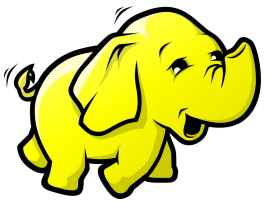
Scalable Computing with Hadoop

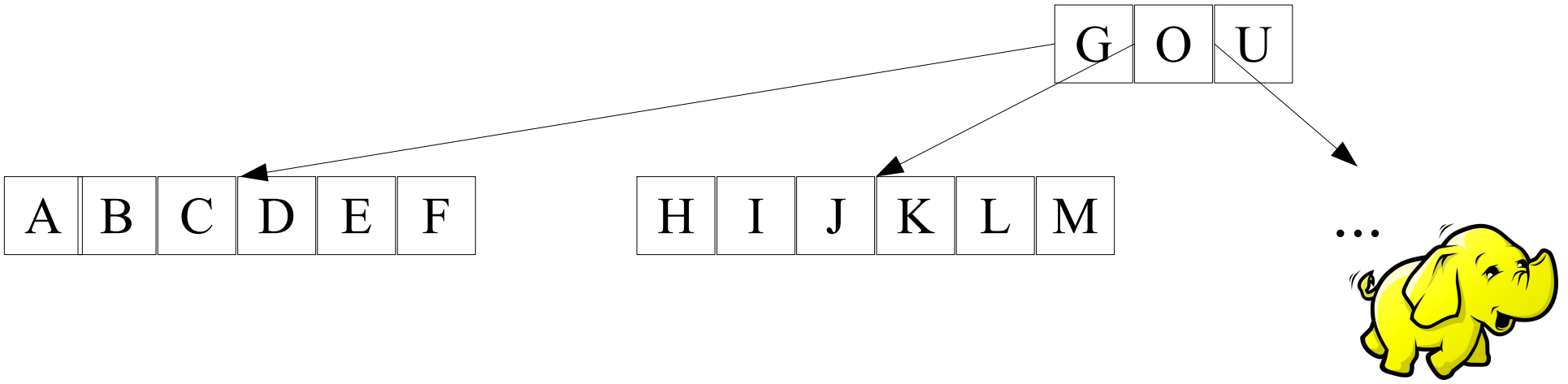
Doug Cutting
cutting@apache.org
dcutting@yahoo-inc.com

5/4/06



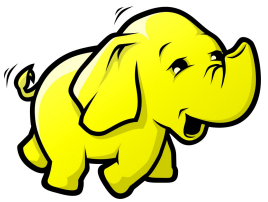
Seek versus Transfer

- B-Tree
  - requires seek per access
  - unless to recent, cached page
  - so can buffer & pre-sort accesses
  - but, w/ fragmentation, must still seek per page



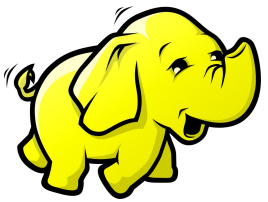
Seek versus Transfer

- update by merging
  - merge sort takes log(updates), at transfer rate
  - merging updates is linear in db size, at transfer rate
- if 10MB/s xfer, 10ms seek, 1% update of TB db
  - 100b entries, 10kb pages, 10B entries, 1B pages
  - seek per update requires 1000 days!
  - seek per page requires 100 days!
  - transfer entire db takes 1 day



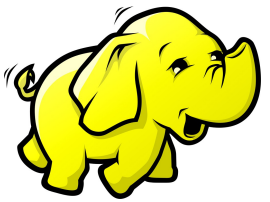
Hadoop DFS

- modelled after Google's GFS
- single *namenode*
  - maps name → <blockId>*
  - maps blockId → <host:port>$^{replication\_level}$
- many *datanodes*, one per disk generally
  - map blockId → <byte>*
  - poll namenode for replication, deletion, etc. requests
- client code talks to both

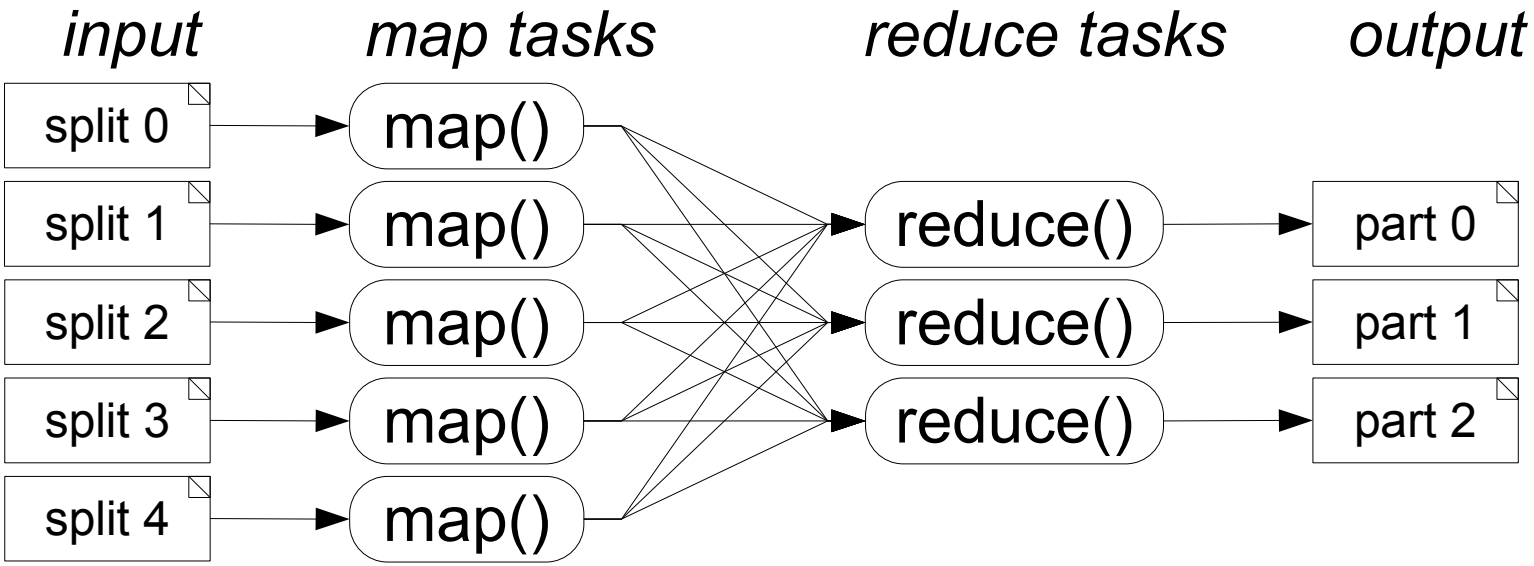


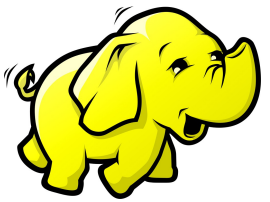
Hadoop MapReduce

- Platform for reliable, scalable computing.
- All data is sequences of <key,value> pairs.
- Programmer specifies two primary methods:
  - map(k, v) → <k', v'>*
  - reduce(k', <v'>*) → <k', v'>*
  - also partition(), compare(), & others
- All v' with same k' are reduced together, in order.
  - bonus: built-in support for sort/merge!



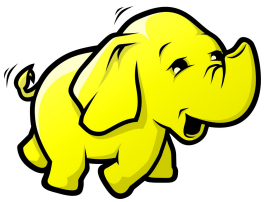
MapReduce job processing





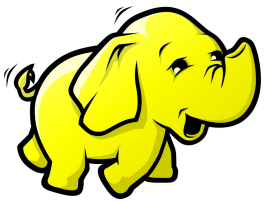
Example: RegexMapper

```
public class RegexMapper implements Mapper {
  private Pattern pattern;
  private int group;

  public void configure(JobConf job) {
    pattern = Pattern.compile(job.get("mapred.mapper.regex"));
    group = job.getInt("mapred.mapper.regex.group", 0);
  }

  public void map(WritableComparable key, Writable value,
                  OutputCollector output, Reporter reporter)
    throws IOException {
    String text = ((UTF8)value).toString();
    Matcher matcher = pattern.matcher(text);
    while (matcher.find()) {
      output.collect(new UTF8(matcher.group(group)),
                     new LongWritable(1));
    }
  }
}
```



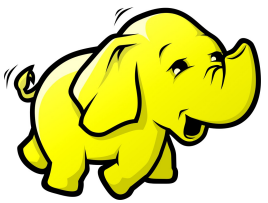
Example: LongSumReducer

```
public class LongSumReducer implements Reducer {

  public void configure(JobConf job) {}

  public void reduce(WritableComparable key, Iterator values,
                     OutputCollector output, Reporter reporter)
    throws IOException {

    long sum = 0;

    while (values.hasNext()) {
      sum += ((LongWritable)values.next()).get();
    }

    output.collect(key, new LongWritable(sum));
  }
}
```



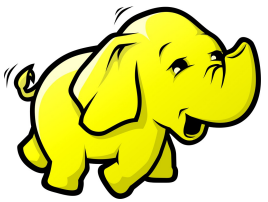
Example: main()

```
public static void main(String[] args) throws IOException {
  NutchConf defaults = NutchConf.get();
  JobConf job = new JobConf(defaults);

  job.setInputDir(new File(args[0]));

  job.setMapperClass(RegexMapper.class);
  job.set("mapred.mapper.regex", args[2]);
  job.set("mapred.mapper.regex.group", args[3]);

  job.setReducerClass(LongSumReducer.class);

  job.setOutputDir(args[1]);
  job.setOutputKeyClass(UTF8.class);
  job.setOutputValueClass(LongWritable.class);

  JobClient.runJob(job);
}
```



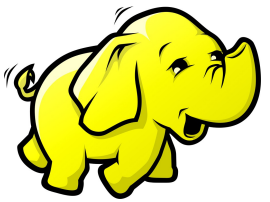
Nutch Algorithms

- *inject* urls into a crawl db, to bootstrap it.
- loop:
  - *generate* a set of urls to fetch from crawl db;
  - *fetch* a set of urls into a *segment*;
  - *parse* fetched content of a segment;
  - *update* crawl db with data parsed from a segment.
- *invert links* parsed from segments
- *index* segment text & inlink anchor text



Nutch on MapReduce & NDFS

- Nutch's major algorithms converted in 2 weeks.
- Before:
  - several were undistributed scalabilty bottlenecks
  - distributable algorithms were complex to manage
  - collections larger than 100M pages impractical
- After:
  - all are scalable, distributed, easy to operate
  - code is substantially smaller & simpler
  - should permit multi-billion page collections



Data Structure: Crawl DB

- CrawlDb is a directory of files containing:

  <URL, CrawlDatum>

- CrawlDatum:

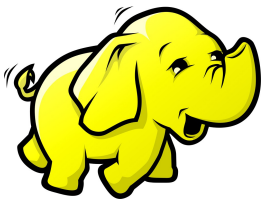
  <status, date, interval, failures, linkCount, ...>

- Status:

  {db_unfetched, db_fetched, db_gone,

  linked,

  fetch_success, fetch_fail, fetch_gone}



Algorithm: Inject

- MapReduce1: Convert input to DB format

  In: flat text file of urls

  Map(line) → <url, CrawlDatum>; status=db_unfetched

  Reduce() is identity;

  Output: directory of temporary files

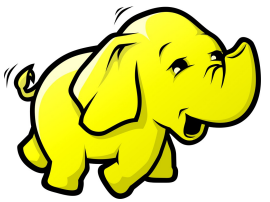
- MapReduce2: Merge into existing DB

  Input: output of Step1 and existing DB files

  Map() is identity.

  Reduce: merge CrawlDatum's into single entry

  Out: new version of DB



Algorithm: Generate

- MapReduce1: select urls due for fetch

  In: Crawl DB files

  Map() → if date≥now, invert to <CrawlDatum, url>

  Partition by value hash (!) to randomize

  Reduce:

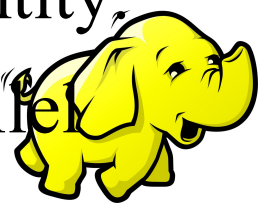
    compare() order by decreasing CrawlDatum.linkCount

    output only top-N most-linked entries

- MapReduce2: prepare for fetch

  Map() is invert; Partition() by host, Reduce() is identity.

  Out: Set of <url,CrawlDatum> files to fetch in parallel



Algorithm: Fetch

- MapReduce: fetch a set of urls

  In: <url,CrawlDatum>, partition by host, sort by hash

  Map(url,CrawlDatum) → <url, FetcherOutput>

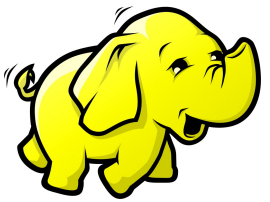
    multi-threaded, async map implementation

    calls existing Nutch protocol plugins

  FetcherOutput: <CrawlDatum, Content>

  Reduce is identity

  Out: two files: <url,CrawlDatum>, <url,Content>



Algorithm: Parse

- MapReduce: parse content

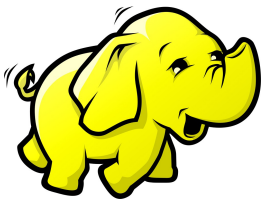
  In: <url, Content> files from Fetch

  Map(url, Content) → <url, Parse>
  calls existing Nutch parser plugins

  Reduce is identity.

  Parse: <ParseText, ParseData>

  Out: split in three: <url,ParseText>, <url,ParseData>
  and <url,CrawlDatum> for outlinks.



Algorithm: Update Crawl DB

- MapReduce: integrate fetch & parse out into db

  In: <url,CrawlDatum> existing db plus fetch & parse out

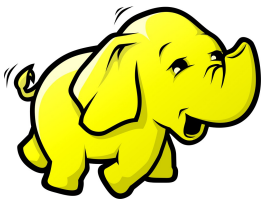
  Map() is identity

  Reduce() merges all entries into a single new entry

    overwrite previous db status w/ new from fetch

    sum count of links from parse w/ previous from db

  Out: new crawl db



Algorithm: Invert Links

- MapReduce: compute inlinks for all urls

  In: <url,ParseData>, containing page outlinks

  Map(srcUrl, ParseData> → <destUrl, Inlinks>

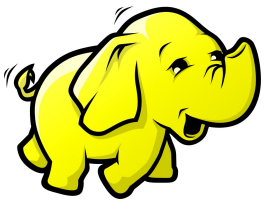
    collect a single-element Inlinks for each outlink

    limit number of outlinks per page

  Inlinks: <srcUrl, anchorText>*

  Reduce() appends inlinks

  Out: <url, Inlinks>, a complete link inversion



Algorithm: Index

- MapReduce: create Lucene indexes

  In: multiple files, values wrapped in <Class, Object>

    <url, ParseData> from parse, for title, metadata, etc.

    <url, ParseText> from parse, for text

    <url, Inlinks> from invert, for anchors

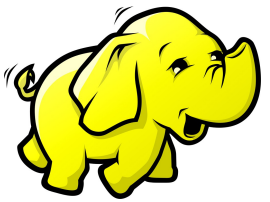
    <url, CrawlDatum> from fetch, for fetch date

  Map() is identity

  Reduce() create a Lucene Document

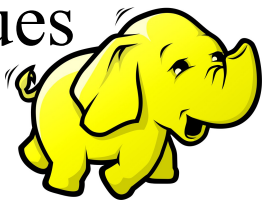
    call existing Nutch indexing plugins

  Out: build Lucene index; copy to fs at end

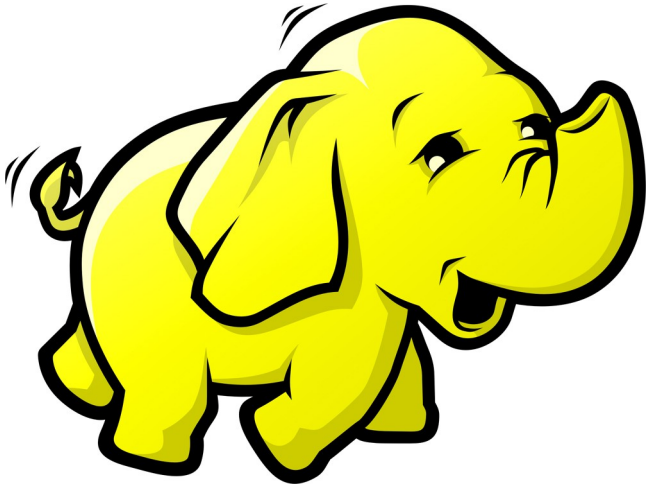


Hadoop MapReduce Extensions

- Split output to multiple files
  - saves subsequent i/o, since inputs are smaller
- Mix input value types
  - saves MapReduce passes to convert values
- Async Map
  - permits multi-threaded Fetcher
- Partition by Value
  - facilitates selecting subsets w/ maximum key values



Thanks!



http://lucene.apache.org/hadoop/