大数据框架与应用评测方面相关研究工作

Lijie Xu

Dec. 2016

A Comparison of Approaches for Large-Scale Data Mining (Utilizing MapReduce in Large-Scale Data Mining)

|  |  |
| --- | --- |
| 来源 | 2010 The University of Texas at Dallas技术报告 |
| 摘要 | In this paper, we analyze existing approaches for large-scale data mining and compare their performance to the MapReduce model. Based on our analysis, a data mining framework that integrates MapReduce and sampling is introduced and discussed. |
| 对比目的 | 三种数据挖掘方法(Sampling, ensemble methods, MapReduce based approaches) 的正确率 |
| 数据集 | 30GB DMOZ ([www.dmoz.org](http://www.dmoz.org)) html pages |
| 应用 | Naïve Bayes Classifier  Sampling和Ensemble方法中使用 CMU Rainbow分类器  MapReduce 方法使用Mahout中的朴素贝叶斯分类器 |
| 测试方法 | 每个实验运行10次，在每次实验中，选取50%作为的数据作为training set，剩下的50%的作为test set。Sampling大小为1000, 2000, 5000, 10000 |
| 研究问题 | 验证增加样本大小是否可以提升正确率：Sample dataset上的数据运算结果与全数据集上的运算结果正确率对比 |
| Ensemble算法发现 | 使用5个sub-model比10个sub-model正确率要高 |
| Mahout算法发现 | 每一轮 (round) 的正确率都有微小变动 |
| 结论 | 1. The experiments show the classifiers built on the entire data set have better accuracy than the one built on sampled data set.  2. Ensemble approach overall has the highest accuracy among the three approaches. But the accuracy of the three approaches are quite close. |
| 相关有趣的参考文献 | 1. C.T.Chu,S.K.Kim,Y.A.Lin,Y.Yu,G.R. Bradski, A. Y. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. pages 281–288. MIT Press, 2006.  2. M. Kearns. Efficient noise-tolerant learning from statistical queries. J. ACM, 45(6):983–1006, 1998. (linear speedup)  3. C. Moretti, K. Steinhaeuser, D. Thain, and N. V. Chawla. Scaling up classifiers to cloud computers. In ICDM ’08 |
| 启发 | 1. 通过sampling数据，测试不同数据量上的算法正确性和运行时间 |

A Comparison of Platforms for Implementing and Running Very Large Scale Machine Learning Algorithms

|  |  |
| --- | --- |
| 来源 | SIGMOD 2014长文 |
| 摘要 | We describe an extensive benchmark of platforms available to a user who wants to run a machine learning (ML) algorithm over a very large data set, but cannot find an existing implementa ion and thus must “roll her own” ML code. |
| 目的 | 对比不同机器学习平台上应用的执行之间，调参需求和编程难度 |
| 贡献 |  |
| 应用选取标准 | We focus on learning relatively complex, hierarchical models. Such models are often characterized by the fact that the Bayesian approach to ML. 并使用马尔科夫链蒙克卡罗方法MCMC方法来求解后验估计问题  (1) A Gaussian mixture model (GMM) for data clustering.  (2) The Bayesian Lasso, a Bayesian regression model.  (3) A hidden Markov model (HMM) for text.  (4) Latent Dirichlet allocation (LDA), a model for text mining.  (5) A Gaussian model for missing data imputation.  Task: 使用马尔可夫链蒙特卡洛方法来计算后验概率 |
| 平台选取 | Spark (MapReduce), SimSQL (a parallel, relational database that supports an SQL-based approach to running large-scale MCMC simulations), GraphLab (Graph-based), Giraph (BSP approach)  GraphLab与Giraph的区别：  Giraph是Pregel的开源版本，是push-based, synchronous的  GraphLab是pull-based, asynchronous的 |
| 实验环境 | AWS EC2 m2.4xlarge (8 virtual cores, two disks, 68GB RAM), 7000 hours |
| GMM数据生成方法 | 文中介绍了Gaussian mixture model数据生成方法 |
| GMM实现方法 | Spark实现（使用了Python版本）  SimSQL实现（使用了UDF）  GraphLab实现其中的Gibbs sampler使用了gather-apply-scatter模型  Giraph实现方式与GraphLab一样  对于每一种平台，分别使用5台、20台、100台机器来计算，每台机器固定生成10,000,000个样本点（10维）。然后再使用一个数据集是的每台机器固定生成1,000,00个样本点（100维），100维的数据只在5台机器上测试 |
| GMM结论 | SimSQL在100维时，运行时间是Spark的两倍。原因是  The reason for this is that the GMM simulation must aggregate one matrix (which is a 10,000 entry matrix for 100-dimensional data) for each data point. In SimSQL, this is performed using a costly GROUP BY, which is slower than the Spark matrix/vector operations.  Spark Java版本和Python版本的运行时间不同：  For the ten-dimensional inference problem, Java takes around 50% of the Python time. But for the 100- dimensional problem, it is more than eight times slower. 原因是the cost of 100-dimensional Java linear algebra.  再使用一个优化方法 (super vertex construction) 以后，SimSQL in particular can be made to run extremely fast using a super vertex construction; the 100-dimensional GMM implementation ran in a time that was only 20% of its nearest competitor (GraphLab). |
| Bayesian Lasso实验方法 | We created a synthetic data set having 103 regressor dimensions and a one-dimensional response. As in GMM experiments, we held the number of data points per machine constant at 105 and tested compute clusters consisting of 5, 20, and 100 machines. |
| Bayesian Lasso实验发现 | In the case of the Bayesian Lasso, the simulation converges very quickly, magnifying the importance of the initialization.  SimSQL’s relatively slow performance can be explained by its lack of support for vector and matrix operations.  SimSQL’s Relatively High Per-Iteration Times. SimSQL is also slow on a per-iteration bases. SimSQL takes about ten times as long as Spark, 20 times as long as GraphLab, and five times as long as Giraph, per iteration. |
| HMM模型测试 | We create a synthetic document database. To create each “document”, we choose two news- group postings from the ubiquitous 20 newsgroups data set and concatenate the postings end-on-end. Since there are 20,000 posts in this data set, it was possible to create up to 400 million different synthetic documents in this way. A dictionary size of 10 thousand words and used K = 20 different states. We used 2.5 million documents on each machine.  Why was Giraph so much faster than GraphLab, and why does it scale better?  Giraph provides a richer programming interface that allowed us to sidestep some of the more serious computational and memory- related problems that are associated with mapping the simulation to a graph. |
| LDA模型测试 | We used the document database used in the HMM experiments, a dictionary size of 10,000 words, and a model size of 100 topics.  Everyone Fails Except for SimSQL. The LDA simulation is quite similar to the HMM simulation, particularly the document-based and super vertex versions. |
| Gaussian imputation | As before, we kept the amount of data constant at 10 million data points per machine, and tested the per-iteration running times at five, 20, and 100 machines. |
| 启发 | 重点关注Spark的扩展性，扩展性测试只扩展了机器数目。  问题原因可能跟cache()有关。  当数据量大时候会出现Out of memory错误  Both Giraph and GraphLab suffer greatly from memory-related difficulties, and for that reason could not be made to run on the largest, most complicated problems. The data sets were not too large to fit into RAM. Our largest data set was around 1TB in size, which fits comfortably in the 7TB of aggregate RAM of our 100 machine compute cluster.  Giraph scaled better than GraphLab, seemingly because Giraph’s BSP-based model maps nicely to the bipartite structure of the models tested. |
| 结论 | Spark + Python was (in our opinion) the most attractive platform simply in terms of ease-of-coding, though it was challenging to get it to work on the larger and more complicated problems.  GraphLab was fast on smaller problems, but it did not scale well. This might be because none of the models we considered naturally map to a graph.  Giraph, on the other hand, which is really more of a BSP platform than a graph platform, did very well, though memory was an issue on the largest problems. |
| 关于Spark的评论 | On the positive side, Spark codes (particularly those written in Python) are incredibly short and beautiful. Spark’s succinctness rivals that of SimSQL’s SQL codes, though many users will find Spark codes to be preferable, since they are imperative/functional.  On the negative side, Spark was slower than the two graph-based platforms. We were a bit worried these results were related to our choice of Python instead of Scala or Java. We had already tested both Python and Java GMM implementations, but just to be sure, we tested a Java Spark LDA implementation as well. The results are shown in Figure 6. The speed is much better than the Python implementation, but we could still not get Spark to run the LDA inference algorithm on 100 machines. The implementation failed on 20 machines after 18 iterations as well.  Spark in general required a lot of tuning and experimentation to get things to work on large and/or complicated problems. There was some disagreement among the authors of this paper as to why. One explanation is that Spark relies greatly upon techniques such as “lazy evaluation” for speed and job scheduling, which looks a lot like pipelining in a database system. However, **database systems often use statistical information to decide when and how to pipeline.** Without this, it is easy to make bad decisions, and in general, we spent a lot of time tuning Spark, doing things such as forcing RDDs to disk. Perhaps the ultimate solution is to make Spark—and other dataflow systems—work more like a database system, carefully planning computational choices such as RDD materialization and pipelining using cost models. |

扩展性测试除了与机器数（cores有关）外，还与系统资源配置（比如Executor大小），处理框架配置（如input split size，partition number），和应用参数（比如树的高度，K-means中K的个数）有关。

看一下所有大数据相关系统论文的扩展性测试方法

Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing

|  |  |
| --- | --- |
| 来源 | NSDI 2012 Spark系统原始论文 |
| 摘要 |  |
| 应用选取标准 | Logistic regression  k-means |
| 平台选取 | Hadoop 0.20.2  HadoopBinMem: A Hadoop deployment that con- verts the input data into a low-overhead binary format in the first iteration to eliminate text parsing in later ones, and stores it in an in-memory HDFS instance.  Spark |
| 实验环境 | 25-100 machines, used m1.xlarge EC2 nodes with 4 cores and 15 GB of RAM. We used HDFS for storage, with 256 MB blocks. Before each test, we cleared OS buffer caches to measure IO costs accurately. |
| 测试应用及方法 | **K-means和logistic regression：**Run both algorithms for 10 iterations on 100 GB datasets using 25–100 machines. 但是没有说数据特征。  **PageRank:** 54GB Wikipedia dump. We ran 10 iterations of the PageRank algorithm to process a link graph of approximately 4 million articles. |
| 发现 | K-means: 计算密集型  Logistic regression: less compute-intensive and thus more sensitive to time spent in deserialization and I/O. |
| K-means和LR实验结果 | **First Iterations:** Spark was moderately faster than Hadoop across experiments. **This difference was due to signaling overheads in Hadoop’s heartbeat protocol between its master and workers.**  **Subsequent Iterations:** Figure 8 shows **how these scaled with cluster size**. For logistic regression, Spark 25.3× and 20.7× faster than Hadoop and HadoopBinMem respectively on 100 machines. For the more compute-intensive k-means application, Spark still achieved speedup of 1.9× to 3.2×.  **Understanding the Speedup:** Hadoop still ran slower due to several factors:  1. Minimum overhead of the Hadoop software stack.  2. Overhead of HDFS while serving data.  3. Deserialization cost to convert binary records to usable in-memory Java objects.  Overhead分析：为了证明上述三个论点，还做了读写本地磁盘与读写HDFS的对比，读写不同格式（纯文本、binary）数据的对比，从内存是文件数据解析转换为Java对象需要的时间。 |
| K-means和LR启发 | 测试迭代型算法 (如k-means和logistic regression) 的时候需要区分第一次迭代和后面迭代的用时 (duration)。  扩展性测试中只对比了不同机器数下的运行时间，没有考虑系统参数和应用参数。 |
| PageRank实验结果 | In-memory storage alone provided Spark with a 2.4× speedup over Hadoop on 30 nodes. In addition, **controlling the partitioning of the RDDs to make it consistent across iteration**, improved the speedup to 7.4×. The results also **scaled nearly linearly** to 60 nodes. 通过控制partition个数可以进一步提高性能，基本可以达到线性扩展能力。 |
| 错误容忍机制测试 | Figure 11 compares the running times for 10 iterations of k-means on a 75-node cluster in normal operating scenario, with one where a node fails at the start of the 6th iteration. Without any failure, each iteration consisted of 400 tasks working on 100 GB of data. 在运行过程中让一个机器强制宕机。 |
| 内存不足时测试 | A natural question is how Spark runs if there is not enough memory to store a job’s data. We present results for various amounts of storage space for logistic regression in Figure 12. We see that performance degrades gracefully with less space. |
| 真实应用的扩展性问题 | Twitter Spam Classification: 自己实现的Logistic regression, 没有达到线性扩展性，原因是The scaling is not as close to linear due to a higher fixed communication cost per iteration. |
| 交互式应用 | Analyze 1TB of Wikipedia **page view logs** (2 years of data)  For this experiment, we used 100 m2.4xlarge EC2 instances with 8 cores and 68 GB of RAM each.  交互式应用目的：We ran queries to find total views of (1) all pages, (2) pages with titles exactly matching a given word, and (3) pages with titles partially matching a word. Each query scanned the entire input data.  Page view logs有用.  Even at 1 TB of data, queries on Spark took 5–7 seconds. This was more than an order of magnitude faster than working with on-disk data; for example, querying the 1 TB file from disk took 170s. |

Scaling Spark in the Real World: Performance and Usability

|  |  |
| --- | --- |
| 来源 | VLDB 2015短文 |
| 摘要 | We describe the main challenges and requirements that appeared in taking Spark to a wide set of users, and usability and performance improvements we have made to the engine in response. |
| Debugging and profiling | 问题：We found that the most challenging issues are in performance debugging: users often do not realize that their work is **concentrated on a few machines**, or that some of their data structures are **memory-inefficient**. |
| Memory management | While external operations for aggregation and joins are well-understood, we found other sources of high memory use. Data records in some applications (e.g., image processing) can be hundreds of megabytes each, requiring careful accounting as each record is read.  Spark initially assumed that the data in each block of a file (typically 128 MB in HDFS) can all fit in memory at once, but for some highly com- pressed datasets, each block could decompress into 3-4 GB. |
| 内存管理改进 | Implemented a per-node allocator that manages all sources of memory usage within each node. Spark initially had a memory manager to **track the size of “cached” data that the user chose to materialize in memory**, evicting old data blocks when a cap was reached.  The original manager did not explicitly track the memory usage for data processing (e.g., scratch space used for joins or aggregations). As a result, a large fraction of the memory exhaustion problems came from processing large joins or aggregations.  To address that, **we implemented a second cap to track hash tables for joins and aggregation.** This cap is allocated dynamically among the threads running these operations as they grow their tables, and threads that are not allowed to take more RAM spill to disk.  Lastly, **a third space was reserved for “unrolling” blocks that are read from disk to see whether the uncompressed data is still small enough to cache. (unrolling区域的解释)**  In all these cases, **we check memory usage every 16 records to handle skewed record sizes**. With these controls, the engine runs robustly across a wide range of workloads. |
| 网络层优化 | 从Java NIO到Netty：  **• Zero-copy I/O: Instruct the kernel to copy data directly from on-disk files to the socket, without going through the user-space memory.** This reduces not only the CPU time spent in context switches between kernel and user space, but also the memory pressure in the JVM heap.  **• Off-heap network buffer management:** Netty maintains a pool of memory pages explicitly outside the Java heap, and as a result eliminates the impact of network buffers on the JVM garbage collector. 有意思  **• Multiple connections: Each Spark worker node maintains multiple parallel active connections (by default 5) for data fetches**, in order to increase the fetch throughput and balance load across the nodes serving data. |

Making Sense of Performance in Data Analytics Frameworks

|  |  |
| --- | --- |
| 来源 | NSDI 2015 |
| 摘要 | In this paper, we develop blocked time analysis, a methodology for quantifying performance bottlenecks in distributed computation frameworks, and use it to analyze the Spark framework’s performance on two SQL benchmarks and a production workload. Contrary to our expectations, we find that (1) CPU (and not I/O) is often the bottleneck, (2) improving network performance can improve job completion time by a median of at most 2%, and (3) the causes of most stragglers can be identified. |
| 贡献 | (1) We develop a methodology for analyzing end-to-end performance of data analytics frameworks; (2) we use our methodology to study performance of two SQL benchmarks and one production workload. |
| 性能瓶颈分析方法 | Blocked time analysis. Blocked time analysis uses **extensive white-box logging** to measure how long each task spends blocked on a given resource. The **per-task measurements** for a particular job allow us to simulate how long the job would have taken to complete if the disk or network were infinitely fast, which provides an upper bound on the benefit of optimizing network or disk performance. |
| 重要发现 | 1. Network optimizations can only reduce job completion time by a median of at most 2%.  2. Optimizing or eliminating disk accesses can only reduce job completion time by a median of at most 19%.  3. Optimizing stragglers can only reduce job completion time by a median of at most 10%, and in 75% of queries, we can identify the cause of more than 60% of stragglers.  (Blocked-time analysis illustrates that the **two leading causes of Spark stragglers** are **Java’s garbage collection** and **time to transfer data to and from disk**.) |
| Benchmark | (1) Big data benchmark (two SQL queries, Join, PageRank-like query)  (2) TPC-DS (reporting, interactive OLAP, data mining queries) 从中挑选了20个queries，在不同情况下运行，在20台机器上运行   1. Store data on-disk using Parquet, scale = 5000 2. In-memory: scale = 100 (SparkSQL’s cache is not well optimized for the type of data used in the TPC-DS benchmark, so **while the data only takes up 17GB in the compressed on-disk format, it occupies 200GB in memory**).   (3) Ad-hoc Spark queries: Input data for the queries includes **a large fact table with over 50 columns**. The workload includes a small number of ETL queries that read input data from an external file system into the memory of the cluster; subsequent queries operate on in-memory data and are b**usiness-intelligence-style queries** that aggregate and summarize data. |
| 性能瓶颈难以分析的原因 | **Tasks use pipelining.** Tasks often use multiple resources simultaneously, and different resources may be the bottleneck at different times in task execution.  Jobs are composed of many tasks that run in parallel, and each task in a job may have a unique performance profile. |
| 分析方法 | **Blocked time analysis.** Quantify how much more quickly a job would complete if tasks never blocked on the disk or the network. |