Lecture 3

Iterators

Iterators

- A fast, safe, ‘lazy’ way to work with data structures.
- **Example:** Let’s take a set of values and double it.

```
fn main() {
 let i = [5, 7, 12];
   let iter = i.into_iter();
   let iter_mapped = iter.map(|x| x * 2);
   let out = iter_mapped.collect::<Vec<i32>>();
   println!("{:?}", out);
}
// Outputs:
// [10, 14, 24]
```

Observing Laziness

- Iterators are lazy.
- use .inspect() calls to observe evaluation.

```
fn main() {
 let i = [1, 2, 3];
  let iter = i.into_iter();
  let iter_mapped = iter.inspect(|&x|
      println!("Pre:\t{}", x))
      .map(|x| x * 10) // This gets fed into...
      .inspect(|&x|println!("First:\t{}",x))
      .map(|x| x + 5)   // ... This.
      .inspect(|&x|println!("Second:\t{}",x));
  iter_mapped.collect::<Vec<i32>>();
}
```

Observing Laziness

```
// Outputs:
Pre:    1
First:  10
Second: 15
Pre:    2
First:  20
Second: 25
Pre:    3
First:  30
Second: 35
```

- **.map()** is only evaluated as iterator is moved through.
- **.inspect()** requires a *&x* to prevent mutation.

Example: infinite or cycling iterators.

```
fn main() {
 let i = [10, 42, 93];
   let iter_cycled = i.into_iter().cycle();
   let out = iter_cycled.take(9).collect::<Vec<&i32>>();
   println!("{:?}", out);
}
// Outputs:
//[10, 42, 93, 10, 42, 93, 10, 42, 93]
```

Iterating over HasMaps

```
use std::collections::HashMap;
fn main() {
 let mut i = HashMap::<i32, i32>::new();
 i.insert(1, 10); i.insert(2, 20);  i.insert(3, 30);

 let iter = i.into_iter();
 let iter_mapped = iter.map(|(key, value)|{
            return (key, value * 10);});
 let out = iter_mapped.collect::<Vec<_>>();
 println!("{:?}", out);
}
//Outputs:
//[(3, 300), (2, 200), (1, 100)]
```

Filter, Map, Reduce... Wait... Fold

- Similar to JS, Rust has .filter(), .map(), .reduce().
- .reduce() is called .fold().

```
fn main() {
      let i = 1..10; //start..end -> start ≤ x < end
      let out = i
      .filter(|&item| item % 2 == 0) // Keep Evens
      .map(|item| item * 2) // Multiply by two.
      .fold(0, |accumulator, item| accumulator +
item);
      println!("{}", out);
}
// Outputs:
// 40
```

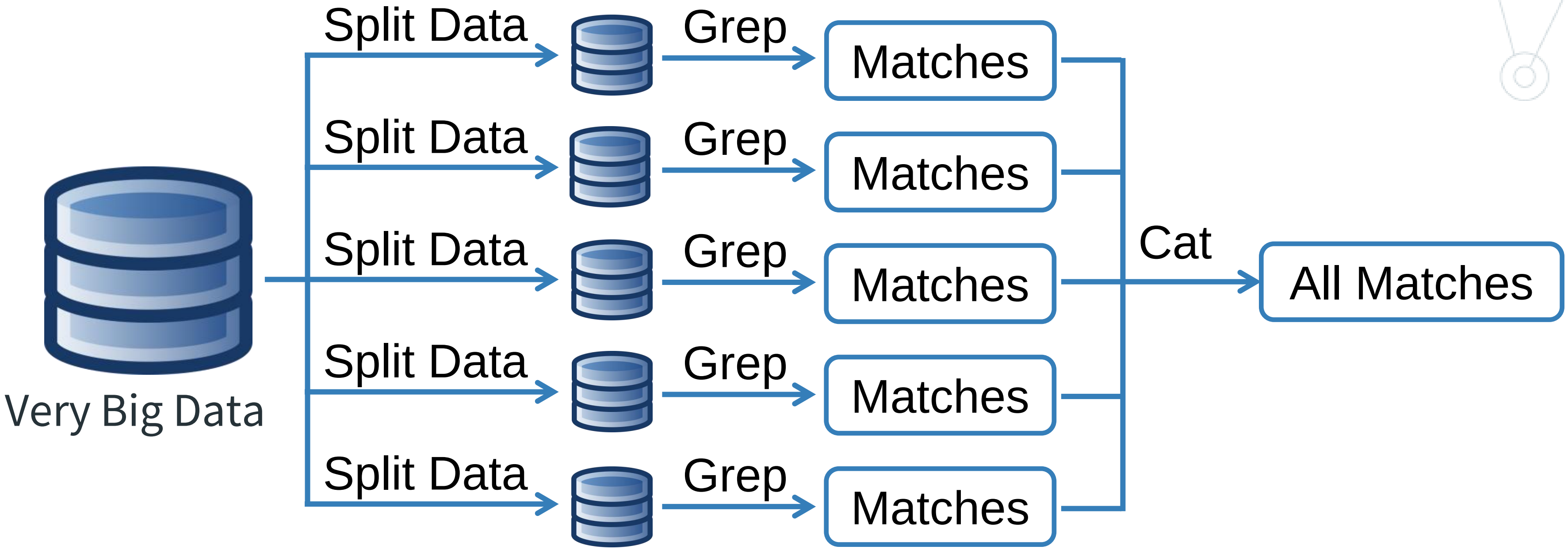
ECE 522

MapReduce

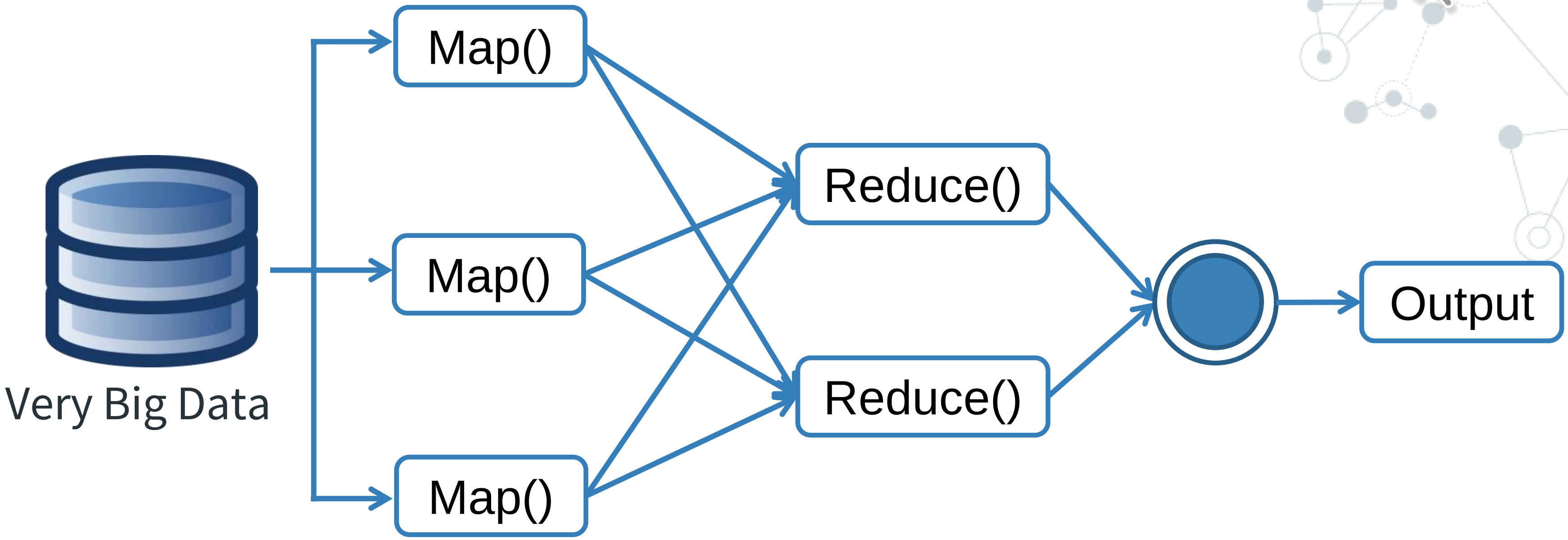
MapReduce

- MapReduce is a programming model that allows us to perform parallel and distributed processing on huge data sets.

Traditional Way



What is MapReduce?



- Consists of two distinct tasks – Map and Reduce.
- In the map job, data is processed to produce key-value pairs as intermediate outputs.
- The reducer receives the key-value pair from multiple map jobs and aggregates those pairs into a smaller set of pairs.

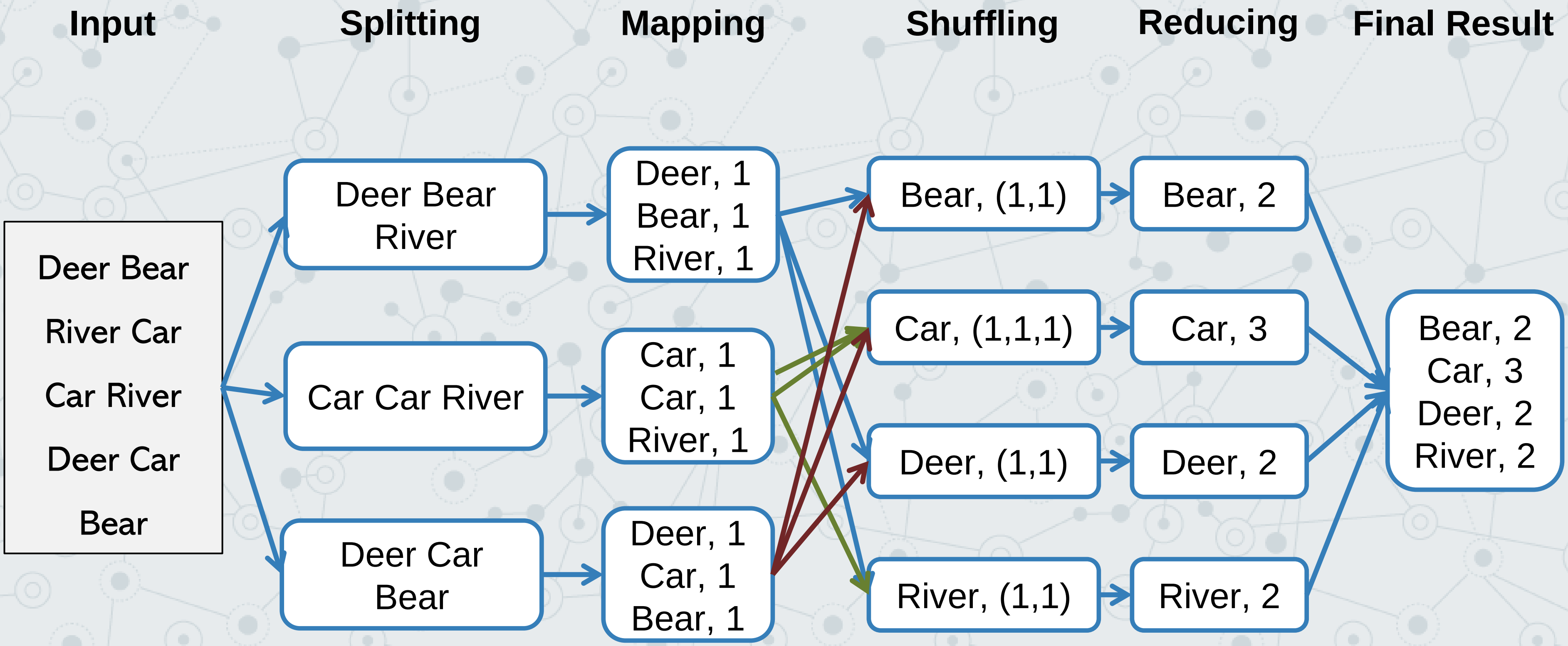
A Word Count Example

- **Input:** a text file called *example.txt* whose contents are as follows:

```
Deer, Bear, River, Car, Car, River, Deer, Car and Bear
```

- **Task:** perform a word count on the sample.txt using MapReduce to find the unique words and the number of occurrences of those unique words.

A Word Count Example


Input
Splitting
Mapping
Shuffling
Reducing
Final Result
Deer Bear
River Car
Car River
Deer Car
Bear
Deer Bear River
Car Car River
Deer Car Bear
Deer, 1
Bear, 1
River, 1
Car, 1
Car, 1
River, 1
Deer, 1
Car, 1
Bear, 1
Bear, (1,1)
Car, (1,1,1)
Deer, (1,1)
River, (1,1)
Bear, 2
Car, 3
Deer, 2
River, 2
Bear, 2
Car, 3
Deer, 2
River, 2