



Hadoop Components

Venkatesh. S.

**Cloud Computing and Data Infrastructure
Yahoo! Bangalore (India)**

YAHOO!



Agenda

- **Hadoop Distributed File System**
 - Architecture
- **Map-Reduce**
 - Architecture
- **Scheduler/Resource Manager**
- **Hadoop Ecosystem**





Hadoop Distributed File System (HDFS)

- **Based on Google's GFS**
- **Redundant storage of massive amounts of data on cheap and unreliable computers**
- **Why not use an existing file system?**
 - Different workload and design priorities
 - Handles much bigger dataset sizes than other file systems
- **Commodity Hardware + Horizontal scaling**
 - Can add servers any time
 - Supports use of unRAIDed SATA drives (JBOD)
 - Use replication across servers to deal with unreliable storage/servers



Assumptions

- **High component failure rates**
 - Inexpensive commodity components fail all the time
- **“Modest” number of HUGE files**
 - Just a few million
 - Each is 100MB or larger; multi-GB files typical
- **Files are write-once, mostly appended to**
- **Large streaming reads - Slightly Restricted file semantics**
 - Focus is mostly sequential access
 - Single writers, no file locking features



HDFS Architecture

- **Single namespace for entire cluster**
 - Managed by a single *namenode* (master).
 - Namenode maintains metadata for files
- **Files stored as chunks**
 - Much larger size than most file systems (default is 128MB)
- **Reliability through Replication**
 - Each chunk replicated across several *data nodes* (3+)
- **Single master (NameNode) coordinates access, metadata**
 - Simple centralized management
 - Metadata-data separation - simple design
- **No data caching, streaming reads**
 - Little benefit due to large data sets, streaming reads



Metadata

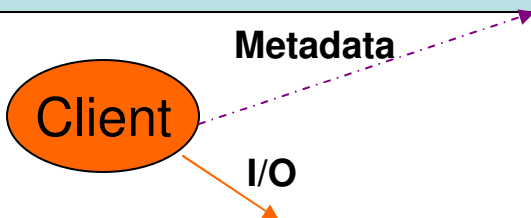
- **Single NameNode stores all metadata**
 - Filenames, locations on DataNodes of each file
- **Maintained entirely in RAM for fast lookup**
- **DataNodes store opaque file contents in “block” objects on underlying local filesystem**
- **Client talks to both namenode and datanodes**
 - Data is not sent through the namenode.
 - Throughput of file system scales nearly linearly with the number of nodes



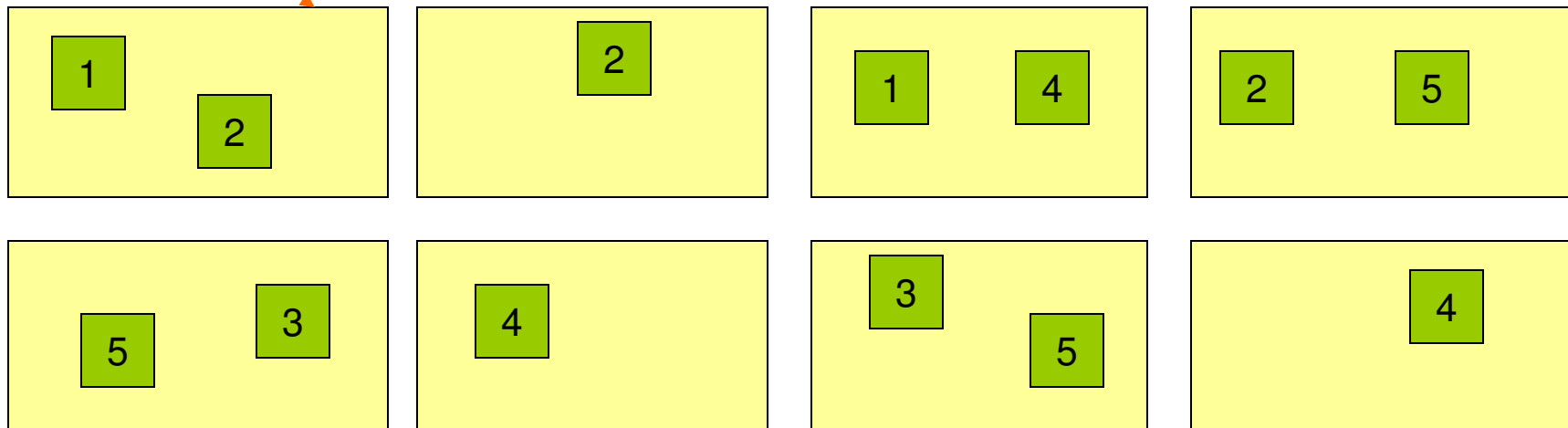
HDFS architecture

Namenode (the master)

name:/users/joeYahoo/myFile - copies:2, blocks:{1,3}
name:/users/bobYahoo/someData.zip, copies:3, blocks:{2,4,5}



Datanodes (the slaves)





Handling failures

- **Namenode failure**
 - A single point of failure.
 - Keeps track of operations through a transaction log.
 - Memory image stored as a file.
 - At startup, reads memory image, applies transaction log, rewrites memory image, starts new transaction log
- **Datanode failures**
 - Namenode detects Datanode failures. Re-replicates blocks.
 - When Datanode comes up, it contacts Namenode
- **Checksums (CRC32) to validate data**
 - File Creation
 - Client computes checksum per 512 byte
 - DataNode stores the checksum
 - File access
 - Client retrieves the data and checksum from DataNode
 - If Validation fails, Client tries other replicas



Block placement

- **Default is 3 replicas, but configurable**
- **Blocks are placed:**
 - One on local node
 - One on same rack
 - Others placed randomly across racks
- **Clients read from closest replica**
- **If the replication for a block drops below target, it is automatically re-replicated.**
- **HDFS provides api to specify block size when you create a file - multiple files in HDFS can use different block sizes**



Records

- **HDFS does not provide record-oriented API and therefore is not aware of records and boundaries between them.**
- **It is the responsibility of the InputSplit's RecordReader to start and end at a record boundary.**
 - For SequenceFile's every 2k bytes has a 20 bytes sync mark between the records.
 - Text files are handled similarly, using newlines instead of sync marks.
- **HDFS supports exclusive writes only.**
 - When the first client contacts the name-node to open the file for writing, the name-node grants a lease to the client to create this file. The second clients request will be rejected.



Distributed processing using Map-Reduce

- **Functional programming meets distributed computing**
- **A batch data processing system**
 - Factors out many reliability concerns from application logic
- **Provides....**
 - Automatic parallelization & distribution
 - Fault-tolerant
 - status and monitoring tools
 - Clean abstraction for programmers
- **Fun to use: focus on problem, let library deal w/ messy details**
- **Several interfaces:**
 - Java, C++, shell scripts



Map/Reduce Programming Model

- **Map/Reduce is a programming model for efficient distributed computing**
 - Borrows from functional programming
 - Users implement interface of two functions:
 - `map (in_key, in_value) -> (out_key, intermediate_value) list`
 - `reduce (out_key, intermediate_value list) -> out_value list`
- **It works like a Unix pipeline:**
 - `cat input | grep | sort | uniq -c | cat > output`
 - **Input | Map | Shuffle & Sort | Reduce | Output**



map

- Records from the data source (lines out of files, rows of a database, etc) are fed into the map function as key*value pairs: e.g., (filename, line).
- map() produces one or more intermediate values along with an output key from the input.
- map (in_key, in_value) -> (out_key, intermediate_value) list
- Example: Upper-case Mapper
letmap(k, v) = emit(k.toUpper(), v.toUpper())
("foo", "bar") ("FOO", "BAR")
("Foo", "other") ("FOO", "OTHER")
("key2", "data") ("KEY2", "DATA")



reduce

- After the map phase is over, all the intermediate values for a given output key are combined together into a list
- `reduce()` combines those intermediate values into one or more final values for that same output key
- (in practice, usually only one final value per key)
- `reduce (out_key, intermediate_value list) -> out_value list`
- **Example: Sum Reducer**

```
let reduce(k, vals) =
```

```
  sum = 0
```

```
  foreach int v in vals:
```

```
    sum += v
```

```
  emit(k, sum)
```

```
  ("A", [42, 100, 312]) ("A", 454)
```

```
  ("B", [12, 6, -2]) ("B", 16)
```





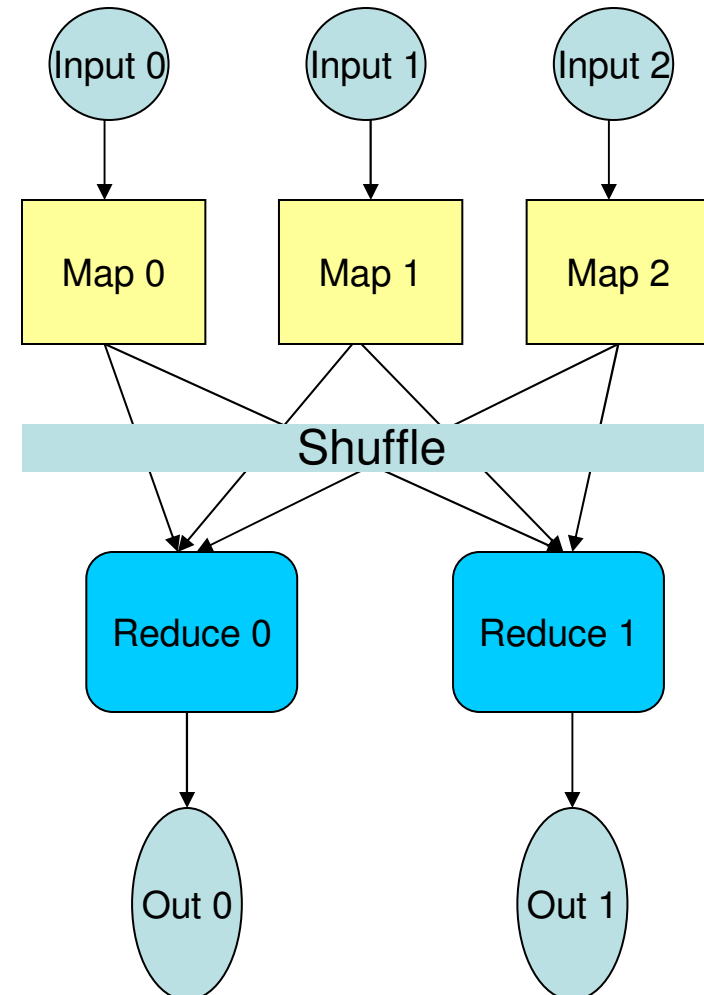
Parallelism

- **map() functions run in parallel, creating different intermediate values from different input data sets**
- **reduce() functions also run in parallel, each working on a different output key**
- **All values are processed independently**
- **Bottleneck: reduce phase can't start until map phase is completely finished.**



Map/Reduce

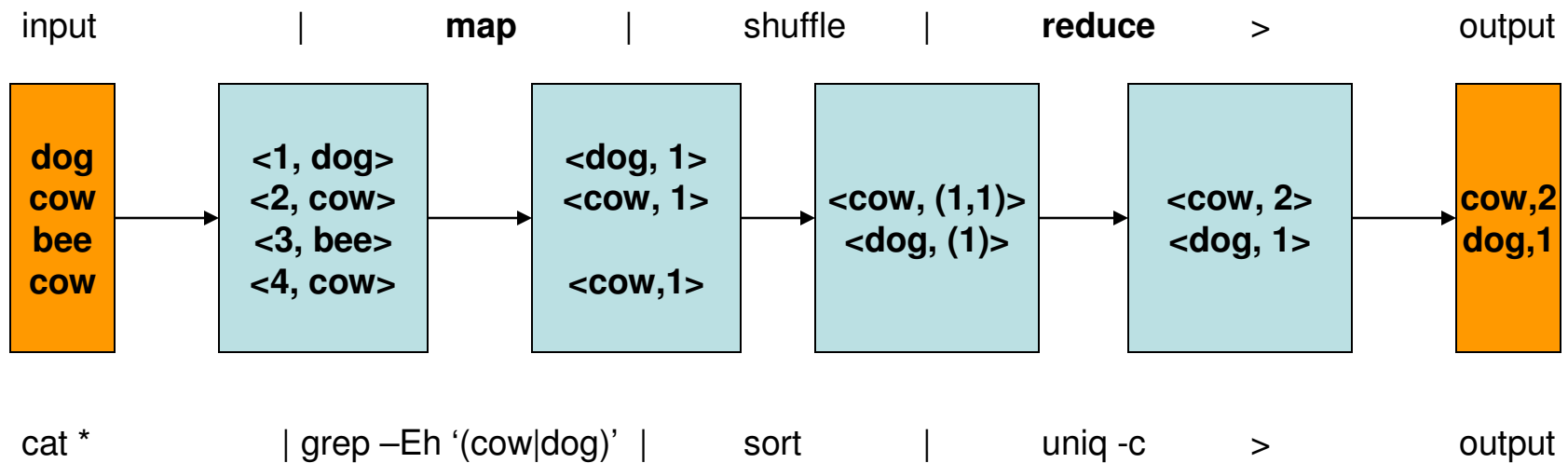
- **Application writer specifies**
 - A pair of functions called *Map* and *Reduce* and a set of input files
- **Workflow**
 - *Input* phase generates a number of *FileSplits* from input files (one per Map task)
 - The *Map* phase executes a user function to transform input kv-pairs into a new set of kv-pairs
 - The framework sorts & *Shuffles* the kv-pairs to output nodes
 - The *Reduce* phase combines all kv-pairs with the same key into new kv-pairs
 - The output phase writes the resulting pairs to files
- **All phases are distributed with many tasks doing the work**
 - Framework handles scheduling of tasks on cluster
 - Framework handles recovery when a node fails





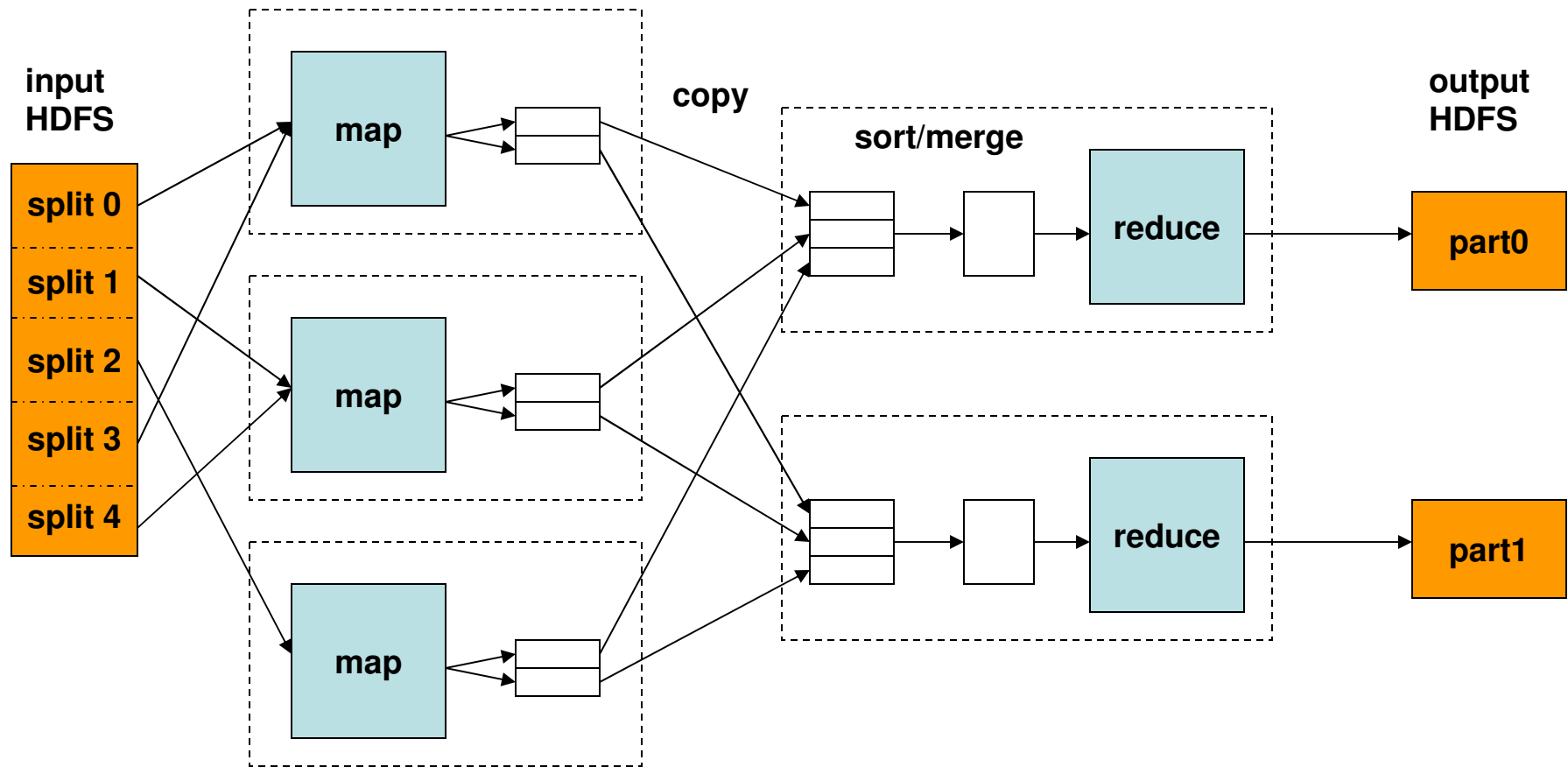
Logical flow

- Ex: grep for # of occurrences of 'cow' and 'dog'





Physical flow



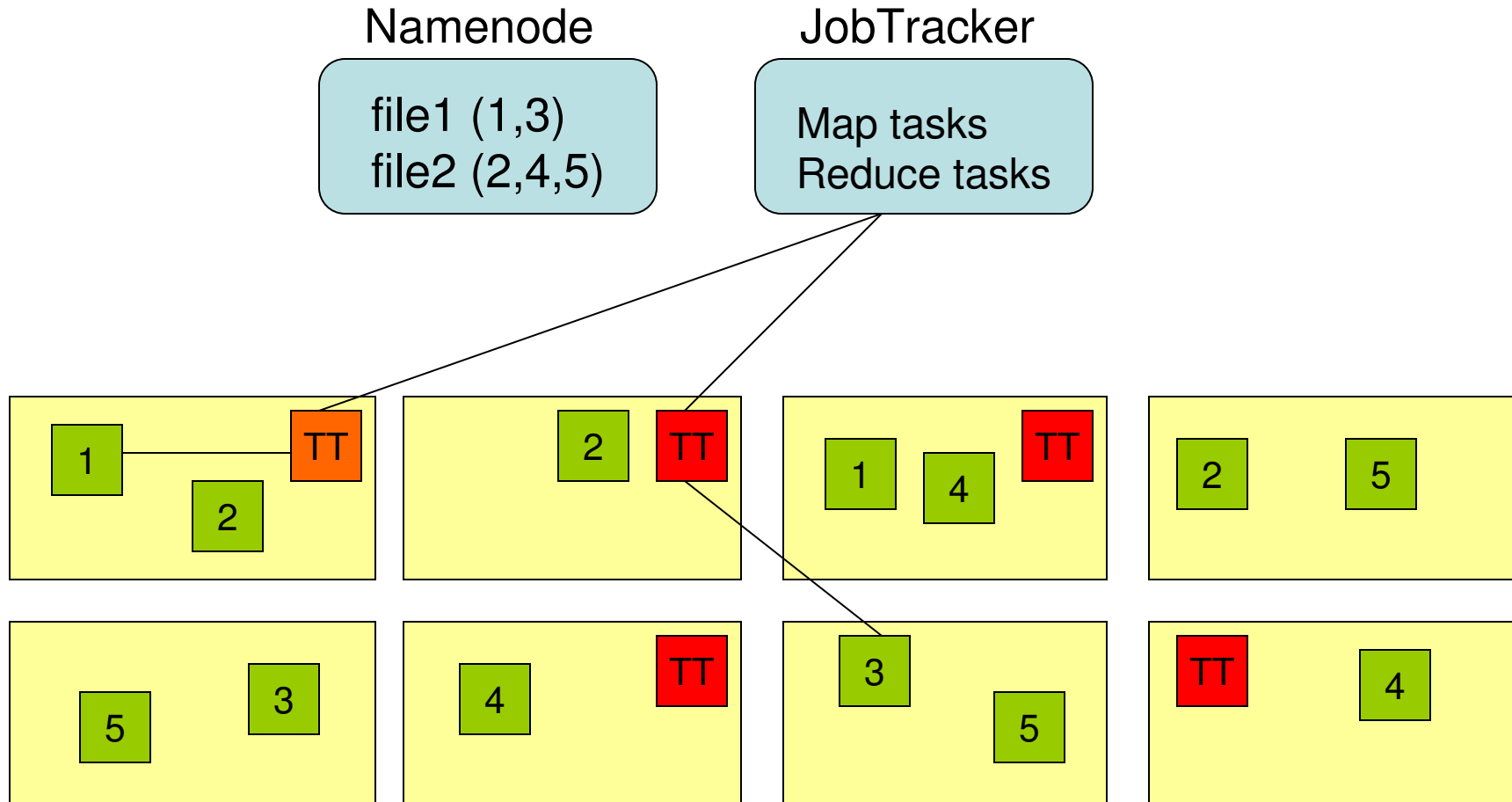


Map-Reduce architecture

- **Master-Slave architecture**
- **Master: JobTracker**
 - Accepts MR jobs submitted by users
 - Assigns Map and Reduce tasks to TaskTrackers (slaves)
 - Monitors task and TaskTracker status, re-executes tasks upon failure
- **Slaves: TaskTrackers**
 - Run Map and Reduce tasks upon instruction from the Jobtracker
 - Manage storage and transmission of intermediate output
- **Greatly simplifies large-scale computations**



Map-Reduce + HDFS





How does Hadoop Scale

- **Distribute state**
 - Namenode keeps meta info, datanodes keep data
- **Keep as much as you can in memory**
 - Namenode's meta info
 - JobTracker only expands jobs as needed
- **Minimize communication**
 - TTs batch task events
 - TTs ask for tasks when free
 - Piggyback stuff on heartbeat calls
- **Good data structures, algorithms**
 - How does JT quickly find which task has data on the same host/rack?
 - Lazy evaluation: batch state changes
- **Keep design simple**



Communication

- **RPC**
- **Threads on a client share socket connection**
- **Server is multi-threaded**
- **Use our own optimized de/serialization format**
 - Will use Avro – a new hadoop sub project



Resource Manager

- **Grid is made up of queues of jobs. Users submit jobs to queues.**
- **Divide cluster resources to queues. Each queue has ‘guaranteed capacity’.**
 - Free capacity can be redistributed
 - Redistributed capacity can be reclaimed
- **To enable fair share, we want to limit the number of resources in a queue that can be used by someone. How?**
 - Limit has upper bound and lower bound
 - Upper bound depends on how many users have submitted jobs.
 - Lower bound is pre-configured.
- **Queues can support priorities for jobs**



Hadoop Ecosystem





Hadoop Software Ecosystem

- **Pig – Yahoo!**
 - Parallel Programming Language and Runtime
- **Zookeeper – Yahoo!**
 - High-Availability Directory and Configuration Service
- **Hbase – Powerset.com**
 - TableStore layered on HDFS
- **JACL – IBM**
 - JSON / SQL inspired programming Hadoop language
- **Mahout – Individual apache members**
 - Machine learning libraries for Hadoop
- **Tashi – Intel, CMU**
 - Virtual machine provisioning service (soon) □
- **Hive – Facebook.com**
 - Data warehousing framework on Hadoop





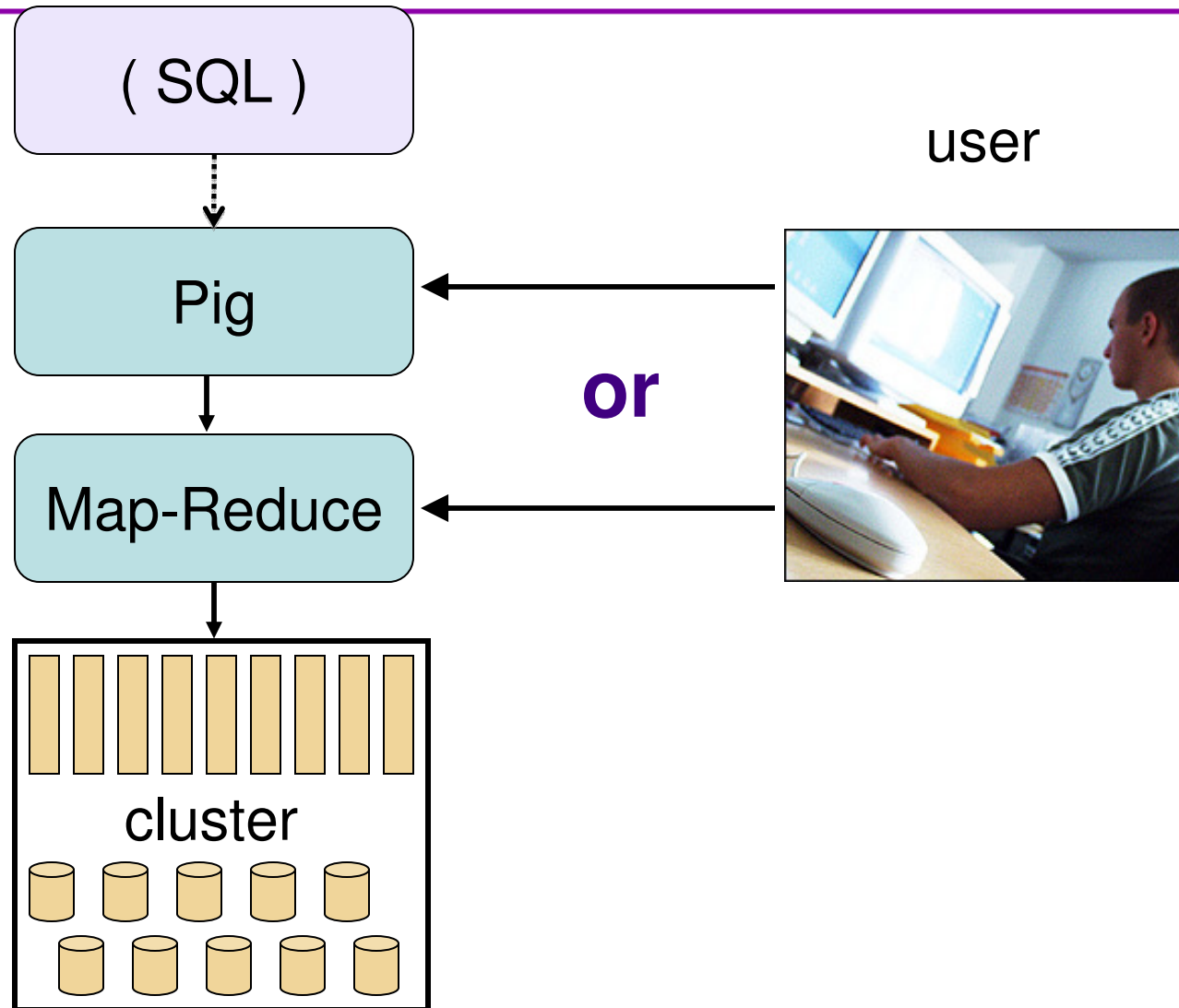
PIG: Parallel Programming Language & Runtime

- **Data-parallel language (“Pig Latin”)**
 - Goals:
 - Relational data manipulation primitives
 - Imperative programming style
 - Users plug in code to customize processing
- **Data-parallel runtime software (“Pig”)**
 - Focus on cross-program optimizations:
 - Combined execution of related programs
 - Reuse of derived data





Map-Reduce as Backend





What's ahead

- **Improved scalability**
 - E.g. 10s K nodes
 - Federated applications across clusters and data centers
- **Improved performance**
- **Enhanced features**
- **Further extensions**
 - on-line service grid?
- **More applications, from many fields!**

The Hadoop eco-system is growing and has the potential to have a lot of impact across the internet industry, and many others!



References

- Website: <http://hadoop.apache.org/core>
- Wiki: (for developers): <http://wiki.apache.org/hadoop/>
- Mailing lists:
 - core-dev@hadoop.apache.org
 - core-user@hadoop.apache.org
- IRC: #hadoop on irc.freenode.org
- Yahoo's Hadoop blog:
<http://developer.yahoo.com/blogs/hadoop/>



Thank You!

