**Part2. Review Summary**

**Lecture2.**

* **[**[**Architecture: Compute+Overall**](http://pages.cs.wisc.edu/~akella/CS744/S19/838-CloudPapers/DCAsAComputer.pdf)**Optional (B+09)]***The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*, L.A. Barroso, U. Holzle, Synthesis Lectures on Computer Architecture, 2009. Chapter 1 and 2.
* **[**[**Architecture: Networks**](http://research.microsoft.com/pubs/80693/vl2-sigcomm09-final.pdf)**: Optional (G+09)]***VL2: A Scalable and Flexible Data Center Network*, Greenberg et al., SIGCOMM 2009.
* **[**[**Architecture: Storage**](http://pages.cs.wisc.edu/~akella/CS744/S19/838-CloudPapers/hdfs.pdf)**(S+10)]***The Hadoop Distributed File System*, Schvachko et al, MSST, 2010.

**The Hadoop Distributed File System**

The paper describes the architecture of Hadoop Distributed File System (HDFS) which is a distributed file system for large clusters with thousands of servers that stores very large data sets.The two key components of the Hadoop Distributed File System are **NameNodes** and **DataNodes. NameNode -** Files and directories are represented by inodes which also contains other fields such as permissions, namespaces, etc. Files are split into blocks (default 128 MB) and replicated on the datanodes (typical replication factor is 3). The NameNode maintains namespace tree and mapping of file blocks to DataNodes. A single NameNode is maintained per cluster. **DataNode** - Block replicas are maintained as two files. First file for actual data and second file for metadata - checksum + generation stamp. During startup, handshake process with NameNode to verify namespace ID and software version (crucial given that in large clusters it is easy to overlook software upgrades for some machines). Namespace ID’s are used to maintain file system integrity. Block replicas are identified using block reports sent to NameNode. Heartbeat mechanism is used for communication between NameNode and DataNode (default three seconds, failure point 10 minutes).

It also introduces other components. HDFS Client, CheckPoint Node and Backup Node.

**HDFS Client:** Code library exporting HDFS interface to facilitate the communication between external applications and cluster.

For read/write operations: The client sends a request to Name Node which provides the location of the data nodes where the blocks constituting the file are present and then the client can directly talk to the data nodes to access content of the block.

For write request, the client requests Name Node to host 3 data nodes for writing each block of the file which thereby gets written in pipeline fashion.

**Check Point Node :** Responsible for keeping an up-to-date checkpoint which it does by taking the current checkpoint and applying the transactions in journal thereby creating new checkpoint and empty journal.

**BackUp Node :** In-memory image of the file system always synchronized with the current state of Name Node. It is sort of only read-only Name Node and does not contain information about block locations.

**File I/O Operations and Replica Management**

HDFS is a single writer, multiple reader model. There is a lease mechanism that handles file write permission and time. The nodes in the cluster are grouped into racks with a switch for each rack. There are core switches that connect the racks. The block placement policy of the HDFS ensures there is a fair tradeoff between performance, reliability and availability. For example, the first 2 replicas of file blocks are placed on nodes from two different racks and third third id placed in one of the 2 racks.

The NameNode has a policy to ensure there are enough replicas of the file block. When the replicas are over-replicated or under-replicated, the NameNode interferes and manages the situation. There is also a load balancer that works to ensure that there is no few nodes whose disk utilization is very high and others that are have very little data. Also, each DataNode runs a block scanner periodically to check if the block is not corrupt. If so, its promptly reported to the NameNode.

**Lecture 3 – MapReduce**

* **[**[**Execution: MR**](http://static.googleusercontent.com/media/research.google.com/en/archive/mapreduce-osdi04.pdf)**(G+04)]***MapReduce Simplified Data Processing on Large Clusters*, Dean and Ghemawat, OSDI, 2004.

Programming model

Input: a set of key/value pairs

Output: a set of key/value pairs

**Map:** user function, takes an input pair and produces a set of intermediate key/value pairs. The library groups together all intermediate values associated with the same intermediate key and passes them to the Reduce function.

**Reduce:** user function, accepts an intermediate key and a set of values for that key. It merges these values and to form a set of values.

Execution

Map splits the input files into M pieces. It then starts up many copies of the program on a cluster of machines.

One of the copies is the master, and the rest are workers. There are M map tasks and R reduce tasks.

A worker for a map task reads the contents of the corresponding input split. It parses key/value pairs out of the input data and passes each pair to the user-defined Map function. The intermediate key/value pairs are buffered in memory.

The buffered pairs are periodically written to local disk, partitioned into R regions by the partitioning function. The locations of the data are passed back to the master, who is responsible for forwarding these locations to the reduce workers.

When a reduce worker is notified by the master about the locations, it uses remote procedure calls to read the buffered data from the local disks of the map workers. It sorts intermediate data by the intermediate keys with same keys grouped together. External sort is used for large data.The reduce worker iterates over the sorted intermediate data and for each unique intermediate key, it passes the key and the corresponding set of intermediate values to the Reduce function. The output is appended to a final output file.

Upon completion, the master wakes up the user program.

**Master Data Structure**

For each completed map task, the master stores the locations and sizes of the R intermediate file regions produced by the map task. The information is pushed incrementally to workers that have in-progress reduce tasks.

**Fault Tolerance**

Worker Failure: The master pings every worker periodically. Without response a certain amount of time, the work is marked failed. Completed map tasks are re-executed because output on local disks becomes inaccessible. Completed reduce tasks are not re-executed because the output is stored in a global file system.

Master Failure: Make master write periodic checkpoints. If the master dies, a new copy can be started from the last checkpointed state. The failure is unlikely: therefore, the current implementation aborts and retries the operation.

Semantics: When map and reduce operators are deterministic, it relies on atomic commits of map and reduce task outputs. When map and/or reduce operators are non-deterministic, weaker but still reasonable semantics are provided.

Locality Network bandwidth is scarce. Most input data are read locally.

**Task Granularity**

M is roughly 16 MB to 64 MB and R is a small multiple of the number of worker machines expected to use.

**Backup**

Alleviate the problem of stragglers: Master schedules backup executions of the remaining in-progress tasks. The task is marked as completed whenever either the primary or the backup execution completes.

Lecture 7.

* **[[ResourceNeg: DRF](https://www.usenix.org/legacy/events/nsdi11/tech/full_papers/Ghodsi.pdf" \t "_blank) (G+11)]***Dominant Resource Fairness: Fair Allocation of Multiple Resource Types*, Ghodsi et al, NSDI, 2011.

DRF (Dominant Resource Fairness) is a generalization of max-min fairness for multiple resources. In a multi-resource environment, DRF seeks to maximize the minimum dominant share across all users, where the dominant share is the maximum share that user has been allocated of any resource.

The paper proposed several important properties

**Sharing incentive:** Each user should be better off sharing the cluster, then exclusively using her own partition of the cluster.

Strategy-proof ness: User should not be able to benefit by lying about their resource demands

Envy-freeness: A user should not prefer the allocation of another user.

Pareto efficiency: It should not be possible to increase the allocation of a user without decreasing the allocation of at least another user.

And some nice-to-have properties for a resource allocating algorithm:

Single resource fairness: For a single resource, the solution should reduce to max-min fairness.

Bottleneck fairness: If there is one resource that is percent-wise demanded most of by every user, then the solution should reduce to max-min fairness for that resource.

Population monotonicity: When a user leaves the system and relinquishes her resources, none of the allocations of the remaining users should decrease.

Resource monotonicity: If more resources are added to the system, none of the allocations of the existing users should decrease.

What DRF does is actually maximizing the allocations while equalizing dominant shares. In application, DRF will track the total resources allocated to each user as well as the user's dominant share, select the user with the lowest dominant share and run its tasks. The author also compares DRF with other strategies, such as Asset Fairness and Competitive Equilibrium from Equal Incomes:

Asset Fairness tries to equalize the aggregate resource value allocated to each user

Competitive Equilibrium from Equal Incomes tries to maximize Nash product

while Asset Fairness violates the sharing incentive property and Competitive Equilibrium from Equal Incomes is not strategy-proof. However, DRF satisfies all those requirements excluding resource monotonicity, which is not a big problem.

lecture 8 - batch analytics – spark-sql

* **[**[**SQL: SparkSQL**](http://dl.acm.org/citation.cfm?id=2742797&CFID=542382153&CFTOKEN=47526471)**(A+15)]***Spark SQL: Relational Data Processing in Spark*, Armburst et al, SIGMOD, 2015.
* **[**[**SQL: Hive**](http://dl.acm.org/citation.cfm?id=2588555.2595630)**(H+14)]***Major technical advancements in Apache Hive*, Huai et al, SIGMOD, 2014.

Summary for sparkSQL;

In the paper "Spark SQL: Relational Data Processing in Spark", Michael Armbrust and Co. discuss Spark SQL, a new module in Apache Spark that integrates relational processing with Spark's functional programming API. Current big data applications require a combination of processing techniques, data sources, and storage formats. To meet these needs, MapReduce was created. However, MapReduce still had many limitations. Namely, the low-level procedural programming interface burdened users with many manual optimizations in order to gain performance. In response to this, systems such as Pig, Hive, Dremel, and Shark rose to answer these concerns. However, Armbrust wants to take it a step further and provide support for writing declarative queries and advanced analytics such as machine learning and graph processing. Spark SQL enables users to get the best of both worlds - the benefits of relational processing and the benefits of complex analytics libraries in Spark. Specifically, there are two main contributions of SparkSQL: it offers a tighter integration between relational and procedural processing and includes a high extensible optimizer that make it easy to add composable rules, control code generation, and define extension points. If one could label SparkSQL, it would be a evolution of SQL-on-Spark and Spark itself.  
  
Spark SQL has four goals that it wants to achieve in order to avoid the original problems that its ancestor, Shark had:  
1) Support relational processing on Spark programs  
2) Provide high performance with standard DBMS techniques  
3) Easily support new data sources  
4) Enable an extension to advanced analytics algorithms (ML, Graph Processing)  
Keeping these goals in mind, the structure of Spark SQL (and the paper) is the following:  
1) Programming Interface: Spark SQL runs as a library on top of Shark. It exposes the SQL interface and allows users to intermix procedural and relational code. Furthermore, Spark SQL's main abstraction, DataFrame, is equivalent to a table in a relational database and can be manipulated in a similar way. DataFrames keep track of their schema and support various operations with optimized execution. Spark SQL also adapts the nested data model from Hive for DataFrames and supports all major SQL data types. Some additional important things are that Spark SQL can materialize "hot data" in memory using columnar storage and supports user defined functions (as they are a vital extension for database systems).  
2) Catalyst Optimizer: Typical work flow looks like the following: SQL Query & DataFrame -> Tree construction -> Retrieval of an optimized logical plan -> Physical plan -> Use a cost model to determine the best physical plan -> Obtain RDDs.   
3) Advanced Analytics Features: 3 features were added to support analytics:  
i) Spark SQL includes a schema inference algorithm for JSON and other semi-structured data  
ii) Spark SQL is incorporated into a new high-level API for Spark’s machine learning library  
iii) Spark SQL supports query federation, allowing a single program to efficiently query disparate sources  
4) Evaluation: As one might expect, Spark SQL has better performance than Shark and Impala in terms of all possible things one might do in a declarative SQL query - although UDFs are more or less the same (very slightly faster). Additionally, to justify the use of DataFrames, they were evaluated alongside Python APIs and Scala APIs that deal with distributed aggregation. DataFrames greatly outperformed them.  
  
Much like other papers, this paper had some drawbacks. The first drawback is the misplacement of the "Related Work" section. It was placed near the end of the paper rather than the beginning of the paper. Since they talk about Shark, Extensible Optimizers, and Advanced Analytics in both this section and the introduction, it would have been great to use this as a way to facilitate discussion earlier on. Another drawback refers to my disappointment of a lack of future work. Spark SQL seems to be at its early stages and does not fully support all current DBMS techniques (such as user defined types) - something that can be handled in the future. To circumvent this, they made it open source, but it seems like they were simply avoiding the issue by doing so.

Apache Hive has become de-facto standard for SQL-on-Hadoop. With ever increasing demands, of users and organizations to provide more analytical power, Hive has undergone major enhancements, since its creation at Facebook in 2008. This paper explores issues and enhancements in Hive, contrasting with Impala, Shark and HAWQ. Analyzing future direction of Hive project, this paper attempts to study the potential of Hive in becoming enterprise SQL data warehouse at Hadoop scale with low-cost.

Lecture 9. geo-distributed and spot market analytics.

1. Clarinet: <https://www.usenix.org/conference/osdi16/technical-sessions/presentation/viswanathan>

You can hear the presentation up there. Slides and paper are also available there. This makes a case for geo-distributed analytics.

2. QOOP: <https://www.usenix.org/conference/osdi18/presentation/mahajan>

Same as above. This makes a case for query re-planning in the face wild resource variations, e.g., spot markets.

Lecture 10. stream analytics - Storm+Heron.

* **[**[**Streaming: Storm**](https://cs.brown.edu/courses/csci2270/archives/2015/papers/ss-storm.pdf)**(T+14)]***Storm @Twitter*, Toshniwal et al, SIGMOD, 2014.
* **[**[**Streaming: Heron**](http://dl.acm.org/citation.cfm?id=2742788)**Optional (KB+15****)]***Twitter Heron: Stream Processing at Scale*, Kulkarni et al, SIGMOD, 2015.

**Storm @Twitter**

Storm is a real-time distributed stream data processing engine at Twitter that powers the real-time stream data management tasks. Storm is designed to be scalable, resilient, extensible, efficient, and easy to administer.

The basic architecture consists of streams of tuples flowing through topologies which are directed graph where the vertices represent computation and the edges represent the data flow between various computation components. There are 2 types of vertices - spouts and bolts. Spouts are tuple sources for the topology, while bolts process the incoming tuples and pass them to the next set of bolts downstream.

**Architecture:**

**Nimbus** – It is a master node in Storm and clients submit topologies to it. Nimbus is responsible for distributing and coordinating the execution of the topology. Nimbus is an Apache Thrift service and Storm topology. The user describes the topology as a Thrift object and sends that object to Nimbus, thus allowing any programming language to create a Storm topology. Nimbus uses a combination of the local disk and Zookeeper to store state about the topology.

Zookeeper - All coordination between Nimbus and the Supervisors is done using Zookeeper. Nimbus and the Supervisor daemons are fail-fast and stateless, and all their state is kept in **Zookeeper** or on the local disks. This design allows the workers to continue to work even if the Nimbus service fails. While Nimbus is down, the user cannot submit new topologies.

Supervisor – It runs on each node and manages workers by spawning workers, monitoring their health, respawning if needed. It receives tasks from Nimbus and has 3 events for operation – heartbeat, synchronize supervisor, and synchronize process.

**Workers and Executors** – A worker process runs several executors inside a JVM. These executors are basically threading, and each executor can run several tasks. Each worker process has two dedicated threads - a worker receive thread and a worker send thread. Each executor consists of two threads namely the user logic thread and the executor send thread.

Nimbus and Supervisor daemons are stateless. This would make backups easy to handle and the snapshots of the memory state are kept locally in the Zookeeper.

The evaluation could have been better. No comparison with other similar systems, along with the manual configuration of the number of instances for spouts and bolts by the programmer makes it harder to evaluate the system.

Comparatively weaker semantics guaranteed with at least once and at most once semantics. The bitwise XOR logic looks neat, but a more detailed explanation would have been helpful.

For the second paper, how is **Heron** better than **Storm**?

1. The provisioning of resources is abstracted from the duties of the cluster manager so that Heron can play nice with the rest of the shared infrastructure.
2. Each Heron Instance executes only a single task so is easy to debug.
3. The design makes transparent which component of the topology is failing or slowing down as the metrics collection is granular and can easily map an issue to a specific process in the system.
4. Heron allows a topology writer to specify exactly the resources for each component, avoiding over-provisioning.
5. Having a Topology Manager per topology enables them to be managed independently and failure of one topology does not affect the others.
6. The backpressure mechanism gives a consistent rate of delivering results and a way to reason about the system. It is also a key mechanism that supports migration of topologies from one set of containers to another.
7. It does not have any single point of failure.

In the evaluation in section 7 of the paper, Heron beats Storm in every single metric.

Lecture 12. spark streaming (and flink)

* **[**[**Streaming: SparkStreaming**](https://cs.stanford.edu/~matei/papers/2013/sosp_spark_streaming.pdf)**(ZD+13)]***Discretized Streams: Fault-Tolerant Streaming Computation at Scale*, Zaharia et al, SOSP, 2013. Also read this introduction to [Structured Streaming](https://databricks.com/blog/2016/07/28/structured-streaming-in-apache-spark.html).
* **[**[**Streaming: Flink**](http://sites.computer.org/debull/A15dec/p28.pdf)**: Optional (C+15)]***Apache Flink: Stream and Batch Processing in a Single Engine*, Carbone et al, Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 2015.

**1.Paper Review for sparksteming**: Discretized Streams: Fault-Tolerant Streaming Computation at Scale.

Spark Streaming is a batch process system based on Spark, trying to simulate streaming by divide real time events into micro-batches with time intervals. In this paper, the authors propose a large-scale, big data processing in real time using a parallel recovery system in a distributed environment. The proposed system addresses the features lacking in the existing approaches, most importantly sub-second fault and stragglers recovery by defining an in-memory data structure (called RDD) instead of a replicated on-disk state recovery mode. The experimental results using a Spark Engine-based machine show the system is able to sub-secondly process over 60 million records per second on a 100-node running scenario.

Advantage: the main advantage of the proposed system that as the authors claim, many of the other work do not support is the address the issues with both faults, and stragglers. Also, unlike centralized approaches, this work is proposing a parallel recovery mechanism in a distributed environment which enables huge scalability. Plus, the other major vantage of the proposed system is the sub-second recovery latency for both faults and stranglers.

Cons for this paper is firstly user still need to figure out batch interval and the memory consumption is huge.

**2.Paper Review for spark-streaming**

This paper presents **Apache Flink**, which is an open-source system for processing streaming and batch data. Flink is built on the philosophy that many classes of data processing applications, including real-time analytics, continuous data pipelines, historic data processing, and iterative algorithms can be expressed and executed as pipelined fault-tolerant dataflows. The paper present's Flink's architecture and expand on how a set of use cases can be unified under a single execution model.   
This paper first presents the architecture of Flink. A Flink cluster comprises three types of processes: **the client, the Job Manager, and at least one Task Manager**. The client takes the program code, transforms it to a dataflow graph, and submits that to the Job-Manager. The Job-Manager coordinates the distributed execution of the dataflow. It tracks the state and progress of each operator and stream, schedules new operators, and coordinates checkpoints and recovery. A Task Manager executes one or more operators that produce streams, and reports on their status to the Job Manager. The Task Managers maintain the buffer pools to buffer or materialize the streams, and the network connections to exchange the data streams between operators. The dataflow graph is a directed acyclic graph that consists of stateful operators and data streams that represent data produced by an operator and are available for consumption by operators. Streams distribute data between producing and consuming operators in various patterns.

Flink’s intermediate data streams are the core abstraction for data-exchange between operators. An intermediate data stream represents a logical handle to the data that is produced by an operator and can be consumed by one or more operators. Pipelined intermediate streams exchange data between concurrently running producers and consumers resulting in pipelined execution. When a data record is ready on the producer side, it is serialized and split into one or more buffers that can be forwarded to consumers. Apart from exchanging data, streams in Flink communicate different types of control events. Flink uses lots of special types of control events, including checkpoint barriers, watermarks, and iteration barriers. Flink offers reliable execution with strict exactly-once-processing consistency guarantees and deals with failures via checkpointing and partial re-execution. The checkpointing mechanism of Apache Flink builds on the notion of distributed consistent snapshots to achieve exactly-once-processing guarantees. Recovery from failures reverts all operator states to their respective states taken from the last successful snapshot and restarts the input streams starting from the latest barrier for which there is a snapshot. ABS provides several benefits: it guarantees exactly-once state updates without ever pausing the computation, it is completely decoupled from other forms of control messages, and it is completely decoupled from the mechanism used for reliable storage, allowing state to be backed up to file systems, databases, etc., depending on the larger environment in which Flink is used. Since Flink’s runtime supports pipelined data transfers, continuous stateful operators, and a fault-tolerance mechanism for consistent state updates, overlaying a stream processor on top of it is to implement a windowing system and a state interface. Batch computations are executed by the same runtime as streaming computations. Flink optimizes batch processing by optimizing execution using a query optimizer and by implementing blocking operators that gracefully spill to disk in the absence of memory.

Lecture 14.

**[**[**Streaming: rStreams**](https://www.usenix.org/system/files/conference/nsdi16/nsdi16-paper-lin-wei.pdf)**(L+16)]***StreamScope: Continuous Reliable Distributed Processing of Big Data Streams*, Lin et al, NSDI,

This paper introduces steamscope, which is thought as a reliable distributed stream computation engine that has been deployed in shared 20,000-server production clusters at Microsoft. STREAMS provides a continuous temporal stream model that allows users to express complex stream processing logic naturally and declaratively. STREAMS supports business-critical streaming applications that can process tens of billions (or tens of terabytes) of input events per day continuously with complex logic involving tens of temporal joins, aggregations, and sophisticated userdefined functions, while maintaining tens of terabytes inmemory computation states on thousands of machines. STREAMS introduce two abstractions, rVertex and rStream, to manage the complexity in distributed stream computation systems. The abstractions allow efficient and flexible distributed execution and failure recovery, make it easy to reason about correctness even with failures, and facilitate the development, debugging, and deployment of complex multi-stage streaming applications.

**rStream** reliably maintains a sequence of events with continuous and monotonically increasing

sequence numbers, supporting multiple writers and readers. (it guarantees Uniqueness, Validity and Reliability.)

**rVertex** supports operations for vertex (Load(s) starts an instance of the vertex at snapshot s. Execute() executes a step from the current snapshot. GetSnapshot() returns the current snapshot)

Also in this paper, it introduces how to implement rStream and rVertex. STREAMS takes a principled approach to distributed fault-tolerant cloud scale stream computation with new abstractions rVertex and rStream. Its implementation and deployment in production not only provide the insights that validate the design choices, but also offer valuable engineering experiences that are key to the success of such a cloud scale stream computation system.

Lecture 15

* **[[GraphProc:Pregel](https://kowshik.github.io/JPregel/pregel_paper.pdf" \t "_blank) (M+10)]***Pregel: A System for Large-Scale Graph Processing*, Malewicz et al, SIGMOD, 2010.

Before reading this paper, we know graphs are hard because its poor locality of memory access/ very little work peer vertex/ changing degree of parallelism and running over many machines make the problem worse.

In this paper, Pregel is proposed as a new computation model for graph processing. Pregel provides a scalable and fault tolerant platform with an API that can be used to solve large graph problems.

**The computation of the model:** Pregel computations consists of a sequence of iterations called supersteps. The input to a Pregel computation is a directed graph in which each vertex is uniquely identified by a vertex identifier. Each vertex is associated with a modifiable, user defined value. The directed edges are associated with their source vertices, and each edge consists of a modifiable, user defined value and a target vertex identifier. Vertices can be active or inactive. In initial superstep, every vertex is given the active state and are able to participate in the computation. Then a vertex can deactivate itself by voting to halt. The algorithm as a whole terminates when all vertices are simultaneously inactive and there are no messages in transit.

The communication between vertices in the graph is by using message passing system. During every iteration, the vertex reads all the messages that are in the previous iterations, performs user defined functions on the vertex. The result would be updating the value of the vertex or using the outgoing edges to send the results to the destination vertex.

**C++ API: Message Passing :** Vertices communicate with each other through message passing consisting of a value and the name of the destination vertex. A vertex receives all the messages sent to it during superstep S in superstep S+1 when its Compute() method is called. **Combiners :** In some cases, it is possible to combine messages to be sent to a vertex on another machine. This helps in reducing the overhead incurred during sending messages over the network.

**Aggregators :** Each vertex can provide a value to an aggregator in a superstep S, it combines these values using a reduction operator and the resulting value is made available to all the vertices in superstep S+1.

**Topology Mutation :** The Compute() function of each vertex is capable of modifying the graph topology by adding or removing a vertex or an edge. If there are conflicting requests, there are some special mechanisms to resolve it.

**Input and Output:** The Pregel API is flexible in the kind of input that it takes. It does not impose any specific input format. User can also define the format of output it wants.

Lecture 16.

* **[[GraphProc: PowerGraph](https://www.usenix.org/system/files/conference/osdi12/osdi12-final-167.pdf" \t "_blank): Optional (GL+12)]***PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs*, Gonzalez et al, OSDI, 2012.

Large scale graph structured computation is useful for cases from targeted advertising to natural language processing, thus several graph-parallel abstractions including Pregel and GraphLab has been developed. However, these systems ignored the fact that natural graphs commonly found in real-world have highly skewed power-law degree distributions (a small fraction of the vertices are connected to most part of the graph). This paper proposes PowerGraph which exploits the internal structure of graph programs to address the challenges of computing on natural graphs. By leveraging PowerGraph abstraction, it introduces a new approach to distributed graph placement and representation that exploits the structure of power graphs. It explicitly factors computation over edges instead of vertices. As an consequence it has substantially greater parallelism, reduces network communication and storage cost and provides highly effective approach to distributed graph placement.

A graph-parallel abstraction consists of sparse graph G = {V, E} and a vertex-program Q which is executed in parallel on each vertex v and can interact through shared state or messages with neighboring instances. To address the challenges of computation on power-law graphs, PowerGraph eliminates the degree dependence of the vertex program by directly exploiting the GAS decomposition to factor vertex program over edges. By lifting the Gather and Scatter phases into the abstraction, PowerGraph is able to retain the natural "think-like-a-vertex" philosophy while distributing the computation of a single vertex program over the entire cluster.

lecture 17 graphX

* **[[GraphProc: GraphX](https://amplab.cs.berkeley.edu/wp-content/uploads/2014/09/graphx.pdf" \t "_blank) (G+14)]***GraphX: Graph Processing in a Distributed Dataflow Framework*, Gonzalez et al, OSDI, 2014.

GraphX is created with the goal of creating a general-purpose distributed graph processing framework that approaches the performance of existing application-specific graph processing libraries. They achieve this by making a number of optimizations to the graph system.

Graphx is able to achieve performance parity with the specialized graph processing systems by recasting the graph specific optimizations on top of the Spark framework. GraphX represents graphs as a pair of vertex and edge collections built on the spark RDD abstraction.

**Vertex Collection:** Hash-partitioned using the vertex-ids. For supporting across vertex collection joins, vertices are stored in a local hash index within each partition. Bit bask is used to store the visibility of each index enabling soft deletions and reuse of index.

**Edge Collection:** Edges are horizontally partitioned by the user-defined partition function. This enables vertex-cut partitioning. The mapping between the vertices and edge collections are maintained in the routing table.

**Index reuse:** Since GraphX works on top of spark RDDs, all graph operators create logically new collections rather than destructively modifying existing ones. Thus, edge and vertex collections can share indices to reduce memory overhead and accelerate local graph operations.

Lecture 18.

* **[**[**ML: ParamServ**](http://www.cs.cmu.edu/~muli/file/parameter_server_osdi14.pdf)**(LA+14)]***Scaling Distributed Machine Learning with the Parameter Server*, Li et al, OSDI, 2014.

With the vast amount of training data available for machine learning models, single machine computation becomes infeasible and impractical. Parameter servers are a distributed framework to globally connect worker nodes by maintaining shared parameters accessible by the workers. This paper presents a new framework for the parameter server to represent parameters as dense or sparse matrices, and an asynchronous data communication management system that supports flexible consistency, scalability, and 1 second fault tolerance.

The  framework presents five key contributions:

1. Efficient communication - non blocking and optimized for machine learning tasks to reduce overhead
2. Flexible consistency models - relaxed consistency hides synchronization cost and latency, allowing the developer to manage the tradeoff
3. Elastic scalability - new nodes can be added without restart
4. Fault tolerance and durability - within 1s recovery without interrupting computation with well defined post-failure behavior
5. Ease of use - parameter representations work well with linear algebra data structures

The architecture is designed such that each parameter server can run multiple algorithms. Server nodes maintain a partition of the shared parameters, and communicate with each other to replicate and migrate for reliability and scaling. There is a single manager node to maintain consistent metadata of the server nodes (such as liveness and assigments)

Worker node groups run applications, each by storing a portion of the training data locally for computing statistics (e.g. gradients). Workers do not communicate with other workers, but only server nodes. Each worker group has a scheduler to assign tasks, monitor progress, and reschedule unfinished tasks.

Parameter servers store shared parameters as key,value vectors to facilitate linear algebra operations. By moving to vectors rather than a list of pairs, the framework can provide key linear algebra optimized functionality such as vector addition, multiplication, norm calculation, and others. This is possible by assuming the keys are ordered.

Data is sent using batch push and pull operations, and is optimized using key range-based push and pulls. These are issued by workers to servers as tasks, which are executed asynchronously. It will then continue with computation and once a callee's reply has been acknowledged, the task is marked finished.

Flexible consistency allows for three different models of implementation by task dependency. These are **Sequential**, **Eventual**, and **Bounded Delay**, depending on the needs of the user.

Fault tolerance for key-value pairs is embodied by chain replication. Additionally, each key-value pair is associated with a vector clock that records the time of each individual node on this key-value pair. This allows tracking aggregation status or rejecting duplicates. Messages may also be sent between nodes which consist of lists of key-value pairs in a key range and associated with a vector clock.

For replication, server nodes stores a replica of the k counterclockwise neighbor key ranges relative to the one it owns. To add a new node to a worker pool, server managers assign the new node a key range to serve as master. The node then fetches the range of data to maintain as master and k additional ranges to keep as slave. Then, the server manager broadcasts the node changes so the recipients may update their data based on key ranges.

Lecture 19.

* **[**[**ML: STRADS**](http://www.pdl.cmu.edu/PDL-FTP/Storage/strads-kim-eurosys16.pdf)**(K+16)]***STRADS: A Distributed Framework for Scheduled Model Parallel Machine Learning*, Kim et al, EuroSys, 2016.

This paper introduces a framework called STRADS which is for Scheduled Model Parallel Machine Learning. Model parallelism is motivated by two challenges that data-parallelism

does not usually address: (1) parameters may be dependent, thus naive concurrent updates can introduce errors that slow convergence or even cause algorithm failure; (2) model parameters converge at different rates, thus a small subset of parameters can bottleneck ML algorithm completion. It is also proposed scheduled model parallelism (SchMP), a programming approach that improves ML algorithm convergence speed by efficiently scheduling parameter updates,

taking into account parameter dependencies and uneven convergence. To support SchMP at scale, we develop a distributed framework STRADS which optimizes the throughput of SchMP programs, and benchmark four common ML applications written as SchMP programs: LDA topic modeling, matrix factorization, sparse least-squares (Lasso) regression and sparse logistic regression. By improving ML progress per iteration through SchMP programming whilst

improving iteration throughput through STRADS we show that SchMP programs running on STRADS outperform nonmodel-parallel ML implementations: for example, SchMP LDA and SchMP Lasso respectively achieve 10x and 5x faster convergence than recent, well-established baselines.

lecture 20 – SLAQ

* **[**[**ML: SLAQ**](https://www.cs.princeton.edu/~mfreed/docs/slaq-socc17.pdf)**(Z+17)]***SLAQ: Quality-Driven Scheduling for Distributed Machine Learning*, Zhang et al, SoCC, 2017.

When assign a distributed machine learning (ML) application resources at the application level, those resources are allotted for many hours. However, loss improvements usually occur during the first part of the application execution, so it is very likely that the application is underutilizing the resources for the rest of the time. (Some ML jobs are retraining of an already trained DNN, or compacting of a DNN by removing unused parameters, etc., so blindly giving more resources at the beginning and pulling some back later may not work well.) To avoid this, SLAQ allocates resources to ML applications at the task level, leveraging the iterative nature of ML training algorithms. Each iteration of the ML training algorithm submits tasks to the scheduler with running times around 10ms-100ms. This is how Spark based systems operate readily anyways.

SLAQ collects "quality" (measured by "loss" really) and resource usage information from jobs, and using these it generates quality-improvement predictions for future iterations and decides on future iteration task scheduling based on these predictions.

The paper equates "quality" with "loss", and justifies this by saying:

1) "quality" cannot be defined unless at the application level; so to keep it general let's use "loss"

2) for exploratory training jobs, reaching 90% accuracy is sufficient for quality, and SLAQ enables to get there in a shorter time frame.

On the other hand, there are drawbacks to that. While delta improvements on loss may correspond to improvements on the quality, the long-tail of the computation may still be critical for "quality", even when loss is decreasing very slowly. This is especially true for non-convex applications.

The paper normalizes quality/loss metrics as follows: For a certain job, SLAQ normalizes the change of loss values in the current iteration with respect to the largest change it has seen for that job so far.

SLAQ predicts an iteration's runtime simply by how long it would take the N tasks/CPUs can process through S the size of data processed in an iteration. (minibatch size.)

For scheduling based quality improvements, the paper considers couple metrics, like maximizing the total quality and maximizing the minimum quantity. The paper includes a good evaluation section.

In conclusion, SLAQ improves the overall quality of executing ML jobs faster, particularly under resource contention, by scheduling at a finer granularity task-level based on the observed loss improvements.

lecture 21 – tensorflow

* **[**[**ML: TensorFlow**](https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf)**(AB+16)]***TensorFlow: A System for Large-Scale Machine Learning*, Abadi et al, OSDI, 2016.

In this lecture, two classmates share with two PPT, one if for machine learning the other shows tensorflow paper:

Part 1: Machine Learning

1. Neuron

2. Motivating Depth of a Neural Net

(1) Suppose there is a k1 dimensional input to be mapped to a k2 dimensional output

(2) We have a budget of (m+n) intermediate (hidden) neurons.

(3) If we put all of them in a single layer, we have k1k2(m+n) paths from input to output.

(4) If we have 2 hidden layers of m and n neurons, k1k2mn paths from input to output.

More depth => More impressive!

3. Output Layers

(1) Binary classification : y = δ(WTh + b) where sigmoid δ(z) = 1/(1 + exp(-z))

(2) Output is a value in (0, 1)

4. Hidden Layers:

(1) o = r(WTx + b)

(2) Typical activation function r

a. Threshold t(z) = II[z >= 0]

b. Sigmoid δ(z) = 1/(1 + exp(-z))

c. Tanh tanh(z) = 2δ(2z) - 1

d. ReLU relu(z) = max{z, 0}

(3) Problem with sigmoid/tanh: saturation (twoo small gradient)

(4) Activation function ReLU (rectified linear unit)

a. ReLU(z) = max{z, 0}

(5) Gradient decent in weight space.

Given a training set D = {(x1, y1), (), .... (xm, ym)} we can specify an error measure that is a function of our weight vector w

(6) Backpropagation notation

we will use

a. o to indicate the output, net the linear combination of inputs

b. subscripts on o, net to indicate which unit they refer to

c. subscripts to indicate the unit a weight emanates from and goes to

d. each weight is changed by

(7) Backpropagation illustrated

a. calculate error for hidden layer

b. determine updates for weights to hidden units using hidden-unit errors.

(8) Convolutional neural networks

a. Strong empircial performance

<1> Especially for computer vision

<2> Also used for speech recognition, natural language processing.

b. Use convolution in place of general matrix/tensor multiplication in at least one of their layers.

for specific kind of weight matrix/tensor W.

(9) Discrete convolution: 2-D example.

(features extractions,)

(10) Discrete convolution: padding

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Motivating Convolution

a. Suppose there is a 1024\*1024\*3 image, to be scaled down to 1000 dimensions

b.Needs a matrix of size 1000\*1024\*1024\*3 ~ 3\*10^9

c. Too many parameters!

d. Can significantly reduce "parameter sharing" - use a small number of parameters to aggregate neighborhood-level information.

e. Other benefits

(1) Sparse connectivity

(2) Parameter sharing

(3) Tanslation equivariance

(4) Arbitrary input sizes

How neural networks tranfers to convolutional networks.

Translation equivariance.

mapping, filter, GPU, parameters,

More conventional neural network : https://github.com/vdumoulin/conv\_arithmetic

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ConvNets have also been used for

(1)Image Segmentation (2) Object Recognition (3) Speech (4) Text

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Recurrent neural networks;

New Cell State;

Output;

Long short-term memory;

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

lecture 22 - gandiva - multi-model scheduling

running ML atop GPUs, specifically about multiplexing many GPUs across jobs. It is a very interesting paper.

Last summer paper.

Multr-tenant ML (Gandiva)

GPU properties

ML jobs fine-grained properties

constraints, objective.

------------------------

GPU armple parallelism.

batch size of images.

--------------------------

multri-GPU machines.

1.PCIe 2.Nvlink (irregular)+ PCIe ≈ 25 GB

GPU to GPU links.

> (faster) network.

some are PCIE machine ; some are cluster machines.

Job Group 1,

JG2,

---------------------------------

Linited GPU 1.Efficiency | 2.Virtualize

------------------------------------

(1) Network - communication bound

(2) Memory is a coslimit.

(3) Periordic | iterate ---> Predictable

evaluate should decusion.

(4) Recursion Use.

suspending rame | nytubum

------------------------------------

Communication? --> scheduling work --> reactive. (as many as free GPU as possible)

job --- How many cansdidated GPUs.

1.CPU socket 2. machine 3.Only free GPU.

summary : introspectively.

this is all the machinism for

GPU applications for jobs, using migration.

------------------------

compare with tensorflow.

-----------------------------

job group is monitored.

mirgration. (迁移)

Monitor the CPU/Memory/Network usage

lecture 23 - serverless computing

* **[**[**Serverless: PyWren**](http://shivaram.org/publications/pywren-socc17.pdf)**(J+17)]***Occupy the Cloud: Distributed Computing for the 99%*, Jonas et al, SoCC, 2017

Motivations : The paper reports that an informal survey at UC Berkeley found that the majority of machine learning graduate students have never written a cluster computing job due to complexity of setting up cloud platforms.

Some cloud computing virtualized a lot of things, like VMs. However, there are still too many choices to make and things to configure before you can get your code to deploy and run at the cloud. (eg, we still do not have a “cloud button”) where we can push our single machine code deployed and running on the cloud in a few seconds.

We are getting there (the goal showed above). AWS Lambda and Google Cloud Functions aim to solve this problem.

This paper pushes the envelope on the serverless model further in order to implement distributed data processing and analytics applications. The paper is a vision paper, so it is low on details at some parts, however a prototype system, PyWren, developed in Python over AWS Lambda, is made available as opensource.

Summarize the code injection approach, in order to build a data processing system, the paper dynamically injects code into these stateless AWS Lambda functions to circumvent its limits and extend its capabilities. The model has one simple primitive: users submit functions that are executed in a remote container; the functions are stateless; the state as well as input, and output is relegated to the shared remote storage. (This fits well with the rising trend of the disaggregated storage architecture.) Surprisingly, the paper finds that the performance degradation from using such an approach is negligible for many workloads. Next we will introduce how PyWren can implement increasing more sophisticated data processing applications ranging from Map, to Map and monolithic reduce, and MapReduce, and finally a hit of parameter server implementation.

PyWren serializes the user submitted Python function using cloudpickle. PyWren submits the serialized function along with each serialized datum by placing them into globally unique keys in S3, and then invokes a common Lambda function. On the server side, PyWren invokes the relevant function on the relevant datum, both extracted from S3. The result of the function invocation is serialized and placed back into S3 at a pre-specified key, and job completion is signaled by the existence of this key. In this way, PyWren is able to reuse one registered Lambda function to execute different user Python functions and mitigate the high latency for function registration, while executing functions that exceed Lambda’s code size limit.

Next, the paper shows how to implement Map, MapReduce, and the Parameter- Server.

**Map implementation.** Many scientific and analytic workloads are embarrassingly parallel. The map primitive provided by PyWren makes addressing these use cases easy. Calling the map launches as many stateless functions as there are elements in the list that one is mapping over.

**Map + Monolithic reduce.** An easy way to implement MapReduce is to do the Reduce in just one machine. For this one machine to perform reduce, they use a dedicated single r4.16xlarge instance. This machine offers a very large amount of CPU and RAM for $14 an hour.

**MapReduce via BSP.** To perform Reduce over many workers, we can use the bulk synchronous processing (BSP) model. To implement the BSP model and data shuffling across the stages PyWren leverages the high-bandwidth remote storage AWS S3 provides. To showcase this approach, they implemented a word count program in PyWren and found that on 83M items, it is only 17% slower than PySpark running on dedicated servers.

lecture 24 & 25

* **[**[**ML: Clipper**](https://www.usenix.org/system/files/conference/nsdi17/nsdi17-crankshaw.pdf)**(C+17)]***Clipper: A Low-Latency Online Prediction Serving System*, Crankshaw et al, NSDI, 2017.
* **[[Approx: BlinkDB](https://www.cs.berkeley.edu/~sameerag/blinkdb_eurosys13.pdf" \t "_blank) (AM+13)]***BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data*, Agarwal et al, Eurosys, 2013.