A Query Processing Framework for Array-Based Computations

Leonidas Fegaras

University of Texas at Arlington

http://lambda.uta.edu/

Data-intensive Scientific Computing

- Current scientific applications must analyze enormous amounts of array data using complex mathematical data processing methods
- Arrays are very important data structures for scientific computing
- Scientists use numerical analysis tools but are not familiar with Big Data analysis and distributed computing
- Need a declarative distributive query language capable of expressing complex mathematical operations on arrays
  - could help them develop their data analysis applications without any prior knowledge of distributed computing
- Our goal: develop a framework for optimizing queries on large vectors and matrices

Inter-Operation Optimization Requires Breaking Abstractions

- Library approach: provide a set of predefined operations (API)
  - Implement these operations using efficient parallel processing algorithms
- Problem: missed opportunities for inter-operation optimization
- Many data analysis algorithms use complex operations with many matrix and vector operations
- Example: matrix factorization using gradient descent

$$\begin{aligned} E &\leftarrow R - P \times Q^T \\ P &\leftarrow P + \gamma(2E \times Q^T - \lambda P) \\ Q &\leftarrow Q + \gamma(2E \times P^T - \lambda Q) \end{aligned}$$

- Evaluating $2E \times Q^T$ using library operations
  vs coding a new program from scratch:
  a program that calculates $\sum_k 2 * E_{ik} * Q_{jk}$ directly
- Various solutions have been proposed:
  - using rewrite rules
  - breaking the abstractions: unfold-simplify-fold
  - using lazy evaluation

Our Approach

- We don't treat matrix operations as opaque library functions
- Matrices are expressed as regular data types using regular language syntax
- We provide a very small number of powerful higher-order physical algorithms that generalize some concrete distributed algorithms
  - general enough to cover many similar operations
  - ... but making sure that the same efficient implementations that apply to the concrete algorithms can also apply to the higher-order operations
- We provide rules to transform regular terms in our algebra to these physical algorithms

MRQL: A Query Language for Big Data Analytics

- A powerful and efficient query processing system for complex data analysis on Big Data
- More powerful than existing query languages
  - a richer data model (nested collections, trees, ...)
  - arbitrary query nesting
  - more powerful query constructs
  - user-defined types and functions
- Platform-independent

Matrix Multiplication in MRQL

- Example of a matrix representation:
  - A sparse matrix M is a collection of triples, (v, i, j), for $v = M_{ij}$
  - *Note:* Our approach is independent of the matrix representation
- Matrix multiplication between two matrices X and Y is $\sum_k X_{ik} * Y_{kj}$:

```
select ( sum(z), i, j )
  from (x,i,k) in X, (y,k,j) in Y, z = x*y
 group by i, j
```

- before the group-by: `z` $= X_{ik} * Y_{kj}$
- after group-by: `z` is lifted to a bag of values $X_{ik} * Y_{kj}$
  - that is, for each group `i,j`, this bag contains all $X_{ik} * Y_{kj}$, for every k
- **sum**`(z)` sums up all these `z` values, for each group `i,j`

Naive Evaluation of the Matrix Multiplication Query

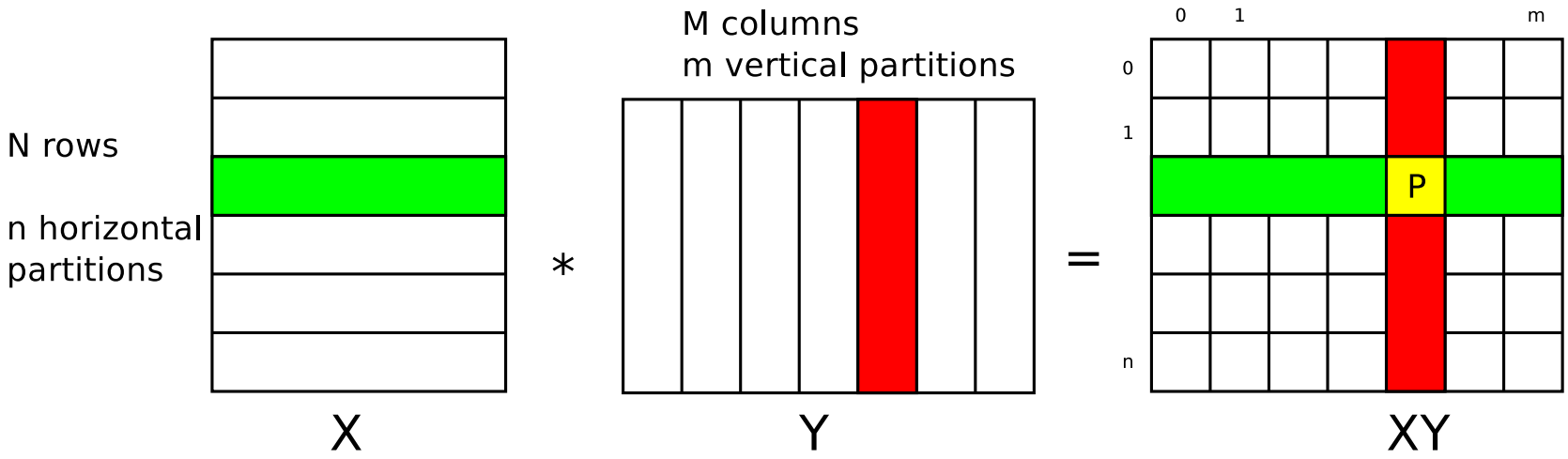
- Let X be an $N * K$ matrix and Y be an $K * M$ matrix
- Naive evaluation: equi-join followed by a group-by with aggregation
- The intermediate result of the join is of max size $N * K * M$

  n^3 for square matrices
- These data need to be shuffled to cluster nodes for the group-by operation

  very expensive

The SUMMA Algorithm

The SUMMA Algorithm [Geijn & Watts, 1997]



- X is an $N * K$ matrix and Y is an $K * M$ matrix
- distribute the data of X and Y as a grid of $m * n$ partitions
- each partition contains N/n full rows from X and M/m full columns from Y
- can be implemented using 1 map-reduce job

The GroupByJoin Operation

Generalization of matrix multiplication:

```
GroupByJoin( jx, jy, gx, gy, acc, zero, h, X, Y )
```

which corresponds to the MRQL query:

```
select h( k, reduce(acc,zero,z) )
  from x in X, y in Y, z = (x,y)
 where jx(x) = jy(y)
 group by k: ( gx(x), gy(y) )
```

from two bags `X` and `Y` to a bag result, where

- `jx` and `jy` are the join key functions
- `gx` and `gy` are the group-by functions
- `h` is the head function
- `reduce(acc,zero,z)` reduces a bag `z` using an accumulator `acc` and a `zero` value:

$$\text{reduce}(\text{acc}, \text{zero}, \{z_1, z_2, \ldots, z_n\}) = \text{acc}(z_1, \text{acc}(z_2, \ldots, \text{acc}(z_n, \text{zero})))$$

Map-Reduce Implementation of GroupByJoin

- One mapper for each input matrix, X and Y
- Each mapper emits pairs (key, (tag, data))
  - data is the input data
  - tag is the source number: 1 for X, 2 for Y
  - key is a triple (partition, joinkey, tag)
    - partition is one of the $n * m$ partitions
    - joinkey is the join key value, jx(x) or jy(y)
- A value x ∈ X is sent to all row partitions (gx(x) mod n, *)
- A value y ∈ Y is sent to all column partitions (*, gy(y) mod m)
- Custom partitioning, grouping, and sorting functions:
  - the partition function returns the partition value of the mapper key,
  - the grouping function returns the pair (partition, joinkey), and
  - the sorting is based on:
    - major order: partition
    - minor order: joinkey
    - sub-minor order: tag

Map-Reduce Code for GroupByJoin(jx, jy, gx, gy, acc, zero, h, X, Y)

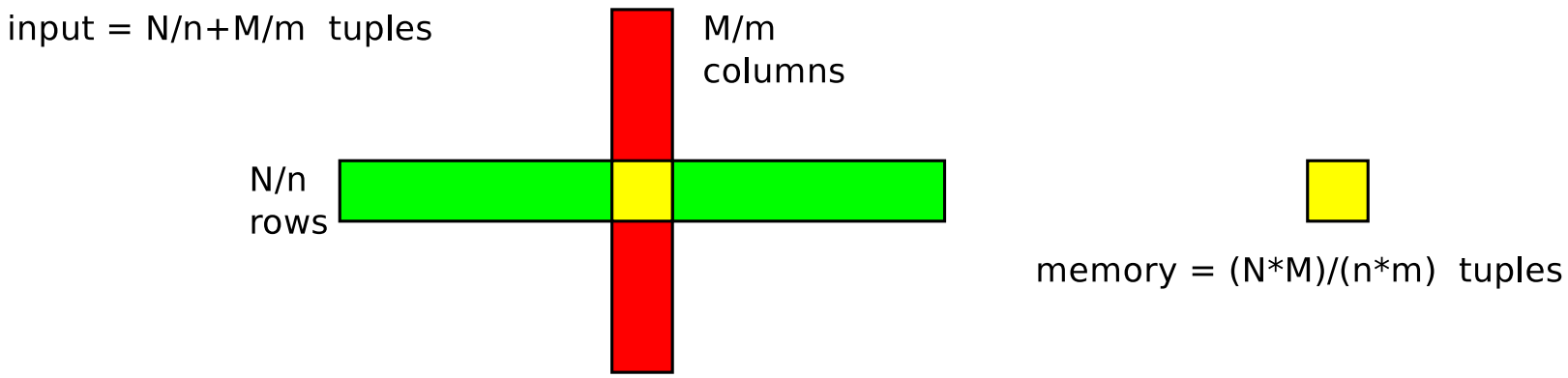
```
mapLeft ( x ):
  for each i in 0..m−1
      emit ( ((hashCode(gx(x)) % n)∗m+i, jx(x), 1), (1,x) )

mapRight ( y ):
  for each i in 0..n−1
      emit ( ((hashCode(gy(y)) % m)+m∗i, jy(y), 2), (2,y) )
```

```
reduce ( ( partition , joinkey , tag ), values ):
  if ( partition != current_partition )
     flush (H)
     current_partition ← partition
  for each leading (1,x) tuple in values
     insert x into xs
  for each (2,y) tuple in the rest of values
     for each x in xs
        key ← (gx(x), gy(y))
        if (H[key] is null )
           H[key] ← zero
        H[key] ← acc( (x,y), H[key] )
```

```
flush (H):
  for each (key , value ) in H
     emit H(key,value)
  clear H
```

```
cleanup ( ):  flush (H)
```

Optimal Number of Partitions $n * m$



- Data replication is $N * K * m + K * M * n$ tuples

  $\Rightarrow$ we want to minimize $N/n + M/m$
- but ... the hash table H must have enough space for $(N * M)/(n * m)$ tuples
- Assuming each worker node has enough memory to fit $\mathcal{T}$ tuples
- Optimal solution: $N/n = M/m = \sqrt{\mathcal{T}}$
- Number of worker nodes $\leq n * m$
  - it can even be just 1: will process one partition at a time

The MRQL Algebra

- **cMap**(f, S): map f over S and concatenate the results

$$\mathrm{cMap}(\lambda x.\, \{x+1\},\ \{1,2,3\}) \ = \ \{2,3,4\}$$

- **groupBy**(S): group-by a bag of pairs by the first component

$$\mathrm{groupBy}(\{(1,10),(2,20),(1,30),(1,40)\}) \\ = \ \{(1,\{10,30,40\}),(2,\{20\})\}$$

- **join**(j_x, j_y, h, X, Y): an equijoin

$$\mathrm{join}(j_x, j_y, h, X, Y) \\ = \ \{\, h(x,y) \,|\, x \in X,\ y \in Y,\ j_x(x) = j_y(y) \,\}$$

Deriving GroupByJoin operations from MRQL Queries

- Optimizations are done at the algebraic level
  - eg, fuse two cascaded cMaps into a nested cMap

$$\mathrm{cMap}(f, \mathrm{cMap}(g, S)) \rightarrow \mathrm{cMap}(\lambda x.\, \mathrm{cMap}(f, g(x)), S)$$

- Then, algebraic terms are mapped to physical plans
- A GroupByJoin operation is derived by the rule:

```
cMap( λ(k,s). { h(k,reduce(acc,zero,s)) },
      groupBy( join( jx, jy,
                     λ(x,y). ( (gx(x),gy(y)), (x,y) ),
                     X, Y ) ) )
  →  GroupByJoin( jx, jy, gx, gy, acc, zero, h, X, Y )
```

Example

- matrix multiplication $X \times Y$:

```
select ( sum(z), i, j )
  from (x,i,k) in X, (y,k,j) in Y, z = x*y
 group by i, j
```

→ cMap(λ((i,j),s). {(reduce(λ(v,c). c+v, 0, s), i, j)},
groupBy(join(λ(x,i,k). k, λ(y,k,j). k,
λ((x,i,k),(y,l,j)). ((i,j), x*y),
X, Y)))

- matrix transpose Y^T:

```
select (y,j,i) from (y,i,j) in Y
```

→ cMap(λ(y,i,j). {(y,j,i)}, Y)

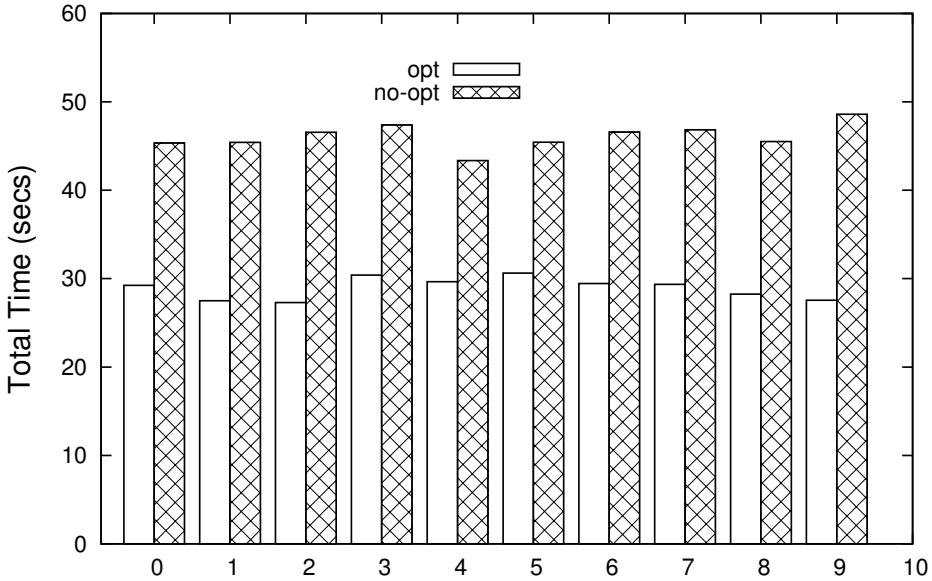
Example (cont.)

- $X \times Y^T$ after fusing the cMaps:

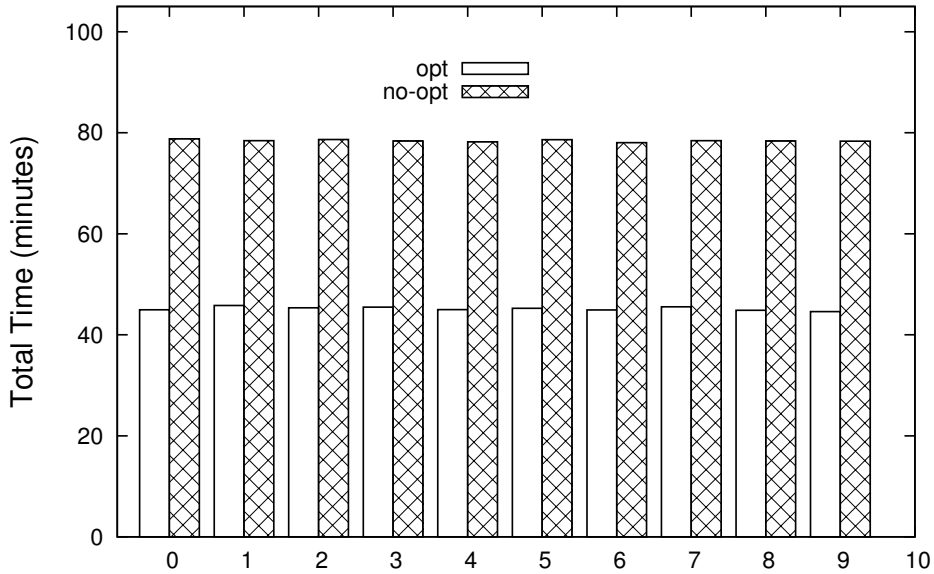
$\rightarrow$ cMap(λ((i,j),s). {(reduce(λ(v,c). c+v, 0, s), i, j)},
groupBy(join(λ(x,i,k). k, λ(y,j,k). k,
λ((x,i,k),(y,j,l)). ((i,j), x*y),
X, Y)))

$\rightarrow$ GroupByJoin(λ(x,i,k). k, λ(y,j,k). k, λ(x,i,k). i, λ(y,j,l). j,
λ((x,y),c). c+x*y, 0, λ((i,j),c). (c,i,j), X, Y)

Performance Evaluation



(A) A simple query data with/without optimization



(B) Matrix factorization with/without optimization

Conclusion

- We let programmers express their array operations using normal SQL-like syntax
  - or using macros that are unfolded during compilation
- May use any kind of matrix representation and storage
- Our framework translates these queries into efficient distributed operations
  - achieving inter-operation optimization
  - which is infeasible if these operations were expressed as black boxes