ILLINOIS INSTITUTE OF TECHNOLOGY

CS520
Data Integration, Warehousing, and Provenance

7. Big Data Systems and Integration

IIT DBGroup

Boris Glavic

http://www.cs.iit.edu/~glavic/

http://www.cs.iit.edu/~cs520/

http://www.cs.iit.edu/~dbgroup/

Outline

ILLINOIS INSTITUTE OF TECHNOLOGY

0) Course Info
1) Introduction
2) Data Preparation and Cleaning
3) Schema matching and mapping
4) Virtual Data Integration
5) Data Exchange
6) Data Warehousing
7) **Big Data Analytics**
8) Data Provenance

3. Big Data Analytics

ILLINOIS INSTITUTE OF TECHNOLOGY

- **Big Topic, big Buzzwords ;-)**
- Here
  - **Overview of two types of systems**
    - Key-value/document stores
    - **Mainly**: Bulk processing (MR, graph, …)
  - **What is new compared to single node systems?**
  - **How do these systems change our approach to integration/analytics**
    - Schema first vs. Schema later
    - **Pay-as-you-go**

3. Big Data Overview

ILLINOIS INSTITUTE OF TECHNOLOGY

- **1) How does data processing at scale (read using many machines) differ from what we had before?**
  - Load-balancing
  - Fault tolerance
  - Communication
  - New abstractions
    - Distributed file systems/storage

3. Big Data Overview

ILLINOIS INSTITUTE OF TECHNOLOGY

- **2) Overview of systems and how they achieve scalability**
  - **Bulk processing**
    - MapReduce, Shark, Flink, Hyracks, …
    - Graph: e.g., Giraph, Pregel, …
  - **Key-value/document stores = NoSQL**
    - Cassandra, MongoDB, Memcached, Dynamo, …

3. Big Data Overview

ILLINOIS INSTITUTE OF TECHNOLOGY

- **2) Overview of systems and how they achieve scalability**
  - **Bulk processing**
    - MapReduce, Shark, Flink,
  - **Fault tolerance**
    - Replication
    - Handling stragglers
  - **Load balancing**
    - Partitioning
    - Shuffle

3. Big Data Overview

- **3) New approach towards integration**
  - **Large clusters enable directly running queries over semi-structured data (within feasible time)**
    - Take a click-stream log and run a query
  - One of the reasons why **pay-as-you-go** is now feasible
    - **Previously:** designing a database schema upfront and designing a process (e.g., ETL) for cleaning and transforming data to match this schema, then query
    - **Now:** start analysis directly, clean and transform data if needed for the analysis

3. Big Data Overview

ILLINOIS INSTITUTE OF TECHNOLOGY

- **3) New approach towards integration**
  - **Advantage of pay-as-you-go**
    - More timely data (direct access)
    - More applicable if characteristics of data change dramatically (e.g., yesterdays ETL process no longer applicable)
  - **Disadvantages of pay-as-you-go**
    - Potentially repeated efforts (everybody cleans the click-log before running the analysis)
    - Lack of meta-data may make it hard to
      - Determine what data to use for analysis
      - Hard to understand semantics of data

- **Scalable systems**
  - Performance of the system scales in the number of nodes
    - Ideally the per node performance is constant independent of how many nodes there are in the system
    - This means: having twice the number of nodes would give us twice the performance
  - Why scaling is important?
    - If a system scales well we can "throw" more resources at it to improve performance and this is cost effective

- **What impacts scaling?**
  - Basically how parallelizable is my algorithm
    - **Positive example**: problem can be divided into subproblems that can be solved independently without requiring communication
      - E.g., array of 1-billion integers [i_1, …, $i_{1,000,000,000}$] add 3 to each integer. Compute on n nodes, split input into n equally sized chunks and let each node process one chunk
    - **Negative example**: problem where subproblems are strongly intercorrelated
      - E.g., Context Free Grammar Membership: given a string and a context free grammar, does the string belong to the language defined by the grammar.

- **New problems at scale**
  - DBMS
    - running on 1 or 10's of machines
    - running on 1000's of machines
- Each machine has low probability of failure
  - If you have many machines, failures are the norm
  - Need mechanisms for the system to cope with failures
    - Do not loose data
    - Do not use progress of computation when node fails
  - This is called **fault-tolerance**

3. Big Data – Processing at Scale

- **New problems at scale**
  - DBMS
    - running on 1 or 10's of machines
    - running on 1000's of machines
- Each machine has limited storage and computational capabilities
  - Need to **evenly** distribute data and computation across nodes
    - Often most overloaded node determine processing speed
  - This is called **load-balancing**

3. Big Data – Processing at Scale

- **Building distributed systems is hard**
  - Many pitfalls
    - Maintaining distributed state
    - Fault tolerance
    - Load balancing
  - Requires a lot of background in
    - OS
    - Networking
    - Algorithm design
    - Parallel programming

- **Building distributed systems is hard**
  - Hard to debug
    - Even debugging a parallel program on a single machine is already hard
      - Non-determinism because of scheduling: Race conditions
      - In general hard to reason over behavior of parallel threads of execution
    - Even harder when across machines
  - Just think about how hard it was for you to first program with threads/processes

3. Big Data – Why large scale?

- **Datasets are too large**
  - **Storing a 1 Petabyte dataset requires 1 PB storage**
    - **Not possible on single machine even with RAID storage**
- **Processing power/bandwidth of single machine is not sufficient**
  - **Run a query over the facebook social network graph**
    - **Only possible within feasible time if distributed across many nodes**

3. Big Data – User's Point of View

- **How to improve the efficiency of distributed systems experts**
  - Building a distributed system from scratch for every store and analysis task is obviously not feasible!
- **How to support analysis over large datasets for non distributed systems experts**
  - How to enable somebody with some programming but limited/no distributed systems background to run distributed computations

3. Big Data – Abstractions

- **Solution**
  - Provide higher level abstractions
- **Examples**
  - **MPI** (message passing interface)
    - Widely applied in HPC
    - Still quite low-level
  - **Distributed file systems**
    - Make distribution of storage transparent
  - **Key-value storage**
    - Distributed store/retrieval of data by identifier (key)

3. Big Data – Abstractions

ILLINOIS INSTITUTE OF TECHNOLOGY

- **More Examples**
  - **Distributed table storage**
    - Store relations, but no SQL interface
  - **Distributed programming frameworks**
    - Provide a, typically, limited programming model with automated distribution
  - **Distributed databases, scripting languages**
    - Provide a high-level language, e.g., SQL-like with an execution engine that is distributed

3. Distributed File Systems

ILLINOIS INSTITUTE OF TECHNOLOGY

- **Transparent distribution of storage**
  - Fault tolerance
  - Load balancing?
- **Examples**
  - **HPC distributed filesystems**
    - Typically assume a limited number of dedicated storage servers
    - GPFS, Lustre, PVFS
  - **“Big Data” filesystems**
    - Google file system, HDFS

3. HDFS

ILLINOIS INSTITUTE OF TECHNOLOGY

- **Hadoop Distributed Filesystem (HDFS)**
- Architecture
  - One nodes storing metadata (name node)
  - Many nodes storing file content (data nodes)
- Filestructure
  - Files consist of blocks (e.g., 64MB size)
- Limitations
  - Files are append only

3. HDFS

- **Name node**
- Stores the directory structure
- Stores which blocks belong to which files
- Stores which nodes store copies of which block
- Detects when data nodes are down

- Clients communicate with the name node to gather FS metadata

3. HDFS

- **Data nodes**
- Store blocks
- Send/receive file data from clients
- Send heart-beat messages to name node to indicate that they are still alive

- Clients communicate data nodes for reading/ writing files

3. HDFS

- **Fault tolerance**
  - n-way replication
  - Name node detects failed nodes based on heart-beats
  - If a node if down, then the name node schedules additional copies of the blocks stored by this node to be copied from nodes storing the remaining copies

3. Distributed FS Discussion

ILLINOIS INSTITUTE OF TECHNOLOGY

- **What do we get?**
  - Can store files that do not fit onto single nodes
  - Get fault tolerance
  - Improved read speed (caused on replication)
  - Decreased write speed (caused by replication)
- **What is missing?**
  - Computations

3. Frameworks for Distributed Computations

- **Problems**
  - Not all algorithms do parallelize well
  - How to simplify distributed programming?
- **Solution**
  - Fix a reasonable powerful, but simple enough model of computation for which scalable algorithms are known
  - Implement distributed execution engine for this model and make it fault tolerant and load-balanced

3. MapReduce

ILLINOIS INSTITUTE OF TECHNOLOGY

- **Data Model**
  - Sets of key-value pairs $\{(k_1,v_1), \ldots, (k_n,v_n)\}$
  - **Key** is an identifier for a piece data
  - **Value** is the data associaed with a key
- **Programming Model**
  - We have two higher-level functions map and reduce
    - Take as input a user-defined function that is applied to elements in the input key-value pair set
  - Complex computations can be achieved by chaining map-reduce computations

3. MapReduce Datamodel

- **Data Model**
  - Sets of key-value pairs $\{(k_1,v_1), \ldots, (k_n,v_n)\}$
  - **Key** is an identifier for a piece data
  - **Value** is the data associaed with a key
- **Examples**
  - Document **d** with an **id**
    - (id, d)
  - Person with name, salary, and SSN
    - (SSN, “name, salary”)

3. MapReduce Computional Model

ILLINOIS INSTITUTE OF TECHNOLOGY

- **Map**
  - Takes as input a set of key-value pairs and a user-defined function `f:(k,v) -> {(k,v)}`
  - Map applies f to every input key-value pair and returns the union of the outputs produced by f

$\{(k_1,v_1),\ldots,(k_n,v_n)\}$

$\rightarrow$

$f((k_1,v_1)) \cup \ldots \cup f((k_n,v_n))$

3. MapReduce Computional Model

- **Example**
  - **Input**: Set of (city,population) pairs
  - **Task**: multiply population by 1.05
- **Map function**
  - `f: (city,population) -> {(city,population*1.05)}`
- Application of f through map
  - Input: `{(chicago, 3), (nashville, 1)}`
  - Output: `{(chicago, 3.15)} ∪ {(nashville, 1.05)}`

    `= {(chicago, 3.15), (nashville, 1.05)}`

3. MapReduce Computional Model

ILLINOIS INSTITUTE OF TECHNOLOGY

- **Reduce**
  - Takes as input a key with a list of associated values a user-defined function

    `g: (k,list(v)) -> {(k,v)}`

  - Reduce groups all values with the same key in the input key-value set and passes each key and its list of values to **g**. and returns the union of the outputs produced by **g**

$\{(k_1,v_{11}),\ldots,(k_1,v_{1n1}),\ \ldots\ (k_m,v_{m1}),\ldots,(k_m,v_{mnm})\}$

$\rightarrow$

$g((k_1,(v_{11},\ldots,v_{1n1}))\ \cup\ \ldots\ \cup\ g((k_m,(v_{m1},\ldots,v_{mnm}))$

3. MapReduce Computional Model

- **Example**
  - **Input**: Set of (state, population) pairs one for each city in the state
  - **Task**: compute the total population per state
- **Reduce function**
  - `f: (state,[p1, …, pn]) -> {(state,SUM([p1,…,pn])}`
- Application of f through map
  - Input: `{(illinois, 3), (illinois, 1), (oregon, 15)}`
  - Output: `{(illinois, 4), (oregon, 15)}`

- **Workflows**
  - Computations in MapReduce consists of map phases followed by reduce phases
    - The input to the reduce phase is the output of the map phase
  - Complex computations may require multiple map-reduce phases to be chained together

3. MapReduce Implementations

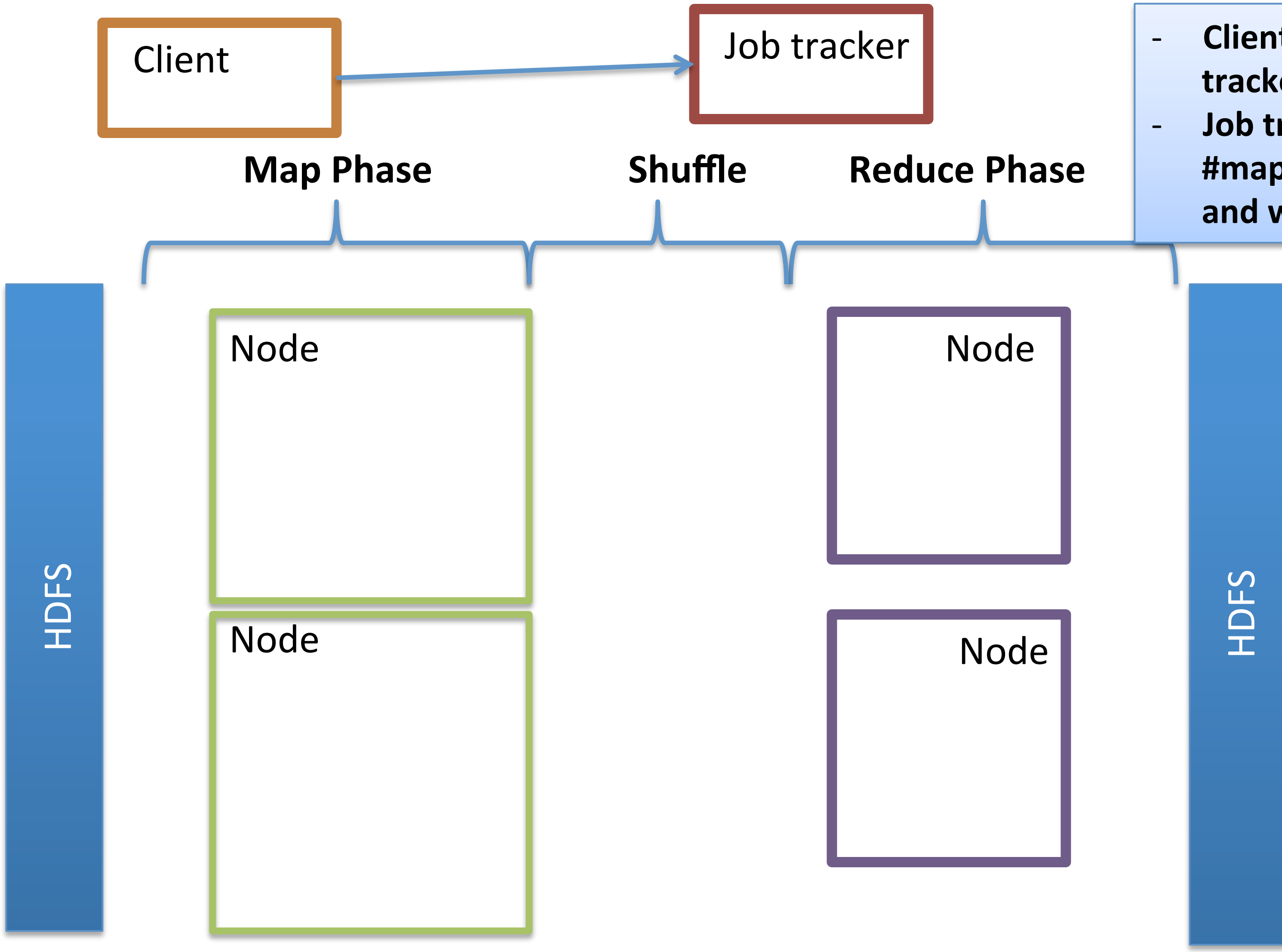
- **MapReduce**
  - Developed by google
  - Written in C
  - Runs on top of GFS (Google's distributed filesystem)
- **Hadoop**
  - Open source Apache project
  - Written in Java
  - Runs on-top of HDFS

- **Anatomy of a Hadoop cluster**
  - **Job tracker**
    - Clients submit MR jobs to the job tracker
    - Job tracker monitors progress
  - **Task tracker aka workers**
    - Execute map and reduce jobs
- **Job**
  - **Input**: files from HDFS
  - **Output**: written to HDFS
  - Map/Reduce UDFs

- **Fault tolerance**
  - **Handling stragglers**
    - Job tracker will reschedule jobs to a different worker if the worker falls behind too much with processing
  - **Materialization**
    - Inputs are read from HDFS
    - Workers write results of map jobs assigned to them to local disk
    - Workers write results of reduce jobs to HDFS for persistence

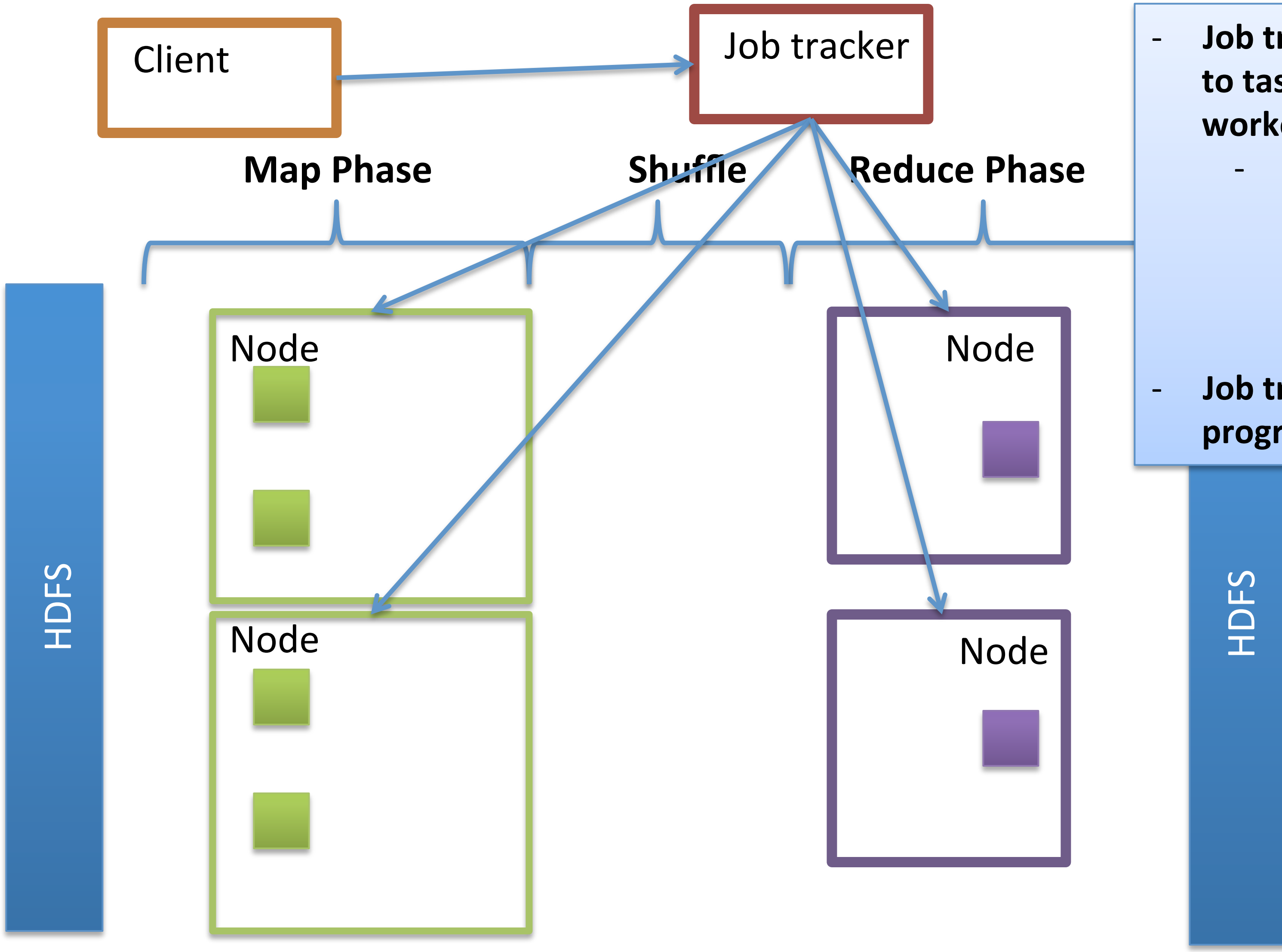
3. Hadoop – MR Job

ILLINOIS INSTITUTE OF TECHNOLOGY



- Clients sends job to job tracker
- Job tracker decides #mappers, #reducers and which nodes to use

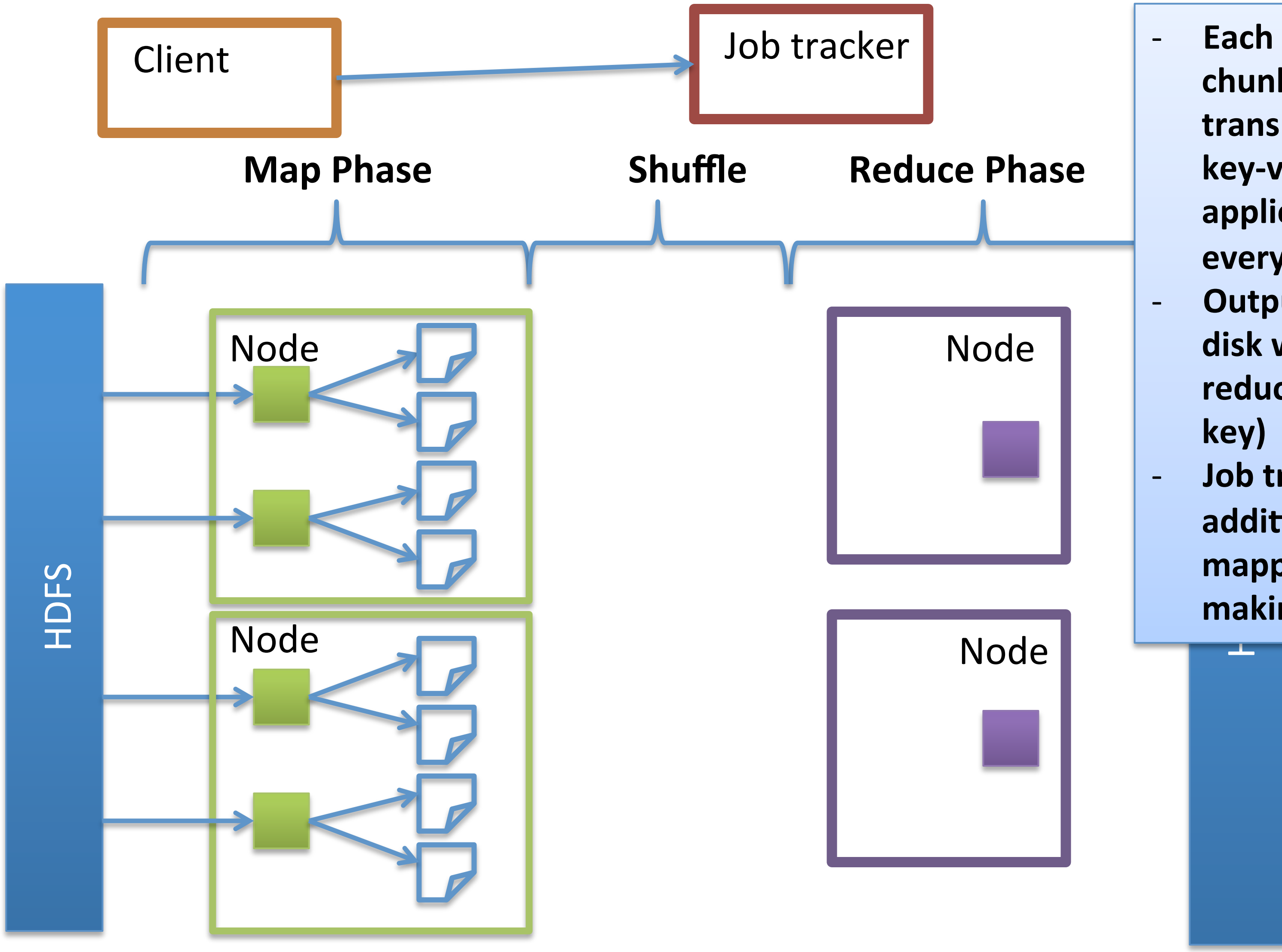
3. Hadoop – MR Job



- Job tracker sends jobs to task tracker on worker nodes
  - Try to schedule map jobs on nodes that store the chunk processed by a job
- Job tracker monitors progress

3. Hadoop – MR Job

ILLINOIS INSTITUTE OF TECHNOLOGY

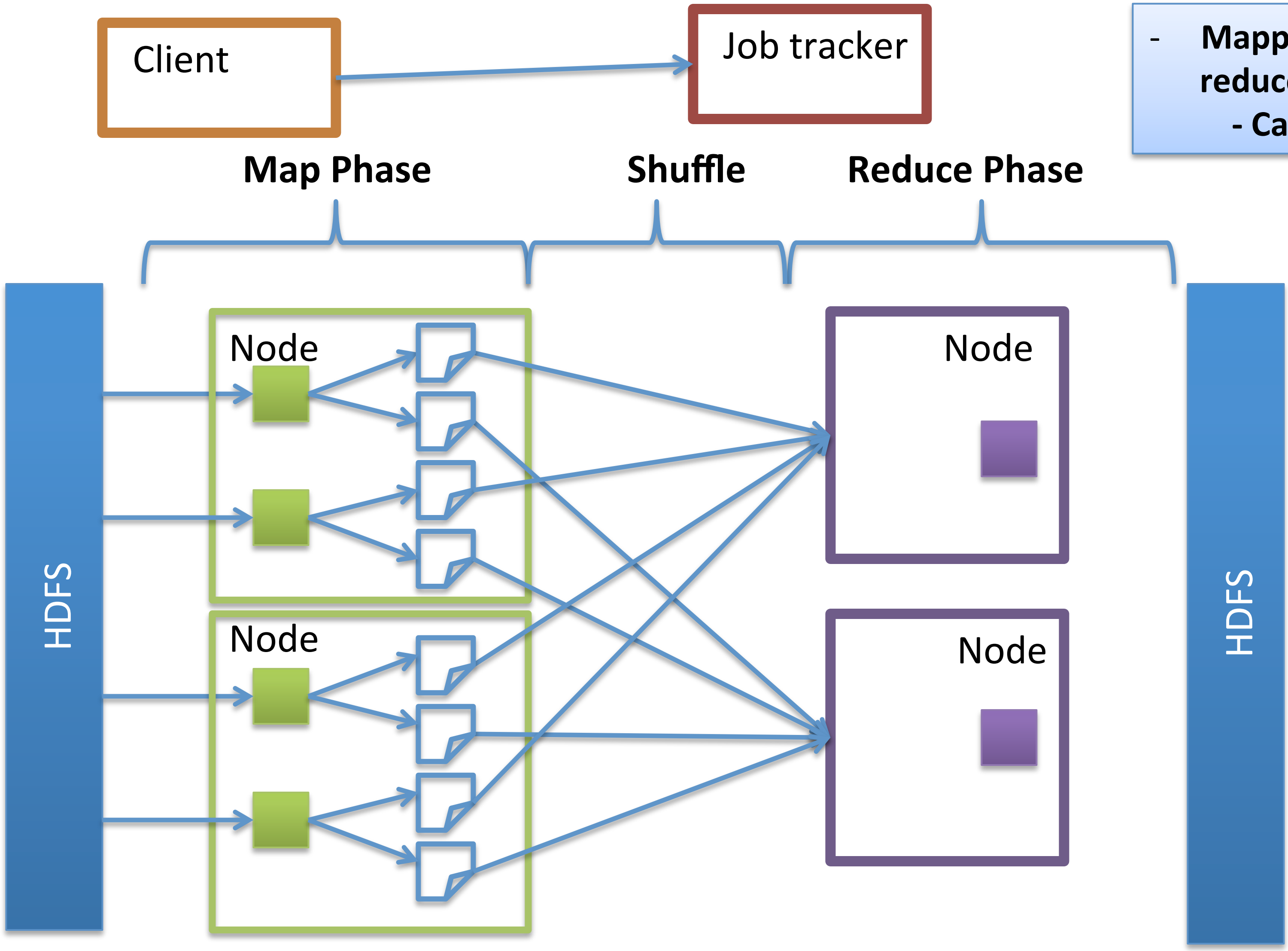


- Each mapper reads its chunk from HDFS, translates the input into key-value pairs and applies the map UDF to every (k,v)
- Outputs are written to disk with one file per reducer (hashing on key)
- Job tracker may spawn additional mappers if mappers are not making progress

3. Hadoop – MR Job

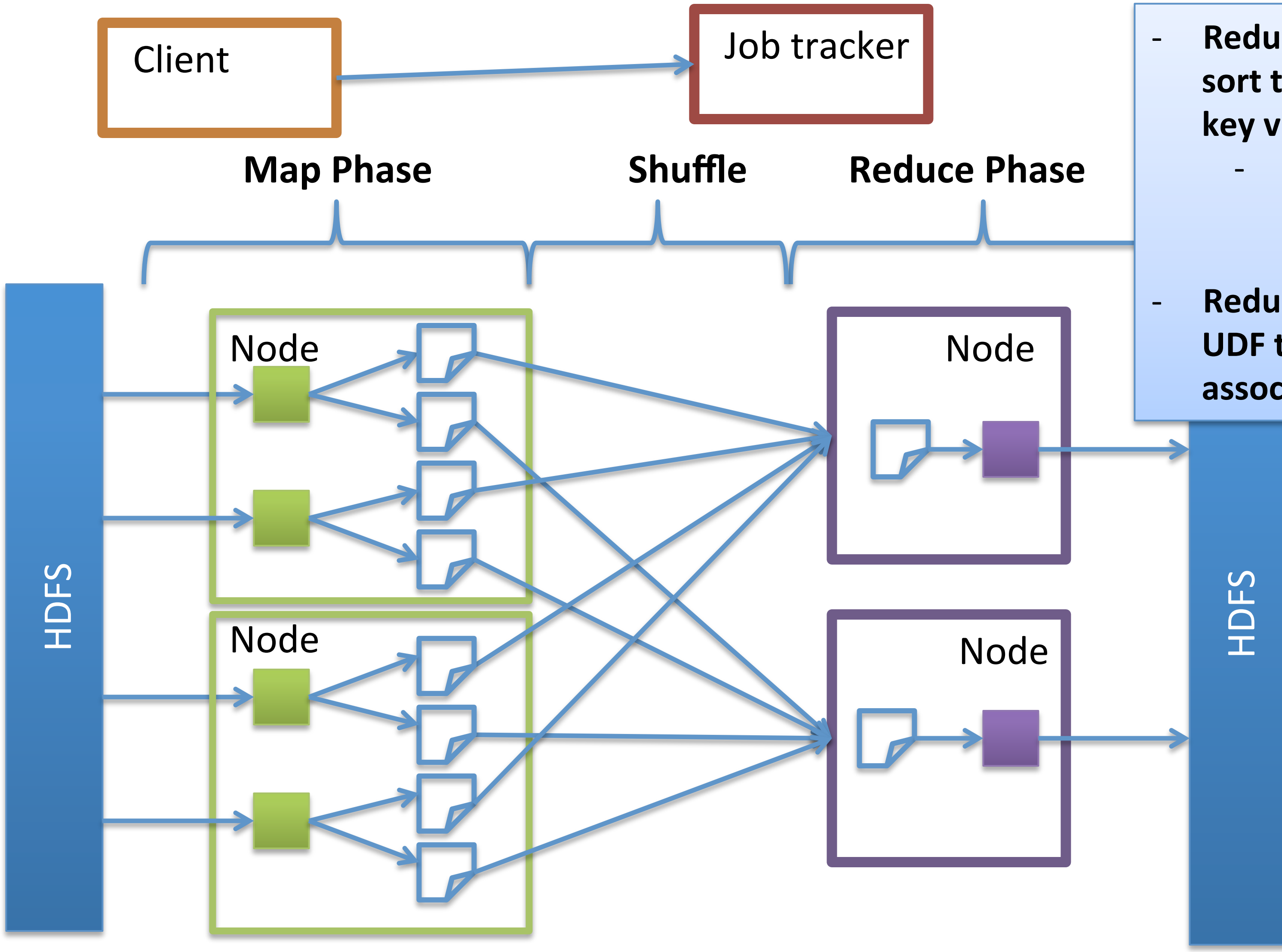
ILLINOIS INSTITUTE OF TECHNOLOGY

- Mappers send files to reducers (scp)
  - Called shuffle



3. Hadoop – MR Job

ILLINOIS INSTITUTE OF TECHNOLOGY



- Reducers merge and sort these input files on key values
  - External merge sort where runs already exists
- Reducer applies reduce UDF to each key and associated list of values

3. Combiners

- Certain reduce functions lend themselves to pre-aggregation
  - E.g., SUM(revenue) group by state
    - Can compute partial sums over incomplete groups and then sum up the pre-aggregated results
  - This can be done at the mappers to reduce amount of data send to the reducers
- Supported in Hadoop through a user provided combiner function
  - The combiner function is applied before writing the mapper results to local disk

3. Combiners

- Certain reduce functions lend themselves to pre-aggregation
  - E.g., SUM(revenue) group by state
    - Can compute partial sums over incomplete groups and then sum up the pre-aggregated results
  - This can be done at the mappers to reduce amount of data send to the reducers
- Supported in Hadoop through a user provided combiner function
  - The combiner function is applied before writing the mapper results to local disk

3. Example code – Word count

ILLINOIS INSTITUTE OF TECHNOLOGY

- https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html

```
public static class Map extends MapReduceBase implements Mapper<LongWritable, Text, Text, IntWritable> {
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();

        public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> output, Reporter
reporter) throws IOException {
          String line = value.toString();
          StringTokenizer tokenizer = new StringTokenizer(line);
          while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            output.collect(word, one);
          }
        }
}
```

3. Example code – Word count

ILLINOIS INSTITUTE OF TECHNOLOGY

- https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html

```
public static class Reduce extends MapReduceBase implements Reducer<Text, IntWritable, Text, IntWritable> {
        public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text, IntWritable> outp
ut, Reporter reporter) throws IOException {
          int sum = 0;
          while (values.hasNext()) {
            sum += values.next().get();
          }
          output.collect(key, new IntWritable(sum));
        }
}
```

3. Example code – Word count

ILLINOIS INSTITUTE OF TECHNOLOGY

```
public static void main(String[] args) throws Exception {
        JobConf conf = new JobConf(WordCount.class);
        conf.setJobName("wordcount");

        conf.setOutputKeyClass(Text.class);
        conf.setOutputValueClass(IntWritable.class);

        conf.setMapperClass(Map.class);
        conf.setCombinerClass(Reduce.class);
        conf.setReducerClass(Reduce.class);

        conf.setInputFormat(TextInputFormat.class);
        conf.setOutputFormat(TextOutputFormat.class);

        FileInputFormat.setInputPaths(conf, new Path(args[0]));
        FileOutputFormat.setOutputPath(conf, new Path(args[1]));

        JobClient.runJob(conf);
}
```

3. Systems/Languages on top of MapReduce

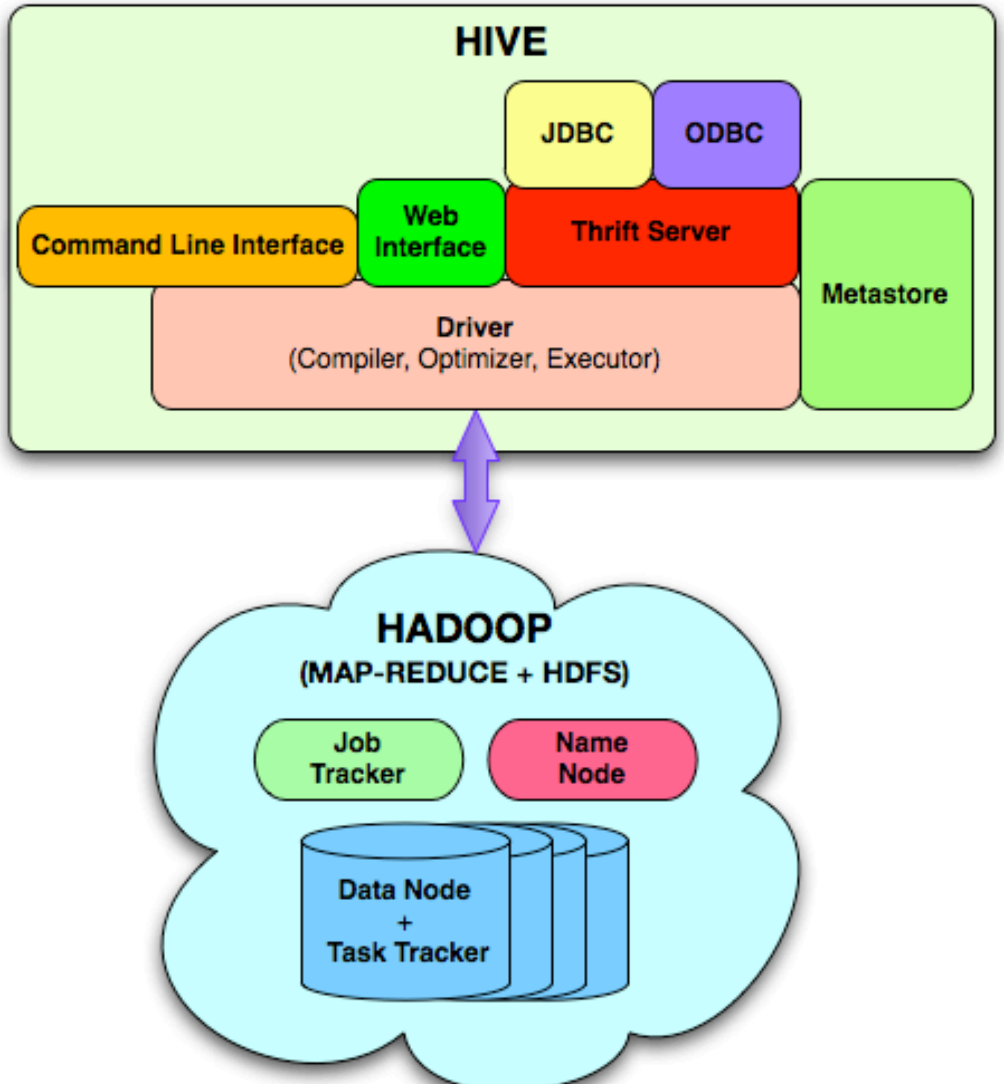
- Pig
  - Scripting language, compiled into MR
  - Akin to nested relational algebra
- Hive
  - SQL interface for warehousing
  - Compiled into MR
- …

- **Hive**
  - HiveQL: SQL dialect with support for directly applying given Map+Reduce functions as part of a query
  - HiveQL is compiled into MR jobs
  - Executed of Hadoop cluster

```
FROM (
        MAP doctext USING 'python wc_mapper.py' AS (word, cnt)
        FROM docs
        CLUSTER BY word
) a
REDUCE word, cnt USING 'python wc_reduce.py';
```

3. Hive Architecture

ILLINOIS INSTITUTE OF TECHNOLOGY



3. Hive Datamodel

ILLINOIS INSTITUTE OF TECHNOLOGY

- **Tables**
  - Attribute-DataType pairs
  - User can instruct Hive to partition the table in a certain way
- **Datatypes**
  - Primitive: integer, float, string
  - Complex types
    - Map: Key->Value
    - List
    - Struct
  - Complex types can be nested
- **Example**:

  CREATE TABLE t1(st string, fl float, li list<map<string, struct<p1:int, p2:int>>);
- **Implementation**:
  - Tables are stored in HDFS
  - Serializer/Deserializer - transform for querying

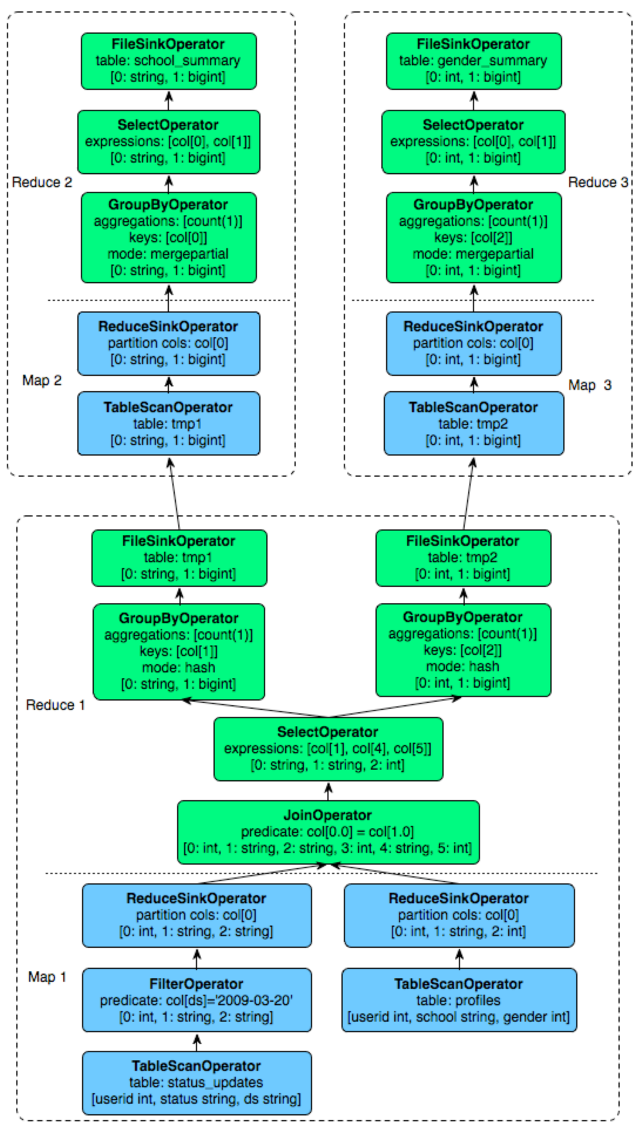
3. Hive - Query Processing

ILLINOIS INSTITUTE OF TECHNOLOGY

- Compile HiveQL query into DAG of map and reduce functions.
  - A single map/reduce may implement several traditional query operators
    - E.g., filtering out tuples that do not match a condition (selection) and filtering out certain columns (projection)
  - Hive tries to use the partition information to avoid reading partitions that are not needed to answer the query
    - For example
      - table **instructor**(name,department) is partitioned on department
      - **SELECT** name **FROM** instructor **WHERE** department = ‘CS’
      - This query would only access the partition of the table for department ‘CS’

- **Join implementations**
  - **Broadcast join**
    - Send the smaller table to all nodes
    - Process the other table partitioned
      - Each node finds all the join partners for a partition of the larger table and the whole smaller table
  - **Reduce join (partition join)**
    - Use a map job to create key-value pairs where the key is the join attributes
    - Reducer output joined rows

3. Example plan



Spark

- MR uses heavy materialization to achieve fault tolerance
  - A lot of I/O
- **Spark**
  - Works in main memory (where possible)
  - Inputs and final outputs stored in HDFS
  - Recomputes partial results instead of materializing them - **resilient distributed datasets** (*RDD*)
    - **Lineage**: Need to know from which chunk a chunk was derived from and by which computation

Summary

- Big data storage systems
- Big data computation platforms
- Big data "databases"
- How to achieve scalability
  - Fault tolerance
  - Load balancing
- Big data integration
  - Pay-as-you-go
  - Schema later

Outline

ILLINOIS INSTITUTE OF TECHNOLOGY

0) Course Info
1) Introduction
2) Data Preparation and Cleaning
3) Schema matching and mapping
4) Virtual Data Integration
5) Data Exchange
6) Data Warehousing
7) Big Data Analytics
8) Data Provenance