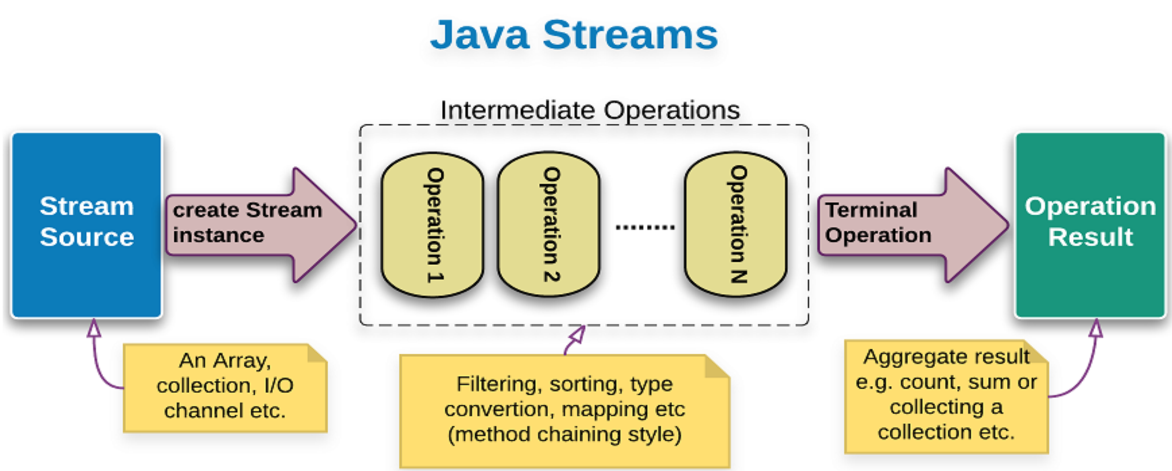
# Java Streams

# Streams

- A stream represents a sequence of elements and supports different kind of operations to perform computations upon those elements.
- Streams let you group and process data.



# Streams

- For example:
  - You might want to create a collection of banking transactions to represent a customer's statement. Then, you might want to process the whole collection to find out how much money the customer spent.

```
public class Transaction {
        private double value = 0;
        private int type;
        private int id;
        public Transaction(int id,int type, double v) {
        this.type = type;
        value = v;
        this.id = id;
}
```

# Java 8 Stream

- *Java 8 added Stream* that lets you process data in a declarative way.

- Furthermore, streams can leverage multi-core architectures without you having to write a single line of multithread code.
  - parallel streams

# Stream

- **Sequence of elements**
  - A stream provides an interface to a sequenced set of values of a specific element type.
  - Streams don’t actually store elements; they are computed on demand.
- **Source**
  - Streams consume from a data-providing source such as collections, arrays, or I/O resources.
- **Aggregate operations**
  - Streams support SQL-like operations and common operations from functional programing languages, such as filter, map, reduce, find, match, sorted, and so on.

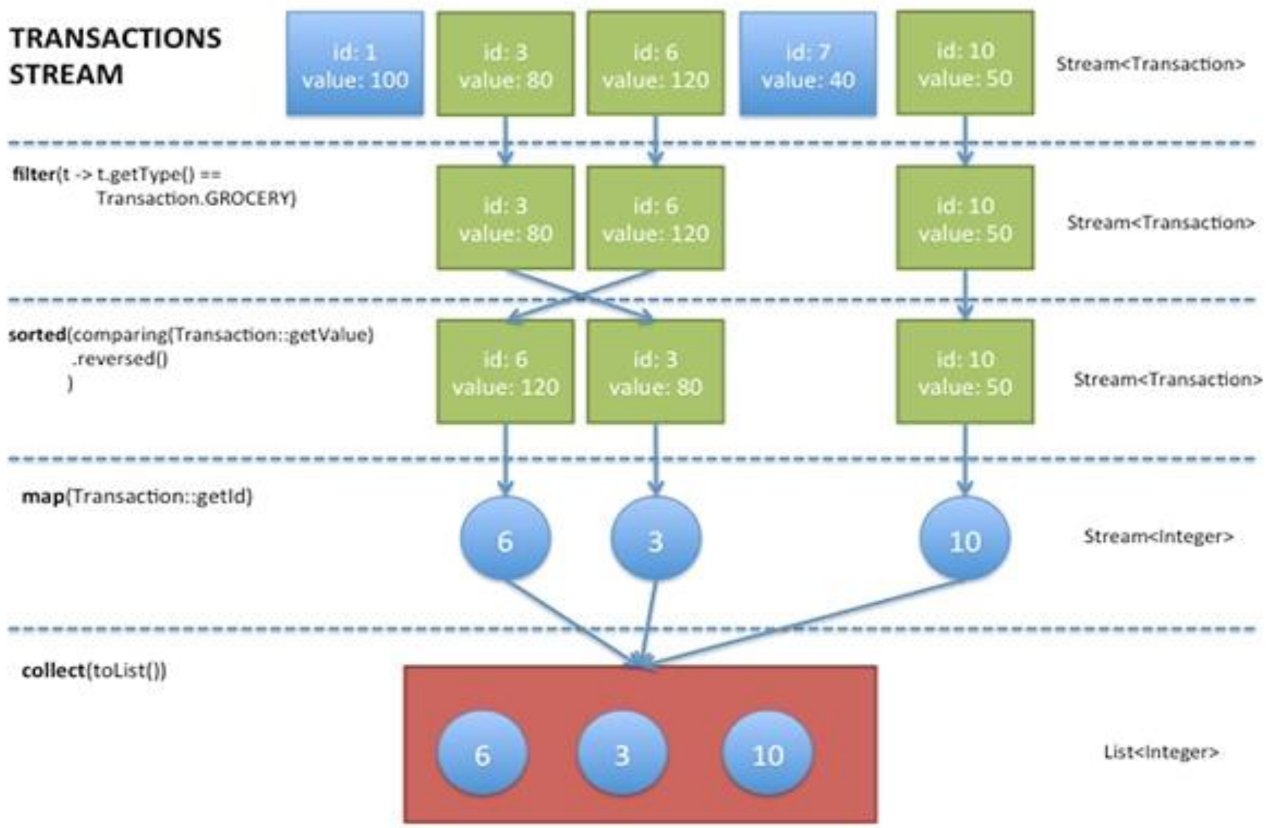
# Stream

- Two fundamental characteristics that make them very different from collection operations:

- **Pipelining:** Many stream operations return a stream themselves. Operations to be chained to form a larger pipeline. This enables certain optimizations, such as *laziness* and *short-circuiting.*

- **Internal iteration:** In contrast to collections, which are iterated explicitly (*external iteration*), stream operations do the iteration behind the scenes for you.

# Stream Pipeline



# Streams vs Collections

- The difference between collections and streams has to do with *when* things are computed.

- A collection is an in-memory data structure, which holds all the values that the data structure currently had.

- Every element in the collection has to be computed before it can be added to the collection.

- In contrast, a stream is a conceptually fixed data structure in which elements are computed on demand.

# Streams vs Collections

- Using the Collection interface requires iteration to be done by the user
  - Foreach loop: external iteration.

- The Streams library uses internal iteration— it does the iteration for you and takes care of storing the resulting stream value somewhere; you merely provide a function saying what's to be done.

# Streams vs Collections

**Collections:**

```
List<String> transactionIds = new ArrayList<>();
for(Transaction t: transactions){
    transactionIds.add(t.getId());
}
```

**Streams:**

```
List<Integer> transactionIds =
        transactions.stream()
        .map(Transaction::getId)
        .collect(toList());
```

- Collections explicitly iterates the list of transactions sequentially to extract each transaction ID and add it to an accumulator.
- Streams builds a query, map extracts the transaction IDs and the collect converts the resulting Stream into a List.

# Stream Operations

- *intermediate operations*
  - filter, sorted, and map
  - Can be connected together to form a pipeline
  - Intermediate operations do not perform any processing until a terminal operation is invoked on the stream pipeline. They are "lazy".
- *terminal operations*
  - Collect
  - Closes a pipeline

# Lazy Evaluation

```
List<Integer> numbers = Arrays.asList(1,2,3,4,5,6,7,8);
List<Integer> t = numbers.stream()
    .filter(n -> {print("filtering " + n);
                  return n % 2 == 0; })
    .map(n -> { print("mapping " + n);
                  return n * n; })
    .limit(2)
    .collect(toList());
```

output:

```
filtering 1
filtering 2
mapping 2
filtering 3
filtering 4
mapping 4
```

Computes two even square numbers from a given list of numbers.

# Quiz

```
List<Integer> numbers =  Arrays.asList(1,2,3,4,5,6,7,8);
List<Integer> mystery =
       numbers.stream()
      .filter(n ->  n % 2 == 0)
      .map(n ->   n * 2)
      .collect(Collectors.toList());
```

**The value of List mystery:**

A. [1, 2, 3, 4]
B. [4, 8, 12, 16]
C. [2, 4, 6, 8]
D. 40

# Quiz

```
List<Integer> numbers =  Arrays.asList(1,2,3,4,5,6,7,8);
List<Integer> mystery =
       numbers.stream()
      .filter(n ->  n % 2 == 0)
      .map(n ->   n * 2)
      .collect(Collectors.toList());
```

**The value of List mystery:**

A. [1, 2, 3, 4]
B. [4, 8, 12, 16]
C. [2, 4, 6, 8]
D. 40

# Stream Operations: Filtering

- Filter elements from a stream:
- filter(Predicate)
  - Takes a predicate (java.util.function.Predicate) as an argument and returns a stream including all elements that match the given predicate
- Distinct
  - Returns a stream with unique elements (according to the implementation of equals for a stream element)
- limit(n)
  - Returns a stream that is no longer than the given size n
- skip(n)
  - Returns a stream with the first n number of elements discarded

# Stream Operations: Filtering

- Distinct
  - Returns a stream with unique elements (according to the implementation of equals for a stream element)

```
List<String> list = Arrays.asList(
                 "A", "B", "C", "D", "A", "B", "C");

// Get collection without duplicate i.e. distinct only
List<String> distinctElements =
        list.stream().distinct()
        .collect(Collectors.toList());

//Let's verify distinct elements
System.out.println(distinctElements);

Output is:[A, B, C, D]
```

# Stream Operations: Filtering

- limit(n)
  - Returns a stream that is no longer than the given size n

```
Stream.of(1,2,3,4,5,6,7,8,9)
        .peek(x->System.out.print("\nA"+x))
        .limit(3)
        .peek(x->System.out.print("B"+x))
        .forEach(x->System.out.print("C"+x));
```

# Stream Operations: Filtering

- limit(n)
  - Returns a stream that is no longer than the given size n

```
Stream.of(1,2,3,4,5,6,7,8,9)
        .peek(x->System.out.print("\nA"+x))
        .limit(3)
        .peek(x->System.out.print("B"+x))
        .forEach(x->System.out.print("C"+x));
```

A1B1C1
A2B2C2
A3B3C3

# Stream Operations: Filtering

- skip(n)
  - Returns a stream with the first n number of elements discarded

```
Stream.of(1,2,3,4,5,6,7,8,9)
    .peek(x->System.out.print("A"+x))
    .skip(6)
    .peek(x->System.out.print("B"+x))
    .forEach(x->System.out.println("C"+x));
```

# Stream Operations: Filtering

- skip(n)
  - Returns a stream with the first n number of elements discarded

```
Stream.of(1,2,3,4,5,6,7,8,9)
     .peek(x->System.out.print("A"+x))
     .skip(6)
     .peek(x->System.out.print("B"+x))
     .forEach(x->System.out.println("C"+x));
```

A1A2A3A4A5A6A7B7C7
A8B8C8
A9B9C9

# Quiz: What is the output?

```
Stream.of(1,2,3,4,5,6,7,8,9)
    .peek(x->System.out.print("A"+x))
    .limit(4)
    .skip(2)
    .forEach(x->System.out.print("B"+x));
```

A. A1B1A2B2A3B3A4B4
B. A1A2A3B3A4B4
C. A3B3A4B4
D. A1A2A3A4B3B4

# Quiz: What is the output?

```
Stream.of(1,2,3,4,5,6,7,8,9)
     .peek(x->System.out.print("A"+x))
     .limit(4)
     .skip(2)
     .forEach(x->System.out.print("B"+x));
```

A. A1B1A2B2A3B3A4B4
B. A1A2A3B3A4B4
C. A3B3A4B4
D. A1A2A3A4B3B4

# Quiz: What is the output?

```
Stream.of(1,2,3,4,5,6,7,8,9)
    .peek(x->System.out.print("A"+x))
    .limit(2)
    .skip(4)
    .forEach(x->System.out.print("B"+x));
```

A. A1A2A3B3A4B4
B. A1A2
C. A1B1A2B2
D. A1A2B1B2

# Quiz: What is the output?

```
Stream.of(1,2,3,4,5,6,7,8,9)
    .peek(x->System.out.print("A"+x))
    .limit(2)
    .skip(4)
    .forEach(x->System.out.print("B"+x));
```

A. A1A2A3B3A4B4
B. A1A2
C. A1B1A2B2
D. A1A2B1B2

# Stream Operations: Finding and matching

- Determining whether some elements match a given property.
  - anyMatch
  - allMatch
  - noneMatch
- They all take a predicate as an argument and return a boolean as the result
  - For example, check if all elements in a stream of transactions have a value higher than 100.

```
boolean expensive = transactions.stream()
        .allMatch(t-> t.getValue() > 100);
```

# Stream Operations: findFirst, findAny

- Retrieves arbitrary elements from a stream.
- They can be used in conjunction with other stream operations such as filter.
- Both `findFirst` and `findAny` return an Optional object

```
Optional<Transaction> = transactions.stream()
  .filter(t-> t.getType() == Transaction.GROCERY)
  .findAny();
```

# Optional<T> Class

- The Optional<T> class (java.util .Optional) is a container class to represent the existence or absence of a value.
- It is possible that findAny doesn't find any transaction of type grocery.
- The Optional class contains several methods to test the existence of an element.

```
transactions.stream()
   .filter(t-> t.getType()== Transaction.GROCERY)
   .findAny()
   .ifPresent(System.out::println);
```

# Stream Operations: Mapping

- takes a function (java.util.function.Function) as an argument to project the elements of a stream into another form.
- The function is applied to each element, “mapping” it into a new element.

```
List<String> words = Arrays.asList(
                  "Oracle", "Java", "Magazine");
List<Integer> wordLengths = words.stream()
      .map(String::length)
      .collect(toList());
```

# Stream Operations: Reducing

- Repeatedly applies an operation (for example, adding two numbers) on each element until a result is produced.
- It's often called a *fold operation* in functional programming.

```
int sum = 0;
for(int x : numbers) { sum += x; }
```

**vs**

```
int sum = numbers.stream().reduce(0, (a,b) -> a + b);
```

- A BinaryOperator<T> to combine two elements and produce a new value

# Stream Operations: Reducing

- The reduce method essentially abstracts the pattern of repeated application.
- Other queries such as “calculate the product” or “calculate the maximum” become special use cases of the reduce method.

**Product:**

```
int product = numbers.stream()
        .reduce(1, (a, b) -> a * b);
```

**Max:**

```
int product = numbers.stream()
        .reduce(1, Integer::max);
```

# IntStream, DoubleStream, LongStream

- Specialize the elements of a stream to be int, double, and long.
- to convert a stream to a specialized version are mapToInt, mapToDouble, and mapToLong.
- return a specialized stream instead of a Stream<T>.

```
int statementSum = transactions.stream()
        .mapToInt(Transaction::getValue)
        .sum();
```

# Range

- range and rangeClosed
  - Static methods of IntStream, DoubleStream, and LongStream

- For examples: use rangeClosed to return a stream of all odd numbers between 10 and 30.

```
IntStream oddNumbers = IntStream
        .rangeClosed(10,30)
        .filter(n -> n % 2 == 1);
```

# Building Streams

- From arrays
  - Stream.of //factory method
  - Arrays.stream

```
Stream<Integer> numbersFromValues = Stream.of(1,2,3,4);
int[] numbers = {1,2,3,4};
IntStream numbersFromArray = Arrays.stream(numbers);
```

# Building Streams

Convert a file into a stream of lines

- Files.lines

```
long numberOfLines =
Files.lines(Paths.get("file.txt"),Charset.defaultCharset())
.count();
```

# Infinite streams

- Because of lazy evaluation, infinite stream is possible
- `Stream.iterate` and `Stream.generate`

```
Stream<Integer> nums = Stream.iterate(0,n->n+10);
```

Process infinite stream:

```
nums.limit(5).forEach(System.out::println);
// 0, 10, 20, 30, 40
```

# Stream Example

```
List<Integer> transactionsIds = transactions.parallelStream()
   .filter(t -> t.getType() == Transaction.GROCERY)

   .sorted(comparing(Transaction::getValue).reversed())

   .map(Transaction::getId)

   .collect(toList());
```

# Parallel Streams

- You can execute streams in serial or in parallel.
- When a stream executes in parallel, the Java runtime partitions the stream into multiple substreams.
- Aggregate operations iterate over and process these substreams in parallel and then combine the results.

# Parallel Streams

```
double average =
   roster.parallelStream()
  .filter(p -> p.getGender() == Person.Sex.MALE)
  .mapToInt(Person::getAge)
  .average()
  .getAsDouble();
```

# Quiz 3: What is the output?

```
public class GFG {
   public static void main(String[] args){
     List<Integer> list = Arrays.asList(0,2,4,6);
     list.stream().peek(System.out::print}
}
```

A. 0246
B. 0
C. No output. Peek is an intermediate operator
D. 0,2,4,6

# Quiz 4: What is the output?

```
public class GFG {
   public static void main(String[] args){
     List<Integer> list = Arrays.asList(0,2,4,6);
     list.stream().peek(System.out::print}//does not print
     long c =list.stream().peek(System.out::print).count();
     System.out.println(c);
}
```

A. 02464
B. 4
C. No output
D. 0,2,4,6,4

# Quiz 5: What is the output

```
IntStream stream = IntStream.of(1, 2, 3,4,5,6,7,8);
List<Integer> test =  stream.skip(2)
                     .skip(3).boxed()
                      .collect(Collectors.toList());

System.out.println(test);
```

A. [4,5,6,7,8]
B. [6, 7, 8]
C. [3,4,5,6,7,8]
D. [ ]

# Quiz 6: What is the output?

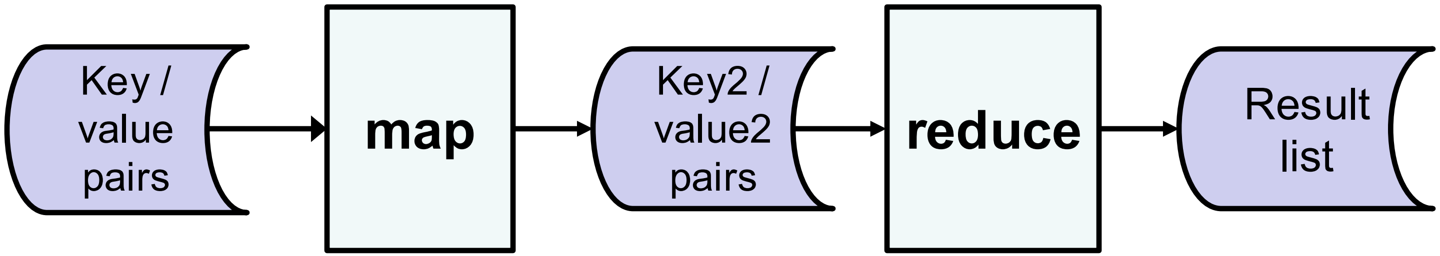
```
IntStream stream = IntStream.of(1,2,3,4,5);
long a = stream.skip(2).count();
long b = stream.skip(3).sum();
System.out.println(a +"," + b);
```

A. 3,9
B. 12
C. Error. count closes the stream
D. 0

# This Class So Far

- *Concurrent* programming in Java
- Exploiting *parallelism* to improve performance
- *Multi-process* programming in Java using actors (akka)
- Next topic: *MapReduce*
  - A "programming model" for processing large data sets in parallel on a cluster
  - Developed by Google researchers in early 2000s
  - Key features
    - Conceptual simplicity
    - Scalability and fault-tolerance of operations

# MapReduce, Conceptually




- Input data consists of key/value pairs
  - E.g. key could be a URL: “www.cs.umd.edu”
  - Value could be the .html code in the file associated with the URL
- MapReduce developer specifies
  - “map” function to produce intermediate set of (possibly) new-key, new-value pairs
  - “reduce” function to convert intermediate data into final result

# What?

- Think of data processed by MapReduce as "tables"
  - The table has two columns: one for keys, the other for values
  - Each key/value pair in the data set corresponds to a row in the table
- So:
  - "map" converts input table into a new, intermediate table
  - "reduce" constructs a new table that aggregates data in the intermediate table

| Key | Value |
|---|---|
| $key_1$ | $value_1$ |
| $key_2$ | $value_2$ |
| … | |
| $key_n$ | $value_n$ |

# Example: Word Counting

- Suppose we want to give a MapReduce application giving an occurrence count for each word in a list of files
  - Input table: file name / file contents pairs (both strings, the second being much longer!)
  - Final table produced by reduce: word / int pairs, where each int is the # of occurrences of the word in the documents
- How do we do this using MapReduce?

# Word Counting: map

*map* converts individual row (file name, file contents) into collection of (word, “1”) rows

| File name (key) | Contents (value) |
|---|---|
| constitution.txt | “We the people …” |
| decl_ind.txt | “When, in the course…” |
| … | |

*map*

| Word (key2) | Count (value2) |
|---|---|
| “When” | 1 |
| “in” | 1 |
| “the” | 1 |
| “course” | 1 |
| … | |

# Word Counting: reduce

*reduce* takes all rows with a given word (key2) and sums the counts (value2), yielding (at most!) one row in output table

| Word (key2) | Count (value2) |
|---|---|
| … | |
| "the" | 1 |
| "the" | 1 |
| … | |
| "the" | 1 |
| … | |

reduce

| Word (key2) | Count (value2) |
|---|---|
| … | |
| "the" | 15 |
| … | |

# Other Applications of MapReduce

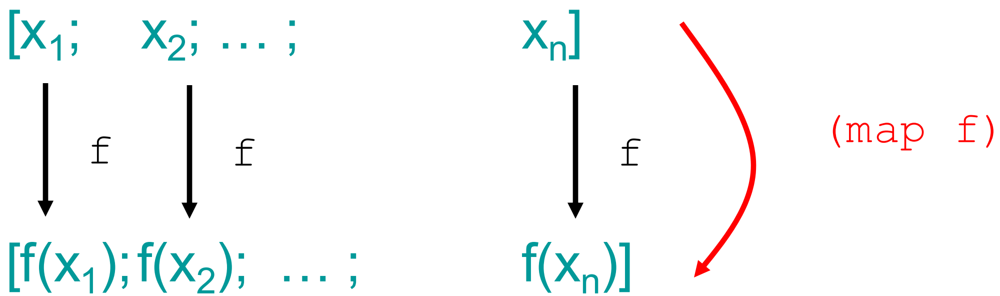
- Distributed grep
- Count of URL access frequency
- “Web-link graph reversal”: compute all URLs with a link to each of a given list of URLs
- Distributed sort
- “Inverted index”: given list of documents, produce output giving, for each word, the documents it appears in
- Used by 1000s of organizations around the world, including Amazon, Google, Yahoo, …

# Foundations of MapReduce

- MapReduce is based on concepts from *functional programming*
  - "map" in functional languages (e.g. OCaml!) converts a function over values to a function mapping lists to lists
    - Given list, (map f) applies f to each element in the list
    - The list of results is then returned
  - "fold" takes a seed / function value as input, returns a function mapping lists to single values as output
    - Actually, two versions: "left" and "right"
    - Point of both is to convert list to single value
- So?
  - Functional languages do not modify variables
  - Mapping can be computed in parallel!
  - MapReduce uses a variant of "fold"; details later

# Functional Map

- Suppose f is a function
- Then (map f) is a new function on lists:




- The $f(x_i)$ can be computed in parallel!
  - The $x_i$ do not share state
  - f cannot modify its arguments

# Map Examples in OCaml

```
# let add1 x = x+1;;
val add1 : int -> int = <fun>
# let g = List.map add1;;
val g : int list -> int list = <fun>
# g [1;2;3];;
- : int list = [2; 3; 4]
# let double x = [x;x];;
val double : 'a -> 'a list = <fun>
# let h = List.map double;;
val h : '_a list -> '_a list list = <fun>
# h [1;2;3];;
- : int list list = [[1; 1]; [2; 2]; [3; 3]]
```

# Functional Fold (Left)

- Suppose f is a *binary* function, s is a value
- Then (fold_left f s) is a function that “iteratively applies” f over lists to produce a single value

  (fold_left f s) [$x_1$; $x_2$; … $x_n$] =

  f ( … f ( f(s, $x_1$), $x_2$ ) …, $x_n$)
- E.g. if f x y = x+y, s = 0, then

  (fold_left f 0) [1;2;3] = ((0+1) + 2) + 3 = 6

# Fold (left) Examples in OCaml

```
# let sum x y = x+y;;
val sum : int -> int -> int = <fun>
# let h = List.fold_left sum 0;;
val h : int list -> int = <fun>
# h [1;2;3];;
- : int = 6
# let prefix tl hd = hd::tl;;
val prefix : 'a list -> 'a -> 'a list = <fun>
# let k = List.fold_left prefix [];;
val k : '_a list -> '_a list = <fun>
# k [1;2;3];;
- : int list = [3; 2; 1]
```

# MapReduce, Logically

- Assumption: input data for MapReduce applications consists of lists of (key, value) pairs (i.e. tables)
- A MapReduce application contains:
  - A "mapper function" converting single (key, value) pairs (i.e. single rows in the old table) to lists of (key2, value2) pairs (i.e. multiple rows in the new table)
  - A "reducer function" converting pairs of form (key2, value2 list) to a list of values (i.e. reducer aggregates data associated to key2 in the intermediate table)
- The MapReduce framework does the following
  - Apply "mapper" to the input data
  - Glue together the resulting lists into a single list of (key2, value2) pairs
  - Rearrange this list into a list (key2, value2 list) pairs, where each distinct key2 appears once
  - Applying "reducer" to each element in the new list
  - Return the aggregate results

# Hadoop

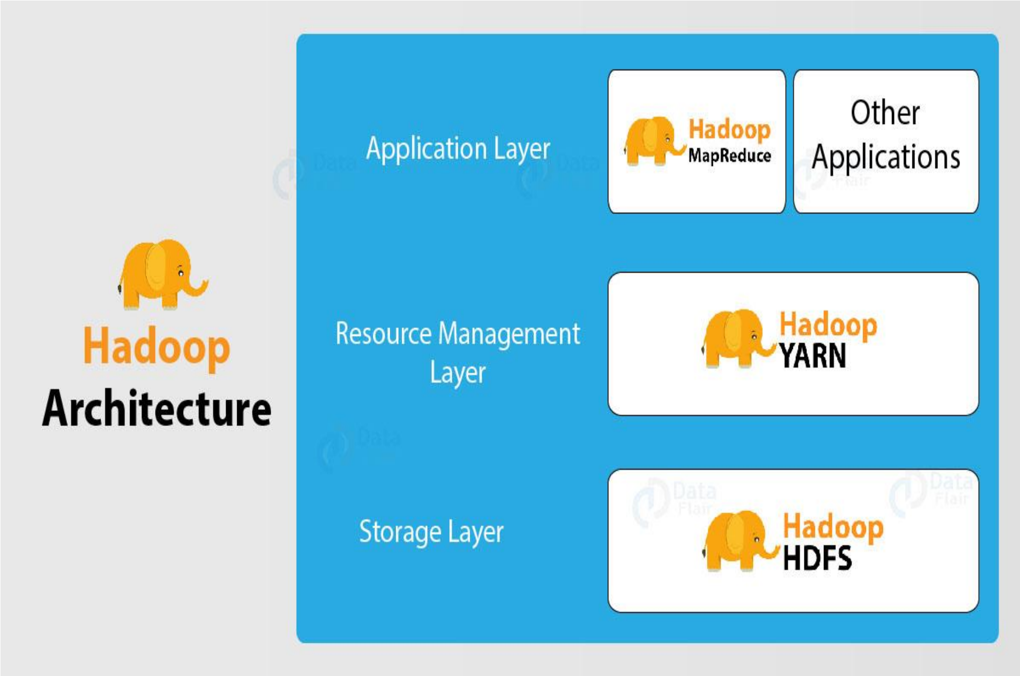
- An open-source implementation of MapReduce
- Design desiderata
  - Performance: support processing of huge data sets (millions of files, GB to TB of total data) by exploiting parallelism, memory within computing clusters
  - Economics: control costs by using commodity computing hardware
  - Scalability: a larger the cluster should yield better performance
  - Fault-tolerance: node failure does not cause computation failure
  - Data parallelism: same computation performed on all data

# Cluster?

- Hadoop is designed to run on a cluster
  - Multiple machines, typically running Linux
  - Machines connected by high-speed local-area network (e.g. 10-gigabit Ethernet)
- Hardware is:
  - High-end (fast, lots of memory)
  - Commodity (cheaper than specialized equipment)

# Hadoop Architecture

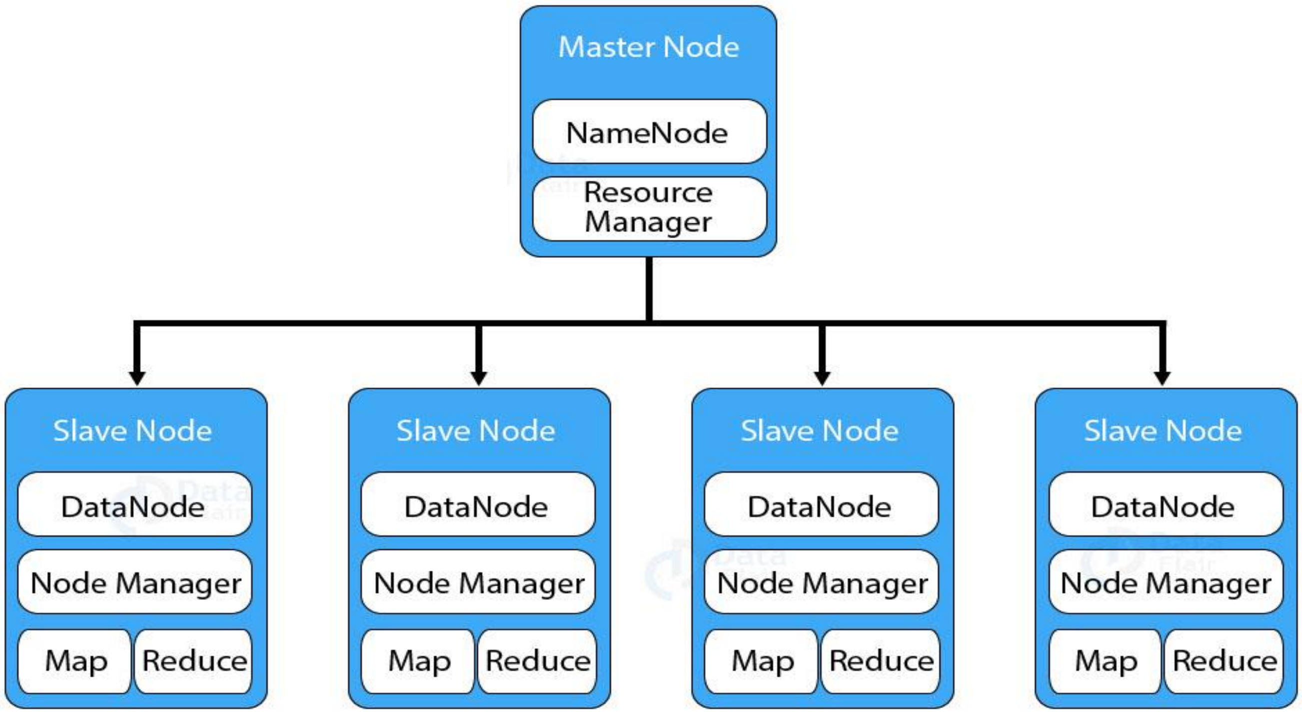
- Hadoop Architecture comprises three major layers.
  - HDFS (Hadoop Distributed File System)
  - Yarn
  - MapReduce



# Hadoop Architecture

- Hadoop has a master-slave topology
  - One master node and multiple slave nodes.
  - Master node’s function is to assign a task to various slave nodes and manage resources.
  - The slave nodes do the actual computing.
  - Slave nodes store the real data whereas on master we have metadata.

# Hadoop Architecture



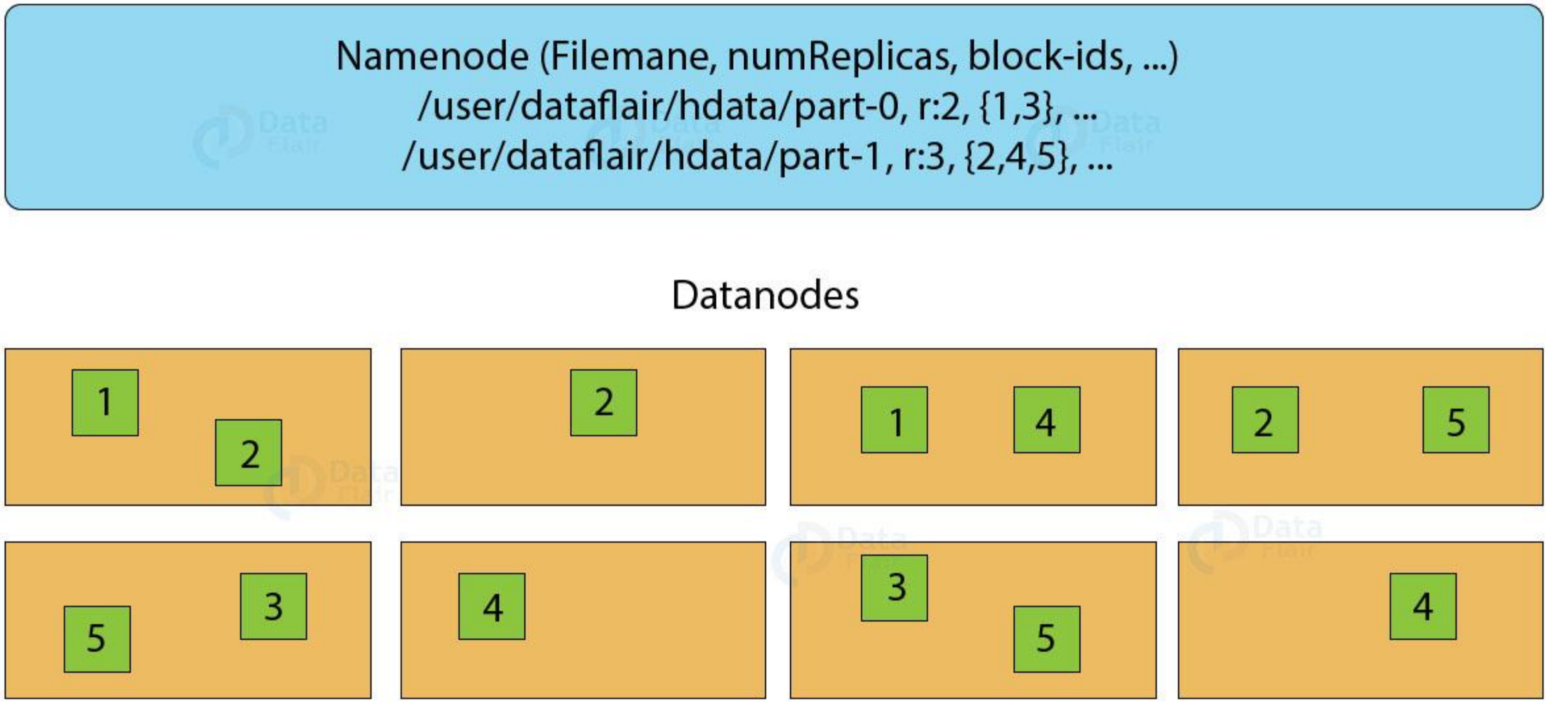
# Principles of Hadoop Design

- *Data is distributed* around network
  - No centralized data server
  - Every node in cluster can host data
  - Data is *replicated* to support fault tolerance
- *Computation is sent to data*, rather than vice versa
  - Code to be run is sent to nodes
  - Results of computations are aggregated at end
- *Basic architecture is master/worker*
  - Master, aka JobNode, launches application
  - Workers, aka WorkerNodes, perform bulk of computation

# Components of Hadoop

- MapReduce
  - Basic APIs in Java supporting MapReduce programming model
- Hadoop Distributed File System (HDFS)
  - Applications see files
  - Behind the scenes:  HDFS handles distribution, replication of data on cluster, reading, writing, etc.

# Block Replication



# Hadoop Execution: Startup

1. MapReduce library in user program splits input files into pieces (typically 16-64 MB), starts multiple copies of program on cluster
2. One copy is master; rest are workers. Work consists of map, reduce tasks
3. Master keeps track of idle workers, assigns them map / reduce tasks

[Discussion adapted from Ravi Mukkamala, “Hadoop: A Software Framework for Data Intensive Computing Applications”; Hadoop 1.2.1 “MapReduce Tutorial”]

# Hadoop Execution:  Map Task

1. Read contents of assigned input split
   Master will try to ensure that input split is "close by"
2. Parse input into key/value pairs
3. *Apply map operation* to each key/value pair; store resulting intermediate key/value pairs on local disk
4. File is sorted on output key, then partitioned based on key values
5. Locations of these files forwarded back to Master
6. Master forwards locations of files to relevant reduce workers
   - Which reduce workers get which files depends on *partition*
   - Partition assigns different key values to different reduce tasks

# Hadoop Execution: Reduce Task

1. Fetch input (files produced by map tasks and sent by master)
2. Sort input data by key
3. For each key, *apply reduce operation* to key / list of values associated with key
4. Write result in file (one output file / key, often, but configurable)
5. Return location of result files to Master

# Configuring MapReduce Execution

- Many configuration parameters to tune performance!
  - Number of maps
  - Number of reduces
  - Splitting of input
  - Sorting, partitioning
  - Etc.
- Hadoop MapReduce tutorial gives a starting point

  https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html

# Fault-Tolerance

- Big clusters = increased possibility of hardware failure
  - Disk crashes
  - Overheating
- Worker failure
  - Master pings worker periodically:  no response = worker marked as failed
  - Tasks assigned to failed worker added back into task pool for re-assignment
  - This works because *functional nature* of MapReduce ensures no shared state, while HDFS ensures *data is replicated* (so data hosted by failed node is still available)
- Master failure
  - Masters write checkpoint files showing intermediate progress
  - If master fails, a new master can be started from the last checkpoint
  - In practice:  job generally restarted

# Setting Up Hadoop

- Three possibilities
  - Local standalone (everything run in one process)
  - Pseudo-distributed (tasks run as separate processes on same machine)
  - Fully distributed (cluster computing)

  - Standalone usually used for development, debugging
  - Pseudo-distributed to ensure no shared memory, analyze performance bottlenecks
  - Fully-distributed used for final job runs

# Quiz

Which one of the following is true about Hadoop?

A. It is a distributed framework
B. The main algorithm used in it is Map Reduce
C. It runs with commodity hardware
D. All are true

# Quiz

Which type of data Hadoop can deal with is

A. Structured
B. Semi - structured
C. Unstructured
**D. All of the above**

# Quiz

Which among the following are the features of Hadoop

A. Open source
B. Fault-tolerant
C. High Availability
**D. All of the above**

# Quiz

What are the advantages of 3x replication schema in Hadoop

A. Fault tolerance
B. High availability
C. Reliability
D. **All of the above**

# Writing a Hadoop Application

- MapReduce
  - One class should extend Mapper<K1,V1,K2,V2>
    - K1, V1 are key/value classes for input
    - K2, V2 are key/value classes for output
  - Another should extend Reducer<K2,V2,K3,V3>
    - K2, V2 are key/value classes for inputs to reduce operation
    - K3, V3 are output key/value classes
- Main driver
  - Need to create an object in Job (Hadoop class) containing configuration settings for Hadoop application
  - Settings include input / output file formats for job, input file-slice size, key/value types, etc.
  - To run the job: invoke **`job.waitForCompletion(true)`**

# Implementing Mapper<K1,V1,K2,V2>

- Key function to implement:

```
public void map(K1 key, V1 value, Context c)
```

  - First two inputs are key / value pair, which map should convert into key2 / value2 pairs
  - "Context"?
    - Used to store key2 / value2 pairs produced by map
    - Context is a Hadoop class
    - To store a newly created key2 / value2 pair, invoke:

```
c.write (key2, value2);
```

- Hadoop takes care of ensuring that pairs written into context are provided to Reducer!

# Implementing Reducer<K2,V2,K3,V3>

- Key function to implement:

```
public void reduce(K2 key, Iterable<V2> values, Context c)
```

  - First args are key / list-of-values pair, which map should convert into (usually at most one) key3 / value3 pairs
  - Context argument used to store these key3 / value3 pairs!
    - Idea is same as for Mapper implementation!
    - To store a newly created key3 / value3 pair, invoke:

```
c.write (key3, value3);
```

- Hadoop takes care of ensuring that pairs written into context are made available for post-processing (i.e. sorting, writing into a file)

# Implementing main()

- Must create a `Job` object (Hadoop class)
  - Job constructor typically requires a `Configuration` argument
  - E.g.:

```
Configuration conf = new Configuration ();
Job job = new Job(conf);
```

- `Job` object must be configured!
  - `Key, Value` classes must be set
  - Mapper, Reducer classes (your implementation!) must be specified
  - Input / output formatting must be given
  - Paths to input files, output files must be given

# Sample main() (from WordCount.java)

```
public static void main(String[] args) throws Exception {
  // Set up and configure MapReduce job.
  Configuration conf = new Configuration ();
  Job job = new Job(conf);
  job.setJobName("WordCount");
  job.setJarByClass(WordCount.class);   // In Eclipse this will not create JAR file

  // Set key, output classes for the job (same as output classes for Reducer)
  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);

  // Set Mapper and Reducer classes for the job. (Combiner often not needed.)
  job.setMapperClass(MapClass.class);
  job.setReducerClass(ReduceClass.class);
```

# Sample main() (cont.)

```
  // Sets format of input files.  "TextInputFormat" views files as a sequence of lines.
  job.setInputFormatClass(TextInputFormat.class);
  // Sets format of output files:  here, lines of text.
  job.setOutputFormatClass(TextOutputFormat.class);

 // Set paths for location of input, output.  Note former is assumed to be
 // initial command-line argument, while latter is second.  No error-checking
 // is performed on this, so there is a GenericOptionsParser warning when run.

  TextInputFormat.setInputPaths(job, new Path(args[0]));
  TextOutputFormat.setOutputPath(job, new Path(args[1]));
 // Run job
  Date startTime = new Date();
  System.out.println("Job started: " + startTime);
  boolean success = job.waitForCompletion(true);
  if (success) {
    Date end_time = new Date();
    System.out.println("Job ended: " + end_time);
    System.out.println("The job took " + (end_time.getTime() - startTime.getTime()) /1000 + " seconds.");
  }
  else { System.out.println ("Job failed."); }
}
```

# Other Tools

- Apache Hive:
  - SQL-like interface to query data stored in various databases and file systems that integrate with Hadoop
- Apache Pig:
  - script.abstracts the programming from the Java MapReduce idiom into a notation which makes MapReduce programming high level,
- Hbase: wide-column database
  - modeled after Google's Bigtable and written in Java
  - Bigtable is a compressed, high performance, and proprietary data storage system built on Google File System
  - Id, {name, address,…}
  - Id,{location, services}

# Other Tools

- Zookeeper:
  - provides a distributed configuration service, synchronization service, and naming registry for large distributed systems.
- Spark
  - an open-source distributed general-purpose cluster-computing framework.
  - provides an interface for programming entire clusters with implicit data parallelism and fault tolerance.