B
  // Aggregates (aliases for various folds
  def minBy, min, sum, product, count, empty, exists, ...
}