Cumulon: Optimizing Statistical Analysis in the Cloud

Botong Huang, Shivnath Babu, Jun Yang
Big data + cloud

- Larger scale
- More sophistication—don’t just report; analyze!
- Wider range of users—not just programmers

- Rise of cloud (e.g., Amazon EC2)
  - Get resources on demand and pay as you go
  - Getting computing resources is easy
    - But that is still not enough!
Development challenge

- Statistical computing with the cloud often requires low-level, platform-specific code
- Why write hundreds lines of Java & MapReduce, if you can simply write this?

```
initialize B, C, E;
repeat
  F ← O ◦ (C × E × Bᵀ);
  E' ← E ◦ (Cᵀ × F × B);
  C' ← (C × (I ◦ E')) ◦ (F × B × E);
  B' ← (B × (I ◦ E')) ◦ (Fᵀ × C × E);
  B, C, E ← B', C', E';
until termination condition;
```

PLSI (Probabilistic Latent Semantic Indexing), widely used in IR and text mining
Deployment challenge

- Maddening array of choices
  - Hardware provisioning
    - A dozen m1.small machines, or two c1.xlarge?
  - System and software configurations
    - Number of map/reduce slots per machine?
    - Memory per slot?
  - Algorithm execution parameters
    - Size of the submatrices to multiply at one time?

<table>
<thead>
<tr>
<th>Machine Type</th>
<th>Compute Unit</th>
<th>Memory (GB)</th>
<th>Cost ($/hour)</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1.small</td>
<td>1</td>
<td>1.7</td>
<td>0.065</td>
</tr>
<tr>
<td>c1.xlarge</td>
<td>20</td>
<td>7.0</td>
<td>0.66</td>
</tr>
<tr>
<td>m1.xlarge</td>
<td>8</td>
<td>15.0</td>
<td>0.52</td>
</tr>
<tr>
<td>cc2.8xlarge</td>
<td>88</td>
<td>60.5</td>
<td>2.40</td>
</tr>
</tbody>
</table>

Samples of Amazon EC2 offerings
Introducing...

Cumulon

Cumulon

- Simplify both development and deployment of matrix-based statistical workloads in the cloud

**Development**
- **DO** let me write matrices and linear algebra, in R- or MATLAB-like syntax
- **DO NOT** force me to think in MPI, MapReduce, or SQL

**Deployment**
- **DO** let me specify constraints and objectives in terms of time and money
- **DO NOT** ask me for cluster choice, implementation alternatives, software configurations, and execution parameters
How Cumulon optimizes a program

- Program $\rightarrow$ logical plan
  - Logical ops = standard matrix ops
    - Transpose, multiply, element-wise divide, power, etc.
  - Rewrite using algebraic equivalences
- Logical plan $\rightarrow$ physical plan templates
  - Jobs represented by DAGs of physical ops
    - Not yet “configured,” e.g., with degree of parallelism
- Physical plan template $\rightarrow$ deployment plan
  - Add hardware provisioning, system configurations, and execution parameter settings

Like how a database system optimizes a query, but ...
Differences from a database system?

- Higher-level linear algebra operators
  - Different rewrite rules and data access patterns
  - Compute-intensive
    - Element-a-time processing kills performance
- Different optimization issues
  - User-facing: costs now in $$$; trade-off with time
  - In cost estimation
    - Both CPU and I/O costs matter
    - Must account for performance variance
  - A bigger, different plan space
    - With cluster provisioning and configuration choices
    - Optimal plan depends on them!
Cumulon storage/execution model

Design goals:
- Support matrices and linear algebra efficiently
  - Not to be “jack of all trades”
- Leverage popular cloud platforms
  - No reinventing the wheels
  - Easier to adopt and integrate with other code
- Stay generic
  - Allowing alternative underlying platforms to be “plugged in”

A simple, general model
Hadoop/HDFS
MapReduce
Used by many existing systems, e.g., SystemML (ICDE '11)
Why not MapReduce?

• Typical use case
  • Input is unstructured/in no particular order
  • Mappers filter, convert, and shuffle data to reducers
  • Reducers aggregate data and produce results

  • Mappers get disjoint splits of one input file

  ➢ But linear algebra ops often have richer access patterns
  • Next: matrix multiply as an example
Multiplying big matrices: $A \times B$

- Multiply matrix splits; then aggregate \((\text{if } f_l > 1)\)
- Each split is read by multiple tasks \((\text{unless } f_m = f_n = 1)\)
- The choice of split factors is crucial
  - Degree of parallelism, memory requirement, I/O
  - Prefer square splits to maximize compute-to-I/O ratio
    - Multiplying a row with a column is suboptimal!

\[ A: m \times l \]
\[ B: l \times n \]
\[ \text{Result: } m \times n \]
$A \times B$ in MapReduce

- Mappers can’t multiply
  - Because multiple mappers need the same split
- So mappers just replicate splits and send to reducers for multiply
  - No useful computation
  - Shuffling is an overkill
- Need another full MapReduce job to aggregate results
  - To avoid it, multiply rows by columns ($f_i = 1$), which is suboptimal

SystemML’s RMM operator ($f_i = 1$)

Other methods are possible, but sticking with pure MapReduce introduces suboptimality one way or another
Let operators get any data they want, but limit timing and form of communication.

- Store matrices in tiles in a distributed store
  - At runtime, a split contains multiple tiles
- Program = a workflow of jobs, executed serially
  - Jobs pass data by reading/writing the distributed store
- Job = set of independent tasks, executed in parallel in slots
- Tasks in a job = same op DAG
  - Each produces a different output split
  - Ops in the DAG pipeline data in tiles
Still use Hadoop/HDFS, but not MapReduce!

- All jobs are map-only
- Data go through HDFS — no shuffling overhead
- Mappers multiply — doing useful work
- Flexible choice of split factors

Also simplifies performance modeling!
Experiments:
Cumulon vs. SystemML: multiply

- Tested different dimensions/sparsities
- Significant improvement in most cases, thanks to
  - Utilizing mappers better and avoiding shuffle
  - Better split factors because of flexibility

<table>
<thead>
<tr>
<th>Program ID</th>
<th>A ((m \times l))</th>
<th>B ((l \times n))</th>
<th>SystemML split factors (f_m, f_l, f_n)</th>
<th>Cumulon split factors (f_m, f_l, f_n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>24k \times 24k</td>
<td>24k \times 24k</td>
<td>8, 1, 8</td>
<td>4, 4, 4</td>
</tr>
<tr>
<td>2</td>
<td>1 \times 100k</td>
<td>100k \times 100k</td>
<td>1, 1, 50</td>
<td>1, 10, 10</td>
</tr>
<tr>
<td>3</td>
<td>100 \times 100k</td>
<td>100k \times 100k</td>
<td>1, 1, 50</td>
<td>1, 10, 10</td>
</tr>
<tr>
<td>4</td>
<td>160k \times 100</td>
<td>100 \times 160k</td>
<td>8, 1, 8</td>
<td>8, 1, 8</td>
</tr>
<tr>
<td>5</td>
<td>100 \times 20000k</td>
<td>20000k \times 100</td>
<td>1, 80, 1</td>
<td>1, 80, 1</td>
</tr>
<tr>
<td>6</td>
<td>200k \times 200k (0.1)</td>
<td>200k \times 1000</td>
<td>100, 1, 1</td>
<td>10, 10, 1</td>
</tr>
<tr>
<td>7</td>
<td>1000k \times 1000k (0.01)</td>
<td>1000k \times 1</td>
<td>100, 1, 1</td>
<td>10, 10, 1</td>
</tr>
</tbody>
</table>

All conducted using 10 m1.large EC2 instances
Experiments:

Cumulon vs. SystemML: GNMF-1

- Dominant step in Gaussian Non-Negative Matrix Factorization
  - SystemML: 5 full (map+reduce) jobs
  - Cumulon: 4 map-only jobs
Cost estimation for optimization

• Key: estimate time
  • Monetary cost = time $\times$ cluster size $\times$ unit price

• Approach
  • Estimate task time by modeling operator performance
    ➢ *Our operators are NOT black-box MapReduce code!*
    • Model I/O and CPU costs separately
    • Train models by sampling model parameter space and running benchmarks
  • Estimate job time from task time
From task to job times:
A strawman approach

- Job time \( \approx \) task time \( \times \) \#waves?
  - Here \#waves = \(
    \lceil \frac{\#tasks}{\#slots} \rceil
  \)
- But actual job cost is much smoother; why?
  - Task completion times vary; waves are not clearly demarcated

A few remaining tasks may just be able to “squeeze in”
From task to job times: Accounting for variance

- Model for (task time $\rightarrow$ job time) considers
  - Variance in task times
  - #tasks, #slots
    - In particular, how “full” last wave is (#tasks mod #slots)
- Simulate scheduler behavior and train model
Finding optimal deployment plan

- Bi-criteria optimization
  - E.g. minimizing cost given time constraint

- Recall the large plan space
  - Not only execution parameters
  - But also cluster type, size, configuration (e.g., #slots per node)
  - As well as the possibility of switching clusters between jobs

- Optimization algorithm
  - Start with no cluster switching, and iteratively ++ #switches
  - Exhaustively consider each machine type
  - Bound the range of candidate cluster sizes

We are in the Cloud!
Experiments: Deployment plan space

- Optimal execution strategy is **cluster-specific**!
  - 4 clusters of different machine types
  - Find optimal plan for each cluster
  - Run each plan on all clusters

- Optimal plan for a given cluster becomes suboptimal (or even invalid) on a different cluster

`Optimal Job Configurations in Different Types of Cluster`

- Not enough memory even with one slot per machine

*Other experiments show that optimal plan also depends on cluster size*
Experiments: Computing Pareto-optimal plans

- Show cost/time tradeoff across all machine types
  - Each point = calling optimizer with a time constraint and machine type
- Users can make informed decisions easily
- Choice of machine type matters!
  - Entire figure took 10 seconds to generate on a desktop
  - Optimization time is small compared with the savings

*Dominant job in PLSI*
Conclusion

Cumulon simplifies both development and deployment of statistical data analysis in the cloud

- Write linear algebra—not MPI, MapReduce or SQL
- Specify time/money—not nitty-gritty cluster setup
- Simple, general parallel execution model
  - Beats MapReduce, but is still implementable on Hadoop
- Cost-based optimization of deployment plan
  - Not only execution but also cluster provisioning and configuration parameters

➢ See paper for details and other contributions, e.g.:
  - New “masked” matrix multiply operator, CPU and I/O modeling, cluster switching experiments, etc.
For more info, search

*Duke dbgroup Cumulon*

Thank you!