Big Data

Ming-Hwa Wang, Ph.D.
COEN 242 Introduction to Big Data
Department of Computer Engineering
Santa Clara University

Introduction

- big data
  - Moore's law: CPU speed doubles every 18 months. Applications automatically got faster without having to change their code.
  - heat dissipation and signal inference stopped making individual processors faster, and switched towards adding more parallel CPU cores. Applications needed to be modified to add parallelism.
  - data grows faster than Moore's law, and data is addictive
  - expensive to collect data explosion: sensors, cameras, public dataset, etc.
- the next generation of successful businesses will be built around data; Hal Varian said, "The ability to take data – to be able to understand it, to process it, to extract value from it, to visualize it, to communicate it – that's going to be a huge important skill in the next generation."
- social, mobile and cloud lead us to the data-enabled world
  - digital exoskeleton or paperless office transfers of pre-existing paper process into the computer by digitization
  - digital nervous system with a large number of inflows and outflows of data and a high level of networking
- the value of big data to an organization falls into two categories: analytical use and enabling new products
- data science: a data-driven application acquires its value from the data itself, and creates more data as a result, or a data product
  - CDDB maps the length of the track to the title
  - Google's PageRank, spelling checking, integrated voice search, track the progress of epidemic, etc.
  - Facebook and LinkedIn use patterns of friendship relationship to suggest other people you may know
  - Amazon saves your searches, correlates with other users and give appropriate recommendations; customers generate a trail of data exhaust that can be mined and put to use
- Five Vs (volume, velocity or rate of data grows, variety or different data types, veracity or uncertainty of available data, and value for business): the web has people spending more time online, and leaving a trail of data wherever they go; mobile applications leave even richer data trail (geolocation, video, audio); point-of-sale devices capture all retail transactions; online retailers compile customers clicks and provide purchase recommendations; financial traders turned systems that cope with fast moving data to their advantages; chatter from social networks, web server logs, traffic flow sensors, satellite imagery, broadcast audio streams, banking transactions, MP3 music, GPS trails, financial data
- data need to be stored and then can be analyzed/mined; when you can, keep everything
- streaming data (fast-moving data, or complex event processing): input data may too fast to store in their entirety (hoping not thrown away anything useful), and analysis must occur as the data streams in, e.g., mobile application, online gaming, etc.
- big data: the information that can't be processed or analyzed using traditional processes or tools
  - since early '80s, processor speed from 10 MHz to 3.6GHz, RAM from $1,000/MB to $25/GB, disk transfer speed 75M/second
  - traditional processing either using single powerful CPU or parallel/distributed processing by transferring data to CPU in order to process, and thus have synchronization, bandwidth, temporal dependency and partial failure issues, not good for big data
  - data locality: it's the program that needs to move, not the data
  - traditional high-performance file servers (NetApp, EMC) have advantages of fast random access and support many concurrent clients, but with high cost per terabyte of storage
  - traditional relational databases with static schemas are designed for consistency and support complex transactions and denormalized (to speed up querying) by time consuming pre-materialization (to predict the queries) into an online analytic processing (OLAP) cube, but not scale well with the size
  - graph database Neo4J: social network relations are graph by nature
  - most data analysis is comparative, new non-relational databases or NoSQL (origin from Google's BigTable or Amazon's Dynamo) store huge unstructured data and provide eventual consistency but not absolute consistency, provide enough structure to organize data, but do not require the exact schema of the data before storing it
- data analysis
  - huge amount of data in different formats and may have errors
  - data conditioning and cleaning (80% of the total effort): get data into a state where it's usable; but data is frequently missing or incongruous, and some are difficult to be parsed, e.g., natural language processing

- agile practice: faster product cycles, close interaction between developers and customers, and testing; MapReduce enabling agile data analysis, and have soft real-time response
- machine learning libraries: PyBrain, Elefant, Weka, Mahout, Google Prediction API, OpenCV (for computer vision), Mechanical Turk (to develop training sets)
- statistics is the grammar of data science
    - open source R language and CRAN
- narrative-driven data analysis or data visualization: making data tell its story:
    - use visualization to see how bad row data is, and use visualization as the first step for data analysis
    - packages: GnuPlot, R, IBM's ManyEyes, Casey Reas' and Ben Fry's Processing (with animations)
- data jiujitsu: use smaller auxiliary problems to solve a large, difficult problem that appear intractable, e.g., CDDB
- the Storage, MapReduce and Query (SMAQ) stack for big data (analogy to LAMP stack – Linux, Apache, MySQL and PHP- for web 2.0)
- the drivetrain approach
    - define a clear objective
    - predictive modeling or optimal decision
    - simulation
    - optimization
- data scraping
    - the uber problem: challenges of data scarping: scale, metadata, data format, source domain complexity
    - tools: ScraperWiki, Junar, Mashery, Apigee, 2scale
    - legal implications of data scraping
        - copyright: the data or facts is not copyrightable, but specific arrangement or selection of the data will be protected
        - terms of service are based in contract law, but their enforceability is a gray area, e.g., YouTube forbids a user from posting copyrighted video
        - trespass to chattels: high-volume web scraping could cause significant monetary damages (e.g., denial of service) to the sites being scraped

Hadoop

- origin from Yahoo (created by Dong Cutting), the top-level Apache open source project Hadoop (http://hadoop.apache.org) written in Java: a computation environment built on top of a distributed clustered file system for very large-scale data operations, and based on Google File System (GFS) and the MapReduce programming paradigm.
    - MapReduce follows the functional programming paradigm for parallelism and locality of data access, work is broken down into mapper and reducer tasks to manipulate data that is stored across a cluster of servers for massive parallelism
- easy to use: to run a MapReduce job, users just need to write map and reduce functions and the Hadoop handles all the rest (e.g., decompose the submitted job into map/reduce tasks, schedule tasks, monitor tasks to ensure successful completion or restart tasks that fails), a tradeoff for flexibility
- a map is a function whose input is a single aggregate and whose output is a bunch of key-value), or (k, v) pairs, wasteful of network and disk I/O, but minimize memory usage in the mappers
- sort and shuffle phase: the master controller sorts the key-value pairs by key, divides among all the reduce tasks by hashing (or by alphanumeric partitioning), and shuffle all pairs with the same key to the same reducer, or optionally via Shuffle/sort, which direct the tasks from mapper to a specific reducer by grouping/sorting, or via Combiner, which perform local aggregation the tasks from mapper before sending to reducer
- a reduce function takes multiple map outputs with the same key and combines their values, i.e., reduces (k, [v_1, v_2, …, v_n]) to (k, v), If the reducer function is associative and commutative, the values can be combined in any order with the same result, when there is no more reduce work, all final key-value pairs are emitted as output, the outputs from all reducer tasks are merged into a single file
- efficiency:
    - partitioning/shuffling: increase parallelism and reduce the data transfer by partitioning the output of the mappers into groups (buckets or regions), my multiple reducers operate on the partitions in parallel with final results merged together
    - combiner: much of the data transfer between mapper and reducer are repetitive (multiple key-value pairs with the same key), a combiner (in essence a reducer function) combines all the data for the same key into a single value
        - a combinable reducer: its output must match its input
        - non-combinable reducer needs to be processed into pipelined map-reduce steps
    - maximum parallelism: use one reducer task to execute each reducer, execute each reducer task at a different compute node
    - reduce skew: different reducers take different time, a significant difference in the amount of time each takes exhibiting skew, we can reduce the impact by using fewer reducer tasks than there are reducers, and using more reduce tasks than there are compute nodes
    - limit reducer tasks to prevent intermediate files explosion

- Hadoop is designed to scan through large complex/structured data set to produce its results through a highly scalable, distributed batch processing system. Hadoop is not about speed-of-thought response times, real-time warehousing, or blazing transactional speed; it is built around a function-to-data model and is about discovery and making the once near-impossible possible; for reliability (self-healing), failures are expected, mean time to failure (MTTF) rate associated with inexpensive hardware is well understood, and with built-in fault tolerance and fault compensation; it is open source, so no vendor lock-in problem.
- Hadoop components (or Hadoop core):
  - Hadoop distributed file system (HDFS): data redundancy
  - Hadoop MapReduce model: programming redundancy
  - Hadoop common: a set of libraries
  - Hadoop YARN (Yet-Another-Resource-Negotiator)
- related projects (Apache project, Apache Incubator projects, and Apache Software Foundation hosted on GitHub)
  - Apache Avro, Protobuffer, Thrift: for data serialization, split-able compression
  - Cassandra, HBase, Mango, and Hypertable: for column-oriented huge databases storing key-value pairs for fast retrieval, act as a source and a sink for MapReduce
  - Chukwa: a monitoring system for large distributed system
  - Mahout: a machine learning library for analytical computation, collaborative filtering, user recommendations, clustering, regression, classification, pattern mining
  - Hive: ad hoc – i.e., no compilation requirement - SQL-like queries for data aggregation and summarization for Hypertable
  - Pig: a high-level Hadoop parallel data-flow programming language for Cassandra, and its User Defined Functions (UDFs)
  - Cloudera Hue (Hadoop User Experience): web console
  - Zookeeper (naming, coordination and workflow services for distributed applications) and Oozie (managing job workflow, scheduling, and dependencies)
  - Ambari (a web-based tool for provisioning, configuration, monitoring, administration, upgrade services, and deployment) and Whirr (running services on cloud platforms)
  - Sqoop (SQL to Hadoop) and Flume: for data ingestion
  - Cascading (provide a wrapper for Java applications) and Cascalog (as a query language)
- free download
  - BigDataUniversity.com
  - www.ibm.com/software/data/infosphere/biginsights/basic.html
- MapReduce programming languages
  - Java
  - scripting languages through Hadoop Streaming interface for simple/short applications
  - Pig (and PigLatin, a dataflow scripting language)
  - Hive and Hive Query Language (HQL): run from Hive shell, Java Database Connectivity (JDBC), or Open Database Connectivity (ODBC) via Hive Thrift Client; enables Hadoop to operate as a data warehouse; Hive with HBase is 4-5 times slower than HDFS and limited store a petabyte in HBase
  - Jaql: a functional, record-oriented, declarative query language for JavaScript Object Notation (JSON)
- data locality: MapReduce assign workloads to servers where the data to be processed is stored without using storage area network (SAN) or network attached storage (NAS)
- data redundancy for high availability, reliability, scalability and data locality: HDFS replicates each block onto three servers by default, a NameNode (and optionally a BackupNode to prevent single point of failure, SPOF, because any data loss in metadata will result in a permanent loss of data in the cluster) keeps track of all the data files in HDFS
  - IBM General Parallel File System (GPFS) based on SAN
  - IBM GPFS extended to a share nothing cluster (GPFS-SNC)
  - Google File System (GFS)
  - Cloudstore: an open-source DFS from Kosmix
- transparent to the application: Hadoop HDFS will contact the NameNode, find the servers that hold the data for the application task, and then send your application to run locally on those node
- MapReduce model: the mappers run parallel converting input into tuples of key and value pairs, and then the reducer combines the results from mappers to get the solutions
  - the master nodes run three daemons: NameNode (optionally a second NameNode to prevent single point of failure), a secondary NameNode does housekeeping for the NameNode, and JobTracker; each slave or worker node runs two daemons: DataNodes and TaskTrackers
  - typically, a cluster with more nodes will perform better than on with fewer, faster nodes; in general, more disks is better; Hadoop nodes are typical disk- and network-bond
  - an application (run on client machine) submits a job to a node (running a daemon JobTracker) in Hadoop cluster, the JobTracker communicates with the NameNode, breaks the job into map and reduce tasks, and TaskTracker schedules the tasks on the node in the cluster where the data resides.

    - a set of continually running daemons TaskTracker agents monitor the status of each task and report to JobTracker
- HDFS is not a POSIX-compliance file system, can't interact with Linux or Unix file system, and you should use the /bin/hdfs dfs <args> file system shell command interface or FsShell (invoked by Hadoop command fs)
    - HDFS shell commands: `put, get, cat, chmod, chown, copyFromLocal, copyToLocal, cp, expunge, ls, mkdir, mv, rm`
    - Java applications need import the org.apache.hadoop.fs package
- no benefit using RAID because HDFS provide build-in redundancy, and RAID striping is slow than JBOD used by HDFS (but RAID is preferred to be used on master node)
- no benefit using virtualization because multiple virtual nodes per node hurts performance
- no benefit using blade servers
- use RedHat Enterprise Linux (RHEL) on master nodes and CentOS on slaves; prefer server distribution (e.g., Ubuntu) to desktop distribution (e.g., Fedora)
- configuring the system:
    - don't use Linux logical volume manager (LVM) to make all your disks appear as a single volume
    - check BIOS settings (e.g., disable IDE emulation when using SATA drives)
    - test disk I/O speed with `hdparm -t /dev/sda1`
    - mount disks with the noatime option to not update access time
    - reduce the swappiness of the system by set vm.swappiness to 0 or 5 in /etc/sysctl.conf
    - increase nofile ulimit in /etc/security/limits.conf to at least 32K
    - disable IPv6, and disable SELinux
    - install and configure the ntp daemon to ensure the time on all nodes are synchronized (especially important for HBase)
- JVM recommendations
    - always use official Oracle JDK (http://java.com/) 1.6.0u24 or 1.6.0u26 (don't use newest version until it is fully tested)
- Hadoop versions
    - http://hadoop.apache.org/
    - Cludera's Distribution for Hadoop (CDH) free and Enterprise editions http://www.cloudera.com/
- integration with streaming data sources
    - Cloudera Flume: a distributed data collection service for flowing data into a Hadoop cluster
        - source and predefined source adapters
        - decorator: IBM Information Server
        - sink: Collector Tier Event sink, Agent Tier sink, Basic sink
    - Facebook Scribe
- NoSQL databases contain MapReduce functionality:
    - CouchDB: semi-structured document-based storage with JavaScript
    - MongoDB: better performance with JavaScript
    - Riak: high availability with JavaScript or Erlang
- integration with SQL databases
    - Sqoop with Java JDBC database API, import data from relational databases into Hadoop
    - Massively parallel processing (MPP) databases: Greenplum based on PostreSQL DBMS, Aster Data's nCluster
- data warehouse integrated with MapReduce (e.g., Zynga's Vertica, IBM's Netezza) or use a hybrid solution: using on-demand cloud resources to supplement in-house deployments
- search with MapReduce: Solr, ElasticSearch
- initiatives
    - need for scale/interaction: interactive 5s-1m, non-interactive 1m+, batch 1h+
    - next generation execution YARN: improve MapReduce performance, low latency, streaming services
    - HDFS 2.0: with federation (more than one NameNodes) and checkpoint
    - Ambari: for Hadoop 2.0, more scale, root cause analysis
    - Herd: streaming and ETL integration with HCatalog/Hive
    - Continuum: for HDFS snapshots and mirroring for disaster recovery
    - Stinger: enhance Hive for BI use cases
    - Knox: Hadoop gateway for security

Common MapReduce Algorithms

- basic ideas
    - Mapper role is filter, transform and parse (FTP) or extract, transform and load (ETL)
    - MapReduce jobs tend to be relatively short in terms of lines of code, and many of MapReduce jobs use very similar code; reuse of existing Mappers/Reducers, with minor modifications
    - it is typical to combine multiple small MapReduce jobs together in a single workflow using Oozie
    - the following common MapReduce algorithms are the basis for more complex MapReduce jobs
- the famous "Word Count" MapReduce program (used in spell checkers, language detection and translation systems, etc.) from Cloudera:

```
package org.myorg;
import java.io.IOException;
```

```java
import java.util.*;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.util.*;
public class WordCount {
  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);
      }
    }
  }
  public static class Reduce extends MapReduceBase implements
Reducer<Text, IntWritable, Text, IntWritable> {
    public void reduce(Text key, Iterator<IntWritable> values,
OutputCollector<Text, IntWritable> output, Reporter reporter)
throws IOException {
      int sum = 0;
      while (values.hasNext()) {
        sum += values.next().get();
      }
      output.collect(key, new IntWritable(sum));
    }
  }
  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);
  }
}
```

- word count
  - Mapper emits (word, 1) for each word
- sorting large data sets
  - keys are passed to the Reducer in sorted order, so it is well suited for sorting large data sets; for input contains lines with single value
  - Mapper is merely the identity function for the value

    (k, v) → (v, _)
  - Reducer is the identity function

    (k, _) → (k, "")
  - for multiple Reducers, then each Reducer is sorted independently; to generate merged sort order, add a Partitioner

    `if key < “c” then reducer = 1 else reducer = 2`
  - as a speed test for Hadoop framework's I/O
- searching large data sets
  - Mapper compares the line against the pattern, if match, output (line, _) or (filename+line, _), otherwise, output nothing
  - Reducer is the identity function
- secondary sort
  - the key is sorted, but we also want to sort values within each key
  - a naïve solution loop through all values in Reducer and keep track the largest value and emit the largest value at end
  - a better solution arrange the values for a given key to the Reducer in descending sorted order, and Reducer just emits the first value
  - custom comparator classes can compare composite keys (the actual key and the value), and specified in the driver code by

    `job.setSortComparatorClass(MyCC.class);`
  - to pass all values for the same natural key in one call to Reducer by using a grouping comparator class using

    `job.setGroupingComparatorClass(MyGC.class);`
  - extend WritableComparator by overriding compare( ) and using

    `job.setOutputKeyComparatorClass(MyOKCC.class);`
  - extend WritableComparator by overriding compare( ) and using

    `job.setOutputValueGroupingComparatorClass(MyOVGC.class);`
- inverted index
  - assume the input is a set of file containing lines of text; key is the byte offset of the line, and value is the line itself
  - Mapper emits (word, filename) or (word, filename+line) for each word in the line
  - Reducer is the identity function and emits (word, filename_list)
- calculating word co-occurrence

- measure the frequency with which two words appears close to each other in a corpus of documents
- this is at the heart of many data-mining techniques, and provides results for "people who did this, also do that", e.g., shopping recommendations, credit risk analysis, identifying 'people of interest'
- Mapper

```
map(doc_id a, doc d) {
    foreach w in d do
        foreach u near w do
            emit(pair(w, u), 1)
}
```

- Reducer

```
reduce(pair p, Iterator counts) {
    s = 0
        foreach c in counts do
            s += c
        emit(p, s)
}
```

- two-stage map-reduce using pipes-and-filters approach: the output of one stage serves as input to the next
  - reuse: the intermediate output may be useful for different outputs
  - materialized view: the saved intermediate records, often represent the heaviest amount of data access
- term frequency – inverse document frequency (TF-IDF)
  - for how important is this term in a document
  - real-world systems typical perform "stemming" on terms: removal of plurals, tense, possessives, etc.
  - TF-IDF considers both the frequency of a word (term) in a given document (TF) and the number of documents which contains the word (IDF) = log(N/n), where N is total number of documents, and n is the number of documents that contain a term; TF-IDF = TF x IDF
  - Stop-words: some words appear too frequently in all documents to be relevant
  - data mining application example: music recommendation system
  - term weighting function: assign a score (weight) to each term (word) in a document
  - several small jobs add up to full algorithm: 3 MapReduce jobs
    1. compute term frequencies: Mapper: input (doc_id, contents) output ((term, doc_id), 1), Reducer: output((term, doc_id), tf), and we can add a combiner, which will use the same code as the Reducer
    2. compute number of documents each words occurs in: Mapper: input ((term, doc_id), tf) output (term, (doc_id, tf, 1)), Reducer: output ((term, doc_id), (tf, n))
    3. compute TF-IDF: Mapper input ((term, doc_id), (tf, n)), output ((term, doc_id), TF x IDF), Reducer: the identity function
  - working at scale: any time when you need to buffer data, there is a potential scalability bottleneck
    - since we need to buffer (doc_id, tf) in the second MapReduce job, especially for very-high-frequency words, the possible solutions can ignore those words, write out intermediate data to a file, or use another MapReduce pass
- incremental map-reduce
  - new data keeps coming in, thus we need to rerun the computation to keep the output up to date
  - structure a map-reduce computation to allow incremental updates
    - only if the input data changes does the mapper need to be rerun
    - if we are partitioning the data for reduction, then any partition that's unchanged does not need to be re-reduced
    - if changes are additive, then just run the reduce with the existing result and the new additions
    - if there are destructive changes (updates, deletes), break the reduce operation into steps and only recalculating those steps whose inputs have changed using a dependency network

MapReduce Algorithms for Relational Algebra Operations

- traditional relational database uses relational algebra:
  - the set of attributes of a relation is schema, a relation is a table with column headers (attributes), and rows of the relation (tuples) $R(A_1, A_2, \ldots, A_n)$; a relation can be stored as a file in a distributed file system, and the elements of this file are the tuples of the relation
- selection:
  - $\sigma_c(R)$: apply a condition C to each tuple in the relation and produce as output only those tuples that satisfy C
  - map function: for each tuple t in R, test if it satisfies C; if so, produce the key-value pair (t, t)
  - reduce function: identity
- projection:
  - $\pi_s(R)$: for some subset S of the attributes of the relation, produce from each tuple only the components for the attributes in S
  - map function: for each tuple t in R, construct a tuple t' by eliminating from t those components who attributes are not is S, then output the key-value pair (t', t')
  - reduce function removes duplicates (may add combiners for efficiency): turn (t', [t', t', …, t']) to (t', t')

- set union, intersection, and difference for two relations *R* and *S* with same schema:
  - union map function: turn each tuple *t* into a key-value pair (*t*, *t*)
  - union reduce function removes duplicates: turn (*t*, [*t*, *t*]) to (*t*, *t*)
  - intersection map function: turn each tuple *t in R* into a key-value pair (*t*, *t*)
  - intersection reduce function removes duplicates: if (*t*, [*t*, *t*]), produce (*t*, *t*), otherwise, produce nothing
  - difference map function: for a tuple *t* in *R*, produce key-value pair (*t*, *R*), and for a tuple *t* in *S*, produce key-value pair (*t*, *S*)
  - difference reduce function: if (*t*, [*R*]), produce (*t*, *t*), otherwise, produce nothing
- bag union, intersection, and difference for two relations *R* and *S*:
  - bag union: the bag of tuples in which tuple *t* appears the sum of the numbers of times it appears in *R* and *S*
  - bag intersection: the bag of tuples in which tuple *t* appears the minimum of the numbers of times it appears in *R* and *S*
  - bag difference: the bag of tuples in which the number of times a tuple *t* appears is equal to the number of times it appears in *R* minus the number of times it appears in *S*; a tuple that appears more times in *S* than in *R* does not appear in the difference
- natural join:
  - $R \bowtie S$ given two relations *R* and *S*, if the tuples agree to all the attributes that are common to the two schemas, then produce a tuple that has components for each of the attributes in either schema and agrees with the two tuples on each attribute
  - map function: for each tuple (*a*, *b*) of *R*, produce the key-value pair (*b*, (*R*, *a*)), for each tuple (*b*, *c*) of *S*, produce the key-value pair (*b*, (*S*, *c*))
  - reduce function: if (*b*, [(*R*, *a*), (*S*, *c*)]), produce (_, (*a*, *b*, *c*)), otherwise, produce nothing
  - equijoin: the tuple-aggreement condition involves equality of attributes from the two relations that do not necessary have the same name
- grouping and aggregation:
  - $\gamma_X(R)$: where X is a list of elements that are either a grouping attribute or an expression $\theta(A)$, where θ is one of the 5 aggregation operations (SUM, COUNT, AVG, MIN, and MAX) and *A* is an attribute not among the grouping attributes
  - for relation *R*(*A*, *B*, *C*), the map and reduce function of $\gamma_{A,\theta(B)}(R)$ are:
    - map function: for each tuple (*a*, *b*, *c)*, produce the key-value pair (*a*, *b*)
    - reduce function: apply the aggregate operator ⍰ to the list [b_1, b_2, …, b_n] with key *a* and get the result *x*, then produce the pair (*a*, *x*)
  - if there are several grouping attributes, then the key is the list of the values of a tuple for all these attributes; if there is more than one aggregation, then the reduce function applies each of them to the list of values associated with a given key and produces a tuple consisting of the key, including components for all grouping attributes if there is more than one, followed by the results of each of the aggregations
- examples:
  - use paths of length two in the web: a link L_1 from U_1 to U_2 or $L_1(U_1, U_2)$, and a link $L_2(U_2, U3)$, use natural join of links: $\pi_{U1,U3}(L_1 \bowtie L_2)$
  - compute a count of number of friends of each user using grouping and aggregation in social network: $\gamma_{user,count(friends)}(friends)$:

Development

- run develop jobs on a virtual machine in pseudo-distributed mode (all five Hadoop daemons are running on the same single-machine cluster in its own JVM), test before deploying to the real cluster
- compile and submit a MapReduce job

```
$ javac -classpath `hadoop classpath` *.java
$ jar cvf wc.jar *.class
$   hadoop   jar   wc.jar   WordCount   <input_directory_file>
<output_directory_file>
$ hadoop fs -ls <output_directory_file>
$ hadoop fs -cat <output_directory_file>/part-<r>-0000 | less
$ hadoop fs -rm -r <output_directory_file>
```

- stop a job through JobTracker

```
$ mapred job -list
$ mapred job -kill <job_id>
```

Communication Cost Model

- assume the communication cost as the way to measure the efficiency of an algorithm instead of exceptions of execution time dominates (because tasks tend to be very simple) or moving data from disk to memory (or vice versa)
  - example: if relation *R*(*A*, *B*) has size *r* and *S*(*B*, *C*) has size *s*, then the join $R(A, B) \bowtie S(B, C)$ has communication cost O(*r* + *s*)
- communication from map tasks to reduce tasks is likely to be across the interconnect of the cluster
- the computation of multiway joins as single MapReduce jobs is more efficient than the straightforward cascade of 2-way joins
- star join: a chain store keeps a huge fact table with each entry representing a single sale $F(A_1, A_2, …, A_n)$, and keeps a relative small dimension table giving information about the participant (e.g., a customer) $D(F_1, B_{11}, B_{12}, …, B_{1n})$, then analytic queries typical join the fact table with several of the dimension table (a "star" join) and then aggregate the result

- count only input size (normally is huge) and not output size (usually is summarized or aggregated)
- two parameters
  - small reducers size *q*: force many reducers for high degree of parallelism, and running reducers in memory
  - replicate rate *r*: the average communication form map tasks to reduce tasks per input

Hive

- MapReduce code is typically written in Java, other languages can use Hadoop Streaming interface; only a few developers write good MapReduce code, but many other people need to analyze data and a higher-level abstraction on top of MapReduce can provide the ability to query the data without needing to know MapReduce intimately
- Hive (http://hive.apache.org/) was originally developed at Facebook, and is now a top-level Apache Software Foundation project; it provides a very SQL-like (or declarative) language, and it generates MapReduce jobs that run on the Hadoop cluster
  - provides an SQL dialect for querying data stored in HDFS, Amazon S3, and databases (e.g., HBase, Cassandra)
  - makes it easier for developers to port SQL-based data warehouse applications to Hadoop
- Hive Query Language (HiveQL or HQL)
  - Hive translate queries to MapReduce jobs
  - Hive is not a full database and does not conform to the ANSI SQL standard, it does not provide record-level update, insert, delete, or transactions, and you can do is generating new tables from queries or output query results to files
  - Hive queries have high latency (due to Hadoop is a batch-oriented system), cannot provide features required for OLTP (OnLine Transaction Processing) but closer to being an OLAP (OnLine Analytic Processing) tool
    - Use NoSQL for OLTP with large-scale data (e.g., HBase, Cassandra, DynamoDB, etc. on Hadoop, Amazon's Elastic MapReduce EMR, Elastic Computer Cloud EC2, etc.)
- the Hive data model
  - layers table definition on top of data in HDFS directories/files: tables (typed columns, JSON-like data, e.g, arrays, struct, map, etc.), partitions and buckets (hashed for sampling, join optimization)
  - primitive types: INT, TINYINT, SMALLINT, BIGINT, FLOAT, DOUBLE, BOOLEN, STRING, and CDH4 added BINARY, TIMESTAMP
  - type constructors
    - ARRAY <primitive-type>: prices = [1000, 950, 920]
    - MAP <primitive-type, data-type>: bonus = ['alice': 200, 'bob': 150]
    - STRUCT <col-name: data-type>: address<zip:STRING, street:STRING, city: STRING>
- the Hive Metastore: Hive's Metastore is a database containing table definitions and other metadata
  - by default, stored locally on the client machine in a Derby database
  - if multiple people will use Hive, create a shared Metastore in database server
- Hive data physical layout
  - Hive tables are store in subdirectories in HDFS
  - actual data is store in flat files either SequenceFiles (control character-delimited text), or in arbitrary format with the use of a custom serializer/deserializer (SerDes)
- the Hive Shell

```
$ hive
hive> SHOW TABLE;
```

to create a Hive table:

```
hive> CREATE TABLE shakespeare (freq INT, word STRING) ROW FORMAT
DELIMITED FIELDS TERMINATED BY '\t' STORED AS TEXTFILE;
hive> DESCRIBE shakepeare;
```

to create a Hive external table in another directory movie:

```
hive> CREATE EXTERNAL TABLE movie (id INT, name STRING, year INT)
ROW FORMAT DELIMITED FIELD TERMINATED BY '\t' LOCATION
'/user/training/movie';
```

to load data

```
hive> LOAD DATA INPATH "shakespeare_freq" INTO TABLE shakespeare;
```

data can also be loaded using Hadoop's filesystem commands:

```
$ hadoop fs –put shakespeare_freq /user/hive/warehouse/shakespeare
```

Sqoop's -–hive-import option can import data from database into Hive tables by generating the Hive CREATE TABLE statement based on the table definition in RDBMS

```
hive> SELECT * FROM shakespeare LIMIT 10;
hive> SELECT * FROM shakespear WHERE freq > 100 ORDER BY freq ASC
LIMIT 10;
hive> SELECT s.word, s.freq, k.freq FROM Shakespeare s JOIN kjv k
ON (s.word = k.word) WHERE s.freq >= 5;
```

or store the output results by creating a new table then write into it:

```
hive> INSERT OVERWRITE TABLE new_table SELECT s.word, s.freq,
k.freq FROM Shakespeare s JOIN kjv k ON (s.word = k.word) WHERE
s.freq >= 5;
```

Hive support manipulation of data via user-defined functions (UDFs) written in Java or in any language via the TRANSFORM operator

```
hive> INSERT OVERWRITE TABLE u_data_new SELECT TRANSFORM (user_id,
movie_id, rating, unixtime) USING ‘python weekday_mapper.py’ AS
(user_id, movie_id, rating, weekday) FROM u_data;
hive> QUIT;
```

- the batch Hive

```
$ hive –f file_to_execute
```

- Hive limitations
  - not all standard SQL is supported, e.g., no correlated subqueries
  - no support for UPDATE and DELETE
  - no support for INSERTing single rows
- word count in Hive

```
CREATE TABLE docs (line STRING);
LOAD DATA INPATH 'docs' OVERWRITE INTO TABLE docs;
CREATE TABLE word_counts AS
SELECT word, count(1) AS count FROM
(SELECT explode(split(line, '\s')) AS word FROM docs) w
GROUP BY word
ORDER BY word;
```

- Impala
  - an open-source project created by Cloudera to facilitates interactive real-time queries of data in HDFS
  - Impala does not use MapReduce, but uses a language very similar to HiveQL and produces results 5-40x faster

Pig

- Pig (http://pig.apache.org/) was originally developed at Yahoo, and is now a top-level Apache Software Foundation project; Pig translates a high-level PigLatin script run on client machine (without shared metadata) into MapReduce jobs and coordinates their execution
  - MapReduce does not directly support complex *N*-step dataflows, lacks explicit support for combined processing of multiple data set (e.g., joining/grouping datasets) and frequently-needed data manipulation primitives (e.g., filtering, aggregation, top-*k* threshholding)
  - Pig provides a very simple high-level dataflow or data description language (in the spirit of SQL) to encode explicit dataflow graphs with built-in relational-style operations, and gives sufficient control back to the programmer
- Pig features: joining datasets, grouping data, referring to elements by position instead of names, loading non-delimited data using a custom SerDe, creation of user-defined functions in Java, etc.
- Pig compilation and execution stages
  - parser: syntactical analysis, type checking, schema inference; output canonical logical plan/operator arranged in DAG
  - logical optimizer: e.g., projection pushdown
  - MapReduce compiler: convert logical plan into MapReduce jobs
  - MapReduce optimizer: e.g., combiner performs early partial aggregation
  - Hadoop job manager: topological sort the jobs and submitted to Hadoop, monitor the execution status
- Pig Latin features:

```
urls= LOAD ‘dataset’ AS (url, category, pagerank);
groups = GROUP urls BY category;
bigGroups = FILTER groups BY COUNT(urls)>1000000;
result = FOREACH bigGroups GENERATE group, top10(urls);
STORE result INTO ‘myOutput’;
```

  - a step-by-step dataflow language where computation steps are chained together through the use of variables
  - the use of high-level transformations (e.g., GROUP, FILTER)
  - the ability to specify schemas as port of issuing a program
  - the use of user-defined functions (e.g., top10 above)
- data type
  - scalar type: int, long, double, chararray (string, sorted by lexicographical ordering), bytearray (for unknown data type and lazy conversion of types, sorted by byte radix ordering)
  - complex types: map (and its dereference operator #), tuple, and bag (collection of tuples)
  - storage function: to delimit data values and tuples when loading/storing data from/to a file (default: ASCII, binStorage for bytearray, user-defined storage function), and use { } to encode nested complex types
  - declaring type explicitly as part of the AS clause during the LOAD
  - referring type implicitly from operators (e.g., #) or known return type from functions
- modes of user interaction
  - interactive mode through an interactive shell Grunt

```
$ pig [-help] [-h] [-version] [-i] [-execute] [-e]
grunt> emps = LOAD ‘people’ AS (id, name, salary);
grunt> rich = FILTER emps BY salary > 100000;
grunt> srtd = ORDER rich BY salary DESC;
grunt> STORE srtd INTO ‘rich_people’;
```

to display on screen for debugging (not use in cluster)

```
grunt> DUMP srtd;
grunt> DESCRIBE bag_name;
grunt> data1 = LOAD ‘data1’ AS (col1, col2, col3, col4);
grunt> data2 = LOAD ‘data2’ AS (colA, colB, colC);
grunt> jnd = JOIN data1 BY col3, data2 by cola;
grunt> STORE jnd INTO ‘outfile’;
grunt> grpd = GROUP bag1 BY elementX;
grunt> justnames = FOREACH emps GENERATE name;
grunt> summedUP = FOREACH grpd GENERATE group, COUNT(bag1) AS
elementCount;
grunt> QUIT;
```

- batch mode

```
$ pig script.pig
```

- embedded mode with a Java library, which allows Pig Latin commands to be submitted via method invocations from a Java program

- word count in Pig

```
raw = LOAD 'wordcount/input' USING PigStorage as (token:chararray);
raw_grp = GROUP raw BY token;
rst = FOREACH raw_grp GENERATE group, COUNT(raw) as count;
DUMP rst;
```

- choosing between Pig and Hive
  - those with an SQL background choose Hive, those without choose Pig
  - Pig deals better with less-structured data, and Hive is used to query that structured data, so some organizations choose to use both

Integrate Hadoop into the Enterprise Workflow using Sqoop & Flume

- Hadoop can complement enterprise workflow to handle huge query size and make ad-hoc full or partial dataset queries possible, but cannot serve interactive queries (Impala?) and less powerful updates (no transactions, no modification of existing records)
- HDFS can crunch sequential data faster, and offload traditional file server for interactive data access
- don't read from a RDBMS into Mapper directly (as it leads to a distributed denial of service attack DDoS), instead, import the data (by default, a comma-delimited text file) into HDFS beforehand using Sqoop
- Sqoop is an open source tool originally written at Cloudera and now a top-level Apache Software Foundation project;
- Sqoop can input just one table, all tables in a database, or just a portion of a table (by WHERE clause) using a JDBC interface (for any JDBC-compatible database), or use a free custom connector (using a system's native protocols to access data for faster performance), e.g., Netezza, Teradata, Quest Oracle Database
- Sqoop syntax and examples:

```
$ sqoop [import|import-all-tables|list-tables] [--connect|--
username|--password]
$ sqoop export [options]    # send HDFS data to a RDBMS
$ sqoop help [command]
$ sqoop import --username fred --password derf --connect
jdbc:mysql://database.example.com/personnel --table employee --
where “id > 1000”
$ sqoop list-databases --connect jdbc:mysql://localhost --username
training --password  training
$ sqoop list-tables --connect jdbc:mysql://localhost/movielens --
username training --password training
$ sqoop import --connect jdbc:mysql://localhost/movielens --table
movie --fields-terminated-by ‘\t’ --username training --password
training
$ hadoop fs –ls movie
$ hadoop fs –tail movie/part-m-00000
```

- Flume is a distributed, reliable, available open source (initially developed by Cloudera) service (with goals of reliability, scalability, manageability, and extensibility) for efficiently moving large amount of data as it is produced with optionally process incoming data (transformation, suppressions, metadata enrichment), and parallel writes to multiple HDFS file formats (text, SequenceFile, JSON, Avro, others) by agents
- Flume's reliability
  - each Flume agent has a source, a sink, and a channel or queue between the source and the sink (can be in-memory only or durable file channel – will not lost data even if power is lost)
  - data transfer between agents and channels is transactional, and a failed data transfer will make the agent rolls back and retries
  - can configure multiple agents with the same task for redundancy
- Flume's horizontal scalability
  - Increase system performance linearly by adding more agents
- Flume's manageability
  - The ability to control data flows, monitor nodes, modify the settings, and control outputs of a large system
  - configuration can be loaded from a properties file (pushed out to each node by scp, Puppet, Chef, etc.) on the fly
- Flume's extensibility
  - The ability to add new functionality to a system by write their own sources and sinks
- access HDFS from legacy systems with FuseDFS and HttpFS
  - FuseDFS is a user space file system, and allows to mount HDFS as a regular filesystem
  - HttpFS provides an HTTP/HTTPS REST interface to read/write from/to HDFS either from within a program or via command-line tools (curl or wget)

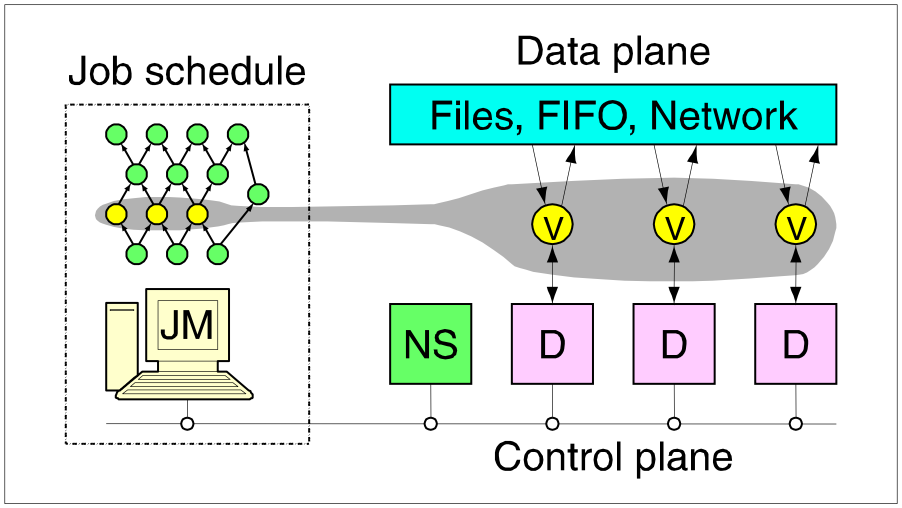
    ```
    Curl http://httpfs-host:14000/webhdfs/v1/user/foo/README.txt
    ```

MapReduce Variations and Extensions

- two-step workflow using acyclic flow graph and minimize communication
- recursive workflow for iterated application of fixed point of matrix-vector multiplication

- Google Pregel and Apache open-source equivalent Giraph: views its data as a graph and use checkpoint

Dryad



- the Microsoft Dryad system (discontinued active development in 2011) is a relative lower-level programming model for execution of data parallel application, allows the developer fine control over the communication graph and the subroutines (in C++) that live at its vertices, and gets better performance
- the Dryad system organization: job manager (JM), name server (NS), daemon (D), and running vertices (V)
- channels can be implemented by files (the default), TCP pipes (need to run at the same time), or shared memory, applications can supply their own serialization and deserialization routines, and the items transported via the channels can be either newline-terminated text strings or tuples of base types
- graph description language: operations with "acyclic" properties (i.e., scheduling deadlock is impossible), and implemented in C++
  - basic graph object: $G = \langle VG, EG, IG, OG \rangle$
  - replicate operator ^: $C = G^k = \langle VG1 \oplus \ldots \oplus VGk, EG1 \cup \ldots \cup EGk, IG1 \cup \ldots \cup IGk, OG1 \cup \ldots \cup OGk \rangle$, where $\oplus$ denotes sequence concatenation
  - concatenation composition operator °: if *VA* and *VB* are disjoint at run time and a new directed edge *Enew* connects *OA* to *IB*, then $C = A \circ B = \langle VA \oplus VB, EA \cup EB \cup Enew, IA, OB \rangle$
  - pointwise composition operator >=:
    - if $|OA| > |IB|$, a single outgoing edge is created from each of *A*'s outputs, assigned in round-robin to *B*'s inputs
    - if $|IB| > |OA|$, a single incoming edge is created to each of *B*'s inputs, assigned in round-robin from *A*'s outputs
  - bipartite composition operator >>: *A* >> *B* forms the complete bipartite graph between *OA* and *IB*
  - merge operator ||: $C = A \,||\, B = \langle VA \oplus^* VB, EA \cup EB, IA \cup^* IB, OA \cup^* OB \rangle$, where * means duplicates removed
- run-time graph refinements - partial aggregation operation: having grouped the inputs into *k* sets, the optimizer replicates the downstream vertex *k* times to allow all of the sets to be processed in parallel
- resource allocation, distribution, and scheduling: topological sorting
- fault-tolerant policy: all vertex programs are deterministic
  - report chain: vertex, daemon, job manager (JM)
  - heartbeat
  - schedule duplicate executions if vertices run too slowly

Low-Latency Cloud

- memcached: provide a general-purpose key-value store entirely in DRAM and its widely used to offload back-end database systems
  - as of 2009, approximately 25% of all online data for Facebook was kept in main memory on memcached servers, providing a hit rate of 96.5%
  - memcached makes no durability guarantees
- RAMCloud
  - Jim Gray's 5-minute rule: pages referenced every five minutes should be memory resident
  - Jim Gray's 5-byte rule: spend five bytes of main memory to save one instruction per second
  - FlashCloud replaces disks with flash memory
    - cheaper and consume less energy than RAMCloud, but write latency is much higher
    - for high query rate and small data set sizes, DRAM is cheapest, for low query rates and large data sets, disk is cheapest, and in the middle ground, flash is cheapest
    - the cost of flash is limited by cost/query/sec, the cost of DRAM is limited by cost/query/bit; cost/query/bit is improving much more quickly than cost/query/sec
  - RAMCloud replaces disks with DRAM, stores data in the main memory and uses thousands of servers to create a large-scale storage system, and disks are only used as backup
  - 100x-1,000x lower latency than disk-based system and 100x-1,000 greater throughput, manipulate data 100x-1,000x more intensively

    - flat storage model: all objects can be access quickly and not affected by data placement, can support random access (MapReduce for sequential access)
    - durability and availability by replication and buffered logging
    - high cost and high energy use per bit, not cost effective for large-scale media storage yet

Berkeley Data Analytics Stack (BDAS)

- support batch, streaming, and interactive computations in a unified analytics stack instead of 3 separated stacks, easy to develop sophisticated algorithms (e.g., graph, machine learning)
- Hadoop stack
    - data processing layer: Hive, Pig, MapReduce, HBase, Storm, …
    - resource management layer: Hadoop Yarn
    - storage layer: HDFS, S3, …
- BDAS stack
    - data processing layer: Spark Streaming, Shark SQL, BlinkDB, GraphX, MLlib, MLBase, …
    - resource management layer: Spark, Apache Mesos
    - storage layer: Tachyon, HDFS, S3, …
- Apache Mesos enables multiple frameworks to share same cluster resources, allow BDAS & Hadoop fit together seamlessly
    - Mesosphere: startup to commercialize Mesos
- Apache Spark: a distributed engine with fault-tolerant, efficient in-memory storage (100x faster than Hadoop MapReduce), and powerful programming models and APIs (Scala, Python, Java with 2-5x less code)
- Tachyon: in-memory fault-tolerant storage system, allow multiple frameworks to share in-memory data

Machine Learning and Mahout

- machine learning is having programs work out what to do instead of telling programs exactly what to do
- the three Cs for machine learning: collaborative filtering (recommendations), clustering, and classification; the first two are unsupervised learning, and the last one is supervised learning
- Mahout is a machine learning data-agnostic library written in Java and some pre-built script to analyze data, algorithms include in Mahout include:
    - Recommendation: Pearson correlation, log likelihood, spearman correlation, Tanimoto coefficient, singular value decomposition (SVD), linear interpolation, cluster-based recommenders
    - clustering: k-means clustering, Canopy clustering, fuzzy k-means, Latent Dirichlet analysis (LDA)
    - classification: stochastic gradient descent (SGD), support vector machine (SVM), Naïve Bayes, complementary naïve Bayes, random forests
- to run Mahout's item-based recommender:
```
$ mahout recommenditembased --input movierating --output recs --
userFile users --similarityClassname SIMILARITY_LOGLIKELIHOOD
$ hadoop fs –cat recs/part-r-00000
```

Semantic Web and Resource Description Format (RDF)

- RDF uses subject-property-object (SPO) triples to model the entity-relationship graph (a pair of entities connected by a named relationship or an entity connected to the values of a named attribute, the typed nodes correspond to entities and the edges to relationships)
- 3 requirements:
    - a high diversity of property names for schema-less data, RDF supports not only SQL select-project-join queries, but also allow wildcard for property names
    - RDF not only does Boolean-match evaluation, but also ranks of search results (for too-many-result situations)
    - RDF support text search with keywords and phrases
- SPARQL
    - each triple pattern has one or two of the SPO components replace by variable names (with prefixed '?'), the dots ('.') between the triple patterns denote logical conjunctions, the triple patterns can augmented by keyword conditions (to match the specified keywords enclosed in the '{ }' pairs (semantic grouping to eliminate inappropriate matches), and using the same variable in different triple patterns denotes join conditions
    - some triples are more important than others, and use statistical weights (the witnesses of a given triple) to construct a non-uniform distribution
    - For example:
```
Select ?x, ?m Where { ?x has type Musician {classical, composer}
. ?x composedSoundtrack ?m . ?m hasType Movie {gunfight, battle}
}
```