(Distributed) Parallelism

with slides by Christophe Bisciglia, Aaron Kimball, & Sierra Michels-Slettvet
and slides by Jeff Dean.

Distribution and Parallelism

- Distribution
  - A program is split into parts that run on multiple computers communicating over a network
- Parallelism
  - Problem divided into smaller ones which are solved concurrently

How to leverage thousands of networked computers to store and process petabytes of data?

Massive Data Sets

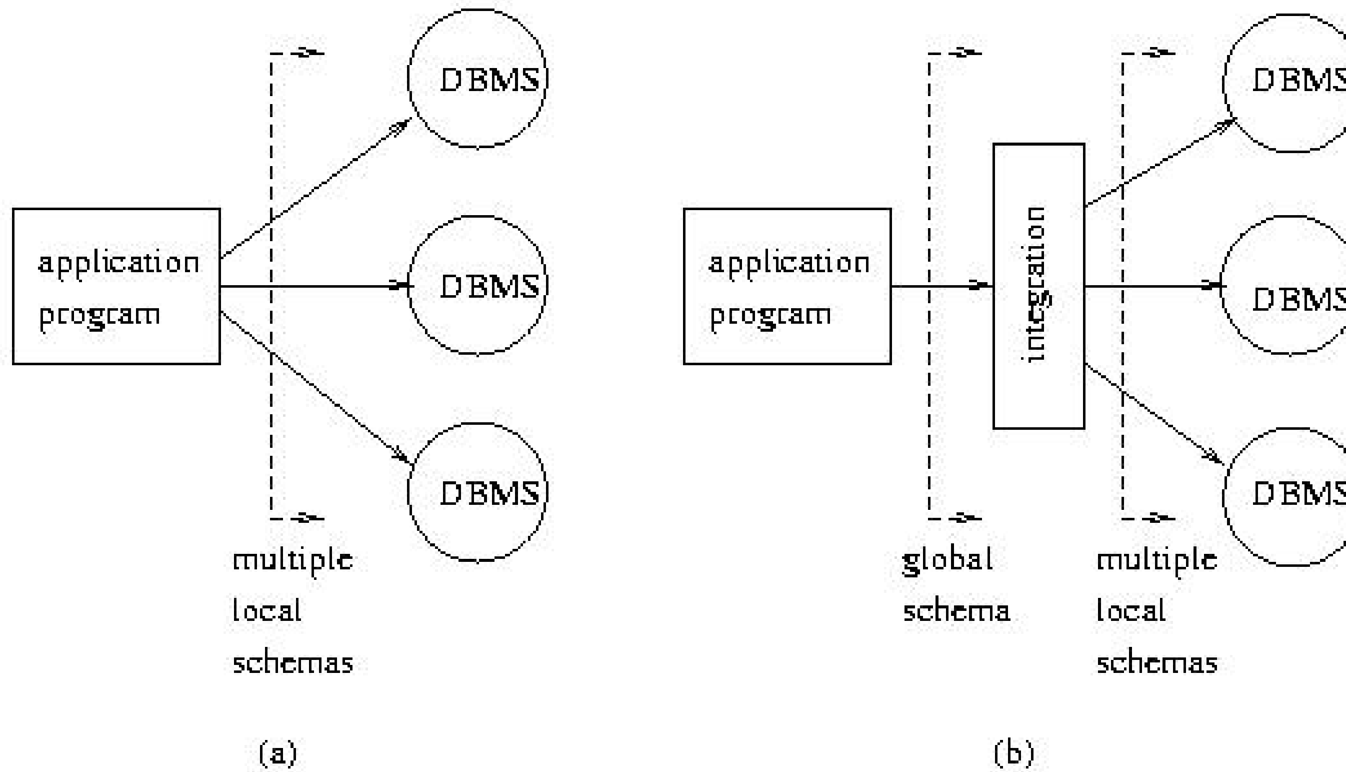
- Example: 20+ billion web pages x 20KB = 400+ terabytes
- One computer can read ~100 MB/sec from disk
  - ~1 month to read the web
  - ~800 hard drives just to store the web
- Even more to do something with the data

Parallelism 101

- Step1:
  - partition the work
- Step2:
  - units of work processed in parallel
- Step3:
  - report result

- Step 1 Problems:
  - **Partition/replicate the data**
  - **Split/cloned processing**

Views of Distributed Data



(a) Multidatabase with local schemas

(b) Integrated distributed database with global schema

Multidatabases

- Application must explicitly connect to each site
- Application accesses data at a site using SQL statements based on that site's schema
- Application may have to do reformatting in order to integrate data from different sites
- Application must manage replication
  - Know where replicas are stored and decide which replica to access

Integration

- Middleware provides integration of local schemas into a global schema
  - Application need not connect to each site
  - Application accesses data using global schema
    - Need not know where data is stored – ***location transparency***
  - Global joins are supported
  - Middleware performs necessary data reformatting
  - Middleware manages replication – ***replication transparency***

Partitioning

- Data can be distributed by storing individual tables at different sites
- Data can also be distributed by decomposing a table and storing portions at different sites – called ***partitioning***
- Partitioning can be ***horizontal*** or ***vertical***

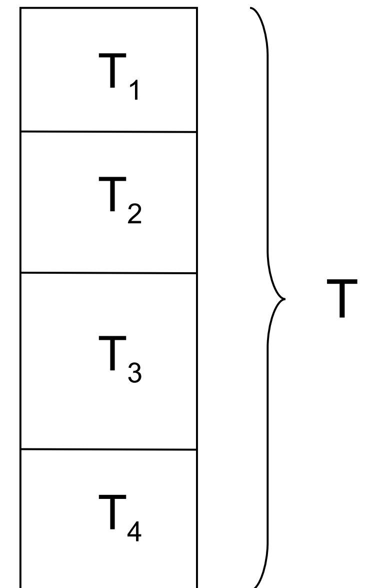
Horizontal Partitioning

- Each partition, T_i , of table T contains a subset of the rows and each row is in exactly one partition:

$$T_i = \sigma_{C_i}(T)$$

$$T = \cup \; T_i$$

  – Horizontal partitioning is lossless



Partitioning Techniques

Round-robin (n sites):

Send the ith tuple inserted in the relation to site *i* mod *n*.

Hash partitioning:

- Choose one or more attributes as the partitioning attributes.
- Choose hash function *h* with range 0…*n* - 1
- Let *i* denote result of hash function *h* applied to the partitioning attribute value of a tuple. Send tuple to site *i*.

Partitioning Techniques

Range partitioning:

- Choose an attribute as the partitioning attribute.
- A partitioning vector [v_o, v_1, ..., v_{n-2}] is chosen.
- Let v be the partitioning attribute value of a tuple. Tuples such that $v_i \leq v_{i+1}$ go to site $I + 1$. Tuples with $v < v_0$ go to disk 0 and tuples with $v \geq v_{n-2}$ go to disk n-1.

Partitioning Techniques

- Evaluate how well partitioning techniques support the following types of data access:

  1. Scanning the entire relation.
  2. Locating a tuple associatively – **point queries**.
     - E.g., $r.A = 25$.
  3. Locating all tuples such that the value of a given attribute lies within a specified range – **range queries**.
     - E.g., $10 \leq r.A < 25$.

- No clustering -- tuples are scattered across all disks

Round robin:

- Advantages
  - Best suited for sequential scan of entire relation on each query.
  - All disks have almost an equal number of tuples; retrieval work is thus well balanced between disks.
- Range queries are difficult to process
  - No clustering -- tuples are scattered across all disks

Partitioning Techniques

Hash partitioning:

- Good for sequential access
  - Assuming hash function is good, and partitioning attributes form a key, tuples will be equally distributed between sites
  - Retrieval work is then well balanced between sites.
- Good for point queries on partitioning attribute. Can lookup single site, leaving others available for answering other queries.
- No clustering, so difficult to answer range queries

Partitioning Techniques

Range partitioning:

- Provides data clustering by partitioning attribute value.
- Good for sequential access
- Good for point queries on partitioning attribute: only one disk needs to be accessed.
- For range queries on partitioning attribute, one to a few disks may need to be accessed
- For range queries on non partitioning attribute, randomized access to sites.

Horizontal Partitioning

- *Example*: An Internet grocer has a relation describing inventory at each warehouse

  Inventory(*StockNum*, *Amount*, *Price*, *Location*)

- It partitions the relation by location and stores each partition locally: rows with *Location* = 'Chicago' are stored in the Chicago warehouse in a partition

  Inventory_ch(*StockNum*, *Amount*, *Price*, *Location*)

- Alternatively, it can use the schema

Inventory_ch(*StockNum*, *Amount*, *Price*)

Vertical Partitioning

- Each partition, T_i, of T contains a subset of the columns, each column is in at least one partition, and each partition includes the key:

  $T_i = \pi_{attr_list_i}(T)$

  $T = T_1 \bowtie T_2 \ldots\ldots \bowtie T_n$

  - Vertical partitioning is lossless
- *Example*: The Internet grocer has a relation

  Employee(*SSnum*, *Name*, *Salary*, *Title*, *Location*)

  - It partitions the relation to put some information at headquarters and some elsewhere:

  Emp1(*SSnum*, *Name*, *Salary*) – at headquarters

  Emp2(*SSnum*, *Name*, *Title*, *Location*) – elsewhere

Replication

- One of the most useful mechanisms in distributed databases
- Increases
  - Availability
    - If one replica site is down, data can be accessed from another site
  - Performance:
    - Queries can be executed more efficiently because they can access a local or nearby copy
    - Updates might be slower because all replicas must be updated

Replication Example

- Internet grocer might have relation

  Customer(*CustNum*, *Address*, *Location*)

  - Queries are executed
    - At headquarters to produce monthly mailings
    - At a warehouse to obtain information about deliveries
  - Updates are executed
    - At headquarters when new customer registers and when information about a customer changes

Example (con't)

- Intuitively it seems appropriate to *either* or *both*:
  - Store complete relation at headquarters
  - Horizontally partition a replica of the relation and store a partition at the corresponding warehouse site
- Each row is replicated: one copy at headquarters, one copy at a warehouse
- The relation can be both distributed *and* replicated

Example (con't): Performance Analysis

- We consider three alternatives:
  - Store the entire relation at the headquarters site and nothing at the warehouses (no replication)
  - Store the partitions at the warehouses and nothing at the headquarters (no replication)
  - Store entire relation at headquarters and a partition at each warehouse (replication)

Example (con't): Performance Analysis - Assumptions

- To evaluate the alternatives, we estimate the amount of information that must be sent between sites.
- Assumptions:
  - The Customer relation has 100,000 rows
  - The headquarters mailing application sends each customer 1 mailing a month
  - 500 deliveries are made each day; a single row is read for each delivery
  - 100 new customers/day
  - Changes to customer information occur infrequently

Example: The Evaluation

- Entire relation at headquarters, nothing at warehouses
  - 500 tuples per day from headquarters to warehouses for deliveries
- Partitions at warehouses, nothing at headquarters
  - 100,000 tuples per month from warehouses to headquarters for mailings (3,300 tuples per day, amortized)
  - 100 tuples per day from headquarters to warehouses for new customer registration
- Entire relation at headquarters, partitions at warehouses
  - 100 tuples per day from headquarters to warehouses for new customer registration

Example: Conclusion

- Replication (case 3) seems best, if we count the number of transmissions.
- Let us look at other measures:
  - If no data stored at warehouses, the time to handle deliveries might suffer because of the remote access (probably not important)
  - If no data is stored at headquarters, the monthly mailing requires that 100,000 rows be transmitted in a single day, which might clog the network
  - If we replicate, the time to register a new customer might suffer because of the remote update
    - But this update can be done by a separate transaction after the registration transaction commits (***asynchronous update***)

Distributed Parallelism 101

- Step1:
  - partition the work
- Step2:
  - units of work processed in parallel
- Step3:
  - report result

- Step 2 Problems:
  - Assign work to processing unit
  - **Synchronization across processing units**
  - Aggregate result
  - **Deal with failure**

Parallelism Patterns

- Master / Workers
  - Master dispatches work to workers
  - Master collects sub-results from workers
- Producer / Consumers
  - Producers create work
  - Consumers process work
  - Producer-consumer mapping
    - 1-1: Network of producers / consumers
    - N-M: via a shared work queue

Example

- Consider the problem of feature extraction from a large data set:
  - For example: find the number of occurences of a given word in a large data set (terrabytes)
- Approach:
  - Partition data set across 100s/1000s of CPU
  - Parallelize word count
    - Dispatch work, aggregate result, deal with failure

The MapReduce Approach

- Run-time engine + Programming model
  - Automatic parallelization & distribution
  - Fault-tolerant
  - Provides status and monitoring tools
  - Clean abstraction for programmers

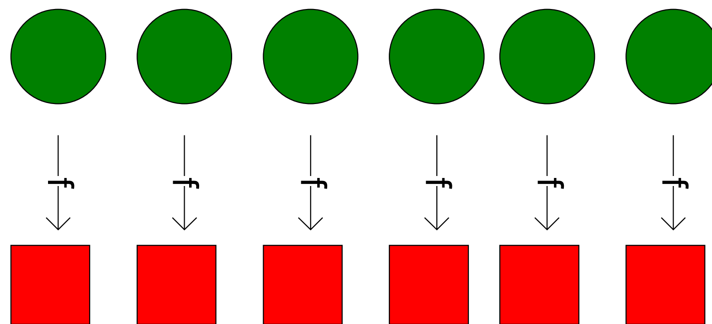
Map

- map :: (a -> b) -> [a] -> [b]

map f [] = []

map f (x:xs) = f x : map f xs

- Creates a new list by applying f to each element of the input list; returns output in order.



Programming Model

- Borrows from functional programming
- Users implement interface of two functions:
  - **map** (in_key, in_value) ->

    (out_key, intermediate_value) list
  - 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.

Programming Model

- **reduce** (out_key, intermediate_value list) -> out_value list
- 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)

Example

```
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));
```

Other examples

- need to count # of times every 5-word sequence occurs in large corpus of documents (and keep all those where count >= 4)
- With MapReduce:
  - map: extract 5-word sequences => count from document
  - reduce: combine counts, and keep if count large enough

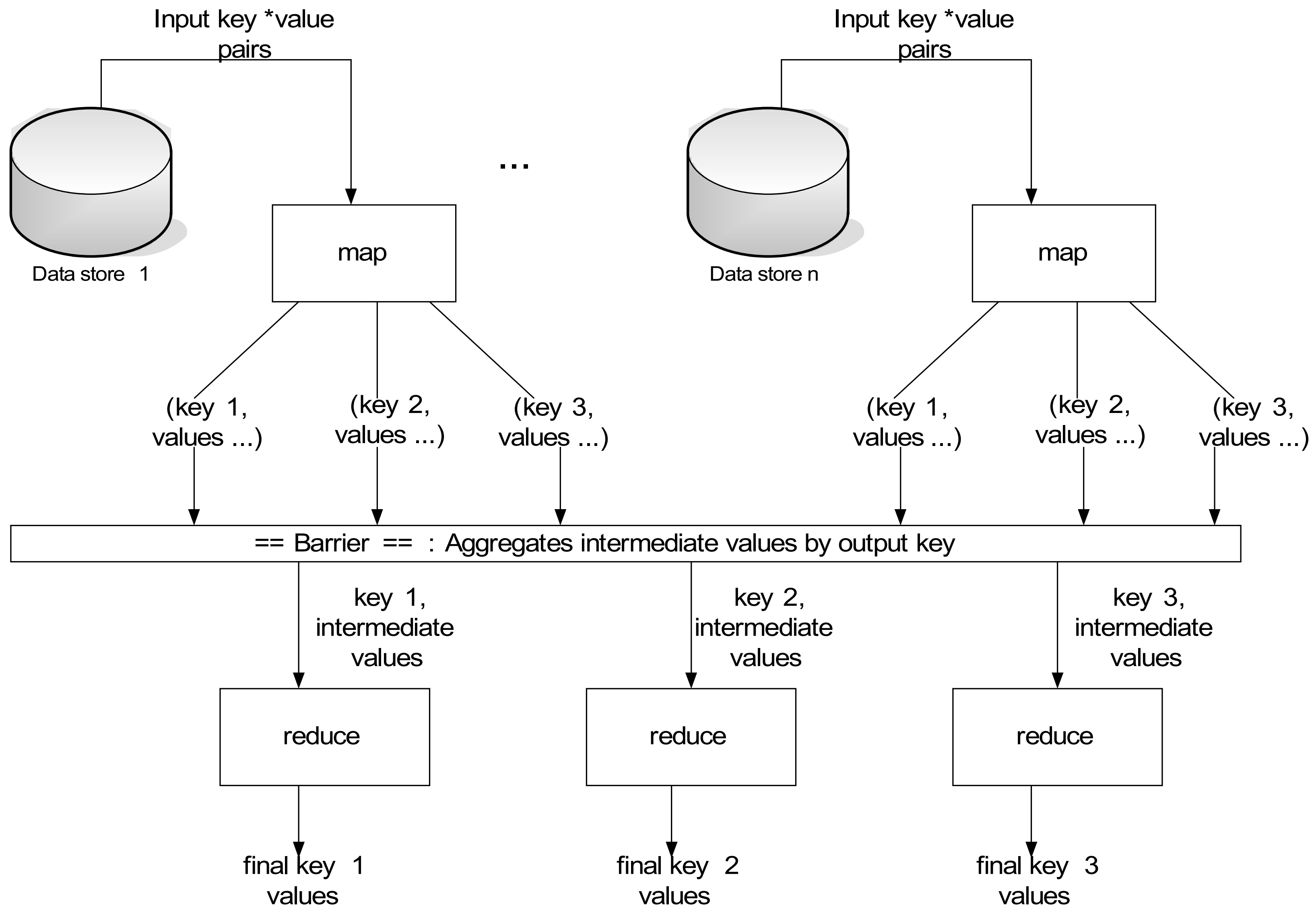
Other Examples

- Example: generate per-doc summary, but include per-host information (e.g. # of pages on host, important terms on host)
  - per-host information might be in per-process data structure, o might involve RPC to a set of machines containing data for all sites
- map: extract host name from URL, lookup per-host info
- combine with per-doc data and emit)

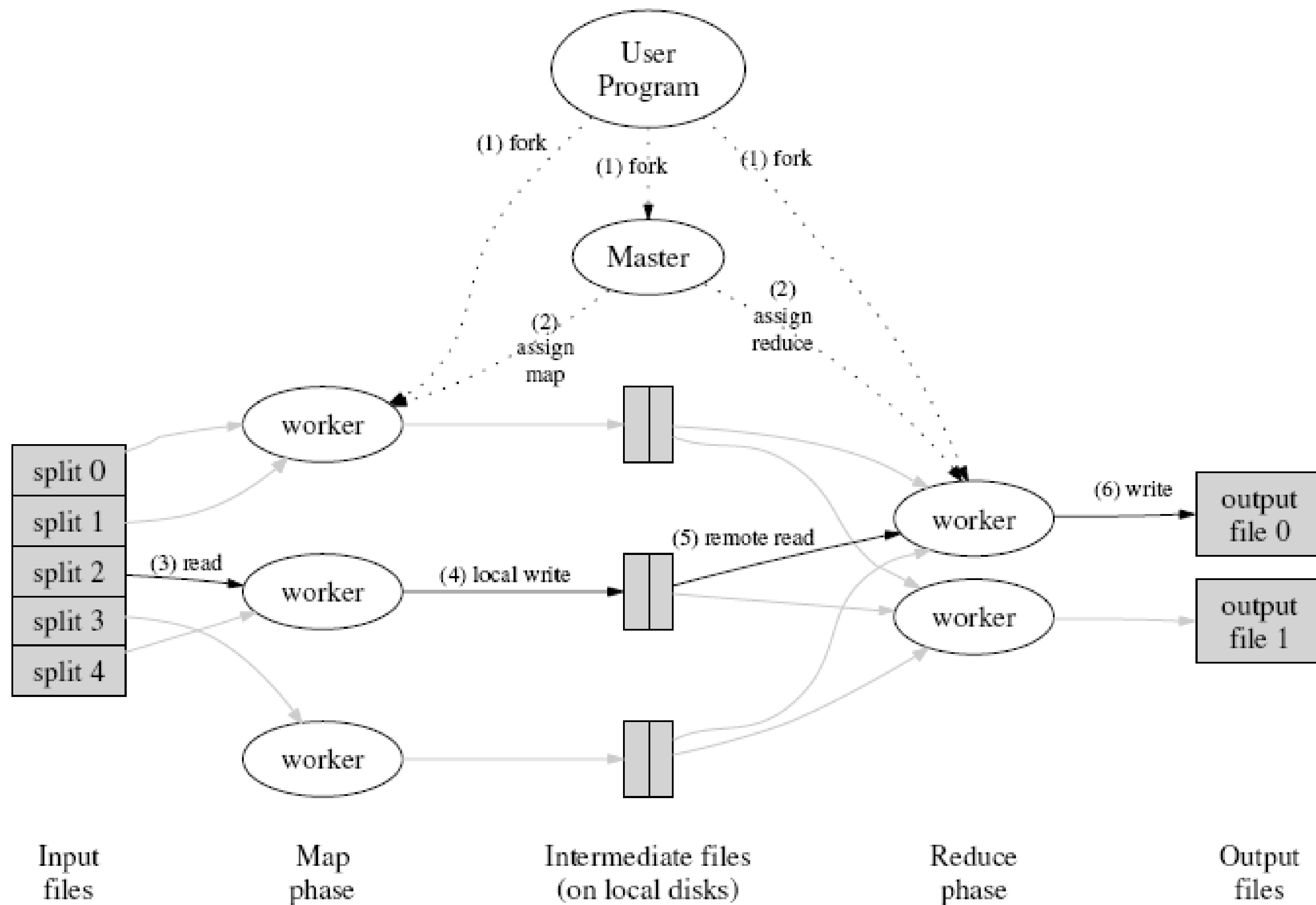
MapReduce Parallelism

- 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



Execution Engine



Failures

- 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!

Optimizations

- 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

Take-away Points

- MapReduce has proven to be a useful abstraction
  - At Google
  - Hadoop (http://hadoop.apache.org/core/)
- Functional programming paradigm can be applied to large-scale applications
- Fun to use: focus on problem, let library deal w/ messy details