Efficient Programs

Group 20 - Optimizing [myjoin] using Rust

Objective: Perform a join operation on files based on a specified key column.

Challenge: Efficiently process large datasets, ensuring low runtime and memory overhead.

Rusty-join:

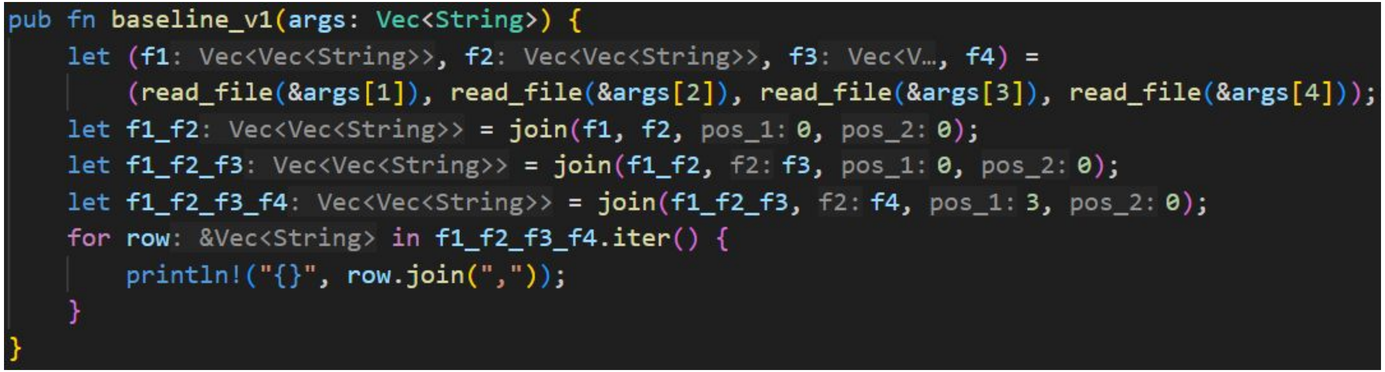
- We implemented the program in Rust
- Compare various algorithmic versions and implementation improvements
- Benchmarking results for multithreaded versions with criterion
- https://github.com/mwage/rusty-join

Overview of Optimization steps

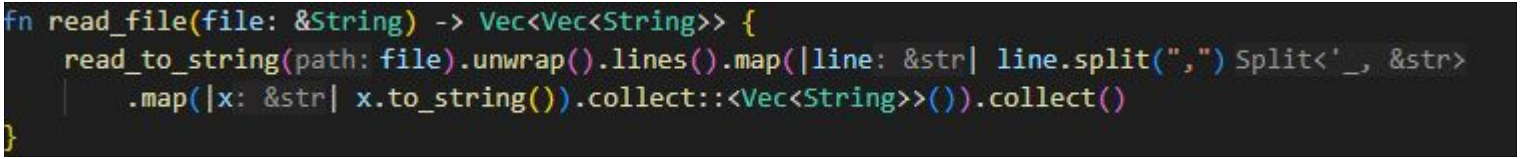
1. Baseline Implementation: Initial naive join.
2. Sorting: Improve algorithm through sorting.
3. Hash-Based Joins : Efficient data structures for faster lookups.
4. Reduced Hash Joins: Reduces number of Hashmaps
5. Multithreading: Exploiting parallelism for further speedups.
6. Polars Library: Comparison with an external library.

Baseline implementation - V1

- Read all four files into vectors of vectors of Strings
- Perform sequential joins using specified key columns.



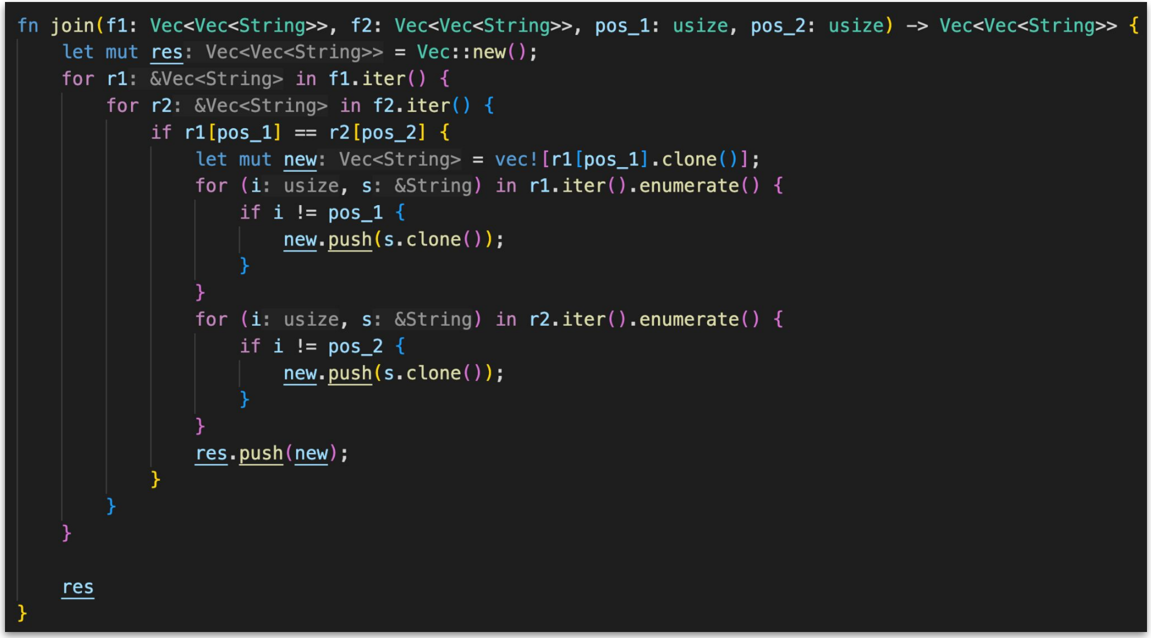
```
pub fn baseline_v1(args: Vec<String>) {
    let (f1: Vec<Vec<String>>, f2: Vec<Vec<String>>, f3: Vec<V…, f4) =
        (read_file(&args[1]), read_file(&args[2]), read_file(&args[3]), read_file(&args[4]));
    let f1_f2: Vec<Vec<String>> = join(f1, f2, pos_1: 0, pos_2: 0);
    let f1_f2_f3: Vec<Vec<String>> = join(f1_f2, f2: f3, pos_1: 0, pos_2: 0);
    let f1_f2_f3_f4: Vec<Vec<String>> = join(f1_f2_f3, f2: f4, pos_1: 3, pos_2: 0);
    for row: &Vec<String> in f1_f2_f3_f4.iter() {
        println!("{}", row.join(","));
    }
}
```



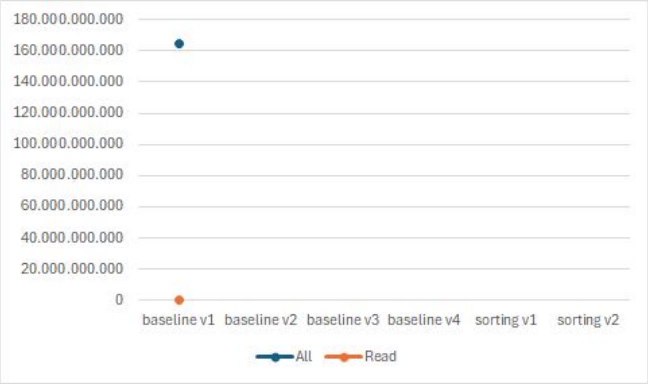
```
fn read_file(file: &String) -> Vec<Vec<String>> {
    read_to_string(path: file).unwrap().lines().map(|line: &str| line.split(",") Split<'_, &str>
        .map(|x: &str| x.to_string()).collect::<Vec<String>>()).collect()
}
```

Baseline implementation - V1

Join: Nested for-loops iterate over rows of two datasets to find matching keys.

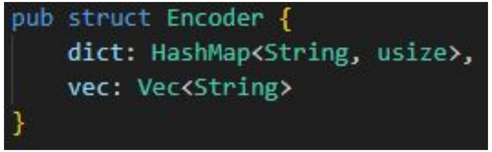


```
fn join(f1: Vec<Vec<String>>, f2: Vec<Vec<String>>, pos_1: usize, pos_2: usize) -> Vec<Vec<String>> {
    let mut res: Vec<Vec<String>> = Vec::new();
    for r1: &Vec<String> in f1.iter() {
        for r2: &Vec<String> in f2.iter() {
            if r1[pos_1] == r2[pos_2] {
                let mut new: Vec<String> = vec![r1[pos_1].clone()];
                for (i: usize, s: &String) in r1.iter().enumerate() {
                    if i != pos_1 {
                        new.push(s.clone());
                    }
                }
                for (i: usize, s: &String) in r2.iter().enumerate() {
                    if i != pos_2 {
                        new.push(s.clone());
                    }
                }
                res.push(new);
            }
        }
    }

    res
}
```

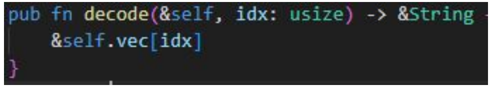


Baseline implementation - V2

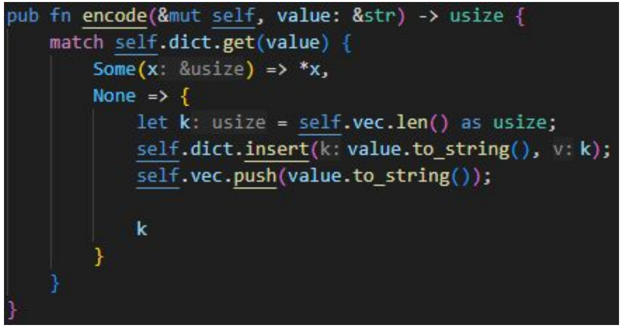
Encoder: Avoid string copy in join by encoding each string as integer



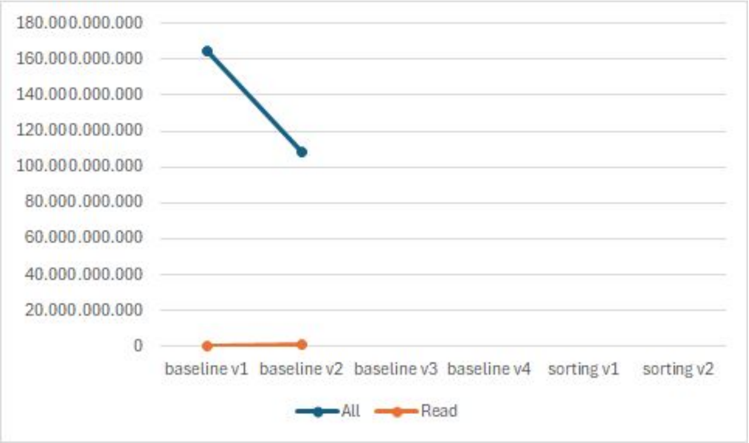
```rust
pub struct Encoder {
    dict: HashMap<String, usize>,
    vec: Vec<String>
}
```

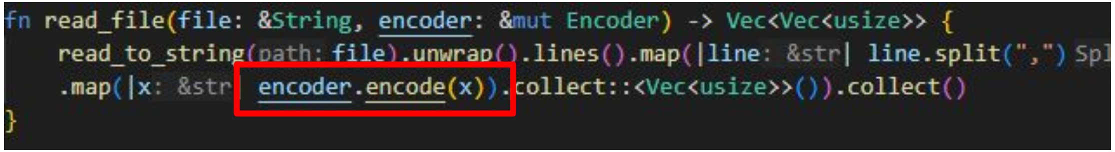


```rust
pub fn decode(&self, idx: usize) -> &String
    &self.vec[idx]
}
```

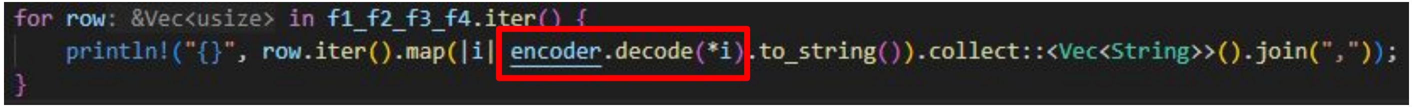


```rust
pub fn encode(&mut self, value: &str) -> usize {
    match self.dict.get(value) {
        Some(x: &usize) => *x,
        None => {
            let k: usize = self.vec.len() as usize;
            self.dict.insert(k: value.to_string(), v: k);
            self.vec.push(value.to_string());

            k
        }
    }
}
```





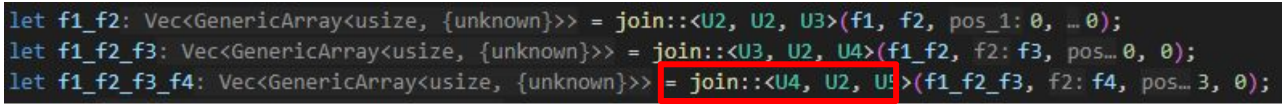
```rust
fn read_file(file: &String, encoder: &mut Encoder) -> Vec<Vec<usize>> {
    read_to_string(path: file).unwrap().lines().map(|line: &str| line.split(",") Spl
    .map(|x: &str  encoder.encode(x)).collect::<Vec<usize>>()).collect()
}
```



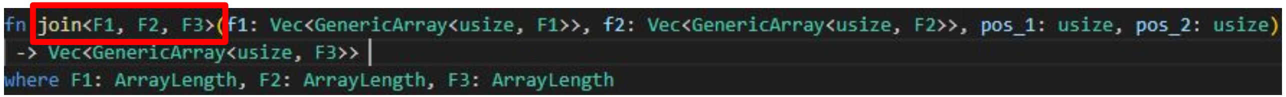
```rust
for row: &Vec<usize> in f1_f2_f3_f4.iter() {
    println!("{}", row.iter().map(|i| encoder.decode(*i).to_string()).collect::<Vec<String>>().join(","));
}
```

Baseline implementation - V3

Generic Arrays: Arrays can be kept on the stack instead of heap (Vec)

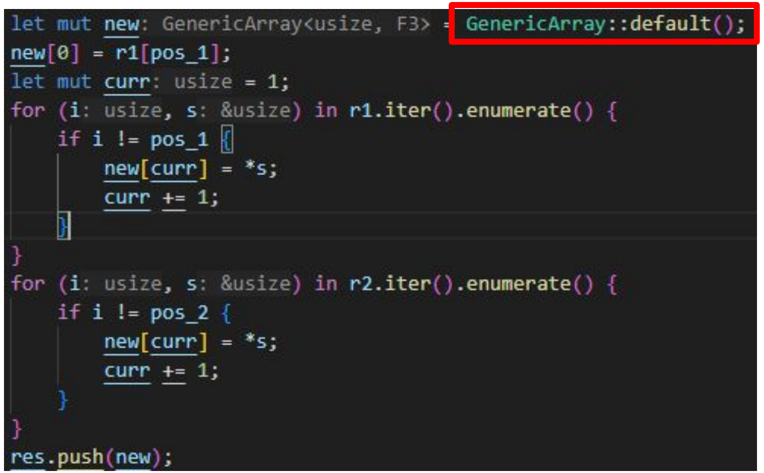


```
let f1_f2: Vec<GenericArray<usize, {unknown}>> = join::<U2, U2, U3>(f1, f2, pos_1: 0, … 0);
let f1_f2_f3: Vec<GenericArray<usize, {unknown}>> = join::<U3, U2, U4>(f1_f2, f2: f3, pos… 0, 0);
let f1_f2_f3_f4: Vec<GenericArray<usize, {unknown}>> = join::<U4, U2, U5>(f1_f2_f3, f2: f4, pos… 3, 0);
```

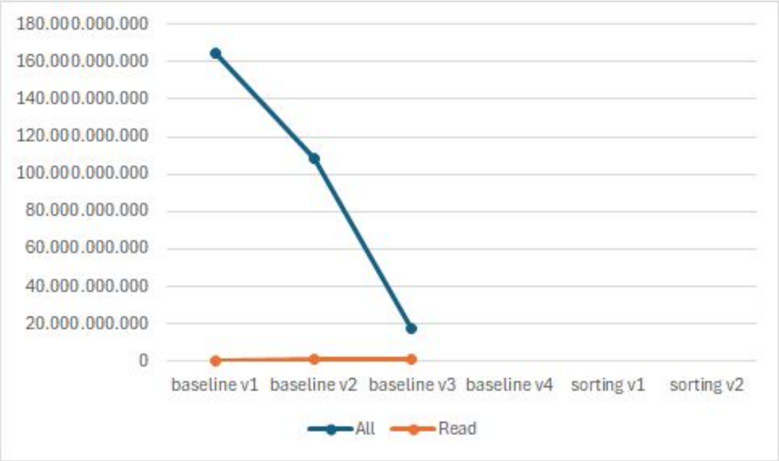


```
fn join<F1, F2, F3>(f1: Vec<GenericArray<usize, F1>>, f2: Vec<GenericArray<usize, F2>>, pos_1: usize, pos_2: usize)
 -> Vec<GenericArray<usize, F3>>
where F1: ArrayLength, F2: ArrayLength, F3: ArrayLength
```

in join:



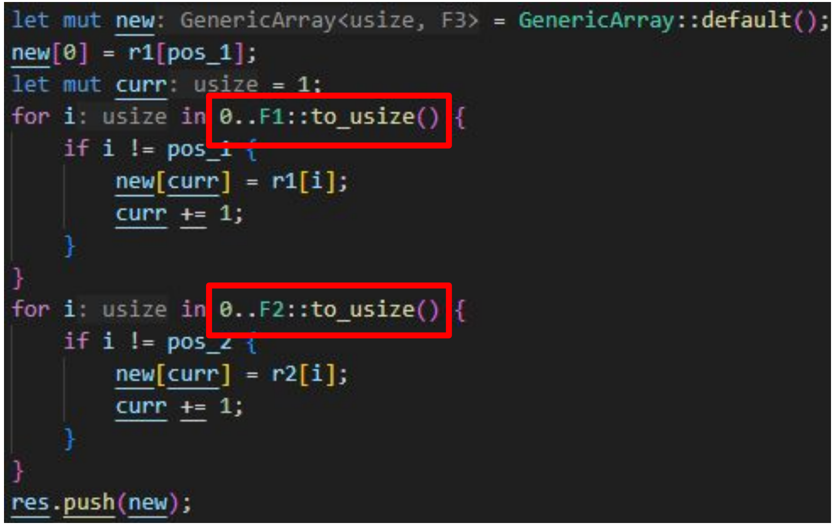
```
let mut new: GenericArray<usize, F3> = GenericArray::default();
new[0] = r1[pos_1];
let mut curr: usize = 1;
for (i: usize, s: &usize) in r1.iter().enumerate() {
    if i != pos_1 {
        new[curr] = *s;
        curr += 1;
    }
}
for (i: usize, s: &usize) in r2.iter().enumerate() {
    if i != pos_2 {
        new[curr] = *s;
        curr += 1;
    }
}
res.push(new);
```



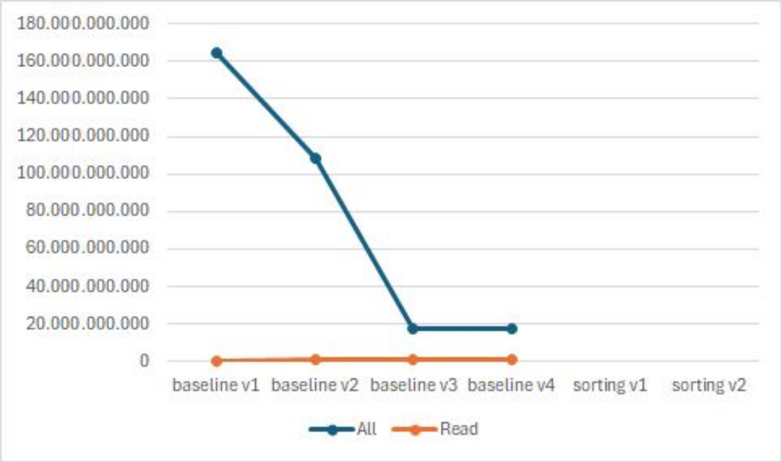
Baseline implementation - V4

Loop Unrolling: “Force” compiler to unroll loops

in join:



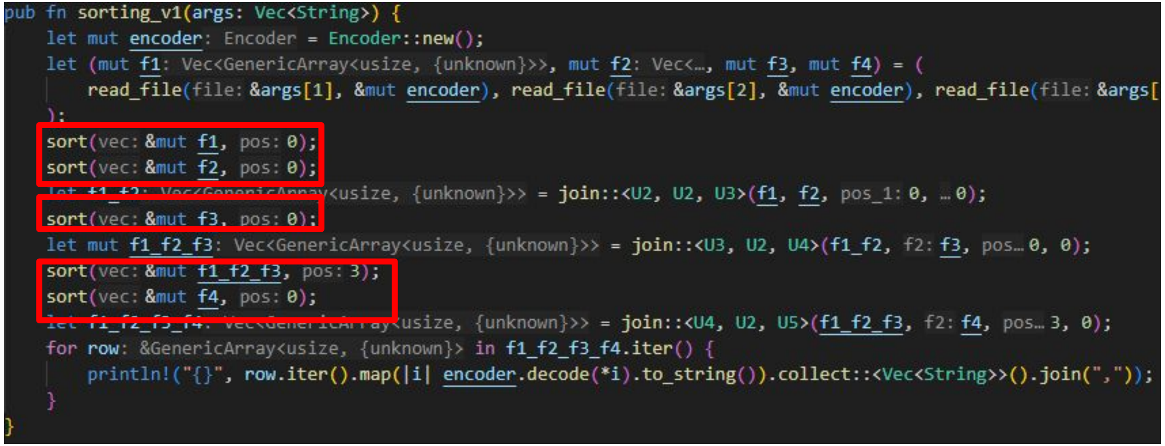
```
let mut new: GenericArray<usize, F3> = GenericArray::default();
new[0] = r1[pos_1];
let mut curr: usize = 1;
for i: usize in 0..F1::to_usize() {
    if i != pos_1 {
        new[curr] = r1[i];
        curr += 1;
    }
}
for i: usize in 0..F2::to_usize() {
    if i != pos_2 {
        new[curr] = r2[i];
        curr += 1;
    }
}
res.push(new);
```



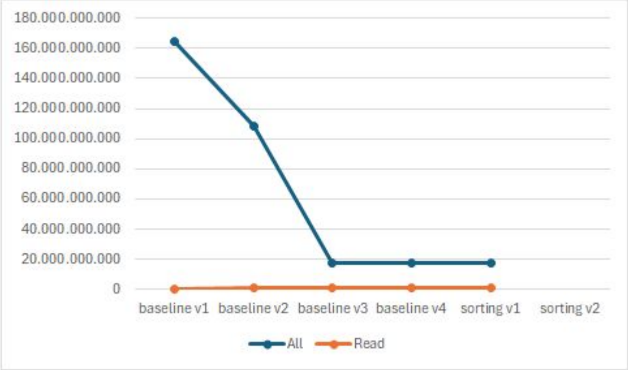
Sorting - V1

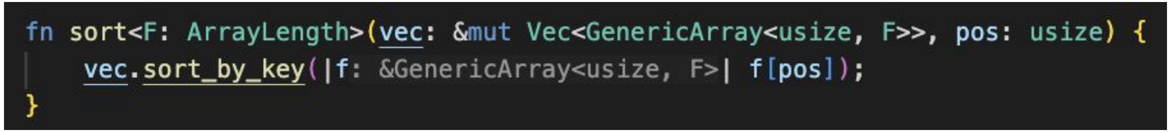
Sort files based on join keys:

- Similar performance => overhead neglectable
- Makes new algorithmic optimizations possible



```
pub fn sorting_v1(args: Vec<String>) {
    let mut encoder: Encoder = Encoder::new();
    let (mut f1: Vec<GenericArray<usize, {unknown}>>, mut f2: Vec<…, mut f3, mut f4) = (
        read_file(file: &args[1], &mut encoder), read_file(file: &args[2], &mut encoder), read_file(file: &args[
    );
    sort(vec: &mut f1, pos: 0);
    sort(vec: &mut f2, pos: 0);
    let f1_f2: Vec<GenericArray<usize, {unknown}>> = join::<U2, U2, U3>(f1, f2, pos_1: 0, … 0);
    sort(vec: &mut f3, pos: 0);
    let mut f1_f2_f3: Vec<GenericArray<usize, {unknown}>> = join::<U3, U2, U4>(f1_f2, f2: f3, pos… 0, 0);
    sort(vec: &mut f1_f2_f3, pos: 3);
    sort(vec: &mut f4, pos: 0);
    let f1_f2_f3_f4: Vec<GenericArray<usize, {unknown}>> = join::<U4, U2, U5>(f1_f2_f3, f2: f4, pos… 3, 0);
    for row: &GenericArray<usize, {unknown}> in f1_f2_f3_f4.iter() {
        println!("{}", row.iter().map(|i| encoder.decode(*i).to_string()).collect::<Vec<String>>().join(","));
    }
}
```

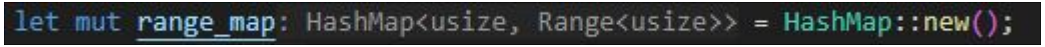




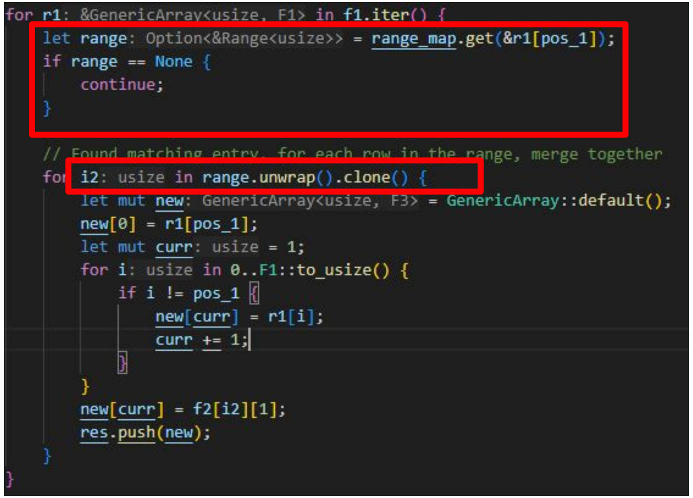
```
fn sort<F: ArrayLength>(vec: &mut Vec<GenericArray<usize, F>>, pos: usize) {
    vec.sort_by_key(|f: &GenericArray<usize, F>| f[pos]);
}
```

Sorting - V2

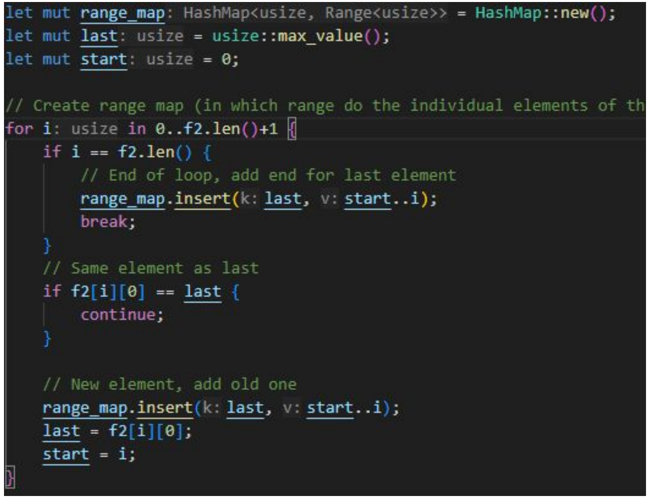
- Uses a HashMap to store the index range in which each value in first column of second dataframe occurs
- only iterates the elements that are necessary



```
let mut range_map: HashMap<usize, Range<usize>> = HashMap::new();
```



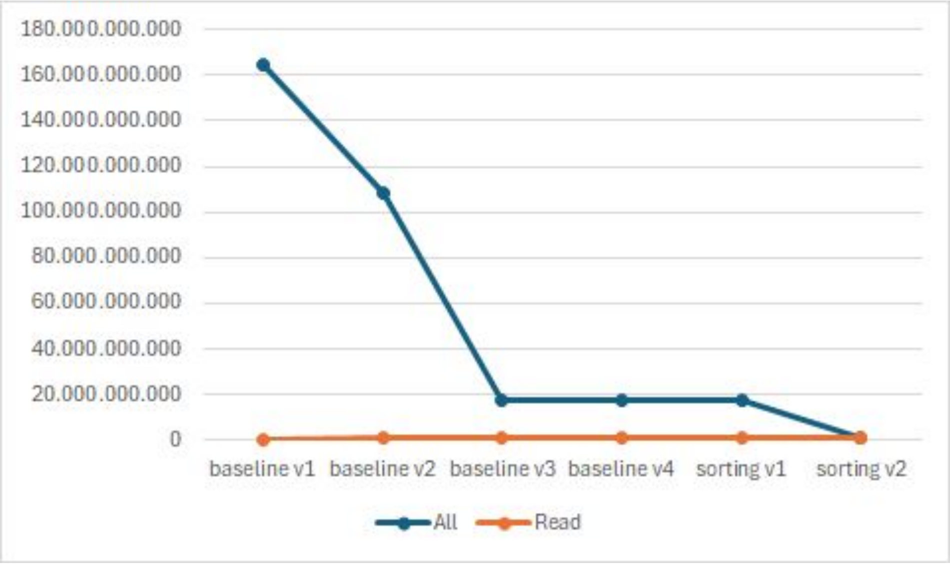
```
for r1: &GenericArray<usize, F1> in f1.iter() {
    let range: Option<&Range<usize>> = range_map.get(&r1[pos_1]);
    if range == None {
        continue;
    }

    // Found matching entry, for each row in the range, merge together
    for i2: usize in range.unwrap().clone() {
        let mut new: GenericArray<usize, F3> = GenericArray::default();
        new[0] = r1[pos_1];
        let mut curr: usize = 1;
        for i: usize in 0..F1::to_usize() {
            if i != pos_1 {
                new[curr] = r1[i];
                curr += 1;
            }
        }
        new[curr] = f2[i2][1];
        res.push(new);
    }
}
```



```
let mut range_map: HashMap<usize, Range<usize>> = HashMap::new();
let mut last: usize = usize::max_value();
let mut start: usize = 0;

// Create range map (in which range do the individual elements of th
for i: usize in 0..f2.len()+1 {
    if i == f2.len() {
        // End of loop, add end for last element
        range_map.insert(k: last, v: start..i);
        break;
    }
    // Same element as last
    if f2[i][0] == last {
        continue;
    }

    // New element, add old one
    range_map.insert(k: last, v: start..i);
    last = f2[i][0];
    start = i;
}
```

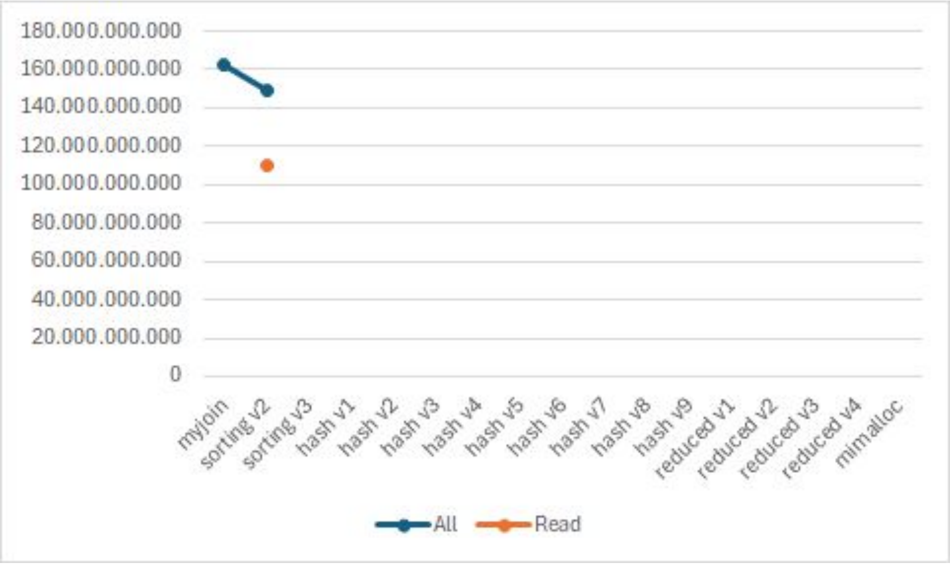
Sorting - V2

Benchmarks with small dataset



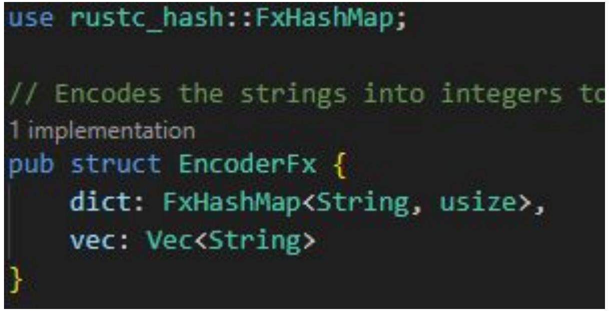
First benchmarks with full dataset:

myjoin and sorting v2

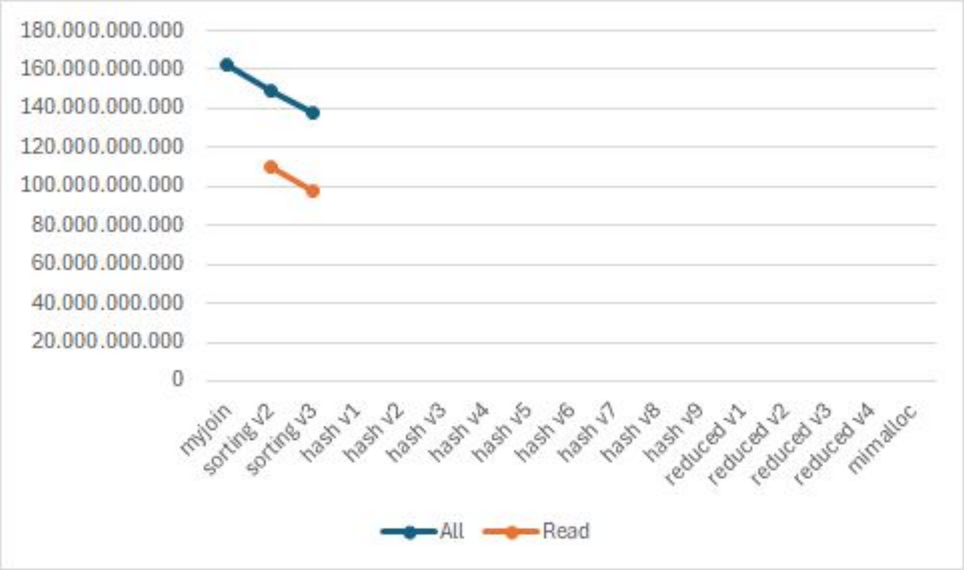


Sorting - V3

- Improve read by using faster Hashmaps:
- Replace standard HashMap with FxHashMap (non-cryptographic hasher)

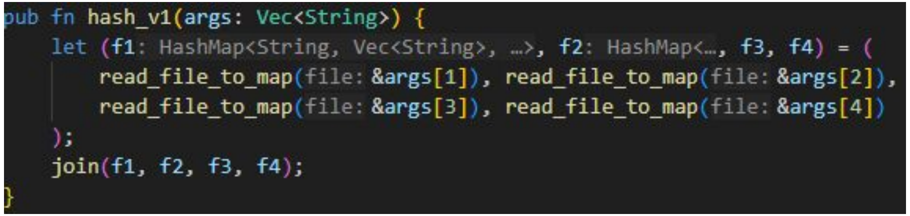


```
use rustc_hash::FxHashMap;

// Encodes the strings into integers to
1 implementation
pub struct EncoderFx {
    dict: FxHashMap<String, usize>,
    vec: Vec<String>
}
```

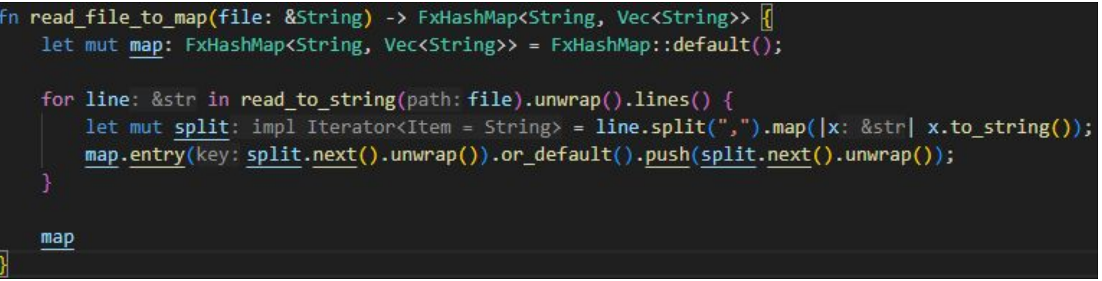


Hash Join - V1

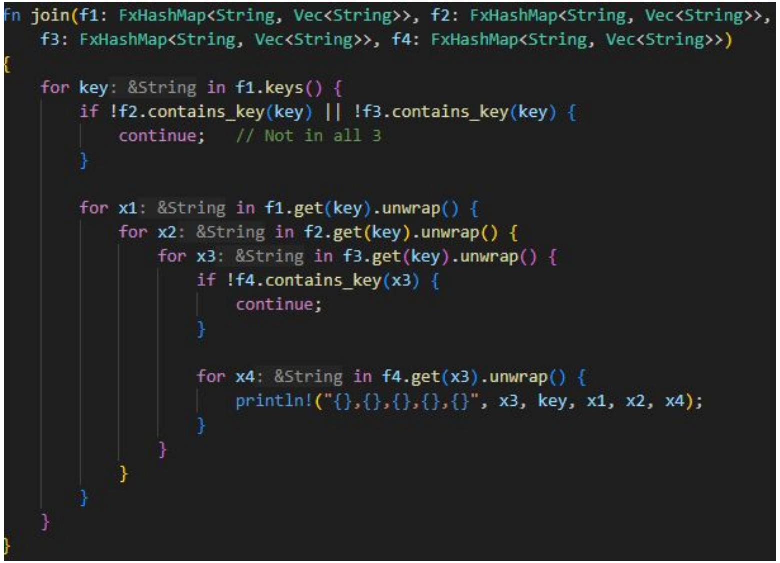
- Read each file into hash maps
- Use String directly => no need to encode to integers anymore
- Join iterates over all hashmaps and looks for the corresponding keys



```
pub fn hash_v1(args: Vec<String>) {
    let (f1: HashMap<String, Vec<String>, …>, f2: HashMap<…, f3, f4) = (
        read_file_to_map(file: &args[1]), read_file_to_map(file: &args[2]),
        read_file_to_map(file: &args[3]), read_file_to_map(file: &args[4])
    );
    join(f1, f2, f3, f4);
}
```

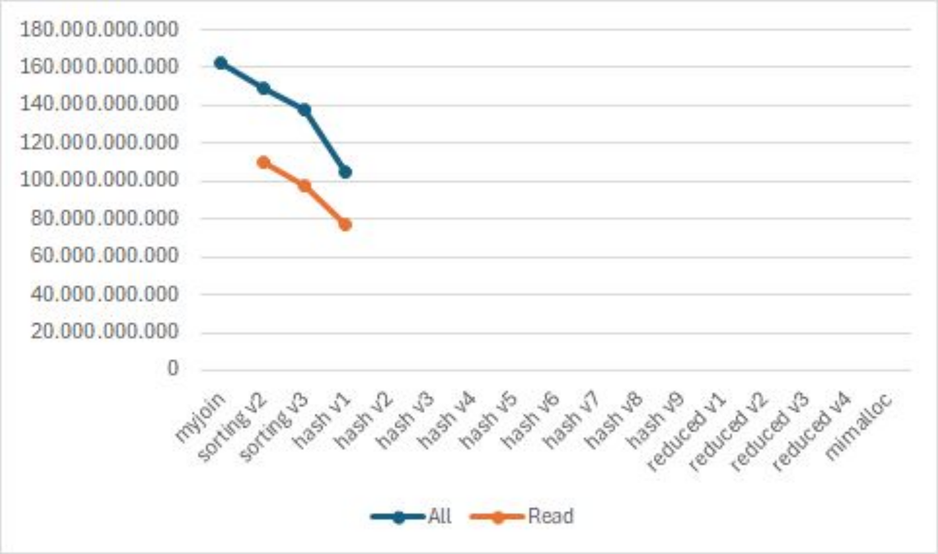


```
fn read_file_to_map(file: &String) -> FxHashMap<String, Vec<String>> {
    let mut map: FxHashMap<String, Vec<String>> = FxHashMap::default();

    for line: &str in read_to_string(path: file).unwrap().lines() {
        let mut split: impl Iterator<Item = String> = line.split(",").map(|x: &str| x.to_string());
        map.entry(key: split.next().unwrap()).or_default().push(split.next().unwrap());
    }

    map
}
```



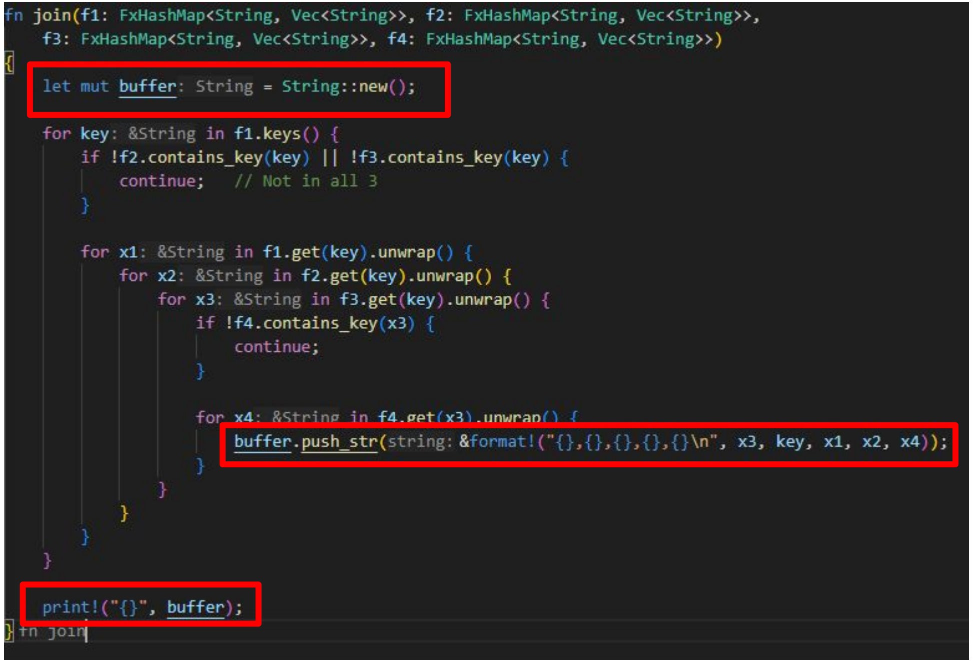
```
fn join(f1: FxHashMap<String, Vec<String>>, f2: FxHashMap<String, Vec<String>>,
    f3: FxHashMap<String, Vec<String>>, f4: FxHashMap<String, Vec<String>>)
{
    for key: &String in f1.keys() {
        if !f2.contains_key(key) || !f3.contains_key(key) {
            continue;   // Not in all 3
        }

        for x1: &String in f1.get(key).unwrap() {
            for x2: &String in f2.get(key).unwrap() {
                for x3: &String in f3.get(key).unwrap() {
                    if !f4.contains_key(x3) {
                        continue;
                    }

                    for x4: &String in f4.get(x3).unwrap() {
                        println!("{},{},{},{},{}", x3, key, x1, x2, x4);
                    }
                }
            }
        }
    }
}
```

Hash Join - V1

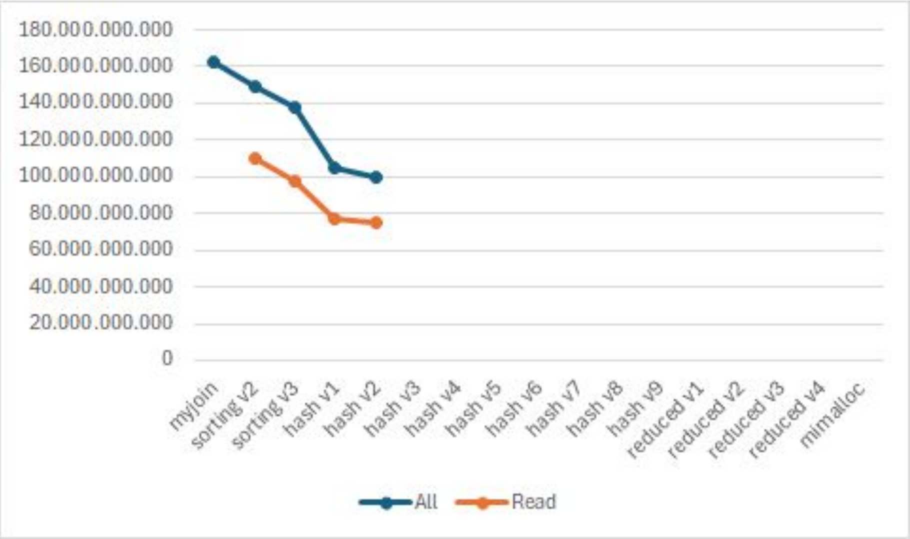

180.000.000.000
160.000.000.000
140.000.000.000
120.000.000.000
100.000.000.000
80.000.000.000
60.000.000.000
40.000.000.000
20.000.000.000
0
myjoin
sorting v2
sorting v3
hash v1
hash v2
hash v3
hash v4
hash v5
hash v6
hash v7
hash v8
hash v9
reduced v1
reduced v2
reduced v3
reduced v4
mimalloc
All
Read

Hash Join - V2

Output string buffer: Write into a String and output at the end

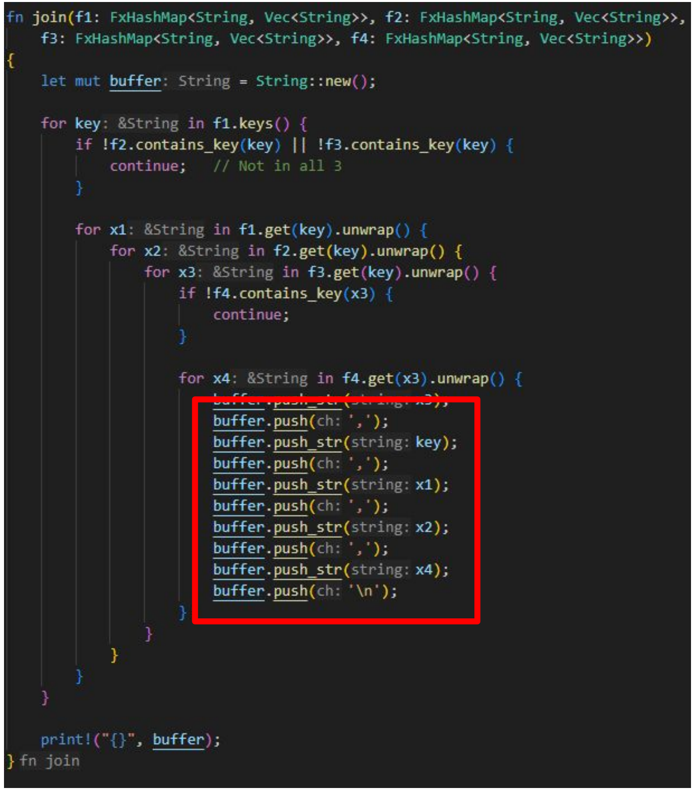


```
fn join(f1: FxHashMap<String, Vec<String>>, f2: FxHashMap<String, Vec<String>>,
    f3: FxHashMap<String, Vec<String>>, f4: FxHashMap<String, Vec<String>>)
{
    let mut buffer: String = String::new();

    for key: &String in f1.keys() {
        if !f2.contains_key(key) || !f3.contains_key(key) {
            continue;   // Not in all 3
        }

        for x1: &String in f1.get(key).unwrap() {
            for x2: &String in f2.get(key).unwrap() {
                for x3: &String in f3.get(key).unwrap() {
                    if !f4.contains_key(x3) {
                        continue;
                    }

                    for x4: &String in f4.get(x3).unwrap() {
                        buffer.push_str(string: &format!("{},{},{},{},{}\n", x3, key, x1, x2, x4));
                    }
                }
            }
        }
    }

    print!("{}", buffer);
} fn join
```

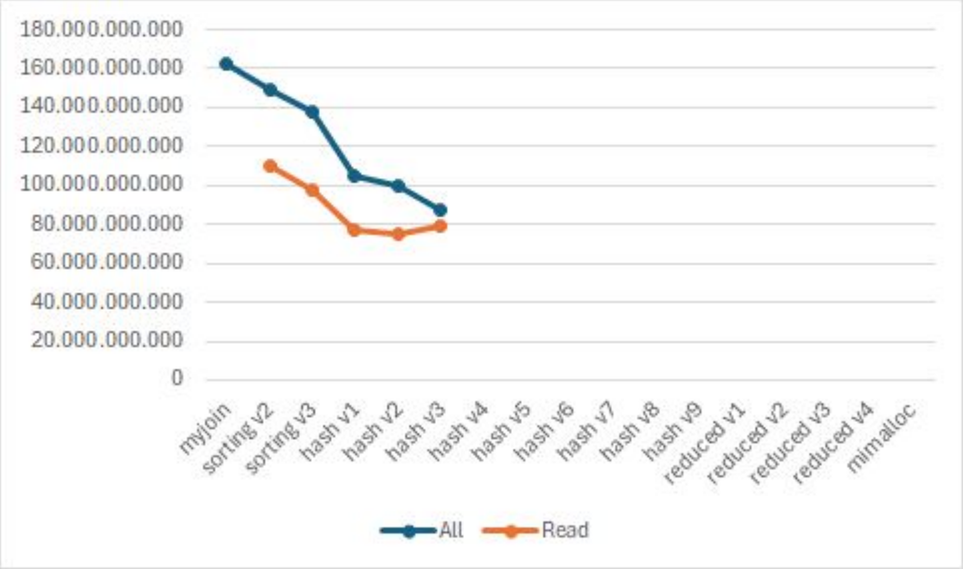


Hash Join - V3

Optimizing the buffer: Directly appends to the buffer without using [format!].

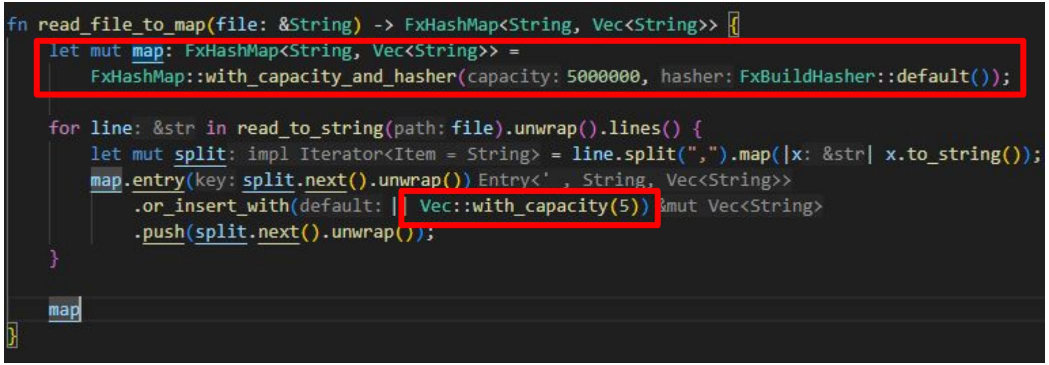


```
fn join(f1: FxHashMap<String, Vec<String>>, f2: FxHashMap<String, Vec<String>>,
    f3: FxHashMap<String, Vec<String>>, f4: FxHashMap<String, Vec<String>>)
{
    let mut buffer: String = String::new();

    for key: &String in f1.keys() {
        if !f2.contains_key(key) || !f3.contains_key(key) {
            continue;   // Not in all 3
        }

        for x1: &String in f1.get(key).unwrap() {
            for x2: &String in f2.get(key).unwrap() {
                for x3: &String in f3.get(key).unwrap() {
                    if !f4.contains_key(x3) {
                        continue;
                    }

                    for x4: &String in f4.get(x3).unwrap() {
                        buffer.push_str(string: x3);
                        buffer.push(ch: ',');
                        buffer.push_str(string: key);
                        buffer.push(ch: ',');
                        buffer.push_str(string: x1);
                        buffer.push(ch: ',');
                        buffer.push_str(string: x2);
                        buffer.push(ch: ',');
                        buffer.push_str(string: x4);
                        buffer.push(ch: '\n');
                    }
                }
            }
        }
    }

    print!("{}", buffer);
} fn join
```

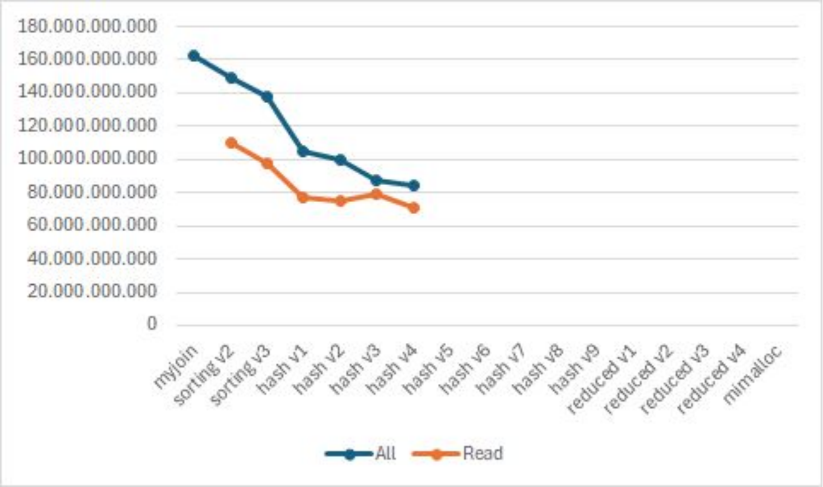


Hash Join - V4

Pre-allocate Vecs and Hashmaps

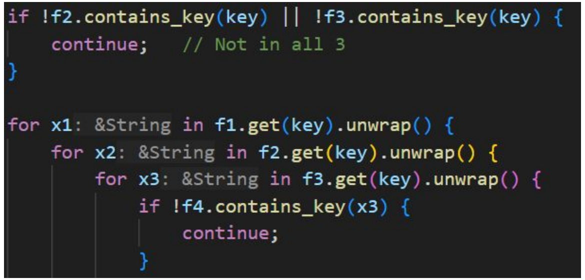


```
fn read_file_to_map(file: &String) -> FxHashMap<String, Vec<String>> {
    let mut map: FxHashMap<String, Vec<String>> =
        FxHashMap::with_capacity_and_hasher(capacity: 5000000, hasher: FxBuildHasher::default());

    for line: &str in read_to_string(path: file).unwrap().lines() {
        let mut split: impl Iterator<Item = String> = line.split(",").map(|x: &str| x.to_string());
        map.entry(key: split.next().unwrap()) Entry<'_, String, Vec<String>>
            .or_insert_with(default: || Vec::with_capacity(5)) &mut Vec<String>
            .push(split.next().unwrap());
    }

    map
}
```

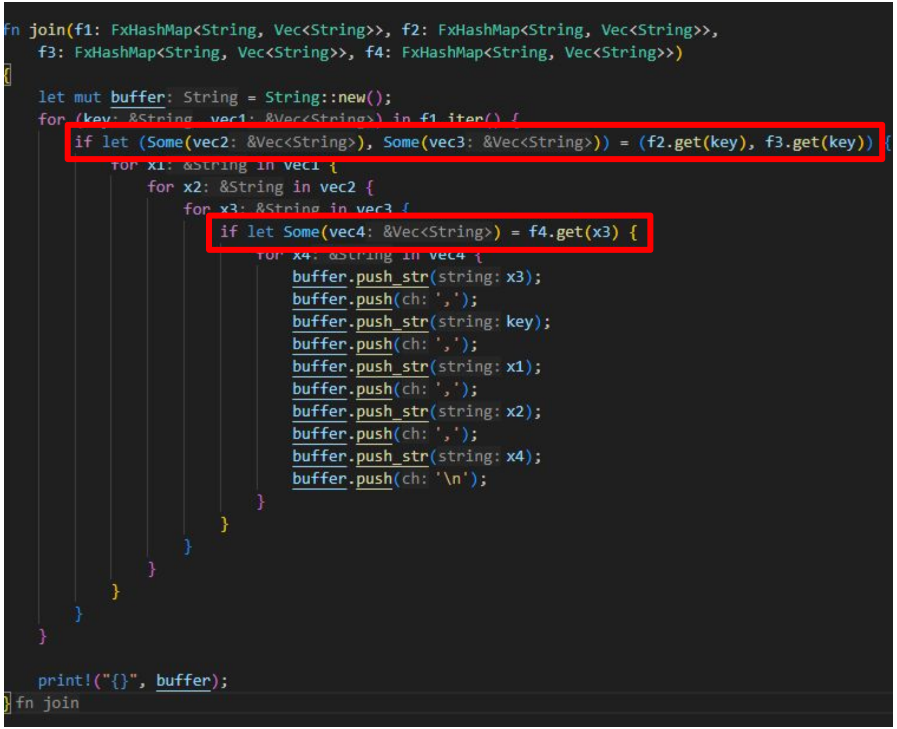


Hash Join - V5

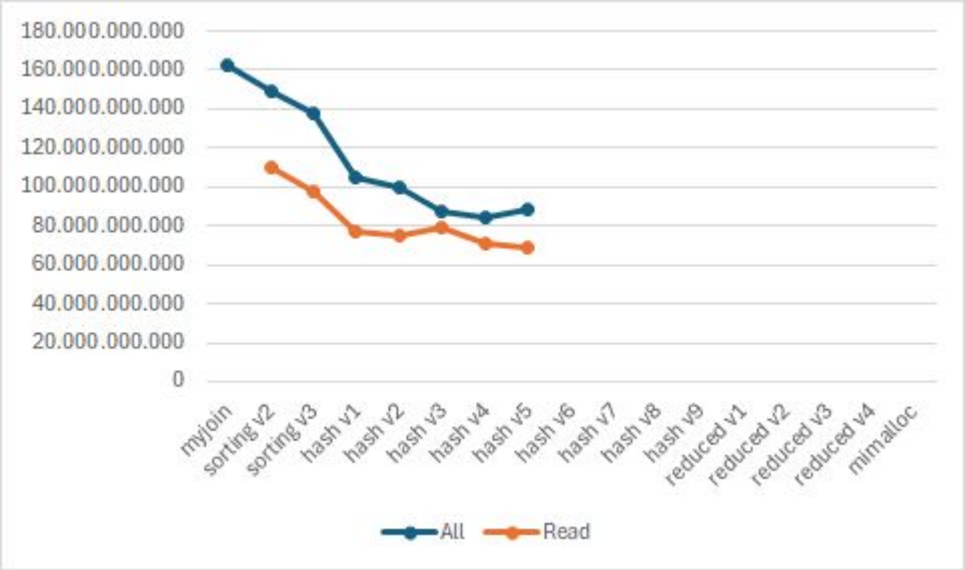
Pattern Matching instead of ifs



```
if !f2.contains_key(key) || !f3.contains_key(key) {
    continue;    // Not in all 3
}

for x1: &String in f1.get(key).unwrap() {
    for x2: &String in f2.get(key).unwrap() {
        for x3: &String in f3.get(key).unwrap() {
            if !f4.contains_key(x3) {
                continue;
            }
```

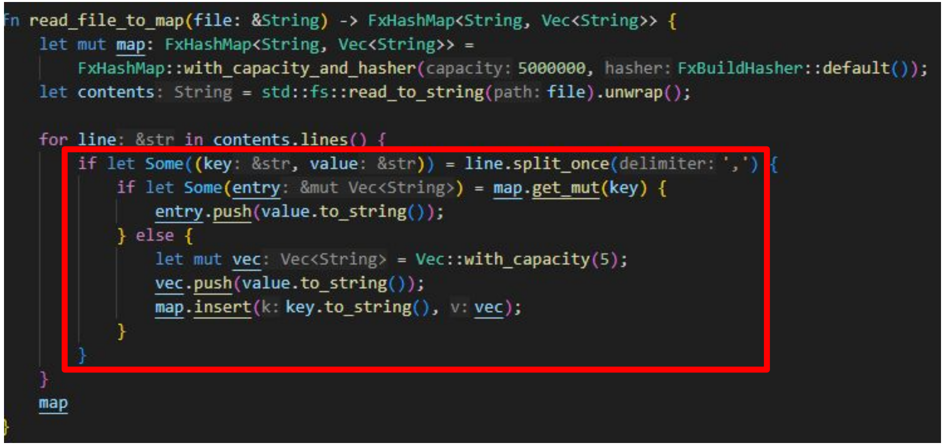


```
fn join(f1: FxHashMap<String, Vec<String>>, f2: FxHashMap<String, Vec<String>>,
    f3: FxHashMap<String, Vec<String>>, f4: FxHashMap<String, Vec<String>>)
{
    let mut buffer: String = String::new();
    for (key: &String, vec1: &Vec<String>) in f1.iter() {
        if let (Some(vec2: &Vec<String>), Some(vec3: &Vec<String>)) = (f2.get(key), f3.get(key)) {
            for x1: &String in vec1 {
                for x2: &String in vec2 {
                    for x3: &String in vec3 {
                        if let Some(vec4: &Vec<String>) = f4.get(x3) {
                            for x4: &String in vec4 {
                                buffer.push_str(string: x3);
                                buffer.push(ch: ',');
                                buffer.push_str(string: key);
                                buffer.push(ch: ',');
                                buffer.push_str(string: x1);
                                buffer.push(ch: ',');
                                buffer.push_str(string: x2);
                                buffer.push(ch: ',');
                                buffer.push_str(string: x4);
                                buffer.push(ch: '\n');
                            }
                        }
                    }
                }
            }
        }
    }

    print!("{}", buffer);
} fn join
```

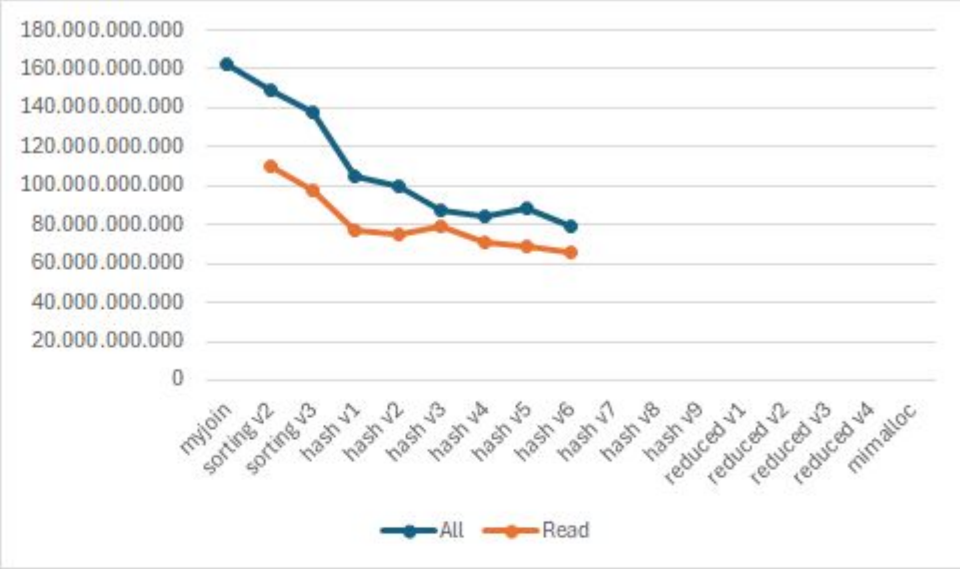


Hash Join - V6

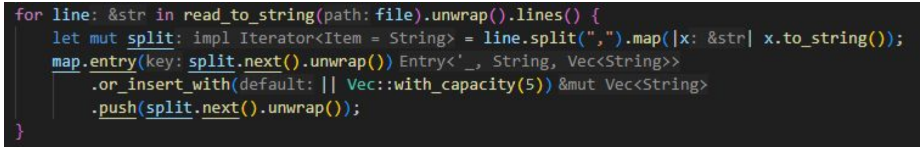
- Avoiding Entry API (overhead)
- Use split_once for parsing lines into key-value tuples



```
fn read_file_to_map(file: &String) -> FxHashMap<String, Vec<String>> {
    let mut map: FxHashMap<String, Vec<String>> =
        FxHashMap::with_capacity_and_hasher(capacity: 5000000, hasher: FxBuildHasher::default());
    let contents: String = std::fs::read_to_string(path: file).unwrap();

    for line: &str in contents.lines() {
        if let Some((key: &str, value: &str)) = line.split_once(delimiter: ',') {
            if let Some(entry: &mut Vec<String>) = map.get_mut(key) {
                entry.push(value.to_string());
            } else {
                let mut vec: Vec<String> = Vec::with_capacity(5);
                vec.push(value.to_string());
                map.insert(k: key.to_string(), v: vec);
            }
        }
    }
    map
}
```



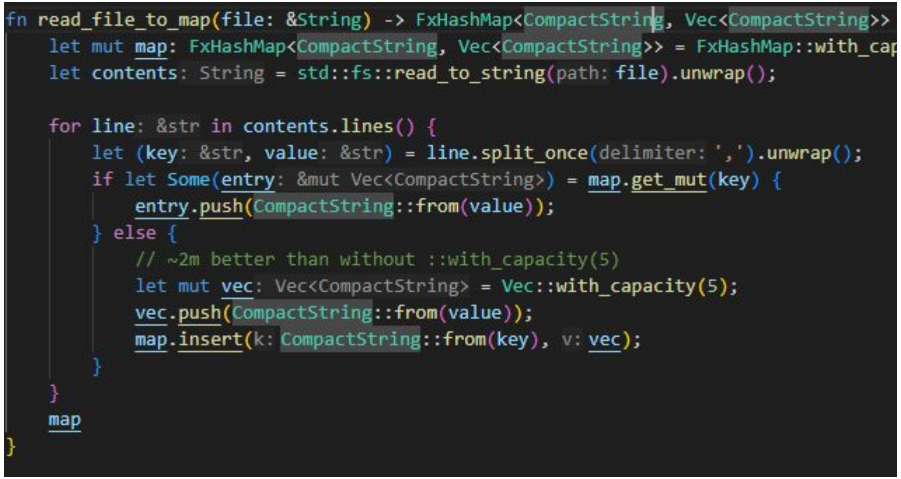
instead of:



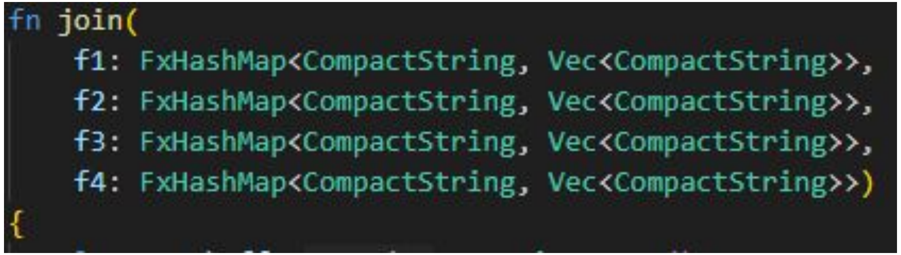
```
for line: &str in read_to_string(path: file).unwrap().lines() {
    let mut split: impl Iterator<Item = String> = line.split(",").map(|x: &str| x.to_string());
    map.entry(key: split.next().unwrap()) Entry<'_, String, Vec<String>>
        .or_insert_with(default: || Vec::with_capacity(5)) &mut Vec<String>
        .push(split.next().unwrap());
}
```

Hash Join - V7

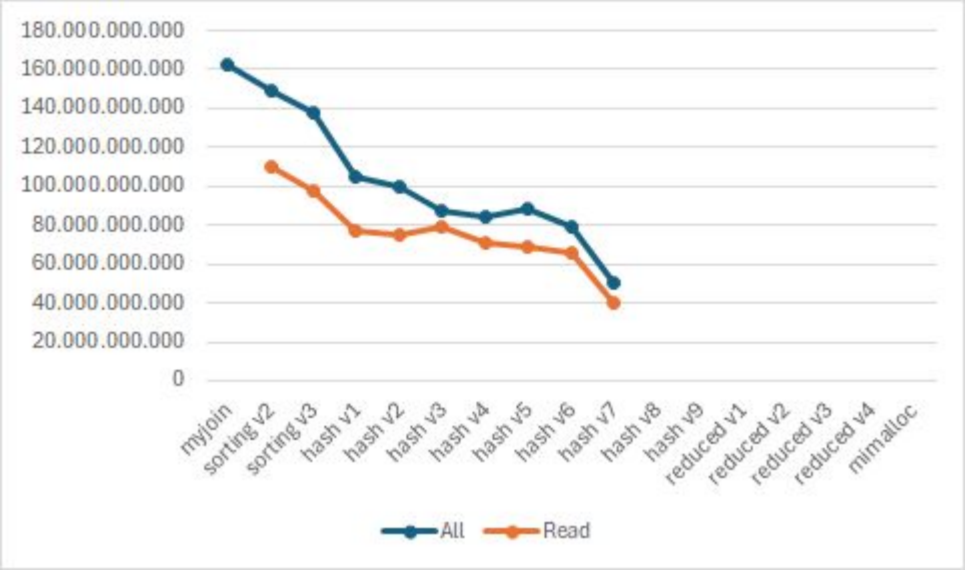
CompactString: Stores small strings on the stack instead of the heap.



```
fn read_file_to_map(file: &String) -> FxHashMap<CompactString, Vec<CompactString>>
    let mut map: FxHashMap<CompactString, Vec<CompactString>> = FxHashMap::with_cap
    let contents: String = std::fs::read_to_string(path: file).unwrap();

    for line: &str in contents.lines() {
        let (key: &str, value: &str) = line.split_once(delimiter: ',').unwrap();
        if let Some(entry: &mut Vec<CompactString>) = map.get_mut(key) {
            entry.push(CompactString::from(value));
        } else {
            // ~2m better than without ::with_capacity(5)
            let mut vec: Vec<CompactString> = Vec::with_capacity(5);
            vec.push(CompactString::from(value));
            map.insert(k: CompactString::from(key), v: vec);
        }
    }
    map
}
```

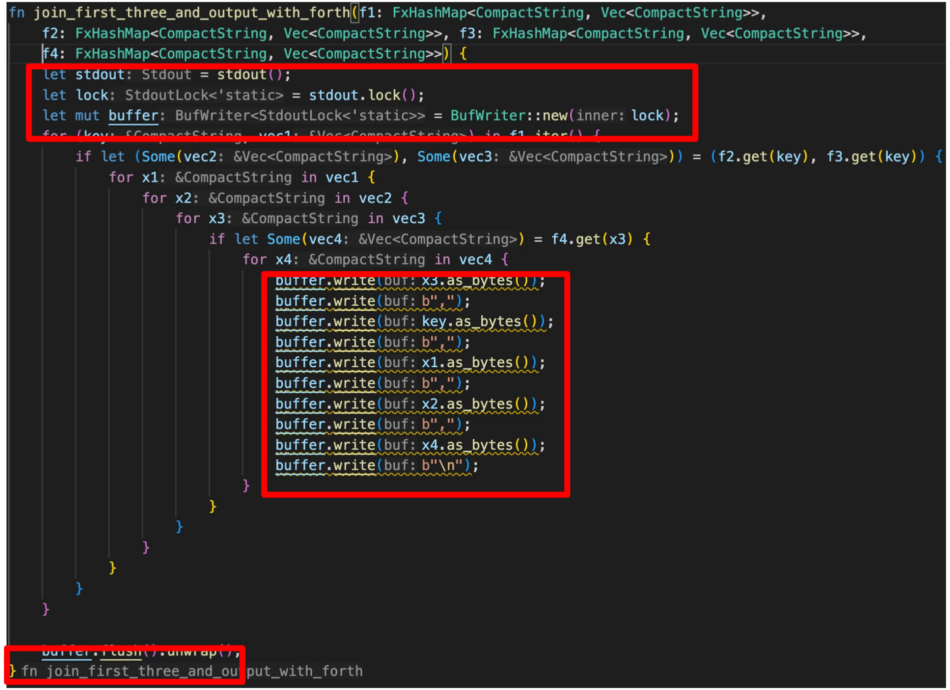


```
fn join(
    f1: FxHashMap<CompactString, Vec<CompactString>>,
    f2: FxHashMap<CompactString, Vec<CompactString>>,
    f3: FxHashMap<CompactString, Vec<CompactString>>,
    f4: FxHashMap<CompactString, Vec<CompactString>>)
{
```

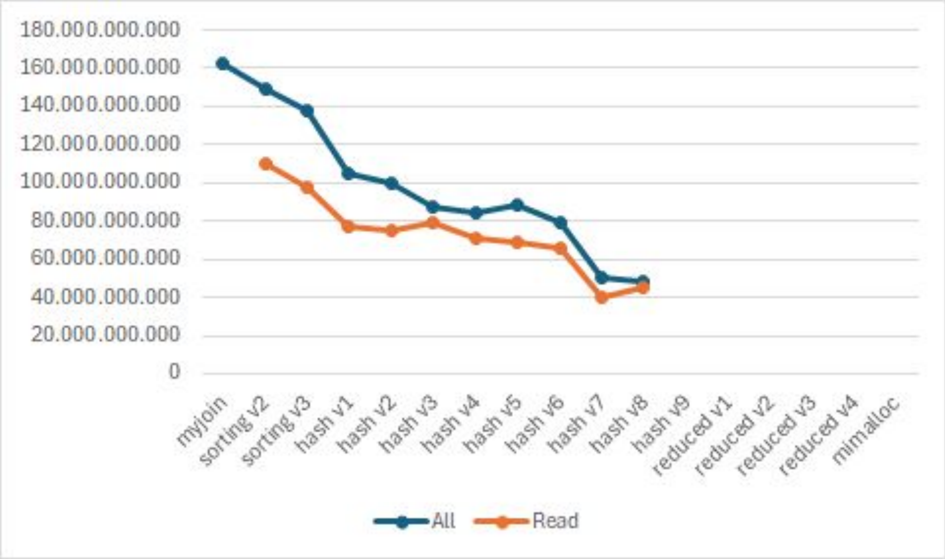


Hash Join - V8

BufWriter: Writes results to stdout using a buffered stream

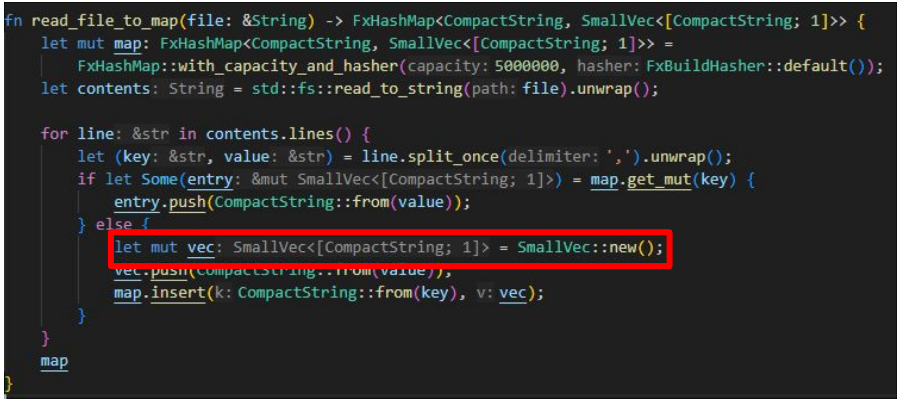


```
fn join_first_three_and_output_with_forth(f1: FxHashMap<CompactString, Vec<CompactString>>,
    f2: FxHashMap<CompactString, Vec<CompactString>>, f3: FxHashMap<CompactString, Vec<CompactString>>,
    f4: FxHashMap<CompactString, Vec<CompactString>>) {
    let stdout: Stdout = stdout();
    let lock: StdoutLock<'static> = stdout.lock();
    let mut buffer: BufWriter<StdoutLock<'static>> = BufWriter::new(inner: lock);
    for (key: &CompactString, vec1: &Vec<CompactString>) in f1.iter() {
        if let (Some(vec2: &Vec<CompactString>), Some(vec3: &Vec<CompactString>)) = (f2.get(key), f3.get(key)) {
            for x1: &CompactString in vec1 {
                for x2: &CompactString in vec2 {
                    for x3: &CompactString in vec3 {
                        if let Some(vec4: &Vec<CompactString>) = f4.get(x3) {
                            for x4: &CompactString in vec4 {
                                buffer.write(buf: x3.as_bytes());
                                buffer.write(buf: b",");
                                buffer.write(buf: key.as_bytes());
                                buffer.write(buf: b",");
                                buffer.write(buf: x1.as_bytes());
                                buffer.write(buf: b",");
                                buffer.write(buf: x2.as_bytes());
                                buffer.write(buf: b",");
                                buffer.write(buf: x4.as_bytes());
                                buffer.write(buf: b"\n");
                            }
                        }
                    }
                }
            }
        }
    }

    buffer.flush().unwrap();
} fn join_first_three_and_output_with_forth
```

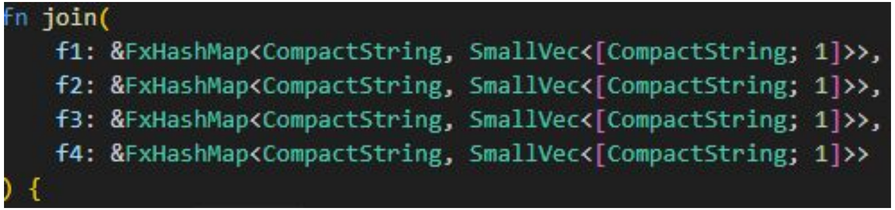


Hash Join - V9

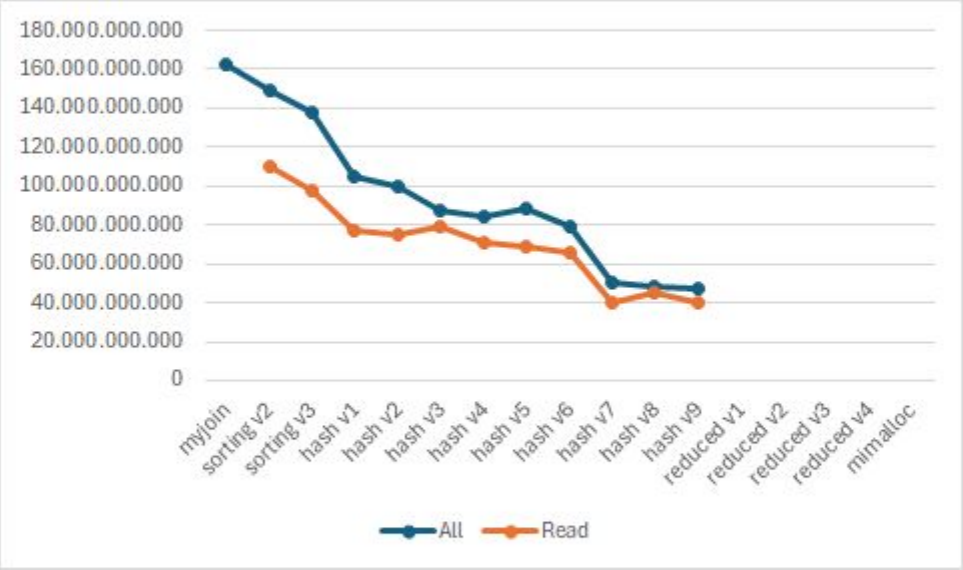
SmallVec: Stores small vectors on the stack, reducing heap allocation



```
fn read_file_to_map(file: &String) -> FxHashMap<CompactString, SmallVec<[CompactString; 1]>> {
    let mut map: FxHashMap<CompactString, SmallVec<[CompactString; 1]>> =
        FxHashMap::with_capacity_and_hasher(capacity: 5000000, hasher: FxBuildHasher::default());
    let contents: String = std::fs::read_to_string(path: file).unwrap();

    for line: &str in contents.lines() {
        let (key: &str, value: &str) = line.split_once(delimiter: ',').unwrap();
        if let Some(entry: &mut SmallVec<[CompactString; 1]>) = map.get_mut(key) {
            entry.push(CompactString::from(value));
        } else {
            let mut vec: SmallVec<[CompactString; 1]> = SmallVec::new();
            vec.push(CompactString::from(value));
            map.insert(k: CompactString::from(key), v: vec);
        }
    }
    map
}
```

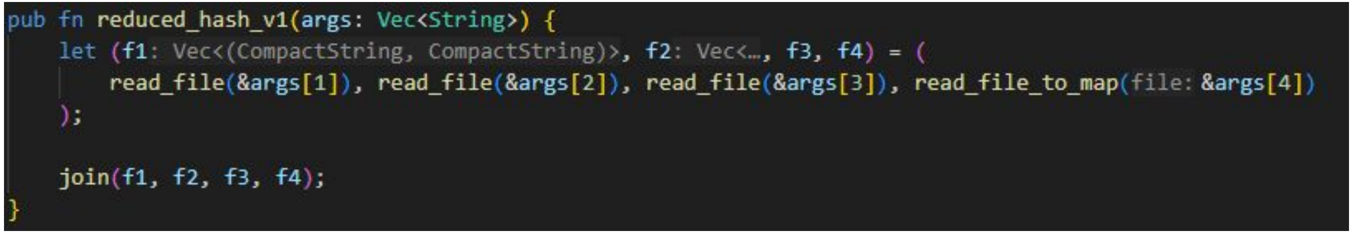


```
fn join(
    f1: &FxHashMap<CompactString, SmallVec<[CompactString; 1]>>,
    f2: &FxHashMap<CompactString, SmallVec<[CompactString; 1]>>,
    f3: &FxHashMap<CompactString, SmallVec<[CompactString; 1]>>,
    f4: &FxHashMap<CompactString, SmallVec<[CompactString; 1]>>
) {
```

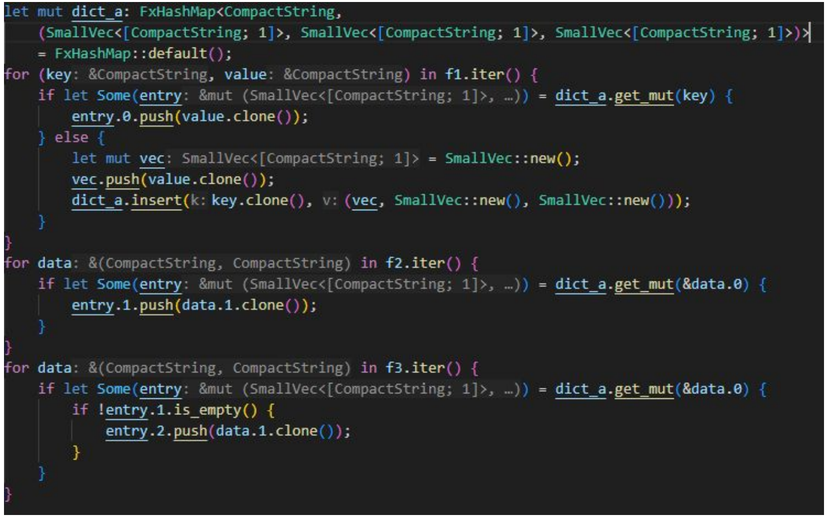


Reduced Hash - V1

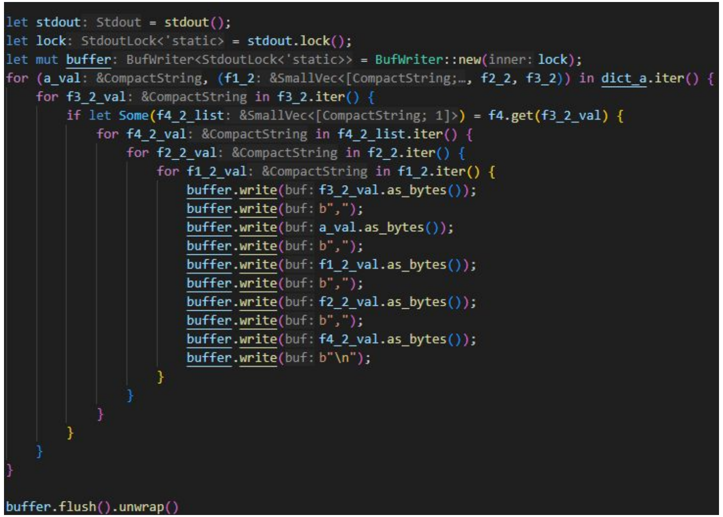
Fewer HashMaps: Uses only one hash map for the join of the first three files



```
pub fn reduced_hash_v1(args: Vec<String>) {
    let (f1: Vec<(CompactString, CompactString)>, f2: Vec<…, f3, f4) = (
        read_file(&args[1]), read_file(&args[2]), read_file(&args[3]), read_file_to_map(file: &args[4])
    );

    join(f1, f2, f3, f4);
}
```



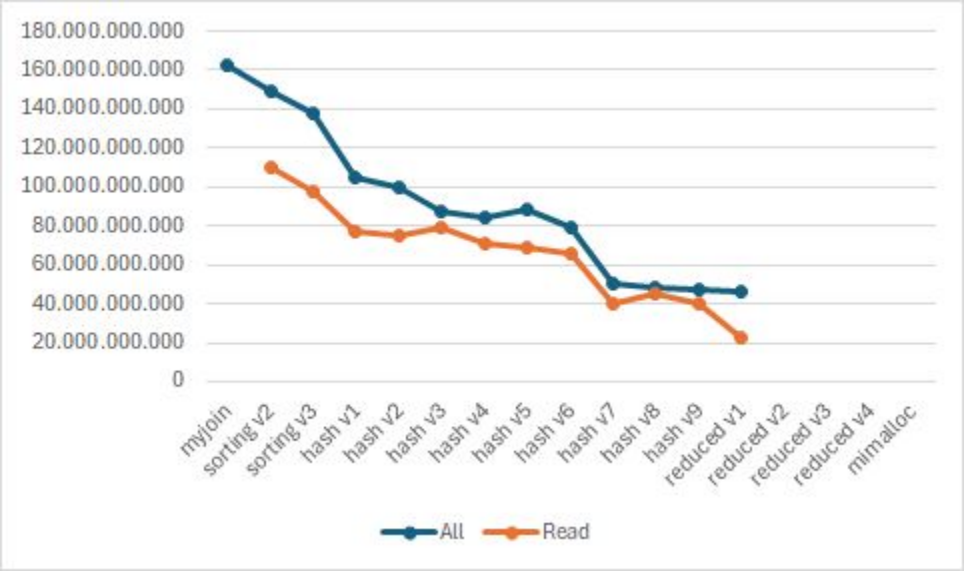
```
let mut dict_a: FxHashMap<CompactString,
    (SmallVec<[CompactString; 1]>, SmallVec<[CompactString; 1]>, SmallVec<[CompactString; 1]>)>
    = FxHashMap::default();
for (key: &CompactString, value: &CompactString) in f1.iter() {
    if let Some(entry: &mut (SmallVec<[CompactString; 1]>, …)) = dict_a.get_mut(key) {
        entry.0.push(value.clone());
    } else {
        let mut vec: SmallVec<[CompactString; 1]> = SmallVec::new();
        vec.push(value.clone());
        dict_a.insert(k: key.clone(), v: (vec, SmallVec::new(), SmallVec::new()));
    }
}
for data: &(CompactString, CompactString) in f2.iter() {
    if let Some(entry: &mut (SmallVec<[CompactString; 1]>, …)) = dict_a.get_mut(&data.0) {
        entry.1.push(data.1.clone());
    }
}
for data: &(CompactString, CompactString) in f3.iter() {
    if let Some(entry: &mut (SmallVec<[CompactString; 1]>, …)) = dict_a.get_mut(&data.0) {
        if !entry.1.is_empty() {
            entry.2.push(data.1.clone());
        }
    }
}
```



```
let stdout: Stdout = stdout();
let lock: StdoutLock<'static> = stdout.lock();
let mut buffer: BufWriter<StdoutLock<'static>> = BufWriter::new(inner: lock);
for (a_val: &CompactString, (f1_2: &SmallVec<[CompactString;…, f2_2, f3_2)) in dict_a.iter() {
    for f3_2_val: &CompactString in f3_2.iter() {
        if let Some(f4_2_list: &SmallVec<[CompactString; 1]>) = f4.get(f3_2_val) {
            for f4_2_val: &CompactString in f4_2_list.iter() {
                for f2_2_val: &CompactString in f2_2.iter() {
                    for f1_2_val: &CompactString in f1_2.iter() {
                        buffer.write(buf: f3_2_val.as_bytes());
                        buffer.write(buf: b",");
                        buffer.write(buf: a_val.as_bytes());
                        buffer.write(buf: b",");
                        buffer.write(buf: f1_2_val.as_bytes());
                        buffer.write(buf: b",");
                        buffer.write(buf: f2_2_val.as_bytes());
                        buffer.write(buf: b",");
                        buffer.write(buf: f4_2_val.as_bytes());
                        buffer.write(buf: b"\n");
                    }
                }
            }
        }
    }
}

buffer.flush().unwrap()
```

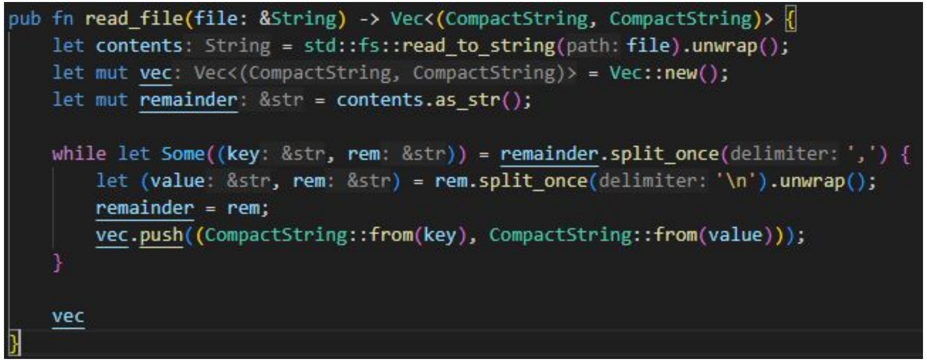
Reduced Hash - V1

Fewer HashMaps: Uses only one hash map for the join of the first three files

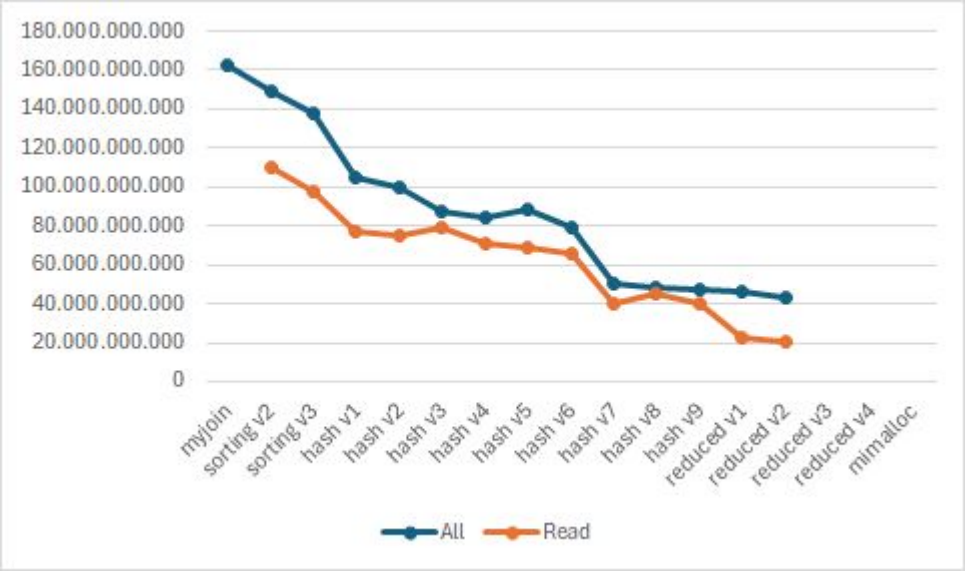


Reduced Hash - V2

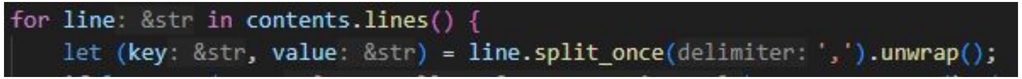
Parsing with split_once instead of lines() function - Not going over the string twice



```
pub fn read_file(file: &String) -> Vec<(CompactString, CompactString)> {
    let contents: String = std::fs::read_to_string(path: file).unwrap();
    let mut vec: Vec<(CompactString, CompactString)> = Vec::new();
    let mut remainder: &str = contents.as_str();

    while let Some((key: &str, rem: &str)) = remainder.split_once(delimiter: ',') {
        let (value: &str, rem: &str) = rem.split_once(delimiter: '\n').unwrap();
        remainder = rem;
        vec.push((CompactString::from(key), CompactString::from(value)));
    }

    vec
}
```



instead of:

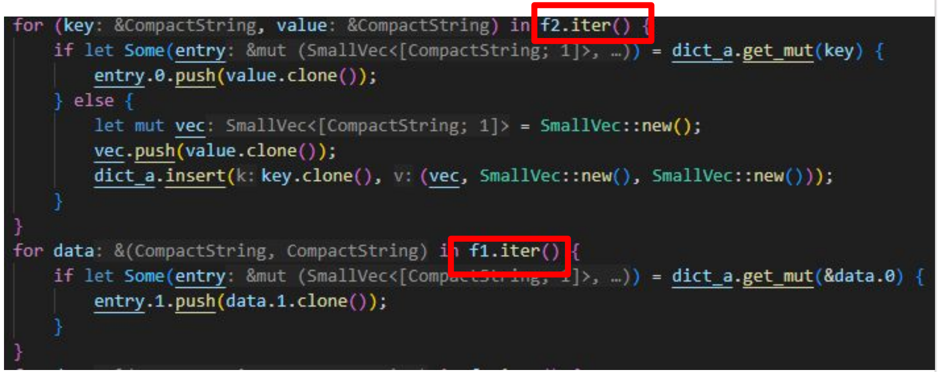


```
for line: &str in contents.lines() {
    let (key: &str, value: &str) = line.split_once(delimiter: ',').unwrap();
```

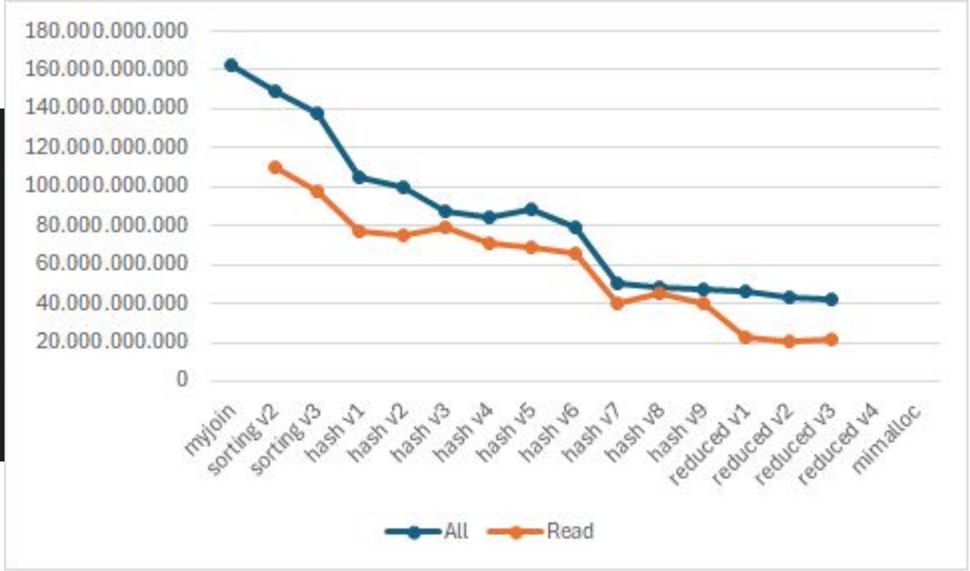
Reduced Hash - V3

- Reordering files:

Second file is smallest -> use second file to initialize the join hash map

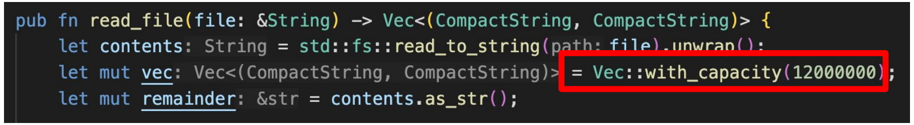


```
for (key: &CompactString, value: &CompactString) in f2.iter() {
    if let Some(entry: &mut (SmallVec<[CompactString; 1]>, …)) = dict_a.get_mut(key) {
        entry.0.push(value.clone());
    } else {
        let mut vec: SmallVec<[CompactString; 1]> = SmallVec::new();
        vec.push(value.clone());
        dict_a.insert(k: key.clone(), v: (vec, SmallVec::new(), SmallVec::new()));
    }
}
for data: &(CompactString, CompactString) in f1.iter() {
    if let Some(entry: &mut (SmallVec<[CompactString; 1]>, …)) = dict_a.get_mut(&data.0) {
        entry.1.push(data.1.clone());
    }
}
```

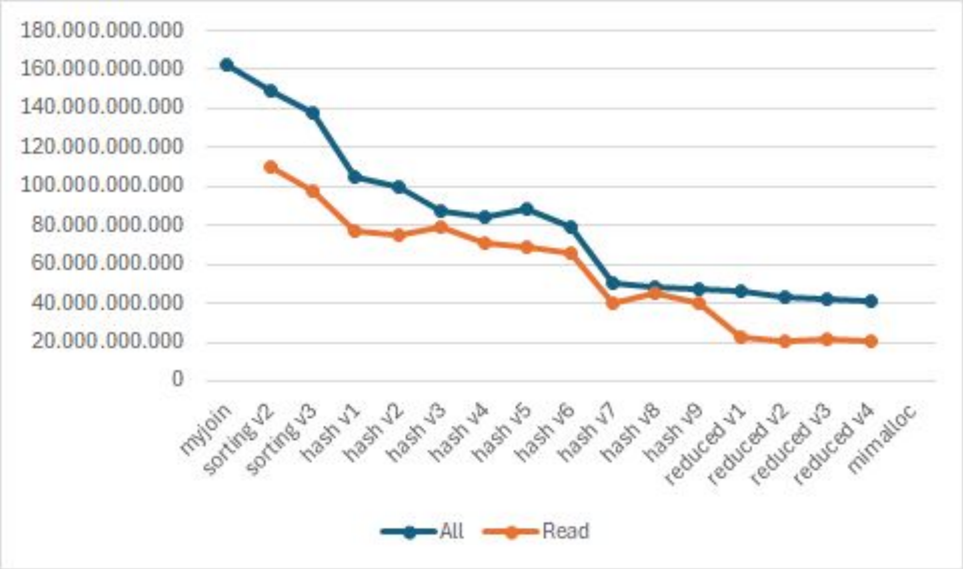


Reduced Hash - V4

Preallocated Vec Capacity: Allocates maximum vector capacity upfront to avoid resizing



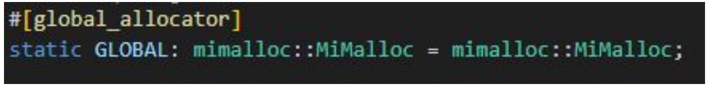
```
pub fn read_file(file: &String) -> Vec<(CompactString, CompactString)> {
    let contents: String = std::fs::read_to_string(path: file).unwrap();
    let mut vec: Vec<(CompactString, CompactString)> = Vec::with_capacity(12000000);
    let mut remainder: &str = contents.as_str();
```



Different allocator

- Tested different allocators:

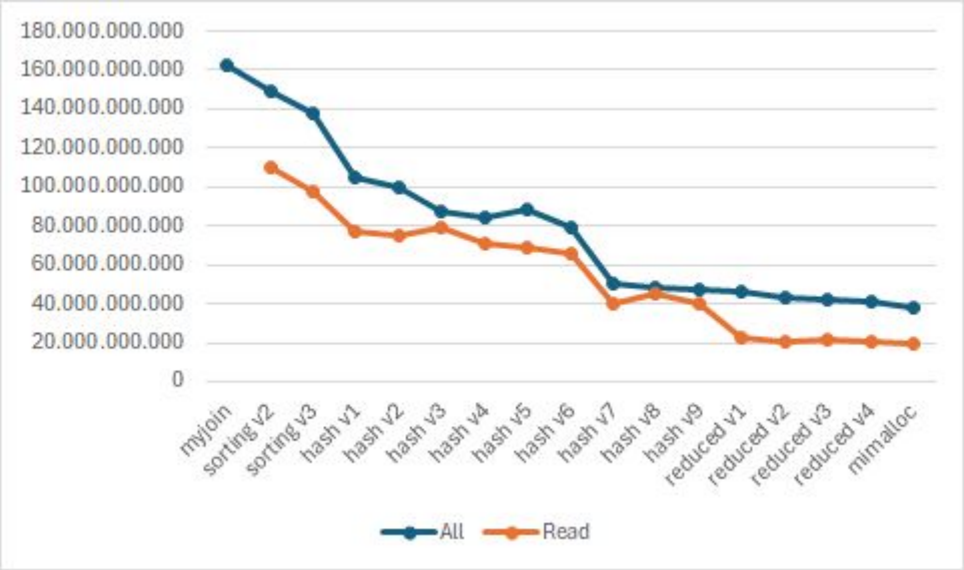
  MiMalloc was faster than Jemallocater (and default)



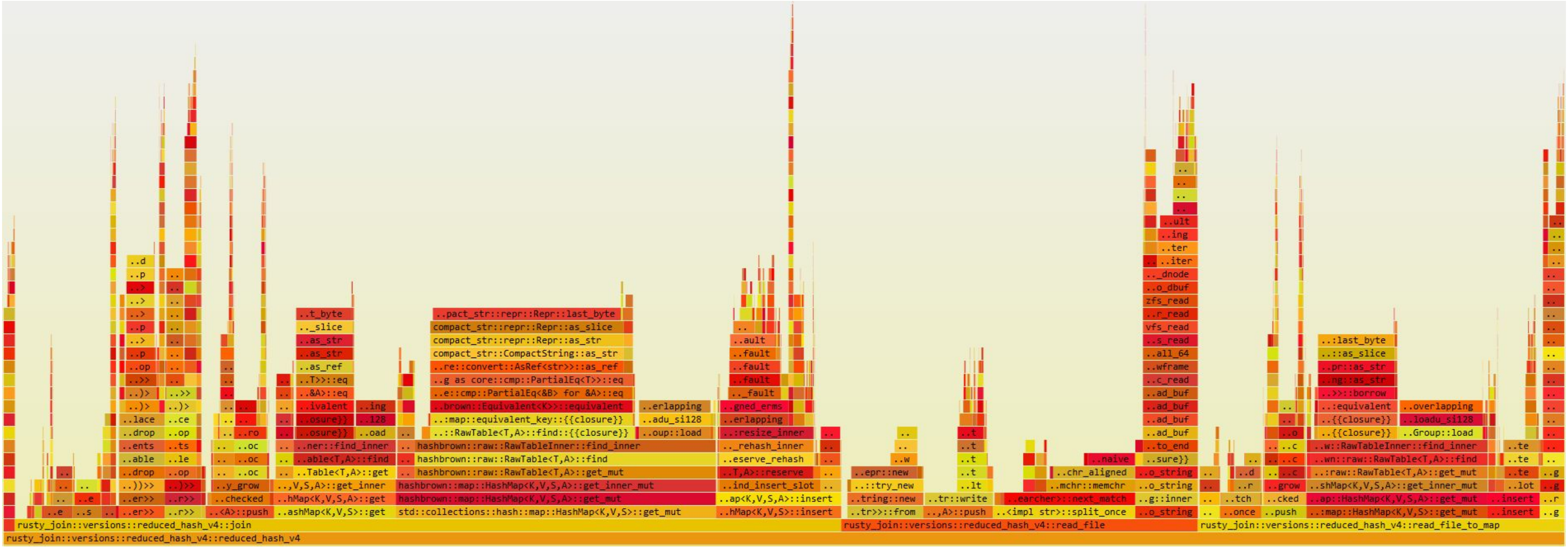
```
#[global_allocator]
static GLOBAL: mimalloc::MiMalloc = mimalloc::MiMalloc;
```

Final result single-threaded:

- rusty-join: 30 to 38.4b cycles
- my-join: 162.1b cycles



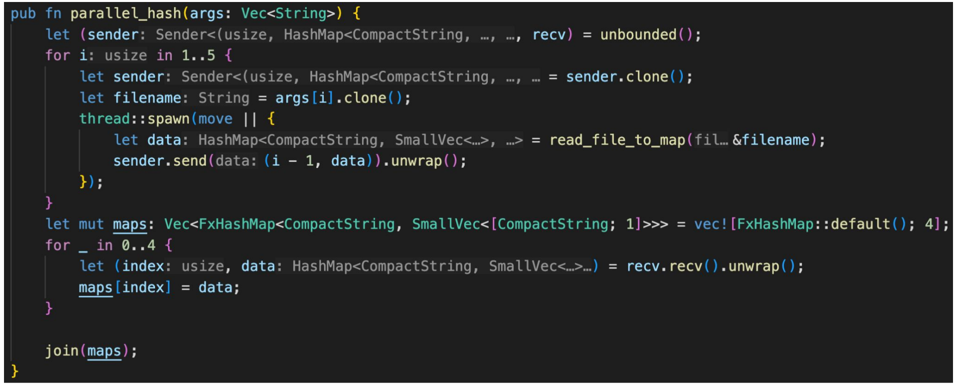
Flamegraph of Reduced Hash v4



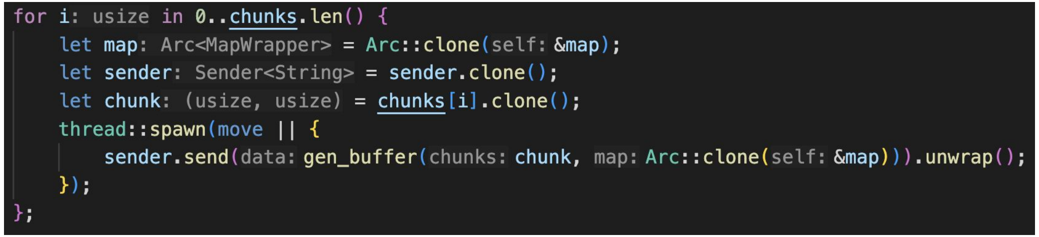
Optimization with Multithreading - V1 - Hash Join

Leverage multithreading to parallelize file reading and joining processes.

- Read files simultaneously
- In join, split first hash map into equally sized chunks
- Each worker performs join with his chunk and writes the result to a string
- Master prints the strings he receives
- Utilizes channels (from kanal crate) for communication between threads.



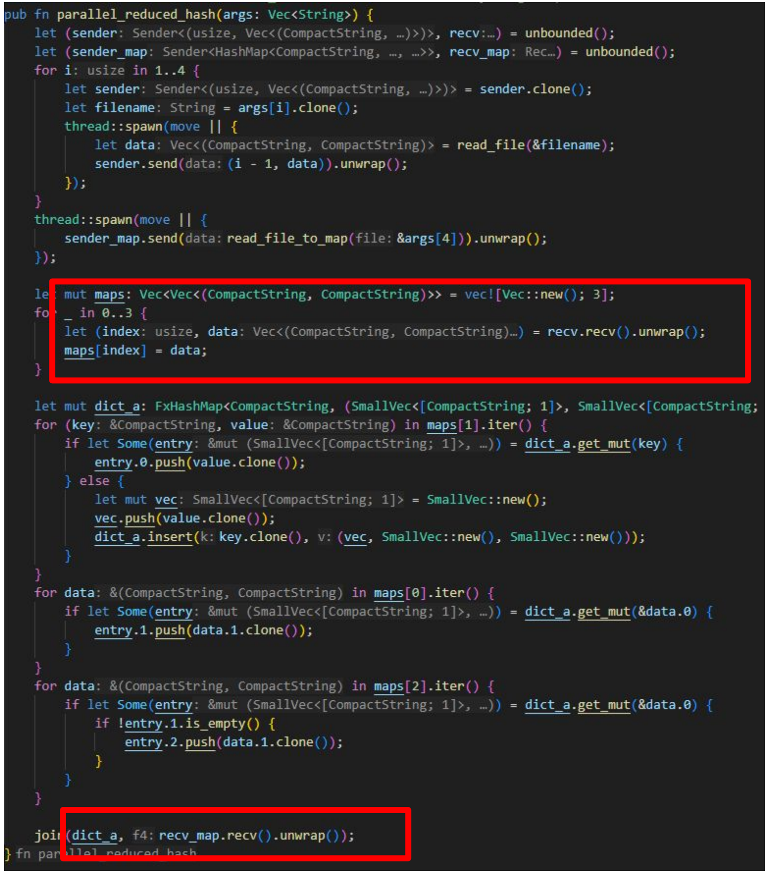
```rust
pub fn parallel_hash(args: Vec<String>) {
    let (sender: Sender<(usize, HashMap<CompactString, …, …, recv) = unbounded();
    for i: usize in 1..5 {
        let sender: Sender<(usize, HashMap<CompactString, …, … = sender.clone();
        let filename: String = args[i].clone();
        thread::spawn(move || {
            let data: HashMap<CompactString, SmallVec<…>, …> = read_file_to_map(fil… &filename);
            sender.send(data: (i - 1, data)).unwrap();
        });
    }
    let mut maps: Vec<FxHashMap<CompactString, SmallVec<[CompactString; 1]>>> = vec![FxHashMap::default(); 4];
    for _ in 0..4 {
        let (index: usize, data: HashMap<CompactString, SmallVec<…>…) = recv.recv().unwrap();
        maps[index] = data;
    }

    join(maps);
}
```



```rust
for i: usize in 0..chunks.len() {
    let map: Arc<MapWrapper> = Arc::clone(self: &map);
    let sender: Sender<String> = sender.clone();
    let chunk: (usize, usize) = chunks[i].clone();
    thread::spawn(move || {
        sender.send(data: gen_buffer(chunks: chunk, map: Arc::clone(self: &map))).unwrap();
    });
};
```

Optimization with Multithreading - V2 - Reduced Hash Join

- Files are read in parallel
- Once files 1-3 finish, create hashmap with them
- Once file 4 finishes, join
- Join parallelized the same way as for parallel hash join



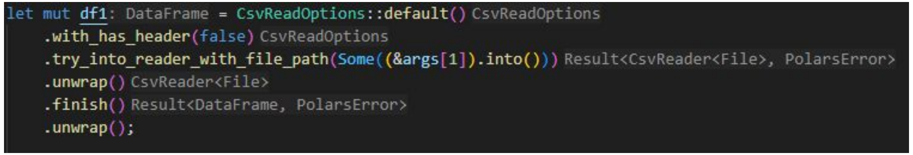
```
pub fn parallel_reduced_hash(args: Vec<String>) {
    let (sender: Sender<(usize, Vec<(CompactString, …)>)>, recv:…) = unbounded();
    let (sender_map: Sender<HashMap<CompactString, …, …>>, recv_map: Rec…) = unbounded();
    for i: usize in 1..4 {
        let sender: Sender<(usize, Vec<(CompactString, …)>)> = sender.clone();
        let filename: String = args[i].clone();
        thread::spawn(move || {
            let data: Vec<(CompactString, CompactString)> = read_file(&filename);
            sender.send(data: (i - 1, data)).unwrap();
        });
    }
    thread::spawn(move || {
        sender_map.send(data: read_file_to_map(file: &args[4])).unwrap();
    });

    let mut maps: Vec<Vec<(CompactString, CompactString)>> = vec![Vec::new(); 3];
    for _ in 0..3 {
        let (index: usize, data: Vec<(CompactString, CompactString)…) = recv.recv().unwrap();
        maps[index] = data;
    }

    let mut dict_a: FxHashMap<CompactString, (SmallVec<[CompactString; 1]>, SmallVec<[CompactString;
    for (key: &CompactString, value: &CompactString) in maps[1].iter() {
        if let Some(entry: &mut (SmallVec<[CompactString; 1]>, …)) = dict_a.get_mut(key) {
            entry.0.push(value.clone());
        } else {
            let mut vec: SmallVec<[CompactString; 1]> = SmallVec::new();
            vec.push(value.clone());
            dict_a.insert(k: key.clone(), v: (vec, SmallVec::new(), SmallVec::new()));
        }
    }
    for data: &(CompactString, CompactString) in maps[0].iter() {
        if let Some(entry: &mut (SmallVec<[CompactString; 1]>, …)) = dict_a.get_mut(&data.0) {
            entry.1.push(data.1.clone());
        }
    }
    for data: &(CompactString, CompactString) in maps[2].iter() {
        if let Some(entry: &mut (SmallVec<[CompactString; 1]>, …)) = dict_a.get_mut(&data.0) {
            if !entry.1.is_empty() {
                entry.2.push(data.1.clone());
            }
        }
    }

    join(dict_a, f4: recv_map.recv().unwrap());
} fn parallel_reduced_hash
```

Polars library for data frames

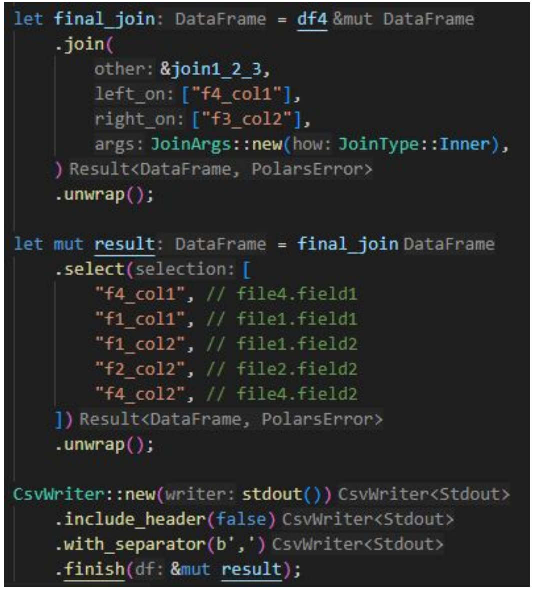
Leverage the Polars library for high-level, DataFrame-based joins.

Steps:

1. Data Loading
2. Join DataFrames
3. Select relevant columns and write the final output to a CSV format.



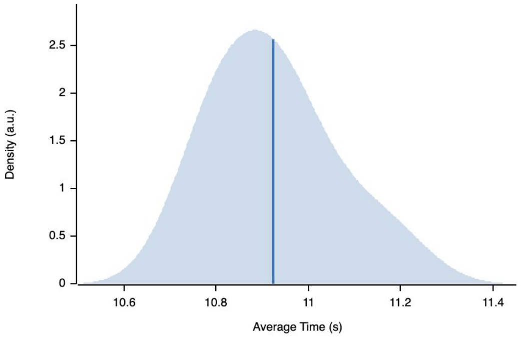
```
let mut df1: DataFrame = CsvReadOptions::default() CsvReadOptions
    .with_has_header(false) CsvReadOptions
    .try_into_reader_with_file_path(Some((&args[1]).into())) Result<CsvReader<File>, PolarsError>
    .unwrap() CsvReader<File>
    .finish() Result<DataFrame, PolarsError>
    .unwrap();
```



```
let final_join: DataFrame = df4 &mut DataFrame
    .join(
        other: &join1_2_3,
        left_on: ["f4_col1"],
        right_on: ["f3_col2"],
        args: JoinArgs::new(how: JoinType::Inner),
    ) Result<DataFrame, PolarsError>
    .unwrap();

let mut result: DataFrame = final_join DataFrame
    .select(selection: [
        "f4_col1", // file4.field1
        "f1_col1", // file1.field1
        "f1_col2", // file1.field2
        "f2_col2", // file2.field2
        "f4_col2", // file4.field2
    ]) Result<DataFrame, PolarsError>
    .unwrap();

CsvWriter::new(writer: stdout()) CsvWriter<Stdout>
    .include_header(false) CsvWriter<Stdout>
    .with_separator(b',') CsvWriter<Stdout>
    .finish(df: &mut result);
```

Criterion Benchmarks (Multithreaded)

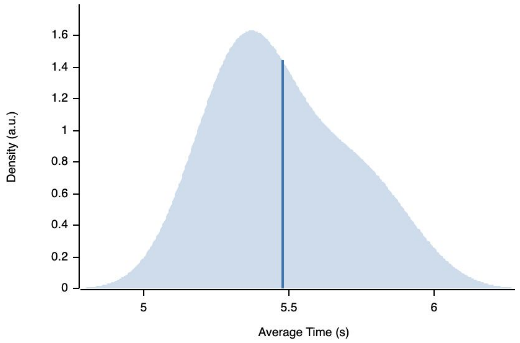
JoiningLarge/AntonErtlVersion/



Additional Statistics:

	Lower bound	Estimate	Upper bound
R²	0.0133855	0.0178964	0.0126196
Mean	10.851 s	10.925 s	11.006 s
Std. Dev.	66.149 ms	132.15 ms	170.86 ms
Median	10.801 s	10.917 s	11.013 s
MAD	28.887 ms	131.15 ms	225.02 ms

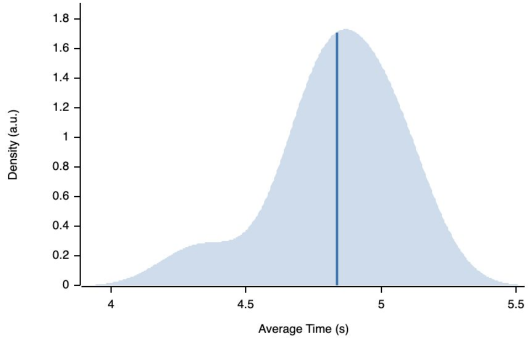
JoiningLarge/parallel_hash/La



Additional Statistics:

	Lower bound	Estimate	Upper bound
R²	0.0096769	0.0130084	0.0092006
Mean	5.3555 s	5.4772 s	5.6106 s
Std. Dev.	109.51 ms	217.18 ms	271.88 ms
Median	5.3168 s	5.4019 s	5.6875 s
MAD	10.760 ms	199.34 ms	349.21 ms

JoiningLarge/polards/SmallSet



Additional Statistics:

	Lower bound	Estimate	Upper bound
R²	0.0064507	0.0092586	0.0070404
Mean	4.6976 s	4.8362 s	4.9542 s
Std. Dev.	99.457 ms	220.47 ms	306.79 ms
Median	4.7382 s	4.8406 s	5.0075 s
MAD	25.878 ms	158.23 ms	342.91 ms

Summary

Worked great!	Not much changed	Didn’t work as expected
Algorithmic optimizations, stack-allocation, buffered output	loop unrolling, inlining	string slices