CSE 598 - Reading Assignment - ML on RDBMS

Sreenivasan Ramesh
sreenivasan.ramesh@asu.edu

[1] Jankov, Dimitrije, et al. "Declarative recursive computation on an RDBMS: or, why you should use a database for distributed machine learning." *Proceedings of the VLDB Endowment* 12.7 (2019): 822-835.

[2] Boehm, Matthias, et al. "Systemml: Declarative machine learning on spark." *Proceedings of the VLDB Endowment* 9.13 (2016): 1425-1436.

[3] Feng, Xixuan, et al. "Towards a unified architecture for in-RDBMS analytics." *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 2012.

Summary of *Declarative Recursive Computation on an RDBMS*

Most currently used Deep Learning frameworks have been designed with data parallelism in mind, where near identical computations are executed in parallel over multiple compute units. The main drawback of this approach is that it assumes that the model can be fit into memory of the computational unit. Handling such models require model parallelism, where the model is partitioned and executed across multiple computational units as a series of parallel bulk-synchronous operations. However, most of the currently used frameworks have limited support for model parallel training.

In this paper, the authors make small set of modifications to an RDBMS to make it suitable for model parallelism. They state that different parts of a model can be stored in a set of tables and any required computations could be expressed through a few simple SQL queries. A departure from the current imperative and DSL style approach taken by current deep learning frameworks, the authors argue that using a RDBMS would provide a declarative programming approach allowing automatic code generation and optimization.

One of the biggest challenges to supporting large scale DL on traditional RDBMS systems is the lack of RDBMS support for recursion. DL tasks require multiple forward and backward passes of a network to propagate errors. It is possible to perform DL computations using Relational Algebra statements; However, we would need to use a table to pass state (storing errors being backpropagated through each neuron) between multiple iterations through the network. This state table can exponentially and materializing this table becomes inefficient. In order to overcome this issue, we need to get rid of the imperative style for-loop and replace it by a declarative query expression behind which sits a pipelined computation - creating new data for the table directly into the next stages. To achieve this, the authors have extended SQL to support multiple versions of a table, where versions are accessed like array styled indexes. This allows the easy unwinding of the recursion and the computation can be expressed as a single but un-optimized relational algebra plan (DAG). The paper then goes into detail about optimizing the plan by decomposing it into frames, reducing the inter-frame communication by probabilistically approximating the number of pipeline breakers, and finding the best plan decomposition using a greedy approximation to the generalized quadratic assignment problem.

Strengths of the paper

- **Declarative programming approach** - allows for automatic code generation and provides more opportunity for automatic optimizations.
- **Large Model Parallelism** - allowing for models with very large hidden layers, which is not possible with TensorFlow and current DL frameworks.
- **Word2Vec and LDA performance for large topics** - With very large dimension/topics, the extended RDBMS approach outperforms TensorFlow and Spark by a huge margin.

Weaknesses of the paper

- **Greedy Approximation** to the generalized quadratic assignment problem depends on the initial node selected. Running the approximation for every starting operation might become expensive.
- **GPU Performance for Neural Networks** - For models which can fit in memory, TensorFlow is much faster.
- **High latency** - The implementation was done on top of a research prototype, and the high latency of the Java/Hadoop system does not make it production ready.

Summary of *SystemML: Declarative Machine Learning on Spark*

Large scale data has made mandatory the use of distributed and data-parallel approaches to machine learning. These requires building a significant number of custom algorithms and also being able to run them in a distributed setting in a data-parallel approach. SystemML provides a platform for declarative large-scale machine learning with automatic generation of hybrid runtime plans. It specifically addresses the problems by providing declarative ML (DML), which allows for flexible algorithm specifications, reusability of code, simplified deployments for different environments and automatic optimization of runtime plans. SystemML's DML supports linear algebra primitives, rich statistical functions and has support for user defined functions. A DAG of high-level operators (HOPs) are created from the DML and an optimizer performs optimizations on the HOP DAGs after which they are decomposed to DAGs of low-level operators (LOPs). An operator selection selects the best physical operator based on memory estimates, data and cluster characteristics to generate an executable of run time instructions. A driver program executes the runtime program and also handles recompilation, runs in-memory node instructions, maintains an in-memory buffer pool and launches Spark or MR jobs. The paper also details several of the Spark specific optimizations such as checkpoint injection and repartition injection. The. Hybrid runtime plans are also compiled with inmemory single node execution type and execution type Spark operators, which allows for high performance gains. SystemML also supports parallel task computation using parallel for loops and hybrid parallelization strategies. The Spark backend allows for lazy evaluation simplifying the runtime execution and allowing for hybrid runtime plans. The paper also details some of the implications of using dynamic recompilation, partition preserving operations and other run time optimizations such as the lazy spark context creation and short circuit read and collect.

Strengths

- **Declarative ML** - A declarative framework allows us to automatically exploit optimizations such as distributed caching and partitioning via rewrites. Also allows for reusability of code and simplifies deployments for different environments due to automatic optimization.
- **Memory Efficient** - The compact data structures along with serialization and CSR techniques avoid cache spilling. It also allows for custom serialization and compression.
- **Use of Spark** allows for general **DAG structure with lazy RDD evaluation**, common data structure and **stateful distributed in-memory caching**. The lazy RDD evaluation allows for automatic grouping of operations into distributed jobs.

Weaknesses

- **Limited Support for Deep Learning tasks** – limited GPU support (no compilation support for cost based or automatic placement; no multi-GPU support) and experimental parameter server and sparsity estimators provide for limited support for deep learning tasks.
- **Shuffling overhead** - can be a problem if the partitioning is performed naively.
- **Performance with large data –** For large data sizes, the cp+spark or cp+mr mode does not provide any significant advantage over the spark only mode.

Summary of *Towards a Unified Architecture for in-RDBMS analytics*

There is an increase in the demand for RDBMS which natively support large scale statistical and data analytics. However, a significant problem is in the engineering of and supporting different statistical techniques as there exists no unified architecture for in-database implementation of these analytics. The authors argue that many of these common analytics tasks can be represented as solving convex programming problems - i.e. minimization problems; and propose a single unified architecture - Bismarck, to solve this class of problems. The unified architecture is based on using Incremental Gradient Descent (IGD) to solve a variety of analytical tasks. They show that IGD can be implemented using User Defined Aggregate (UDA) features, which are commonly used in most RDMBS. The UDA mechanism is used to run the IGD computation and also check for convergence. With this new unified architectural abstraction new models can be added and tested very easily. The authors also study the optimization opportunities such as compression, data ordering, data layouts and parallelism. To avoid the shuffling overhead, shuffling the data once may suffice to achieve a good convergence rate, but may still take a long time to shuffle the dataset once. To avoid this, we can subsample the data using reservoir sampling. The paper proposes a modified version of reservoir sampling – Mixed Reservoir Sampling to reduce the shuffling overhead. They conclude with a performance evaluation which shows their prototype implementation outperforming existing commercial DBMS by a factor of at least 2 for simple tasks and an order of magnitude faster for newly added tasks such as matrix multiplication.

Strengths

- **Reduced engineering effort** - A single framework can be used to implement different algorithms and it also allows for code reusability.
- **Portability between RDBMS** - Since Bismarck is based on UDA, which is supported by most common RDBMS, the algorithms can easily be ported over to other RDBMS.
- **Performance** - Initial prototype implementations are able to outperform existing built in tools provided by current RDBMS by a significant margin.

Weaknesses

- **Using IGD for all convex problems –** Not all problems can be solved by IGD and these problems cannot be tackled by this approach. Also, IGD may not be the optimal way to solve all optimization problems.
- **Small Model Size** - There is a strong assumption that the state/model is able to fit in memory.
- **No support for distributed architectures** - Since Bismarck is designed for a single node multicore system, it has to be redesigned to work with distributed databases. There is also no performance comparison against distributed compute frameworks such as Spark.