Parallel Processing using MapReduce

Prof. Yanlei Diao

Some materials borrowed from talks by Jeff Dean, Sanjay Ghemawat, Christophe Bisciglia, Aaron Kimball, and Sierra Michels-Slettvet

Motivation: Large Scale Data Processing

- Want to process lots of data, *unstructured or structured*
- Want to parallelize across *hundreds/thousands* of commodity computers
  - New definition of cluster computing: *large numbers of low-end processors working in parallel to solve a computing problem.*
  - Parallel DB: *a small number of high-end servers.*
- Want to make this easy

I. MapReduce

- Clean abstraction for programmers
- Automatic parallelization & distribution
- Fault-tolerance
- Status and monitoring tools

MapReduce: Simplified Data Processing on Large Clusters. Jeffrey Dean and Sanjay Ghemawat. OSDI 2004.

Programming Model

- Borrows from functional programming
- Users implement an interface of two functions:

```
– map  (in_key, in_value) ->
    list(out_key, intermediate_value)

– reduce (out_key, list(intermediate_value) ->
    list(out_value)
```

map

- Input: a key-value pair. E.g.,
  - A line out of files (filename, line),
  - A row of a database (row_id, row),
  - A document (doc_name, document)
- map( ) produces one or more *intermediate* values along with an output key from the input.
- map( ) is stateless: one input leaves no state that would affect the processing of the next input.

reduce

- After the map phase is over, all the intermediate values for a given output key are collected into a list
- reduce( ) combines those intermediate values into one or more *final values* for that same output key
- reduce( ) can be stateful: it operates on all the intermediate values of a certain key

Example: Count Word Occurrences

```
map(String input_key, String input_value):
  // input_key: document name
  // input_value: document contents
  for each word w in input_value:
    EmitIntermediate(w, "1");

reduce(String output_key, Iterator intermediate_values):
  // output_key: a word
  // output_values: a list of counts
  int result = 0;
  for each v in intermediate_values:
    result += ParseInt(v);
 Emit(AsString(result));
```

How do we implement this using a relational DBMS? Customized data loading (data may be used only once), then Group By.

Click Stream Analysis: Page Frequencies

```
Clicks(time, url, referral_url, user_id, geo_info…)
```

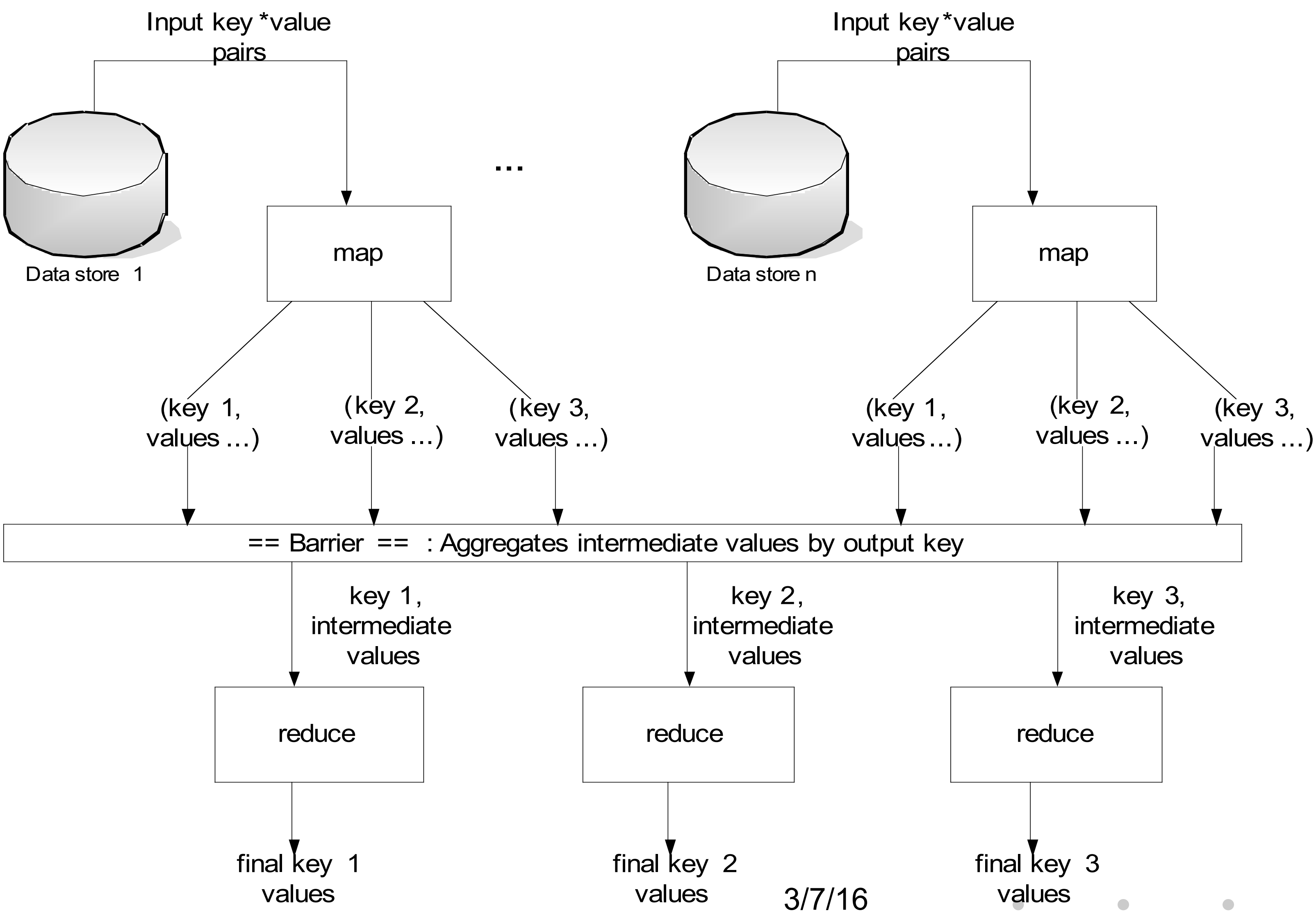
```
map(String tuple_id, String tuple):
   EmitIntermediate(url, "1");

reduce(String url, Iterator list_tuples):
   int result = 0;
   for each t in list_tuples:
    result += ParseInt(t);
   Emit(AsString(result));
```

```
Select count(*)
From Clicks
Group By url;
```

MapReduce Computation Model

Extract (key, value) using map(). **Group data by key.** Then apply reduce().



Parallelism

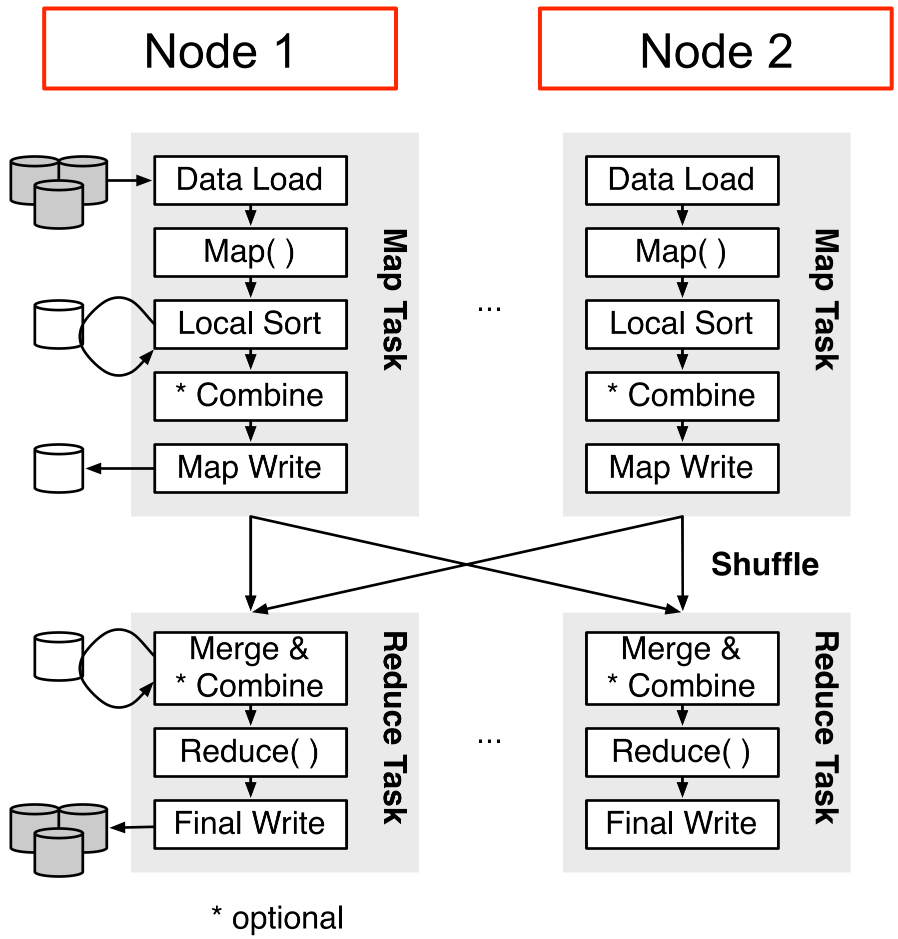
- The map() function is stateless, so many instances can run in parallel on different splits (chunks) of input data
- The reduce() function is stateful, but works on an output key at a time, so many copies can run in parallel on different keys (groups)
- Performance bottleneck: reduce phase can't start until map phase is completely finished.

Optimization: Incremental Computation

- "Combiner" functions can be applied earlier, e.g., right after map() finishes on the same machine
- Causes a mini-reduce phase to occur before the real reduce phase, to save bandwidth
- Common examples: word frequency, url frequency
- Also called partial aggregation

Under what conditions is it sound to use a combiner?

Illustration of the Dataflow



Fault Tolerance

- Fine-grained fault tolerance: materialize map output onto local disk before the map task completes
- Master detects worker failures
  - Re-executes completed & in-progress map() tasks
  - Re-executes in-progress reduce() tasks
- Master notices particular input key/values cause crashes in map(), and skips those values on re-execution.
  - Effect: Can work around bugs in third-party libraries!

Optimization: Redundant Execution

- No reduce can start until map is complete:
  - A single slow disk controller can rate-limit the whole process
- Master redundantly executes "slow-moving" map tasks; uses results of first copy to finish

Refinement: Exploiting Locality

- Master scheduling policy:
  - Asks GFS for locations of replicas of input file blocks
  - Map tasks typically split into 64MB (GFS block size)
  - Map tasks scheduled so GFS input block replica are on same machine or same rack

- Effect
  - Thousands of machines read input at local disk speed
  - Without this, rack switches limit read rate

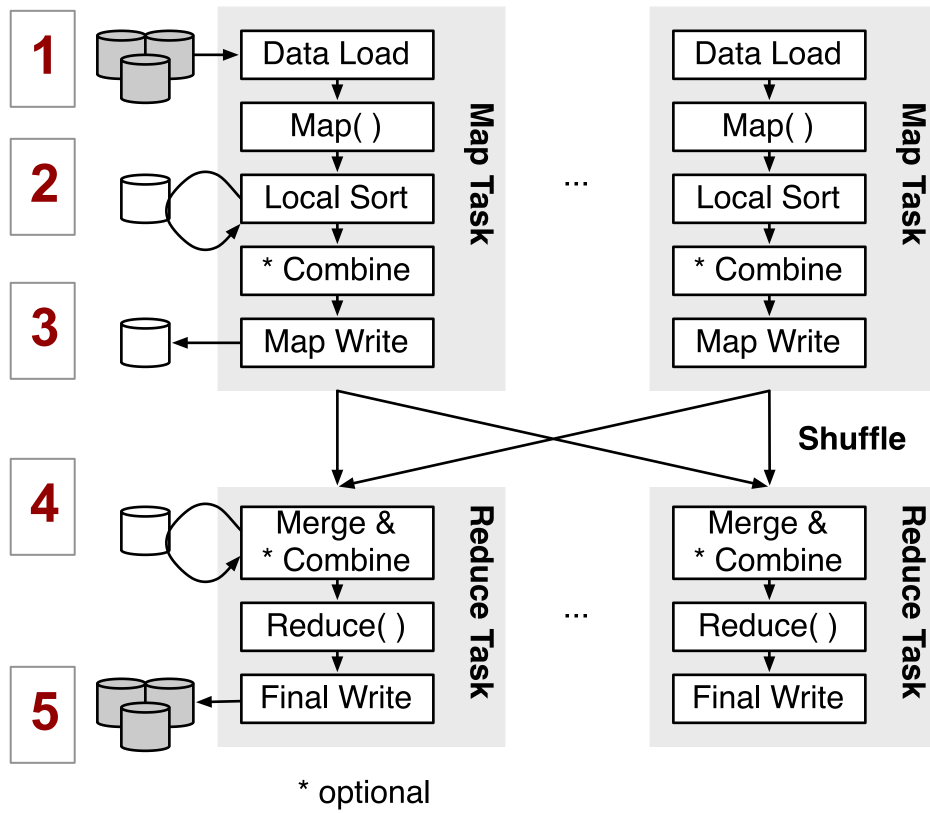
II. Comparison to Parallel Databases

- Let us consider structured data here.
  - Of course, MapReduce can also handle text processing!

1. A closer look at internal implementation of MapReduce
   - Extract (key, value) using map()
   - Group data by key
   - Then apply reduce() to each group
2. Implementing relational operators using MapReduce
   - Parallel sorting?
   - Parallel Join?
   - Parallel group by-aggregation?
3. MapReduce query plans

1. Analysis of Open Source Hadoop

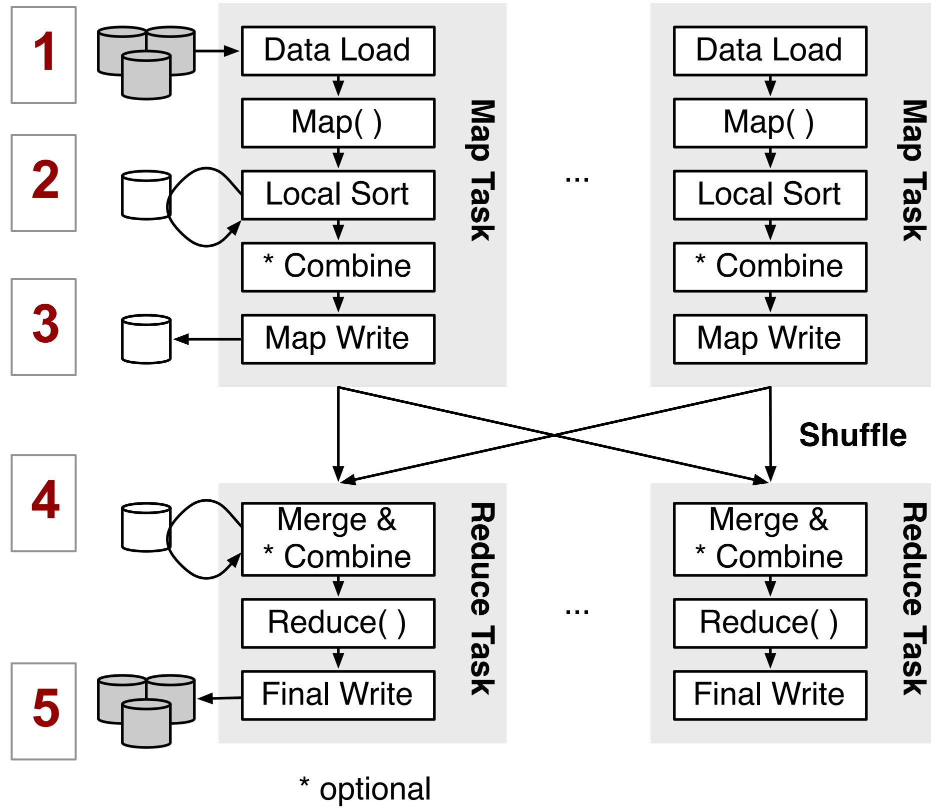
Sort-Merge Implementation of Group-By



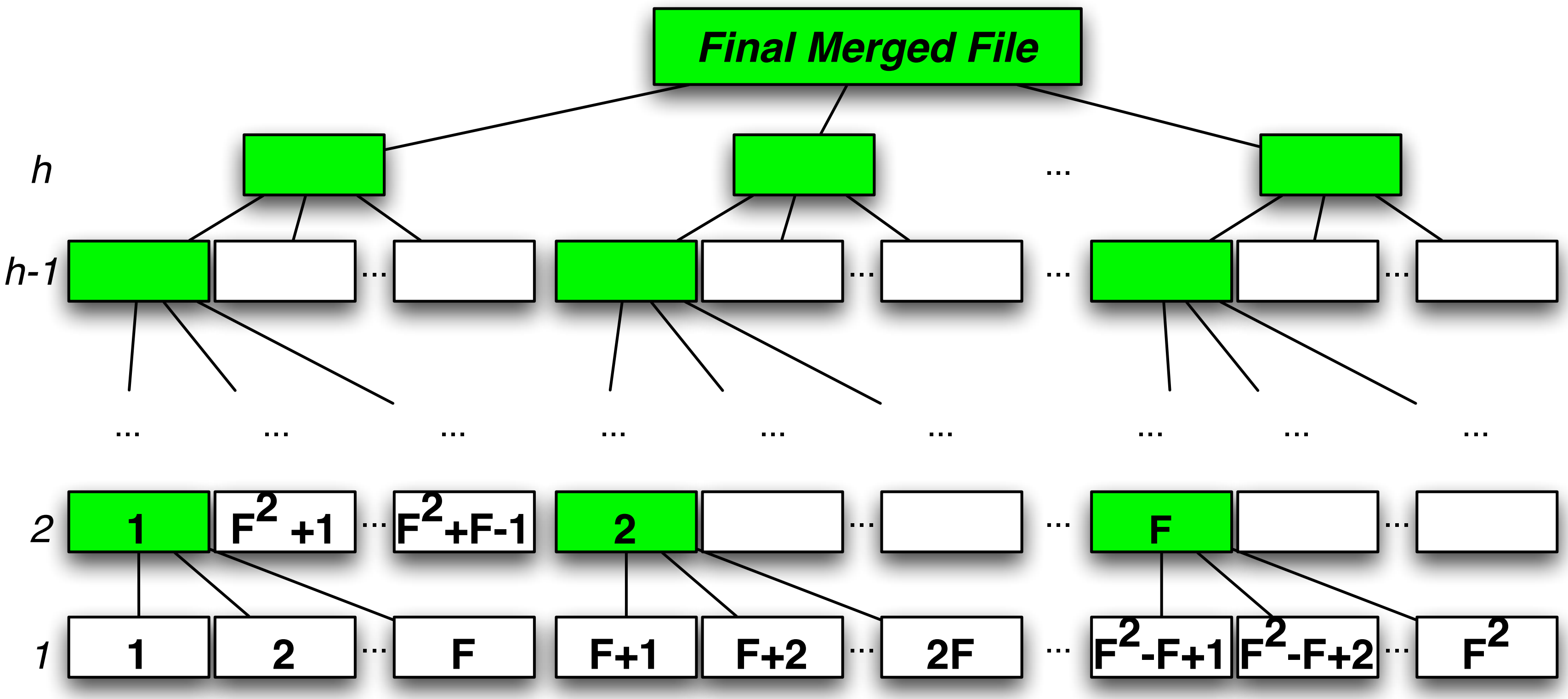
Analysis of Open Source Hadoop

R	Number of reduce tasks per node
C	Map input chunk size
F	Merge factor that controls how often on disk files are merged

Symbol	Description
(1) System Settings	
R	Number of reduce tasks per node
C	Map input chunk size
F	Merge factor that controls how often on-disk files are merged
(2) Workload Description	
D	Input data size
K_m	Ratio of output size to input size for the map function
K_r	Ratio of output size to input size for the reduce function
(3) Hardware Resources	
N	Number of nodes in the cluster
B_m	Output buffer size per map task
B_r	Shuffle buffer size per reduce task
(4) Symbols Used in the Analysis	
U	Bytes read and written per node, $U = U_1 + \ldots + U_5$ where U_i is the number of bytes of the following types 1: map input; 2: map internal spills; 3: map output; 4: reduce internal spills; 5: reduce output
S_i	Number of sequential I/O requests per node for IO type i
T	Time measurement for startup and I/O cost
h	Height of the tree structure for multi-pass merge

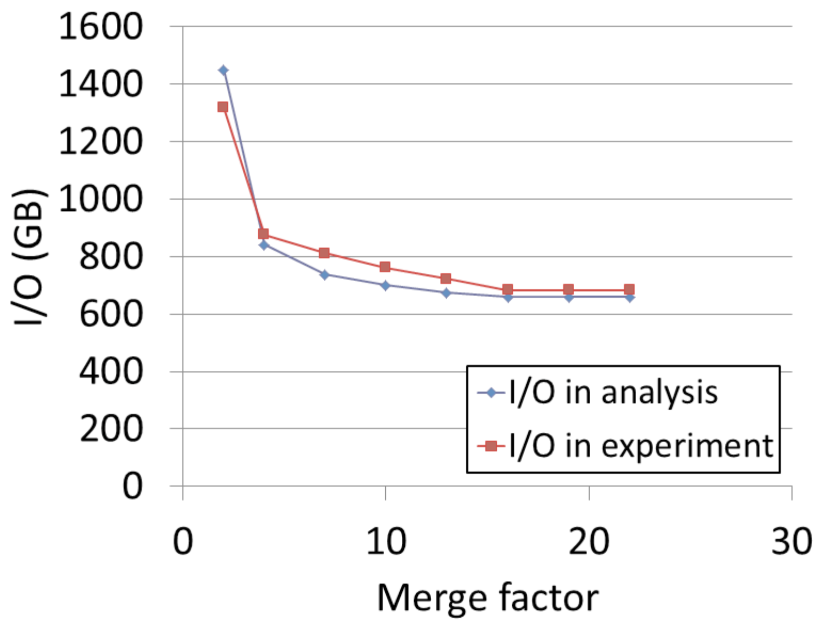


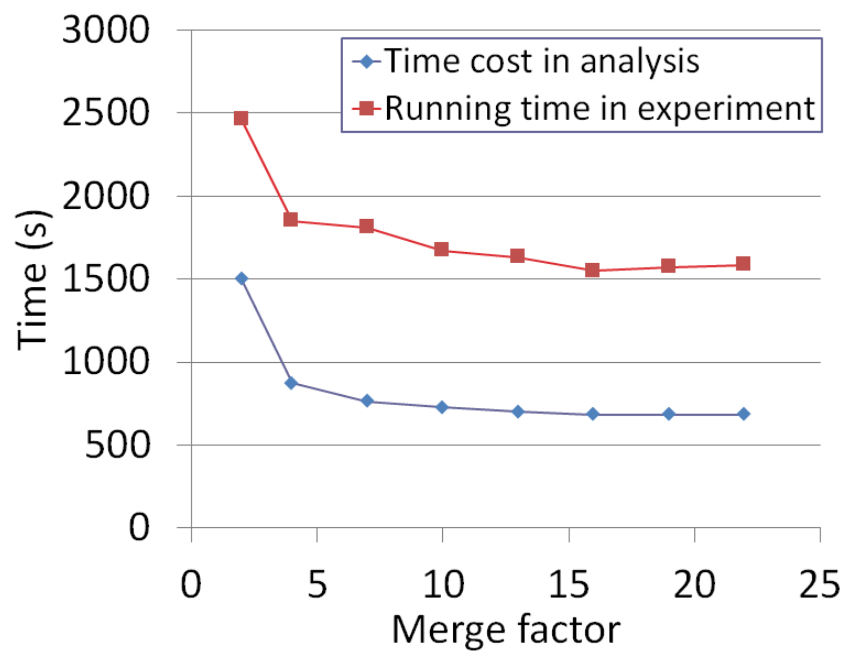
Analysis of Multi-Pass Merge



- Used in a mapper for sorting if map output exceeds memory size
- Used in a reducer unless all data fits in memory

Effect of the Merge Factor F





2. Implementing Relational Operators

- Selection: R.a > "abc"
  - ParallelDB: if range partitioned, use a few nodes and indexes
  - MapReduce: scan all nodes, map() only.
    - *Can be dominated by start-up cost. No indexes in the original impl.*
- Most other operators need *repartitioning* data:
  - ParallelDB: explicit partitioning function
  - MapReduce: more complicated

    (1) Implicit partitioning function, **fn**, controls data shuffling to reducers. (Default is hash partitioning. Can be changed to range partitioning.)

    (2) Each reducer uses an additional mechanism to group data by the key.

❑ Consider the task to range partition data and sort data in each range. What is the key in the MR programming model?

Join Operators

- Equijoin: R.a = S.a
  - ParallelDB: hybrid hash join.
    - I/O and network costs?
  - MapReduce: the programming interface is not natural for joins.
    1. map() annotates tuples with 'r' and 's',
    2. the system groups all data by the join attribute using sort-merge,
    3. reduce() joins 'r' and 's' tuples with the same value of the join attribute.
    - *It is better to change the programming model to make join more natural!*
- Non equijoin: R.a < S.a
  - ParallelDB: fragment-replication
  - MapReduce: simulates fragment-replication. If replicate S,
    - replicate each S tuple *m* times in the mapper
    - tweak the partitioning function, **fn**, for shuffling so that these *m* copies go to different reducers (fn can be customized in Hadoop)

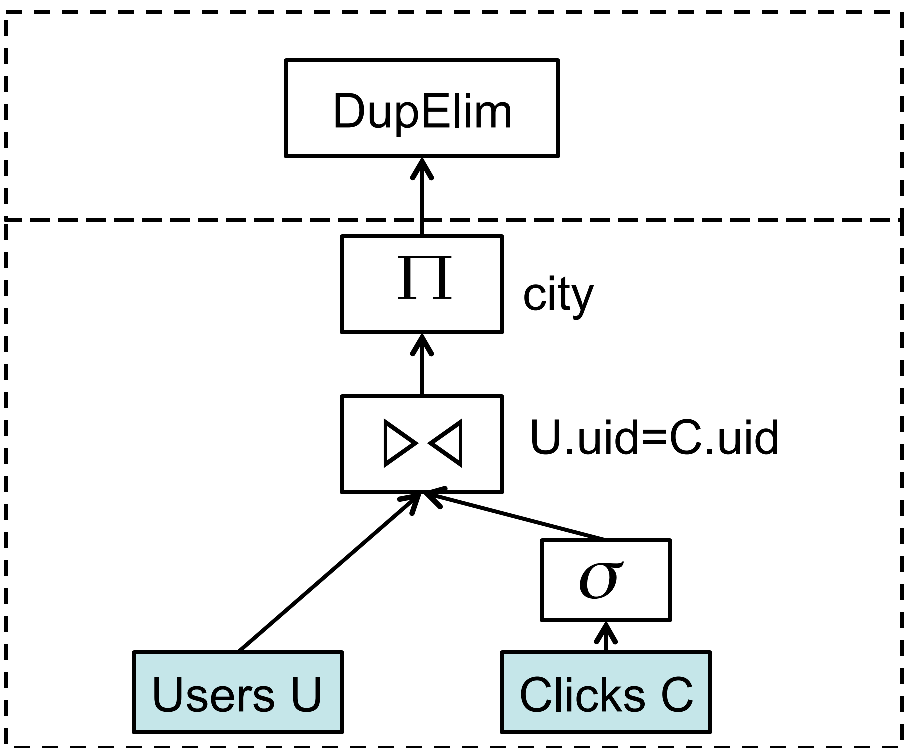
Group By Aggregation

- Scalar aggregate: count(), sum()
  - ParallelDB: partial aggregation + final aggregation
  - MapReduce: map() is empty; use combiner() for partial aggregation; use reduce() for final aggregation
- Group by aggregation: $G_{R.a,\ aggr(R.b)}$
  - ParallelDB: unary input version of hybrid hash join
  - MapReduce:
    - map() simply emits tuples;
    - the system groups data by R.a;
    - reduce() computes sum.
    - should use the combiner() for partial aggregation earlier.

3. MapReduce Query Plans

- How many rounds of map reduce jobs?
- In each round, what is in map(), what is in reduce()?

```
SELECT  DISTINCT U.city
FROM    Users U, Clicks C
WHERE   U.uid=C.uid
   AND  C.url  LIKE '%google%';
```

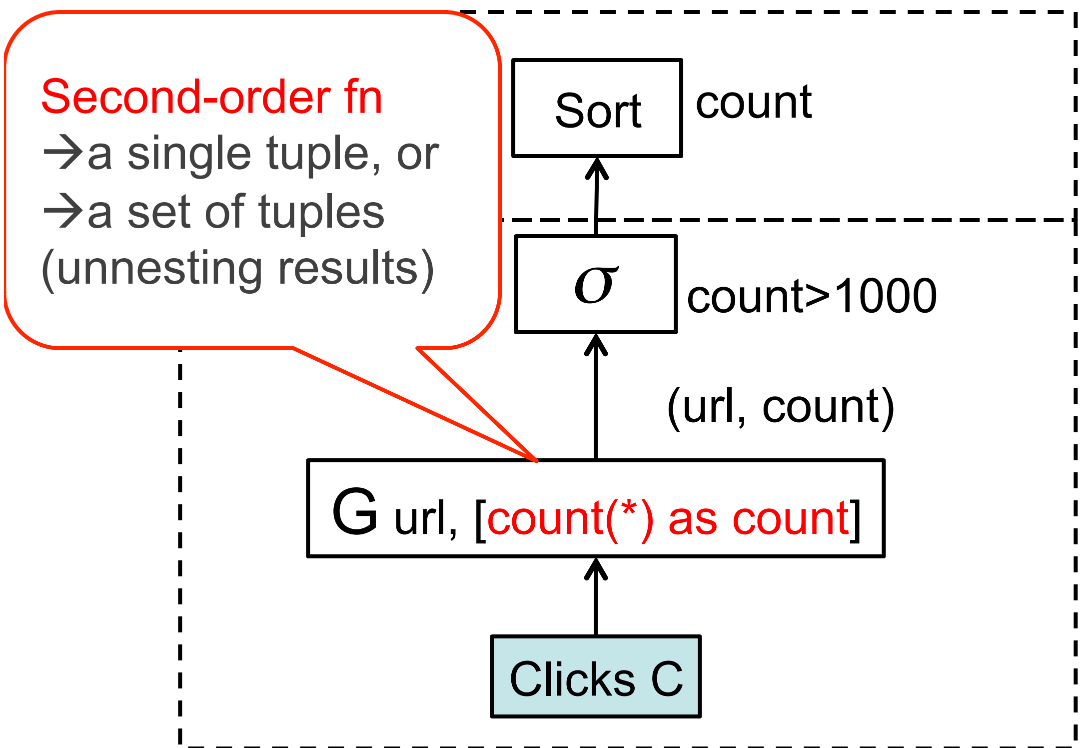


Round 2:
Key: city
Map: emit
Reduce: emit a tuple in each group

Round 1:
Key: uid
Map: (1) selection, (2) create 'u', 'c' tuples with labels
Reduce: (1) join tuples within each group, (2) emit cities

More on Query Plans

```
SELECT url, count(*)
FROM    Clicks C
GROUP BY url
HAVING count(*) > 1000
ORDER BY count(*) DESC;
```



Round 2:
Key: count
Map: emit
Shuffle: range partitioning (set manually)
Reduce: local sort
or (a simple but bad plan)
(**Key**: fixed
Map: emit
Reduce: sort all in a single reducer)

Round 1:
Key: url
Map: emit
Reduce: (1) count, (2) selection, (3) emit (url, count)

References

- MapReduce: Simplified Data Processing on Large Clusters. Jeffrey Dean and Sanjay Ghemawat. OSDI 2004.
- MapReduce and Parallel DBMSs: Friends or Foes?. Michael Stonebraker, Daniel Abadi, David J. DeWitt, Sam Madden, Erik Paulson, Andrew Pavlo, Alexander Rasin. CACM Jan 2010.
- MapReduce: A Flexible Data Processing Tool. Jeffrey Dean, Sanjay Ghemawat. CACM Jan 2010.
- Dryad/Linq: http://research.microsoft.com/en-us/projects/dryadlinq/
- Hyracks: A Flexile and Extensible Foundation for Data-Intensive Computing. Vinayak Borkar, et al. ICDE 2011.
- Llama: Leveraging Columnar Storage for Scalable Join Processing in the MapReduce Framework. Yuting Lin, etc. SIGMOD 2011.
- A Platform for Scalable One-Pass Analytics. Boduo Li, et al. SIGMOD 2011.

Questions