Evolving the computing technology in extremely large eCommerce data platform:
JD.com case study

Dennis Weng & Wensheng Wang
Big Data Platform, JD retail

Platform Overview

- Large scale platform based on highly customized Hadoop stack

XX K servers

Million+ jobs/queries

Towards EB storage

Thousands streaming jobs

  - Batch workloads distribution

MapReduce	Spark	Presto Queries
50%+	~2%	40%+

- Supports
  - Large scale ETLs for data warehousing
  - Online data analytics for business intelligence
  - Machine learning jobs for supply chain optimization, recommendations, advertising, etc.
  - Real-time streaming computation for risk analysis, account security, etc.

Vision

- Decision to evolve to Spark:
  - Getting more adoptions: esp. machine learning related tasks
  - Lots of optimization opportunities with in-memory computation
  - Needs for more performant and versatile engines
  - Workloads co-location asking for more cloud native engine
  - More active spark community

Spark: 40min
Hive : 2.5h

Considerations

- Minimize the interruption to developers
  - Backward compatible with relatively old Hive version
  - No major performance regression
  - No stability regression for different scales of workloads
- Show benefit on resource cost saving
  - Optimization target is not merely job completion time
  - Rather: given the job completion SLA, use the most economical configuration
  - No adverse impact to co-located Hive jobs (Spark is memory hungry)

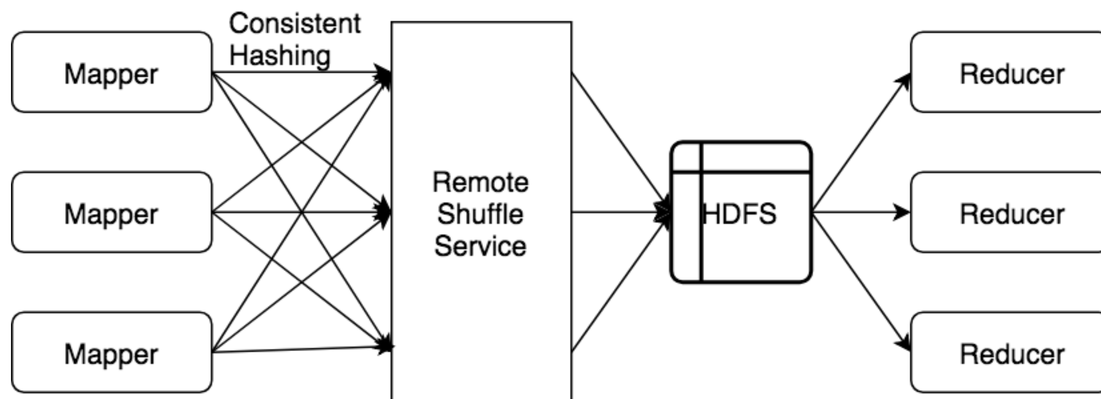
Compatibility

- Biggest challenge: Hive compatibility
  - Existing Hive was modified to follow the behavior of *older* version *a few years ago*
  - Examples: *grouping_id*, default *Decimal* type precision

- Open issue: *Hive UDFs*
  - Some are not thread safe: conflict with Spark task execution model
  - Lack of determinism annotation: potentially cause wrong query plan optimization
  - Need to act *thoroughly* and *conservatively*

Stability

- Shuffle fetch failures
  - Can result in stage re-computation
  - Action: *Store shuffle data to HDFS*



- Out of Memory
  - More frequent in Spark: difficult to eliminate all tail cases
  - Integrate JVM heap dump analysis
  - Tune Spark and JVM jointly*: e.g. tune spark page size accordingly with G1HeapRegionSize to reduce heap fragmentation*

Performance

- Optimization strategy
  - Keep executor memory to CPU ratio reasonable, to avoid memory starvation (especially for MR workers)
  - Prefer smaller executor size to allow dynamic allocation working more effectively.
  - Memory is the bottleneck: minimize (*allocated_memory* * *job_time*) subject to (*job_time* < *defined_SLA*)
  - Average result: running time decreased by > 40%, resource usage decreased by > 30%

Lessons

- Large scale production job migration is a significant effort
  - Good justification and planning
  - Careful execution: detail matters a lot
- Data consistency is critical
  - Even the buggy behavior in the old engine needs to be followed
  - Fast and trustable data validation mechanisms are desired
  - Maintaining clear data APIs and expectations helps long term success
- Process and tools are important
  - Canary/release/rollback process and tools
  - Monitoring and fast feedback
  - Distributed profiling and debugging support
  - Automation is the only way to scale to tens of thousands jobs

Next steps

- Complete the migration effort
- Employ cost based optimization with historical statistics
- Deep JVM optimization for the Spark engine
- Push for the co-location with online workload at larger scale
- Explore more effective and richer columnar data format
- Look into the application of persistent memory to address memory tension

Backups

Co-location

- Goal:
  - Be able to co-locate batch and real-time workloads with Kubernetes
- Action:
  - Early adopter and production deployment of Spark on Kubernetes
- Progress:
  - Several Kubernetes clusters (total 4000+ nodes), running batch/streaming/ML jobs
  - Save 25%+ resources