# Big Data Paper Summary

## CMPT 308N Database Management

- **Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing** Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica University of California, Berkeley
- **Choosing A Cloud DBMS: Architectures and Tradeoffs** Junjay Tan1, Thanaa Ghanem2,↼ , Matthew Perron3, Xiangyao Yu3, Michael Stonebraker3,6, David DeWitt3, Marco Serafini4, Ashraf Aboulnaga5, Tim Kraska3 1Brown University; 2Metropolitan State University (Minnesota), CSC; 3MIT CSAIL; 4University of Massachusetts Amherst, CICS; 5Qatar Computing Research Institute, HBKU; 6Tamr, Inc. junjay@brown.edu, thanaa.ghanem@metrostate.edu, {mperron,yxy,stonebraker}@csail.mit.edu, david.dewitt@outlook.com, marco@cs.umass.edu, aaboulnaga@hbku.edu.qa, kraska@mit.edu
- **Michael Stonebraker on his 10-Year Most Influential Paper Award** at ICDE 2015. http://kdb.snu.ac.kr/data/stonebraker_talk.mp4

By: Brianna Lee

# Main Idea of *Resilient Distributed Dataset*

- Resilient distributed datasets (RDDs) is an efficient, general-purpose and fault-tolerant abstraction for sharing data in cluster applications
  - RDDs can express a wide range of parallel applications, including many specialized programming models that have been proposed for iterative computation
- Generalized cluster computing frameworks like MapReduce and Dryad lack in two areas:
  - **Iterative Algorithms** where immediate results are used across multiple computations
  - **Interactive data analysis** where users run queries data
- Specialized frameworks like Pregel can solve these issues but will lead to loss of generality
  - RDDs can solve these by providing a general purpose, fault-tolerant, distributed memory abstraction

# How Ideas are Implemented from *Resilient Distributed Dataset*

- RDD Abstraction
  - Users can control two other aspects of RDDs: *persistence* and *partitioning*
    - Users can indicate which RDDs they will reuse and choose a storage strategy for them
- Spark Programming Interface
  - Programmers start by defining one or more RDDs through transformations on data in stable storage
    - Can call a *persist* method to indicate which RDDs they want to reuse in future operations
    - Spark keeps persistent RDDs in memory by default, but it can spill them to disk if there is not enough RAM
- Applications Not Suitable for RDDs
  - RDDs are best suited for batch applications that apply the same operation to all elements of a dataset
- Representing RDDS
  - A system implementing RDDS should provide as rich a set of transformation operators as possible and let users compose them in arbitrary ways
  - RDD implementations:
    - HDFS files: the input RDDS in our samples have been files in HDFS and *partitions* returns one partition for each block of the file
    - *map*: calling *map* on any RDD returns a MappedRDD object and this object has the same partitions and preferred locations as its parents but applies the function passed to *map* to the parent's records in its *iterator* method
    - *union*: calling *union* on two RDDs returns an RDD whose partitions are the union of those of the parent and each child partition is computed through a narrow dependency on the corresponding parent
    - *sample*: sampling is similar to mapping, except that the RDD stores a random number generator seed for each partition to deterministically sample parent records
    - *join*: joining two RDDs may lead to either two narrow dependencies, two wide dependencies, or a mix, and in either case, the output RDD has a partitioner
- Implementation
  - Spark provides three options for storage of persistent RDDs: in-memory storage as deserialized Java objects, in-memory storage as serialized data, and on-disk storage

# Analysis on Ideas and Implementations On *Resilient Distributed Dataset*

RDDs provide an interface where functions are applied to all data items, which can then provide a fault-tolerance. However, this makes it unsuitable for certain applications that need asynchronous fine-grained updates to a shared state because RDDs are based on fine-grained reads and coarse-grained updates. The RDDs can be used to express multiple different models and their limitations barely make an impact on parallel applications. RDDs highlight why this framework works best compared to others that cannot offer the same level of generality and lacked sharing data abstractions.

# Main Idea on *Cloud DBMS*

- Storage
  - Focus on single-user workloads to reproduce the common use case of ad-hoc cloud analytics
  - Queries were initiated  by a separate client node
    - Was done to allow parity with Redshift and Athena, which does not allow client code to run on the database nodes
  - Result sets were sent to the client node from the database management system and then on to the client
- Initialization Time
  - Initialization time measures how easy it is to launch and use a particular DBMS system
  - Is advantageous in the cloud to shut down compute resource when they are not being used, but there is then a query latency cost
- Cost
  - Cost is the only way to have a useful metric by which to compare Athena against other systems since Athena's infrastructure is a black box making performance comparison to other systems impossible
  - Storage cost is less flexible than compute cost because it does not allow recurring to contractual means for cost reduction, such as reverse and spot pricing
- Scalability
  - The main theme is that horizontal scaling is generally advantageous, while vertical scaling is generally disadvantageous

# How Ideas are Implemented from *Cloud DBMS*

- Storage
  - The main file systems on AWS (Amazon Web Server) are two block store options like Elastic Block Store (EBS) and Instance Store (InS) and then there is the Simple Storage Service (S3)
    - EBS is a remote network storage that is co=located in specified regions, while InS is physically attached to the compute node
- Costs
  - Total system costs are comprised of node compute cost, storage costs, and data scan costs.
    - Compute Costs
      - Are not provided by AWS at the granularity of individual queries and are calculated costs from the query runtimes and other associated node runtime processes
    - Storage Costs
      - Assume that users destroy this volume when not in use and re-initialize it upon system restart to save on costs
    - Data Scan Costs
      - Can be classified into two categories and pertain only to S3 storage:
        - The first category consists of explicit data scan costs for Spectrum and Athena due to the systems' unique pricing models
        - The second category applies to all other systems that retrieve data from S3
- Scaling
  - Most systems exhibit performance benefits from horizontal scaling, with Spectrum being the exception
  - Vertical scaling tests suggest that larger nodes are generally disadvantageous once moderate to large nodes are already used

# Analysis of Ideas and Implementation On *Cloud DBMS*

Through the experiments conducted, the results mainly focused on the following: query restrictions, system initialization time, query performance, cost, data compatibility with other systems, and scalability. With the experiments, it found that Athena's cost-per-query was on par with other systems which gave it a very good competitive cost, along with its great query performance. Although InS and EBS may run faster than S3 on the same system, the difference depends on how well the system utilizes storage and the workload of S3 beats its performance disadvantage. Essentially with the results, the best way to store data is with an S3. Also, using the portable data format gives one future flexibility to process it with multiple different systems. The use of Athena for the workloads that it can support is best, especially if it can doing less frequent queries.

# Comparison of the Ideas and Implementation of Both Papers

| Cluster Computing | Cloud DBMS |
| --- | --- |
| ● Compares multiple DBMS that are used on the cloud<br>● Between the DBMS, it tests multiple factors like:<br>  ○ Storage<br>  ○ Costs<br>  ○ Query restrictions<br>  ○ Scalability<br>● Looks for the best DBMS to use on the cloud | ● Compares multiple DBMS that are used on the cloud<br>● Between the DBMS, it tests multiple factors like:<br>  ○ Storage<br>  ○ Costs<br>  ○ Query restrictions<br>  ○ Scalability<br>● Looks for the best DBMS to use on the cloud |

# Main Ideas of the *Stonebraker Talk*

- Relational Database Management Systems (RDBMS) were to be as broad and scope as possible and believed to be the answer for everything and to be “one size that fits all”
  - It is not and is now known as “one size fits none”
    - There was no chance that streaming applications could be wedged into a traditional row-store RDBMS implementation
- All major vendors will move onto using column stores as they are two magnitudes faster than row stores
- OLTP engines seem to have a greater streaming market share
  - Programmer preference for function call architecture versus messaging architecture
- There is a huge diversity of engines across all markets
  - Traditional row stores do not work for most markets, hence that “one size fits none”

# Advantages and Disadvantages of *Resilient Distributed Dataset* in the Context to *Cloud DBMS* and the *Stonebraker Talk*

| Advantages | Disadvantages |
| --- | --- |
| • Data flow models like MapReduce can share data through stable storage<br>• RDDs store data in memory rather than the disk, which increases the performance | • While data flow models can share data, it has to suffer the costs of data replication, I/O, and serialization<br>• Cloud DBMS has unlimited storage compared to RDDs which stores data in the memory rather than disk where storage is limited |