Shark: SQL & Rich Analytics at Scale
Deborah Hanus & Victor Lei
What problem does Shark solve?
What is the problem? Why is it hard?

Increasing data volumes
What is the problem? Why is it hard?

- Increasing data volumes
- Increased incidence of stragglers & faults
What is the problem? Why is it hard?

- Increasing data volumes
- Increased incidence of stragglers & faults
- Data analysis more difficult
What is the problem? Why is it hard?

- Increasing data volumes
- Increased incidence of stragglers & faults
- Data analysis more difficult
- Users want to query at interactive speeds
Existing solutions

Queries to MapReduce (eg. Hive, Cheetah)

Highly scalable

Fine-grained fault tolerance

High latency
## Existing solutions

<table>
<thead>
<tr>
<th>Queries to MapReduce (eg. Hive, Cheetah)</th>
<th>Shared-nothing parallel DBs (eg. Impala, PowerDrill)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Highly scalable</td>
<td>Less scalable</td>
</tr>
<tr>
<td>Fine-grained fault tolerance</td>
<td>No fine-grained fault tolerance</td>
</tr>
<tr>
<td>High latency</td>
<td>Low latency</td>
</tr>
</tbody>
</table>
## Existing solutions

<table>
<thead>
<tr>
<th>Queries to MapReduce (eg. Hive, Cheetah)</th>
<th>Shared-nothing parallel DBs (eg. Impala, PowerDrill)</th>
<th>Hybrid approaches (eg. HadoopDB, Osprey)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Highly scalable</td>
<td>Less scalable</td>
<td>Hybird approaches (eg. HadoopDB, Osprey)</td>
</tr>
<tr>
<td>Fine-grained fault tolerance</td>
<td>No fine-grained fault tolerance</td>
<td></td>
</tr>
<tr>
<td>High latency</td>
<td>Low latency</td>
<td></td>
</tr>
</tbody>
</table>
What is the core intuition of Shark’s solution?
What is the core intuition?

RDDs are the best. Modern Databases are OK too.
What is the core intuition?

RDDs are the best.  Modern Databases are OK too.

Shark: Scalable, Fault Tolerant, Interactive query speeds
Solution: Resilient Distributed Datasets (RDDs)

Read-only partitioned collection of records stored in-memory.

Representation:

- Set of partitions – atomic pieces of the data set
- Set of dependencies on parent RDDs
- Function to compute dataset based on its parents
- Metadata about partitioning scheme and data placement

Created by *transformations* of (a) data in stable storage, or (b) other RDDs

Coarse-grained writes make it fault tolerant.
How do RDDs compare to standard distributed shared memory systems?
## RDD’s vs. Distributed Shared Memory Systems

<table>
<thead>
<tr>
<th>Aspect</th>
<th>RDDs</th>
<th>Distr. Shared Mem.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reads</td>
<td>Coarse- or fine-grained</td>
<td>Fine-grained</td>
</tr>
<tr>
<td>Writes</td>
<td>Coarse-grained</td>
<td>Fine-grained</td>
</tr>
<tr>
<td>Consistency</td>
<td>Trivial (immutable)</td>
<td>Up to app / runtime</td>
</tr>
<tr>
<td>Fault recovery</td>
<td>Fine-grained and low-overhead using lineage</td>
<td>Requires checkpoints and program rollback</td>
</tr>
<tr>
<td>Straggler mitigation</td>
<td>Possible using backup tasks</td>
<td>Difficult</td>
</tr>
<tr>
<td>Work placement</td>
<td>Automatic based on data locality</td>
<td>Up to app (runtimes aim for transparency)</td>
</tr>
<tr>
<td>Behavior if not enough RAM</td>
<td>Similar to existing data flow systems</td>
<td>Poor performance (swapping?)</td>
</tr>
</tbody>
</table>
Solution: Shark Architecture

- Master Node:
  - HDFS NameNode
  - Resource Manager Scheduler
  - Master Process
  - Metastore (System Catalog)

- Slave Node:
  - Spark Runtime
    - Execution Engine
    - Memstore
    - Resource Manager Daemon
  - HDFS DataNode

  - Spark Runtime:
    - Execution Engine
    - Memstore
  - Resource Manager Daemon
  - HDFS DataNode
Solution: Executing SQL on RDDs

declarative query

abstract syntax tree

logical plan

physical plan: MapReduce jobs

physical plan: RDDs

select * from X
where Y . . .
Solution: Engine Extensions

1. Partial DAG Execution

Gathers statistics at global & partition granularities

Allows DAG to be altered based on stats

Worker sends stats to master, which optimizes

Handles skew and degree of parallelism

Allows join optimization
Solution: Partial DAG execution & join optimization
Solution: Engine Extensions

2. Columnar Memory Store

Placing objects in memory increases speed.

All columns are JVM arrays.

Each column creates only one JVM object.
Solution: Engine Extensions

3. Distributed Data Loading

Tracks metadata for each task, determining if it should be compressed
Solution: Engine Extensions

4. Data Co-partitioning

If you know the data schema, you can avoid shuffles by co-partitioning.
Solution: Engine Extensions

5. Partition Statistics & Map Pruning

Shark can avoid blocks that only contain data outside of the query range
How does this compare to modern database systems?
How does this compare to modern databases?

- Cache conscious
- Adaptive query patterns (e.g. H2O)
- Parallelism for minimal lock contention
- Shared nothing architecture (e.g. SharedDB)
Experiments

1. Pavlo et al. experiments (2.1 TB dataset)
2. TPC-H Dataset (100GB and 1TB)
3. Sampled real workload from Shark user (1.7TB)
4. Synthetic Machine Learning Dataset (100GB)
Experiments

EC2 $\times$ 100

8 cores  68 GB  1.7 TB
Results

Shark can perform **100x** faster than Hive and Hadoop

Even faster* than MPP databases in some experiments
How does this compare to other machine learning systems?
Advantages of Shark for Machine Learning

Keeping data in memory (RDDs)

Machine Learning as a first-class citizen: incorporating SQL with Machine Learning
Machine Learning

1 B rows

10 columns

100 GB data

Figure 10: Logistic regression, per-iteration runtime (seconds)

Figure 11: K-means clustering, per-iteration runtime (seconds)
Aggregation performance

![Diagram showing aggregation performance on TPC-H 100GB and TPC-H 1TB datasets.]

**Figure 6**: Aggregation queries on `lineitem` table. X-axis indicates the number of groups for each aggregation query.

```
SELECT [GROUP_BY_COLUMN], COUNT(*) FROM lineitem GROUP BY [GROUP_BY_COLUMN]
```
Join strategy optimization

A combination of static query analysis and partial DAG execution led to a 3x performance improvement over a naïve, statically chosen plan.
Real Hive Warehouse Queries

The map pruning technique, on average, reduced the amount of data scanned by a factor of 30.
What else could be done with this paper?
Next steps

Add the two unimplemented areas (bytecode compilation and specialized data structures) and benchmark differences in performance

More extensive real-world machine learning benchmarks (including machine learning tools)

Looking at other machine learning algorithms that can be incorporated into Shark