Introduction to Cloud System Software

Introduction

1. Fundamental technologies underlying cloud computing including the programming frameworks and communication mechanisms
2. Concept of virtualization and associated mechanisms for virtualization in the cloud
3. Cloud storage systems, including key-value stores
4. Resource management: Automated provisioning, load balancing, and scheduling
5. Performance scalability and benchmarking

Setting the Stage

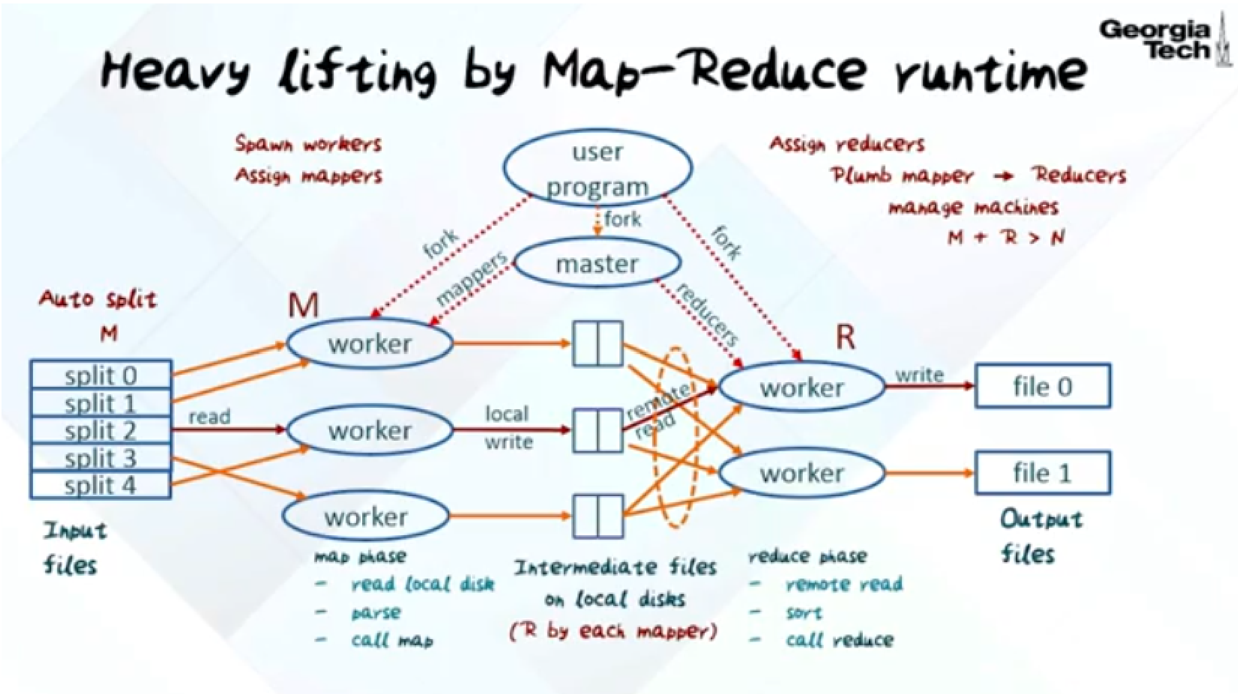
1. How to write large-scale parallel/distributed apps with scalable performance from multicores to clusters with thousands of machines?
    - Make the programming model simple
        - Liberate the developer from fine-grain concurrency control of the application components (e.g., threads, locks, etc.)
        - Dataflow graph model of the application with application components (e.g., subroutines) at the vertices, and edges denoting communication among the components
    - Exploit data parallelism
        - Require the programmer to be explicit about the data dependencies in the compuation
    - Let the system worry about distribution and scheduling of the computation respecting the data dependencies
        - Use the developer provided application component as the unit of scheduling and distribution
2. How to handle failures transparent to the app?
    - In data centers, it is not an "if", it is a "when" question
3. Roadmap
    - Map-reduce
    - Dryad
    - Spark
    - Pig Latin
    - Hive
    - Apache Tez

Map Reduce

1. Input + output to each of map + reduce
    - <key, value> pairs
    - Example: Emit number of occurrence of names in documents
        - Key: filename, Value: contents
        - Map: look for unique names
        - Reduce: aggregate values
        - Key: unique name, Value: number
2. Why Map-Reduce?
    - Several processing steps in giant-scale services expressible
        - Ranks for pages
    - Domain expert writes
        - map
        - reduce
    - Runtime does the rest
        - Instantiating number of mappers, reducers
        - Data movement
3. Map-Reduce Summary
    - Developer resonsibility

  - Input data set
  - Map and reduce functions
- System runtime responsibility
  - Shard the input data and distribute to mappers
  - Use distributed file system for communication between mappers and reducers

Heavy Lifting by Map Reduce Runtime



Map-Reduce Runtime

Issues Handled by the Runtime

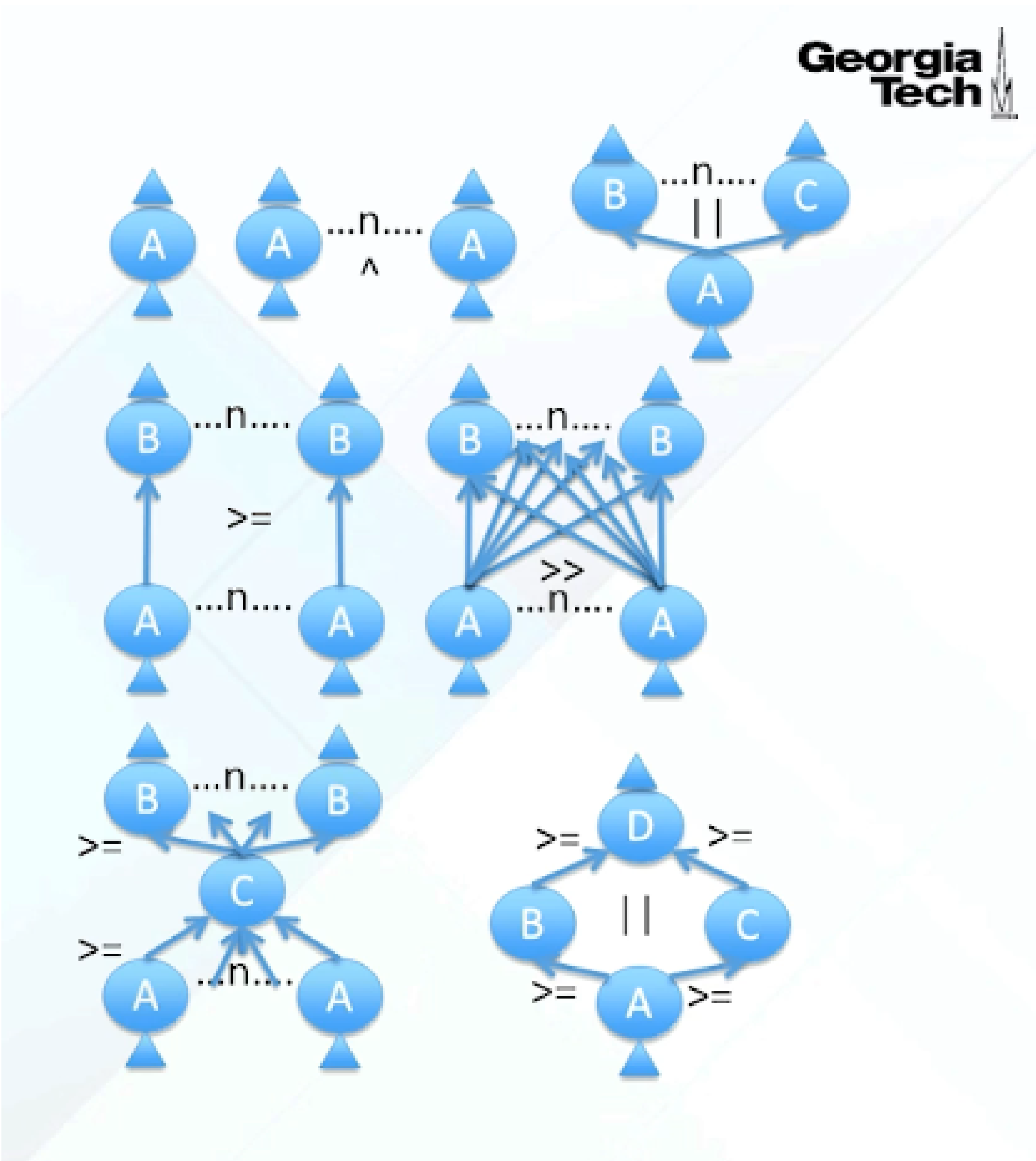
1. Master data structures
   - Location of files created by completed mappers
   - Scoreboard of mapper/reducer assignment
   - Fault tolerance
     - Start new instances if no timely response
     - Completion message from redundant stragglers
   - Locality management
   - Task granularity
   - Backup tasks

Dryad

1. Dryad design principles
   - Map-Reduce aims for simplicity for a large class of applications at the expense of generality and performance
     - e.g., Files for communicaton among application components, two-level graph (map-reduce with single input/output channel)
   - Dryad: General acylic graph representing the application

    - Vertices are application components
        * Arbitrary set of inputs and outputs
    - Edges are application-specified communication channels (shared memory, TCP sockers, files)

2. Dryad primitives
    - Developed at Microsoft
    - App developer writes subroutines
    - Uses graph composition primitives via C++ library to build the application
        - Cloning, merging, composition, fork-join
    - Encapsulate
        - Create a new vertex out of a subgraph
    - Specify transport for edges
        - Shared memory, files, TCP



Dryad

3. Dryad system
   - Application developer creates the graph describing how subroutines communicate
   - Job manager consults the name server to find out which nodes are available and launch portions of the application graph across available nodes
   - Finer granularity control compared to map-reduce

Spark

1. Data center programming challenges
   - Need fault tolerance
     - Map-reduce approach: Use stable storage for intermediate results
       - Make computations idempotent

    - Cons of this approach
        * Disk I/O is expensive
        * Inhibits efficient re-use of intermediate results in different computations
2. Spark design principles
    - Need performance and fault tolerance
        - Keep intermediate results in memory
        - Provide efficient fault tolerance for in-memory intermediate results
3. Spark secret sauce
    - Resilient distributed data (RDD)
        - In-memory immutable intermediate results
        - Fault tolerance by logging the lineage of RDD
            * i.e., the set of transformations that produces the RDD
            * RDD2 <- T2(T1(RDD1))
        - Regenerate the RDD using the lineage upon failure
            * Only the missing portion of the RDD needs regeneration
4. Spark generality
    - Unifies many current programming models
        - Data flow models: Map-reduce, Dryad, SQL
        - Specialized models
            * Batched stream processing, iterative Map-Reduce, iterative graph applications

Pig Latin, Hive, Apache Tez

1. Pig Latin (Yahoo)
    - In between the declarative style of SQL and procedural style of Map-Reduce
    - Break the rigidity of Map-Reduce
        - User-defined functions (UDF) as first class citizens
            * Grouping, joining, filtering, etc.
        - Nested data model
            * Atom, tuples, bags
            * e.g. {(‘bala’, ‘falcons’), (‘drew’, (‘braves’, ‘falcons’))}
2. Hive (Facebook)
    - System for querying and managing structured data built on top of Hadoop
    - Key features:
        - Queries expressed in SQL-like declarative language
        - Allows embedding custom map-reduce scripts
        - Compiles into map-reduce jobs
        - Uses HDF5 for storage
3. Apache Tez (Fast)
    - Similar in spirit to Dryad
        - Express data processing app as a dataflow graph
    - Built on top of Hadoop resource management framework called YARN
    - Used by Pig and Hive as the execution engine

Conclusion

1. Covered Map-Reduce, Dryad, Spark, Pig Latin, Hive, Tez
    - Some common functionality across frameworks, but different programming models are better suited for certain applications