**Rust:Data processing pipelines**

[[](https://medium.com/@technocrat21?source=post_page-----a56290483c22--------------------------------)](https://medium.com/@technocrat21?source=post_page-----a56290483c22--------------------------------)

[[CoderHack.com](https://medium.com/coderhack-com?source=post_page-----a56290483c22--------------------------------)](https://medium.com/coderhack-com?source=post_page-----a56290483c22--------------------------------)

[Amay B](https://medium.com/@technocrat21?source=post_page-----a56290483c22--------------------------------)

·

Follow

Published in

[CoderHack.com](https://medium.com/coderhack-com?source=post_page-----a56290483c22--------------------------------)

·

3 min read

·

Sep 15

22



Photo by [Scott Graham](https://unsplash.com/@homajob?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral)

A data processing pipeline refers to a set of interconnected stages that sequentially transform raw data into meaningful insights. Each stage in the pipeline builds upon the output of the previous stage.

Rust is suitable for data processing pipelines due to:

* Speed and efficiency: Rust compiles to fast, native code with no runtime or garbage collector.
* Safety and correctness: Rust’s ownership and borrowing rules ensure thread safety and prevent data races, making it suitable for concurrent data processing.
* Robust tooling: Cargo, rustup, and Crates.io provide a great experience for managing dependencies, building, and distributing Rust code.

Some core Rust features useful for data processing pipelines are:

* Traits: Use traits to define generic interfaces that can be implemented for multiple types.
* Pattern matching: Use pattern matching to elegantly destructure enums and structs.
* ADTs: Use enums to represent choice/option types in your data.
* Iterators: Use iterators to manipulate collections in a lazy, zero-cost abstractions manner.
* Closures: Use closures to concisely define callback functions that can capture their environment.

**2. Reading data**

Use the csv crate to read CSV files:

use csv;  
  
let mut rdr = csv::Reader::from\_path("data.csv");  
for record in rdr.records() {  
 let record = record?;  
 // Use `record`  
}

Use serde\_json to read JSON:

use serde\_json;  
  
let f = File::open("data.json")?;  
let data: MyStruct = serde\_json::from\_reader(f)?;

Use std::fs::read to read binary data:

use std;   
  
let mut f = std::fs::File::open("data.bin")?;  
let mut buffer = Vec::new();  
f.read\_to\_end(&mut buffer)?;

Use BufReader to incrementally read large files:

use std::io::BufReader;  
  
let f = File::open("large\_file.txt")?;  
let rdr = BufReader::new(f);

Use crates like flate2, brotli, etc. to decompress data.

**3. Parsing and manipulating data**

Use the serde crate to serialize and deserialize data. Define data schemas using:

* Structs:

struct Record {  
 name: String,  
 age: i32,  
 phones: Vec<String>   
}

* Enums:

enum Status {  
 Active,  
 Inactive   
}

Use pattern matching to destructure enums and structs:

let rec = Record {   
 name: "John",  
 age: 30,  
 phones: ["123-123-1234", "234-234-2345"]  
};  
  
match rec {  
 Record { name, age, ref phones } => { // Destructure   
 println!("Name: {}", name);  
 println!("Age: {}", age);   
 for phone in phones { // Iterate over Vec  
 println!("Phone: {}", phone);   
 }  
 }  
}

Use Iterators to manipulate collections in a zero-cost abstraction manner:

let v = vec![1, 2, 3, 4, 5];  
  
let sum = v.iter().sum(); // 15  
let even = v.iter().filter(|&n| n % 2 == 0); // [2, 4]

For performance, collect Iterators into concrete collections like Vec using .collect():

let even: Vec<i32> = v.iter().filter(|&n| n % 2 == 0).collect();

**4. Filtering and aggregating data**

Use .filter() to filter data:

let even = v.iter().filter(|&n| n % 2 == 0);

Use .sum(), .fold(), etc. to aggregate data:

let sum = v.iter().sum(); // 15  
  
let folded = v.iter().fold(0, |acc, &x| acc + x); // Also 15

Use .group\_by() to group and aggregate data:

let groups = v.iter().group\_by(|&n| n % 2 == 0);   
let evens = groups.get(&true).unwrap(); // [2, 4]  
let odds = groups.get(&false).unwrap(); // [1, 3, 5]

**5. Storing data**

Write to CSV/JSON for short term storage:

let wtr = csv::Writer::from\_path("data.csv");   
wtr.serialize(record)?; // Record is a struct

Use a database like Postgres, SQLite, MongoDB for long term storage.

Cache data in memory with Arc> for fast lookup:

use std::sync::{Arc, Mutex};  
  
let cache = Arc::new(Mutex::new(HashMap::new()));  
let cached\_data = cache.lock().unwrap(); // Get lock

**6. Parallelizing data processing**

Use Rayon to parallelize operations over Iterators:

use rayon::prelude::\*;  
  
let sum = v.par\_iter().sum(); // Run in parallel!

Share data across threads with Arc and Mutex:

let cache = Arc::new(Mutex::new(0));  
let handles = (0..10).map(|i| {  
 let cache = cache.clone();  
 thread::spawn(move || {  
 let mut data = cache.lock().unwrap();   
 \*data += i;  
 })  
});

Compare performance of sequential vs parallel processing and choose appropriately.

**7. Putting it all together: A data processing pipeline example**

* Read input CSV
* Filter and parse the data
* Store aggregated results in a database
* Cache calculated results in memory
* Parallelize independent operations
* Handle errors/bad data gracefully

fn main() {  
 let mut rdr = csv::Reader::from\_path("data.csv");  
 let conn = establish\_db\_connection();  
 let cache = Arc::new(Mutex::new(HashMap::new()));  
  
 // Read and parse CSV rows, inserting into DB   
 for record in rdr.records() {  
 let record = match record {  
 Ok(record) => record,  
 Err(err) => { /\* Handle invalid row \*/ }   
 };  
 conn.execute(/\* ... \*/);  
 let cache = cache.clone();  
 thread::spawn(move || {  
 let mut cache = cache.lock().unwrap();  
 cache.insert(/\* ... \*/);  
 });  
 }   
 // Use cache for fast lookups  
}