Hadoop Performance Tuning
A case study

Milind Bhandarkar
(Suhas Gogate)
(Viraj Bhat)
Do we really need to tune it?

- Case study: Recommendation Engine
  - 20 dedicated machines ($160,000 for 3 Years)
  - 3 Hours runtime every day
    - Needed 15% improvement per year
      - Equivalent to 3 additional machines ($24,000 for 3 years)
  - 4 Full-time programmers
    - 2 months for needed improvement ($120,000)

- Throw more hardware at it!
  - Happy Ending: 100 machines, 1 hour runtime = $20 per day
  - $22000 for 3 years
Hadoop

- Highly configurable commodity cluster computing programming framework
  - Hides messy details of parallelization and exposes only sequential map-reduce API for programmers
  - Hadoop run-time system takes care of partitioning, scheduling and program execution
    - Hints from user
    - Allows programmers having little/no experience in parallel programming write parallel programs
    - Exposes various knobs to tuning jobs

- Tuning individual jobs is a non-trivial task
  - 165+ tunable parameters in hadoop framework
  - Tuning a single parameter can have adverse impact on the other
Map-Reduce Application Optimization

- Job Performance – user perspective
  - Reduce end-to-end execution time
    - Yield quicker analysis of user data

- Cluster Utilization – provider perspective
  - Service provider to make use of cluster resources across multiple users
  - Increase overall throughput in terms of number of jobs/unit time
Approach

- Application Agnostic
  - Without instrumentation or modification of user code
  - Instrument the underlying run-time hadoop (map-reduce) framework

- Case Study
  - Model Trainer based on MaxEnt
  - Iterative Application
    - Each iteration is a Map-Reduce job
  - Uses Hadoop Streaming
  - Written in C++
MaxEnt

- Maximum entropy models are considered promising avenues for text classification
  - Long training times make entropy research difficult

- To offset this, trainer uses Hadoop Map-Reduce programming framework to speed execution
  - 200 machines, 16-18 Hours runtime
MaxEnt (contd)

- MaxEnt uses Hadoop Streaming
- C++ Binaries as Mappers and Reducers
- File-based splits
  - large "mapred.min.split.size" to allow file size splits
    - Results 5.9k maps each map operating on 230MB of idx data
MaxEnt baseline

- Optimizations already used
  - Speculative execution
  - Cache intermediate results in the iteration stages using -cacheFile option

- Establish Baseline
  - Benchmark MaxEnt code on dedicated cluster for reproducible results

<table>
<thead>
<tr>
<th>Hadoop Version</th>
<th>Time sec (M:R)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.17</td>
<td>1545 (5900:800)</td>
</tr>
<tr>
<td>(100 nodes)</td>
<td></td>
</tr>
<tr>
<td>0.18</td>
<td>1459 (5900:800)</td>
</tr>
<tr>
<td>(100 nodes)</td>
<td></td>
</tr>
</tbody>
</table>
Methodology for Tuning MaxEnt

- Changing Map/Reduces
- Reducing Intermediate data
  - Use of combiners
- Reducing disk spill on map side
- Compressing map output
- Reducing Reduce side disk spill
- Effect of Increasing Maps Slots
Changing Number of Maps/Reduces

- Optimal number of Maps and Reduces
  - Typically larger number of maps are better
    - Low map re-execution time in case of failure
    - Computation to Communication overlap is better
  - Typically 0.95% of total reduce slots is a good number for reduce tasks
    - If map local o/p is very large (exceeding the local disk capacity)
    - Reduce logic needs scale out (cpu bound)
    - In this case typically use larger cluster for the job.
  - Shuffle requires M * R transfers

<table>
<thead>
<tr>
<th>Hadoop Version</th>
<th>Time sec (M:R)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.17</td>
<td>1545 (5900:800)</td>
</tr>
<tr>
<td>0.17</td>
<td>1740 (12,300:380)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hadoop Version</th>
<th>Time sec (M:R)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.18</td>
<td>1459 (5900:800)</td>
</tr>
<tr>
<td>0.18</td>
<td>1197 (5900:380)</td>
</tr>
<tr>
<td>0.18</td>
<td>1255 (5900:200)</td>
</tr>
</tbody>
</table>
## Use of combiner

<table>
<thead>
<tr>
<th>Hadoop Version</th>
<th>Combiner</th>
<th>Time sec (M:R)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.17</td>
<td>No</td>
<td>1545 (5900:800)</td>
</tr>
<tr>
<td>0.17</td>
<td>Yes</td>
<td>1256 (5900:800)</td>
</tr>
<tr>
<td>0.18</td>
<td>No</td>
<td>1197 (5900:380)</td>
</tr>
<tr>
<td>0.18</td>
<td>Yes</td>
<td>934  (5900:380)</td>
</tr>
</tbody>
</table>
Combiner Efficiency

Significant gain is obtained in using the combiner at both map and reduce stages in the feature count application.

Combiner Efficiency = \[1 - \frac{(\text{Combine O/P Records})}{(\text{Combiner I/P Records})}\] * 100

- Combiner Efficiency at map stage is around 40%
- Combiner Efficiency at reduce stage is around 22%
Avoiding Map side Disk spill

- **P** = Partition
- **KS** = Key Start, **VS** = Value Start
- Sort buffer is a byte array which wraps around
- Partition and Index buffers maintain location of key and value for records for in-memory quick sort
- Data is spilled to disk when the **io.sort.record.percent** exceeds
  - For both the index and data buffers
Reducing disk spill on map side

<table>
<thead>
<tr>
<th>Hadoop Version</th>
<th>io.sort.mb</th>
<th>Time sec (M:R)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.18</td>
<td>256</td>
<td>934 (5900:380)</td>
</tr>
<tr>
<td>0.18</td>
<td>350</td>
<td>921 (5900:380)</td>
</tr>
</tbody>
</table>
## Analyzing effect of minimizing disk spill

<table>
<thead>
<tr>
<th>Time sec (M:R)</th>
<th>Map Local Bytes Read TB</th>
<th>Map Local Bytes Written TB</th>
<th>Map HDFS Bytes Read TB</th>
<th>Avg. Map Time sec</th>
<th>Avg Reduce Time sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>934 (5900:380)</td>
<td>1.6</td>
<td>2.6</td>
<td>1.29</td>
<td>42</td>
<td>872</td>
</tr>
<tr>
<td>921 (5900:380)</td>
<td>0</td>
<td>0.789</td>
<td>1.29</td>
<td>39</td>
<td>853</td>
</tr>
</tbody>
</table>
Effect of compressing map output

<table>
<thead>
<tr>
<th>Hadoop Version</th>
<th>Map Output Compression</th>
<th>Time sec (M:R)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.18</td>
<td>No</td>
<td>921 (5900:380)</td>
</tr>
<tr>
<td>0.18</td>
<td>Yes</td>
<td>1308 (5900:380)</td>
</tr>
</tbody>
</table>

6/29/09
Reducing Reduce side disk spill

- Reduce side disk spill is caused when merging map-output causes the hadoop framework to write merges to local disk

- Reduce side disk spill is difference between total map output bytes (MOB) for all maps and total reducer local bytes written (RLBW)

\[ \sum_{m=1}^{n_{\text{maps}}} \text{MapOutBytes} - \sum_{r=1}^{n_{\text{reducers}}} \text{ReducerLocalBytesWritten} \]

- 2 parameters determine reduce side disk spill
  - fs.inmemory.size.mb determines size of merge buffer allocated by the framework
  - io.sort.factor determines the number of segments to keep in memory before writing to disk

6/29/09
Effect of Minimizing Reduce side disk spill

- `fs.inmemory.size.mb = 350MB` and `io.sort.factor = 380`
- Larger amount of memory allocated for the in-memory file-system used to merge map-outputs at the reduces
- `io.sort.factor` was increased to keep more segments in memory before merging

<table>
<thead>
<tr>
<th>Hadoop Version</th>
<th>Time sec (M:R)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.18</td>
<td>1308 (5900:380)</td>
</tr>
<tr>
<td>0.18</td>
<td>834 (5900:380)</td>
</tr>
</tbody>
</table>
Increasing number of map and reduce slots

- Normal nature of the map-reduce computations require
  - Larger number of maps compared to reduces

- Increase number of map slots depending on nature of map computations
  - If map is cpu intensive increasing map slots could result in slowing of map-tasks
  - If map is io-intensive increasing map slots could result in increasing local disk contention

- **Caveat**
  - Consider resources used by task tracker and data nodes when increasing map or reduce slots
Opt 6: Effect of Increasing Maps Slots

<table>
<thead>
<tr>
<th>Hadoop Version</th>
<th>Map Slots + Reduce Slots</th>
<th>Time sec (M:R)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.18</td>
<td>4 + 4</td>
<td>834 (5900:380)</td>
</tr>
<tr>
<td>0.18</td>
<td>7 + 4</td>
<td>732 (5900:380)</td>
</tr>
</tbody>
</table>
Conclusions

- Application agnostic tuning was beneficial for the MaxEnt application
  - Runtime reduced from 1459 seconds to 732 seconds
  - Total time spent: 2 days

- Additive nature of optimization may not be true
  - Tuning one parameter can have adverse impact on the other
Vaidya

- Hadoop Vaidya (Vaidya in Sanskrit language means "one who knows", or "a physician")
Hadoop Vaidya: Rule based performance diagnostics

- Rule based performance diagnosis of M/R jobs
  - M/R performance analysis expertise is captured and provided as an input through a set of pre-defined diagnostic rules
  - Detects performance problems by postmortem analysis of a job by executing the diagnostic rules against the job execution counters
  - Provides targeted advice against individual performance problems

- Extensible framework
  - Can add more and more hints
    - based on a hint template and published job counters data structures
  - Write complex hints using existing simpler hint
Hadoop Vaidya

- Input Data used for evaluating the rules
  - Job History, Job Configuration (xml)

- Patch committed: contrib/vaidya (Hadoop 0.20.0)
  - [http://issues.apache.org/jira/browse/HADOOP-4179](http://issues.apache.org/jira/browse/HADOOP-4179)

- Future enhancements
  - Online progress analysis of the Map/Reduce jobs
  - Adapter for Chukwa: Performance data collection system
<DiagnosticTest>
  <Title>Balanaced Reduce Partitioning</Title>
  <ClassName>org.apache.hadoop.chukwa.vaidya.postexdiagnosis.tests.BalancedReducePartitioning</ClassName>
  <Description>This rule tests as to how well the input to reduce tasks is balanced</Description>
  <Importance>High</Importance>
  <SuccessThreshold>0.20</SuccessThreshold>
  <Prescription>advice</Prescription>
  <InputElement>
    <PercentReduceRecords>0.85</PercentReduceRecords>
  </InputElement>
</DiagnosticTest>
<TestReportElement>
  <TestTitle>Balanced Reduce Partitioning</TestTitle>
  <TestDescription>This rule tests as to how well the input to reduce tasks is balanced</TestDescription>
  <TestImportance>HIGH</TestImportance>
  <TestResult>POSITIVE</TestResult>
  <TestSeverity>0.16</TestSeverity>
  <ReferenceDetails>* TotalReduceTasks: 4096
  * BusyReduceTasks processing 0.85% of total records: 3373
  * Impact: 0.17</ReferenceDetails>
  <TestPrescription>* Use the appropriate partitioning function
  * For streaming job consider following partitioner and hadoop config parameters
    * org.apache.hadoop.mapred.lib.KeyFieldBasedPartitioner
    * -jobconf stream.map.output.field.separator, -jobconf stream.num.map.output.key.fields</TestPrescription>
</TestReportElement>
Executing and Writing your own tests

- Execute $HADOOP_HOME/contrib/vaidya/bin/vaidya.sh
  - Job's configuration file -jobconf job_conf.xml
  - Job history log file -joblog job_history_log_file
  - Test description file -testconf postex_diagnostic_tests.xml
    - Default test is available at: $HADOOP_HOME/contrib/vaidya/conf/postex_diagnostic_tests.xml
    - Location of the final report file -report report_file

- Writing and executing your own test rules is not very hard
  - Writing a test class for your new test case should extend the org.apache.hadoop.vaidya.DiagnosticTest
  - evaluate()
  - getPrescription()
  - getReferenceDetails()