MapReduce

GVGD: VÕ THỊ HỒNG TUYẾT

Nội dung

MapReduce?

- *Trước MapReduce*
  - Hệ thống đồng thời lớn
  - Điện toán lưới
  - Đưa ra giải pháp của riêng bạn
- *Cân nhắc*
  - Luồng là khó!
  - Làm thế nào để bạn mở rộng quy mô cho nhiều máy hơn?
  - Bạn xử lý lỗi máy như thế nào?
  - Làm thế nào để bạn tạo điều kiện giao tiếp giữa các nút?
  - Giải pháp của bạn có mở rộng quy mô không?
- ***Thu nhỏ lại, không tăng lên!***

Ví dụ

- I will present the concepts of MapReduce using the “typical example” of MR, Word Count
- The input of this program is a volume of raw text, of unspecified size (could be KB, MB, TB, it doesn’t matter!)
- The output is a list of words, and their occurrence count. Assume that words are split correctly, ignoring capitalization and punctuation.
- Example: The doctor went to the store. =>
    - The, 2
    - Doctor, 1
    - Went, 1
    - To, 1
    - Store, 1

Map? Reduce?

- **Mappers** read in data from the filesystem, and output (typically) modified data
- Reducers collect all of the mappers output on the keys, and output (typically) reduced data
- The outputted data is written to disk
- All data is in terms of key value pairs

MapReduce vs Hadoop

- The paper is written by two researchers at Google, and describes their programming paradigm
- Unless you work at Google, or use Google App Engine, you won't use it!
- Open Source implementation is Hadoop MapReduce
  - Not developed by Google
  - Started by Yahoo
- Google's implementation (at least the one described) is written in C++
- Hadoop is written in Java

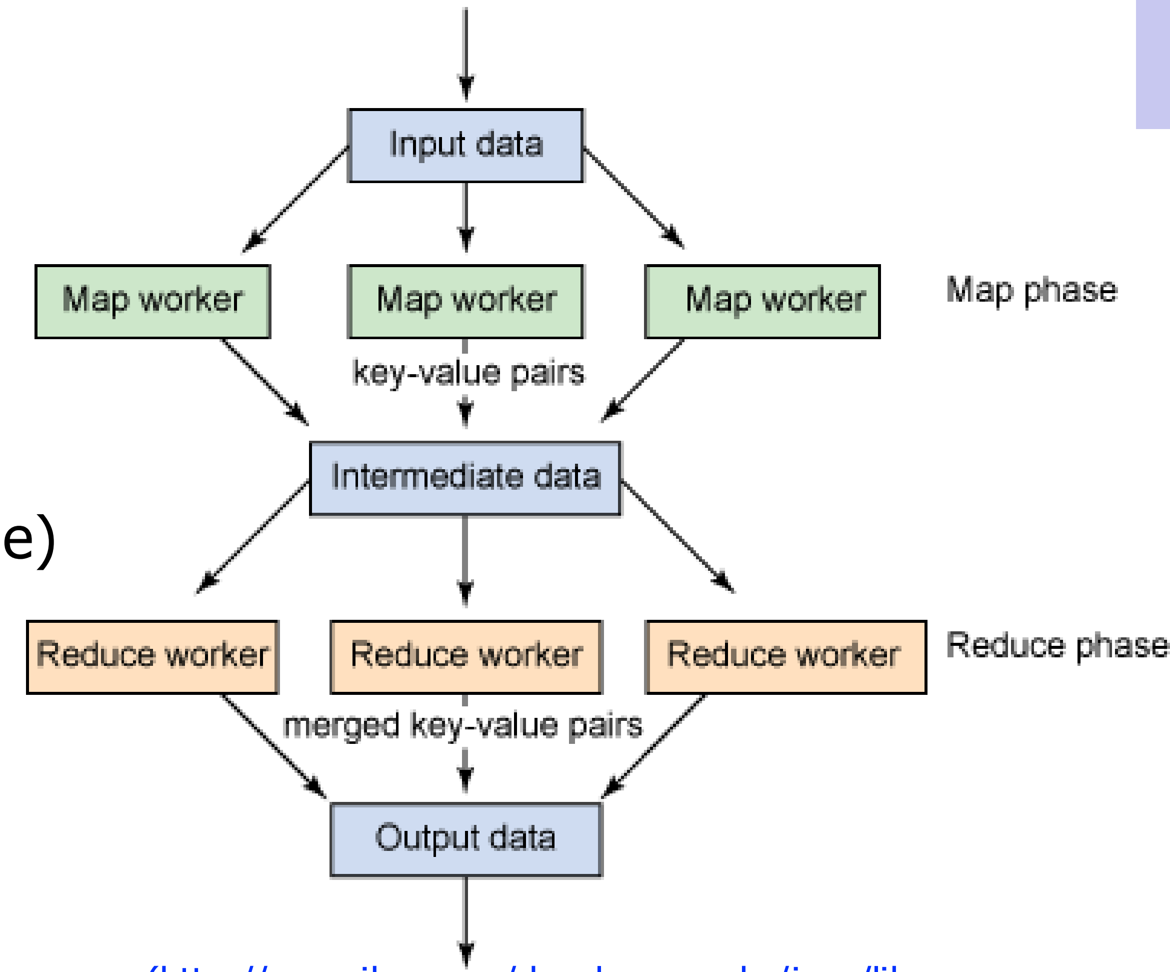
GFS/HDFS

- Google File System (GFS) và Hadoop Distributed File System (HDFS) là các hệ thống tệp được phân phối.
- Có khả năng chịu lỗi thông qua việc sao chép.
- Cho phép dữ liệu toàn cục để tính toán.

Các thành phần chính

- Thành phần người dùng:
  - Mapper
  - Reducer
  - Combiner (Optional)
  - Partitioner (Optional) (Shuffle)
  - Writable(s) (Optional)

- Thành phần hệ thống:
  - Master
  - Input Splitter
  - Output Committer



(http://www.ibm.com/developerworks/java/library/l-hadoop-3/index.html)

https://www.ibm.com/docs/en/SS8G7U_ref/pdf/itm/ITM_Hadoop_reference_guide.pdf

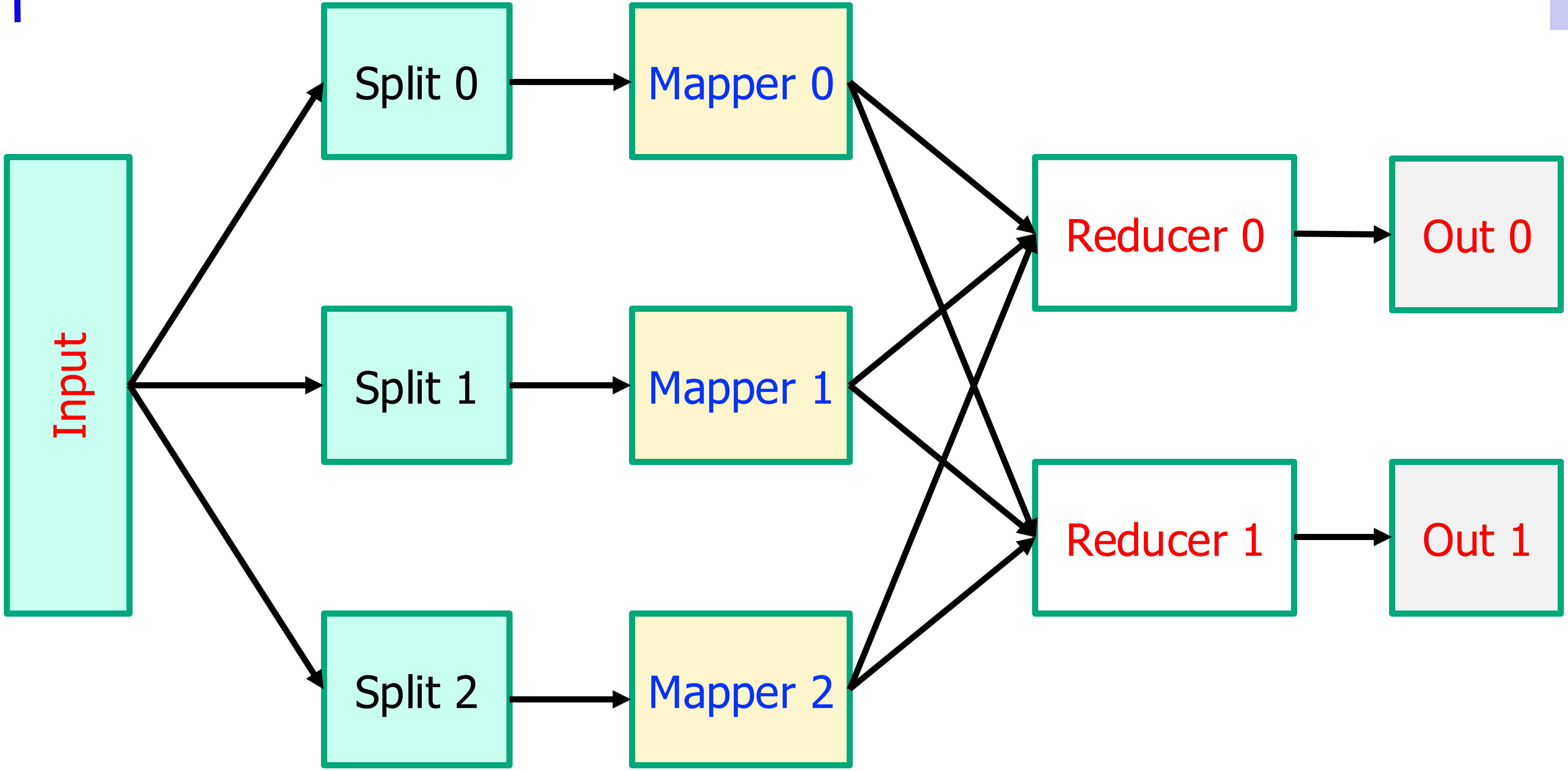
Chú ý

- Mappers và Reducers là luồng đơn và mang tính quyết định
  - Tính quyết định cho phép khởi động lại các công việc thất bại hoặc thực hiện suy đoán
- Cần xử lý nhiều dữ liệu hơn -> Just add more Mappers/Reducers!
  - Không cần xử lý mã đa luồng
  - Vì tất cả chúng đều độc lập với nhau nên bạn có thể chạy (gần như) số lượng nút tùy ý
- Mappers/Reducers chạy trên các máy tùy ý. Một máy thường có nhiều bản đồ và thu nhỏ các vị trí có sẵn cho nó, thường là một vị trí trên mỗi lõi bộ xử lý.
- Mappers/Reducers hoàn toàn độc lập với nhau khi thực thi
  - Trong Hadoop thực thi trong các JVMs riêng biệt

Nguyên tắc cơ bản

- All data is represented in key value pairs of an arbitrary type
- Data is read in from a file or list of files, from HDFS
- Data is chunked based on an input split
  - A typical chunk is 64MB (more or less can be configured depending on your use case)
- Mappers read in a **chunk** of data
- Mappers emit (write out) a set of data, typically derived from its input
- Intermediate data (the output of the mappers) is split to a number of reducers
- Reducers receive each key of data, along with **ALL** of the values associated with it (this means each key must always be sent to the same reducer)
  - Essentially, <key, set<value>>
- Reducers emit a set of data, typically reduced from its input which is written to disk

Data Flow


Input
Split 0
Split 1
Split 2
Mapper 0
Mapper 1
Mapper 2
Reducer 0
Reducer 1
Out 0
Out 1

Input Splitter

- Is responsible for splitting your input into multiple chunks
- These chunks are then used as input for your mappers
- Splits on logical boundaries. The default is 64MB per chunk
  - Depending on what you're doing, 64MB might be a LOT of data! You can change it
- Typically, you can just use one of the built in splitters, unless you are reading in a specially formatted file

Mapper

- **Reads in input pair <K,V> (a section as split by the input splitter)**
- **Outputs a pair <K', V'>**
- **Ex.** For our Word Count example, with the following input: "The teacher went to the store. The store was closed; the store opens in the morning. The store opens at 9am."
- The output would be:
  - <The, 1> <teacher, 1> <went, 1> <to, 1> <the, 1> <store, 1> <the, 1> <store, 1> <was, 1> <closed, 1> <the, 1> <store, 1> <opens, 1> <in, 1> <the, 1> <morning, 1> <the 1> <store, 1> <opens, 1> <at, 1> <9am, 1>

Reducer

- Accepts the Mapper output, and collects values on the key
  - All inputs with the same key *must* go to the same reducer!
- Input is typically sorted, output is output exactly as is
- **For our example, the reducer input would be:**
  - <The, 1> <teacher, 1> <went, 1> <to, 1> <the, 1> <store, 1> <the, 1> <store, 1> <was, 1> <closed, 1> <the, 1> <store, 1> <opens, 1> <in, 1> <the, 1> <morning, 1> <the 1> <store, 1> <opens, 1> <at, 1> <9am, 1>
- **The output would be:**
  - <The, 6> <teacher, 1> <went, 1> <to, 1> <store, 4> <was, 1> <closed, 1> <opens, 2> <morning, 1> <at, 1> <9am, 1>

Combiner

- Essentially an intermediate reducer
- Is optional
- Reduces output from each mapper, reducing bandwidth and sorting
- Cannot change the type of its input
  - Input types must be the same as output types

Output Committer

- Is responsible for taking the reduce output, and committing it to a file
- Typically, this committer needs a corresponding input splitter (so that another job can read the input)
- Again, usually built in splitters are good enough, unless you need to output a special kind of file

Partitioner (Shuffler)

- Decides which pairs are sent to which reducer
- Default is simply:
  - Key.hashCode() % numOfReducers
- User can override to:
  - Provide (more) uniform distribution of load between reducers
  - Some values might need to be sent to the same reducer
    - Ex. To compute the relative frequency of a pair of words <W1, W2> you would need to make sure all of word W1 are sent to the same reducer
  - Binning of results

Master

- Responsible for scheduling & managing jobs
- Scheduled computation should be close to the data if possible
  - Bandwidth is expensive! (and slow)
  - This relies on a Distributed File System (GFS / HDFS)!
- If a task fails to report progress (such as reading input, writing output, etc), crashes, the machine goes down, etc, it is assumed to be stuck, and is killed, and the step is re-launched (with the same input)
- The Master is handled by the framework, no user code is necessary

Master Cont.

- HDFS can replicate data to be local if necessary for scheduling
- Because our nodes are (or at least should be) deterministic
  - The Master can restart failed nodes
    - Nodes should have no side effects!
  - If a node is the last step, and is completing slowly, the master can launch a second copy of that node
    - This can be due to hardware isuses, network issues, etc.
    - First one to complete wins, then any other runs are killed

Writables

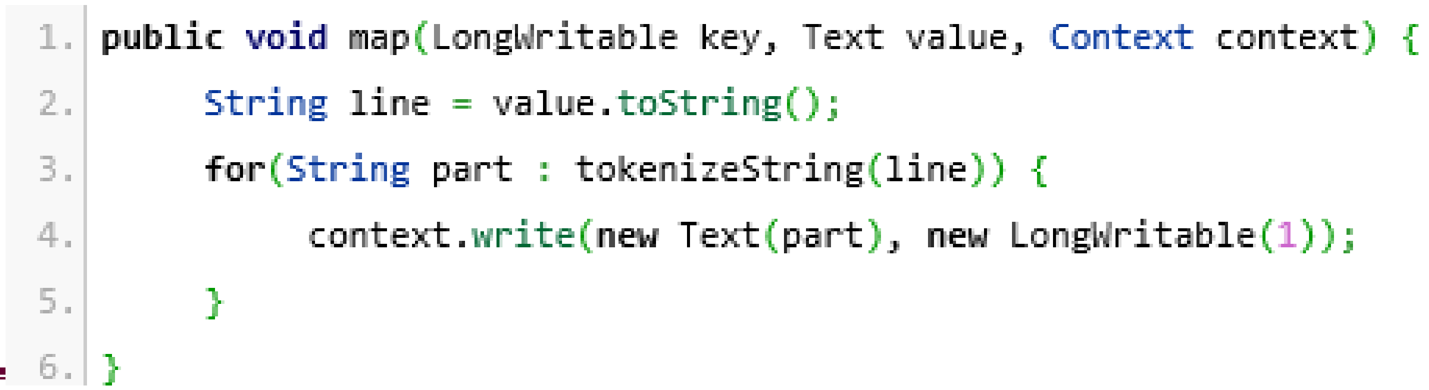
- Are types that can be serialized / deserialized to a stream
- Are required to be input/output classes, as the framework will serialize your data before writing it to disk
- User can implement this interface, and use their own types for their input/output/intermediate values
- There are default for basic values, like Strings, Integers, Longs, etc.
- Can also handle store, such as arrays, maps, etc.
- Your application needs at least six writables
    - 2 for your input
    - 2 for your intermediate values (Map <-> Reduce)
    - 2 for your output

Ví dụ: Mapper Code

Our input to our mapper is <LongWritable, Text>
The key (the LongWritable) can be assumed to be the position in the document our input is in. This doesn’t matter for this example.

Our output is a bunch of <Text, LongWritable>. The key is the token, and the value is the count. This is always 1.

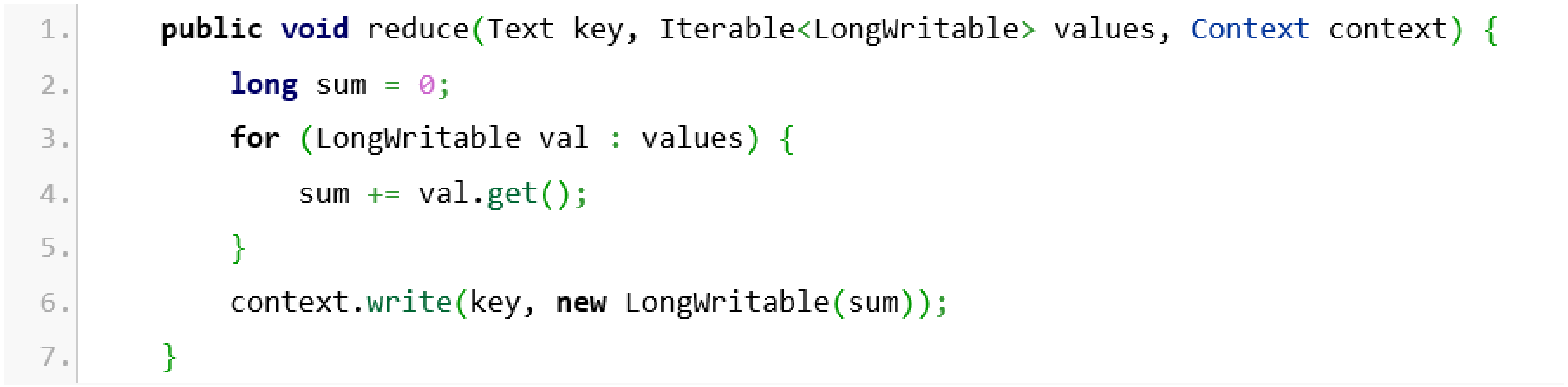
For the purpose of this demonstration, just assume Text is a fancy String, and LongWritable is a fancy Long. In reality, they’re just the Writable equivalents.



```java
public void map(LongWritable key, Text value, Context context) {
      String line = value.toString();
      for(String part : tokenizeString(line)) {
            context.write(new Text(part), new LongWritable(1));
      }
}
```

Ví dụ: Reducer Code

- Our input is the output of our Mapper, a <Text, LongWritable> pair
- Our output is still a <Text,LongWritable>, but it reduces N inputs for token T, into one output <T, N>



```java
    public void reduce(Text key, Iterable<LongWritable> values, Context context) {
        long sum = 0;
        for (LongWritable val : values) {
            sum += val.get();
        }
        context.write(key, new LongWritable(sum));
    }
```

Combiner Code

- Do we need a combiner?
  - No, but it reduces bandwidth.

- Our reducer can actually be our combiner in this case though!

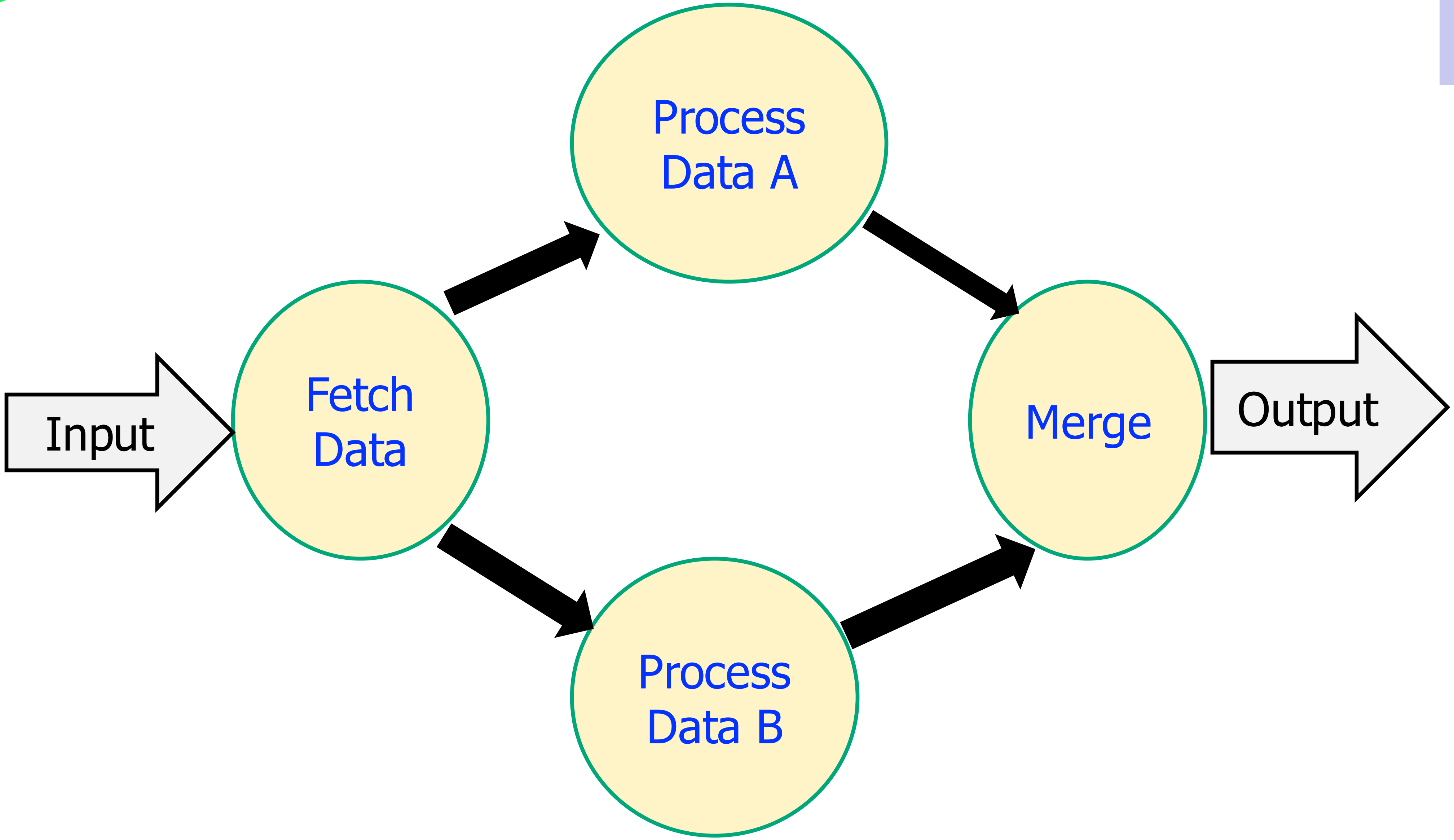
That's it!

- All that is needed to run the above code is an extremely simple runner class.
  - Simply specifies which components to use, and your input/output directories

Quy trình làm việc

- Hadoop YARN: chu kỳ có hướng
- Oozie vẽ biểu đồ các nút

Handling Data By Type



Chú ý

- MapReduce cung cấp một cách đơn giản để mở rộng mô hình ứng dụng của bạn.
- Mở rộng quy mô cho nhiều máy hơn thay vì mở rộng quy mô.
- Dễ dàng mở rộng quy mô từ 1 -> hàng nghìn máy
- Khả năng chịu lỗi và hiệu suất cao
- Điều chỉnh trường hợp sử dụng phù hợp với mô hình.

HẾT