Recommendation Using Large-scale Click-through Data

靳志辉 Rick Jin　　严浩 Charlie Yan　　王刚 Gang Wang　　王益 Yi Wang

A Broad View

- A narrow view of recommendation: recommend something given users
  - e.g., Netflex, Amazon, Taobao, etc.
  - Specific solution for specific question.
- A broad view of recommendation: recommend anything given a clue
  - Search engine: recommend Web pages given query
  - Personalized search: recommend Web pages given query and user info
  - Search Ads: recommend Ads given query
  - Content Ads: recommend Ads given host page content
  - User behavior targeting Ads: recommend Ads given, say, IME history

A Systematic View

- The complete procedure:
  1. From data to information
  2. From data to knowledge
  3. From knowledge to solution
- Example 1: click model in search engine
  - Recover sessions from impression and click log
  - Train click model and ranker from sessions

    Bayesian Browsing Model from Petabyte-scale Data, Chao Liu, KDD'09
  - Rank Web pages given a query

    Smoothing clickthrough data for web search ranking, Jianfeng Gao, SIGIR'09

More On The Example

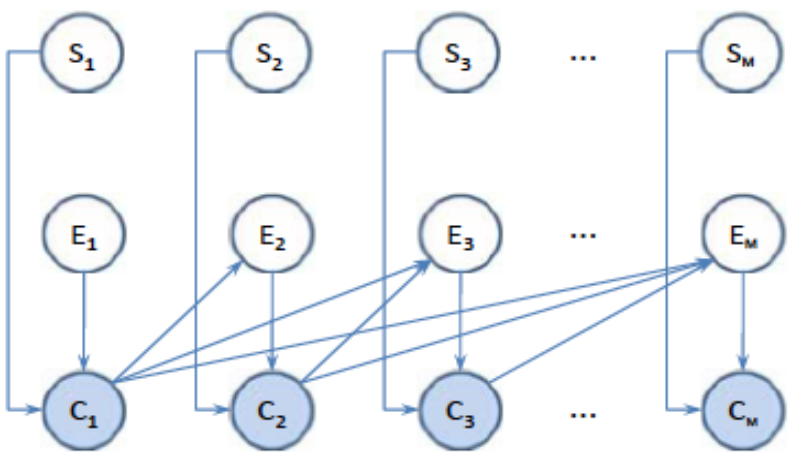
- Recover sessions from impression and click log
  - impression log entry: session_id, query, URL1, URL2, URL3, ...
  - click log entry: session_id, clicked_URL, timestamp
  - The SQL way of join:

    SELECT * from impression, click where impression.session_id = click.session_id
  - The MapReduce way of join

    ImpressionMapper::Map(key, value) { OutputWSK(key, "impr", value); }
    ClickMapper::Map(key, value) { OutputWSK(key, "clik", value); }
    JoinerReducer::Reduce(key, values) { Output(key, values[0] + values[1]); }

More On The Example

- Training of the Click Model



Algorithm 2 : `Map(I)` – Mapping a search instance I

Input: I: current search instance.
$I.qry$: returns the query,
$I.phi[i]$: gives the URL on the ith position,
$I.clk[i]$: indicates click on the ith position.
Output: $((q,u), val)$: intermediate (key, value) pairs for every position

```
01: q = I.qry; r = 0;
02: for each position i = 1, ..., M
03:     u = I.phi(i);
04:     if  I.clk[i] == 1
05:         r = i;
06:         val = 0;
08:     else
09:         d = i − r;
10:         val = r(2M − r − 1)/2 + d;
11:     end
12:     Emit((q, u), val);
13: end
```

Algorithm 3 : `Reduce((q,u), valList)`

Input: (q,u): the intermediate key
$valList$: a list of values associated with (q,u)
Output: $((q,u), \mathbf{e})$: $\mathbf{e}$ is the exponent vector for (q,u)

```
1: e = 0;
2: for each val in valList
3:     e[val] + +
4: end
5: return ((q, u), e)
```

What Is MapReduce

- MapReduce is a programming paradigm
- Programmers define only Map() and Reduce() -- Simple Programming
- The framework takes communication and synchronization -- Less Bugs
- Parallel execution of Map() and Reduce() -- High Performance
- IO and intermediate data have to be key-value pairs -- Not Flexible
- Restart dead tasks, not the whole job -- Auto Fault Recover

The Logic in MapReduce

- For each input key-value pair, invoke Map(key, value)
- Map() can invoke Output(inter-key, inter-value) 0 or more times
  - As no data/comm dependency, so Map() invocations are parallelizable
- Take inter-values with the same inter-key as a reduce-input
  - Need to sort all map outputs by inter-key
- For each reduce-input, invoke Reduce(inter-key, inter-value-set)
  - If certain sets of inter-key go to a certain computer, we can sort on that computer and invoke Reduce() their.

Problems Solvable by MapReduce

- Classic Problems
  - Distributed grep
  - Build invert index
  - Distributed sort
- Our Work
  - Click Model
  - Logistic regression
  - Latent topic model
  - Language Model

Key Elements in MapReduce

- Required
  - Start tasks on remove computers
  - Effecient distributed sorting of map outputs
- Optional
  - Data error tolerance -- need a distributed filesystem
  - Computing error tolerance -- need a scheduler
  - Efficient task rescheduling -- need a master worker
  - Monitoring -- HTTP status page and counter

The Simplest MapReduce Implementation

- Implement MapReduce by a one-line bash script

```
cat input_file |
awk ‘ { for (i = 1; i <= NF; ++i) print $i 1; } ’ |
sort |
awk ‘ {if ($1 != word) {print word, count; word = $1; count = 0;} ++count} ’ >
output_file
```

- Each map input is a text line
- Only one map worker; outputs key = word, value = 1
- Only one reduce worker; sums up occurrences of each word

More Complexity: BashReduce

- Run on more computers, netcat map outputs to reduce machines
- Use remote shell or sshd to start remove task workers
- Invoke Map() and Reduce() written in bash or user programs
- No distribute filesystem

Our Work: MRML

- No distributed filesystem; use mpd or sshd to start task workers
- No explicit copying of map outputs to reduce machines; map worker use MPI or socket API to send outputs to reduce workers
- Reduce workers internally sorts map outputs; use incremental reduction mode to handle many input map outputs:
  - Traditional Reduce API:

    ```
    void Reduce(string key, ValueIterator values);
    ```

  - Incremental API:

    ```
    void* /*partial_result*/ BeginReduce(string key, string first_value)
    void PartialReduce(string key, string value, void* partial_result)
    void EndReduce(string key, void* partial_result);
    ```

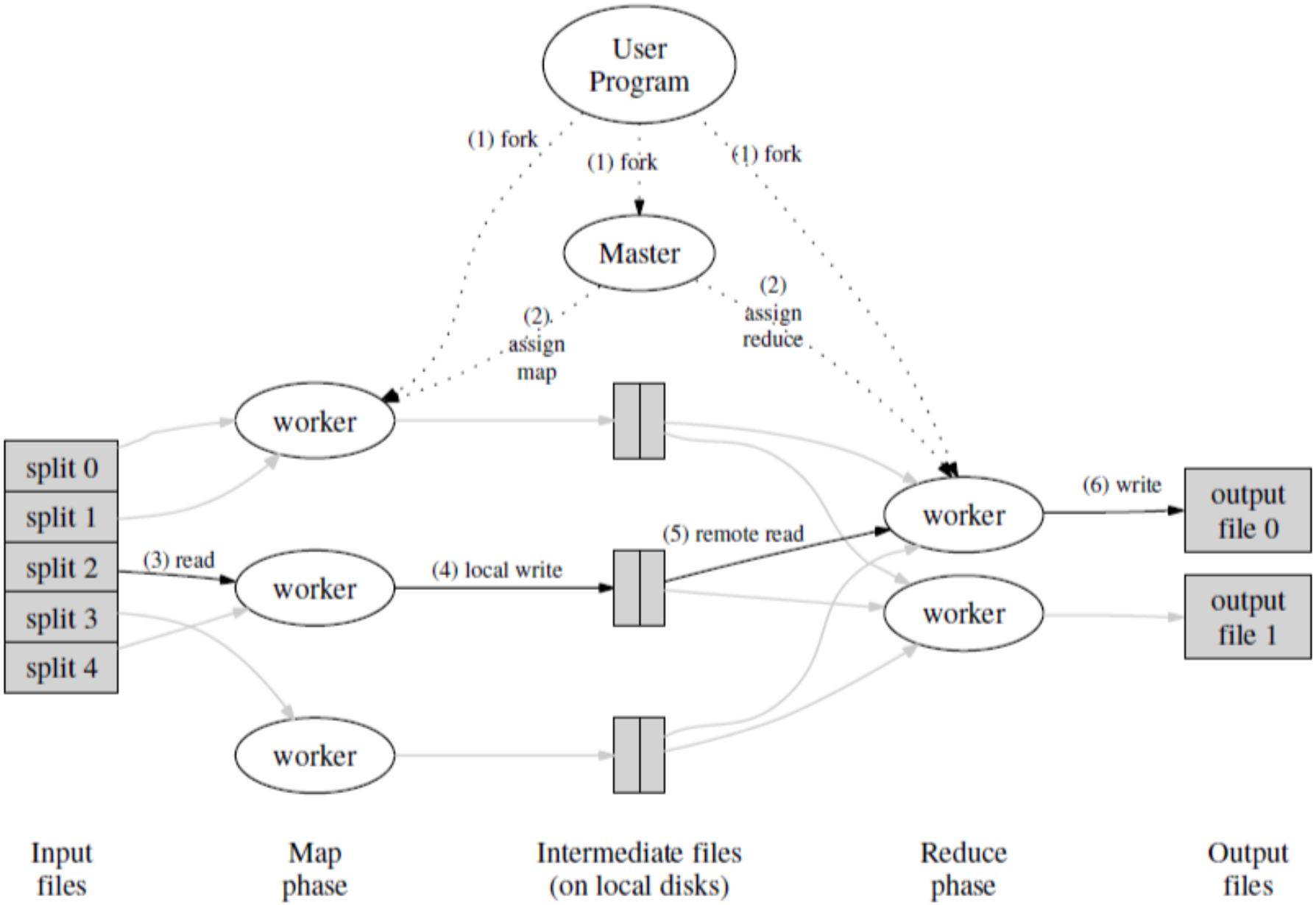
From MRML to MRLite

- MRML avoids disk IO between map phase and reduce phase
  - However, if we have a large map output key vocabulary, MRML crashes
  - Most machine learning algorithms have small vocabs
- Solution 1: No internal sort, but external sort using B-tree
  - Berkeley DB is a stable B-tree implementation
  - However, B-tree is slow for insertion of massive key-value pairs
  - More dangerous, frequent disk block accesses destroy disks
- Solutions 2: external sort using merge sort
  - Sequencial write and read, avoids frequent access to certain disk blocks
  - However, it is slow for map workers waiting reduce worker to write

Distributed OS?

- Solution 3: Map workers write to local disk, reduce workers do merge sort
  - Map workers write small sorted buffers into local files
  - After all map workers done, netcat data to reduce computers
  - Reduce worker do merge sort
  - No more communication dependency between map worker and reduce workers, thus easy fault recovery
    - Job 1: Start map workers and restart failed ones
    - Job 2: Use netcat to copy data
    - Job 3: Start reduce workers and restart failed ones

Google MapReduce : Overview



Google MapReduce : Master Process

- A master process which assign tasks to workers
  - A worker may handle more than one tasks in its life, it can handle both map and reduce tasks
  - Small tasks leads to a small set of map outputs, which can be sorted in memory buffer
- Communication between master and workers
  - assign tasks / report progress / restart failed workers
  - Health checking / Counters aggregation / status report
- Master Data Structure
  - For each task : **state {idle, in-progress, completed}**
    worker_id (if state==in-progress)
    map output file info (if role==map && state==completed)

Google MapReduce: Failure Recovery

- Master periodically ping workers, response timeout means worker failed
  - Reset all map tasks completed by this worker to idle
  - No need to idlelize reduce tasks completed by this worker, since reduce output on distribute filesystem already
  - Idlelize the in-progress task on the worker
- Notice: reduce tasks are started after all map tasks were completed
- Master periodically checkpoints to distributed filesystem
  - The Google paper does not clarify the recovery of master

Google MapReduce : Some Details

- Reduce workers read map outputs using RPC, RPC server is the map worker, so failed worker's completed map tasks need to be idlelized.
- No difference with worker failure and machine failure!

Google MapReduce : Backup Tasks

- If few tasks are slow, the whole job is slow to complete
  - e.g., a disk has bad tracks, checking slows down IO
  - e.g., unexpected many tasks on a computer
- When a MapReduce job is close to complete, many workers are idle, master then use them to execute replica of remaining tasks. One such task completes, master marks the task as completed

Google MapReduce : Semantics

- Considering both fault recovery and backup tasks:
  - Each in-progress task outputs to private temporary file
  - A map task outputs to R local files; a reduce task outputs to a distributed filesystem file
  - A map task notify master about its R output files; if master receives duplicated notifications about a completed task, it ignores such duplica.
  - After a reduce task completes, rename the GFS file; GFS prevents duplicated renaming.

Google MapReduce : Refinements

- Partition Function : reducer_id <- partition_function(key)
- Sorted Outputs can be written to SSTable
- Combiner Function : locally executed reducer
- IO formats : Google MapReduce supports few pre-defined formats; Hadoop supports user defined formats through customized splitter.
- Side-effects :
  - multiple map inputs
  - multiple map outputs
  - multiple reduce outputs

Google MapReduce : Refinements

- Skip bad key-value pairs
  - Workers installs signal handlers to catch SEGFAULT, and tell master the current record. Master tell worker to skip the bad record in re-execution of the task.
- Debug : a locally executed implementation of MapReduce
- Status report: each master or worker has a HTTP server returning status
  - The TCP port is allocated by the distributed OS. Communication, RPC, and status report are all bind to this port.
- Counters : master aggregates workers' counts; be aware of de-duplication.

How about Hadoop

- NextGen Hadoop Plan:
  http://developer.yahoo.com/blogs/hadoop/posts/2011/02/mapreduce-nextgen/
- Split JobTracker into
  - ResourceManager: Google Borg, distributed OS
  - ApplicationMaster: master in Google MapReduce

Thanks!
Q/A