```
MapReduce
=========

What problem is the paper addressing
  Distributed programming is hard
  A lot of common work in many distributed big-data applications

Why are distributed systems hard
  Failure is regular occurrence with thousands of machines
  Tail latency (example of processor with cache disabled)
  Handling dependencies between computations

What is the basic abstraction of map reduce?
  User provides two functions ([x] = list of type x, Haskell-style):
    map :: (k1, v1) -> [(k2, v2)]
    reduce :: (k2, Stream v2) -> [v2]  -- typically outputs 0 or 1 v2
                                       -- guaranteed v2 input sorted
  User provides giant input files
  What you get out: [(k2, v2)]
  User provides some parameters:
    M -- number of pieces into which to split up input
    R -- number of pieces into which output will be broken
  Optional features:
    combiner :: [v2] -> [v2]   -- computation to run on mapper machines
    partition :: k2 -> Int     -- function to partition k2 into R buckets
    input  -- produces stream of (k1,v1) values
    output -- consumes stream of (k2,v2) values
    counters -- used to check progress, sanity check computation

What are some applications that fit this model?
* Grep.  What are the types and map/reduce functions for grep?
    split input files into big pieces
    k1 doesn't matter - maybe position
    v1 line of text
    k2 doesn't matter - maybe position or hash of line of text
    v2 line of text
    map (k1, v1) = v1 contains string ? [(k1 or hash(v1), v1)] : []
    reduce (k2, v2) = [v2]
* Word count - look at appendix
    split input files into big pieces
    k1 doesn't matter, v1 text
    k2 is word, v2 is 1 (int)
    map (k1, v1) = [(word, 1) | word <- words(v1)]
    reduce (word, counts) = [sum counts]
    combine (counts) = sum counts
* Count of URL access frequency
* Reverse web-link graph
* Term-Vector per host
* Inverted Index
* Distributed sort
    Need to use partitioning function
    Means you need rough idea of data distribution (MapReduce pre-pass)

What is basic architecture? (Fig. 1)
  Compile an link your program with MapReduce library, then run it
  Program forks a "master" process
  Master process pushes copies of program to 1000 machines (takes ~1 min)
  Master keeps state per map/reduce task:
    - Idle | InProgress | Completed
    - WorkerId for non-idle tasks
  Master assigns tasks to different machines
    Starts with map tasks, then reduce tasks as input available

What happens if machine fails (3.3)?
  Worker failure
```

```
    How does master detect?  Periodic ping fails
    How does master react?
      InProgress *and* Completed map tasks set to Idle (why completed?)
      InProgress reduce tasks set to Idle
      Note:  reduce output completes via atomic rename
  Master failure?  Too bad, whole computation fails.  Is this okay?
  Semantics
    Deterministic?  No problem, re-executing does nothing differently
    Otherwise?  Different reduce tasks might be from different executions
      Example?  Mappers include timestamps, or socket communication, ...

What are the possible bottlenecks of a MapReduce computation?
  Network bisection bandwidth (less today than at the time)
    Required for reading some (not all) input data (on GFS file system)
    Required for feeding data between mappers and reducers
    Required for writing output data (often multiple times for redundancy)
    Not required for intermediary map output (stored locally)
  Disk bandwidth
  Computation at in mapper and reducer processes
How does system optimize bandwidth?
  Locality optimization - place mappers based on GFS data location

What's the issue with stragglers?
  One slow machine kills performance of whole task
  Solution:  backup tasks
    Run last few tasks redundantly on multiple machines, take first finished
    So need to extend master state to include multiple WorkerIds?

How to skip bad records and why?
  Global variable tracks current record
  Workers install SIGSEGV, SIGBUS handlers
    Last gasp UDP packet on signal tells master what record caused crash
  Master can tell workers to skip bad record when reexecuting

What questions should we ask for evaluation?
  1. Does it in fact simplify programming?
  2. Is performance good?
  3. Is fault tolerance good enough?
How do experiments address these questions?

Figure 3:
  Why two humps for shuffle?
    First batch of reduce tasks take some time, then second
  Why is input rate higher than shuffle higher than output rate?
    Locality optimization for input
    Output replicated 2-way

Is this reproducible research?
  Definitely not--proprietary system, reports usage stats at google
  But high-level model has been validated by widespread usage
    Open source Hadoop project
    Follow-on more complex projects like Dryad
```