Databend

Databend

New Hash Table for Hash Join

徐金凯 Github ID: Dousir9

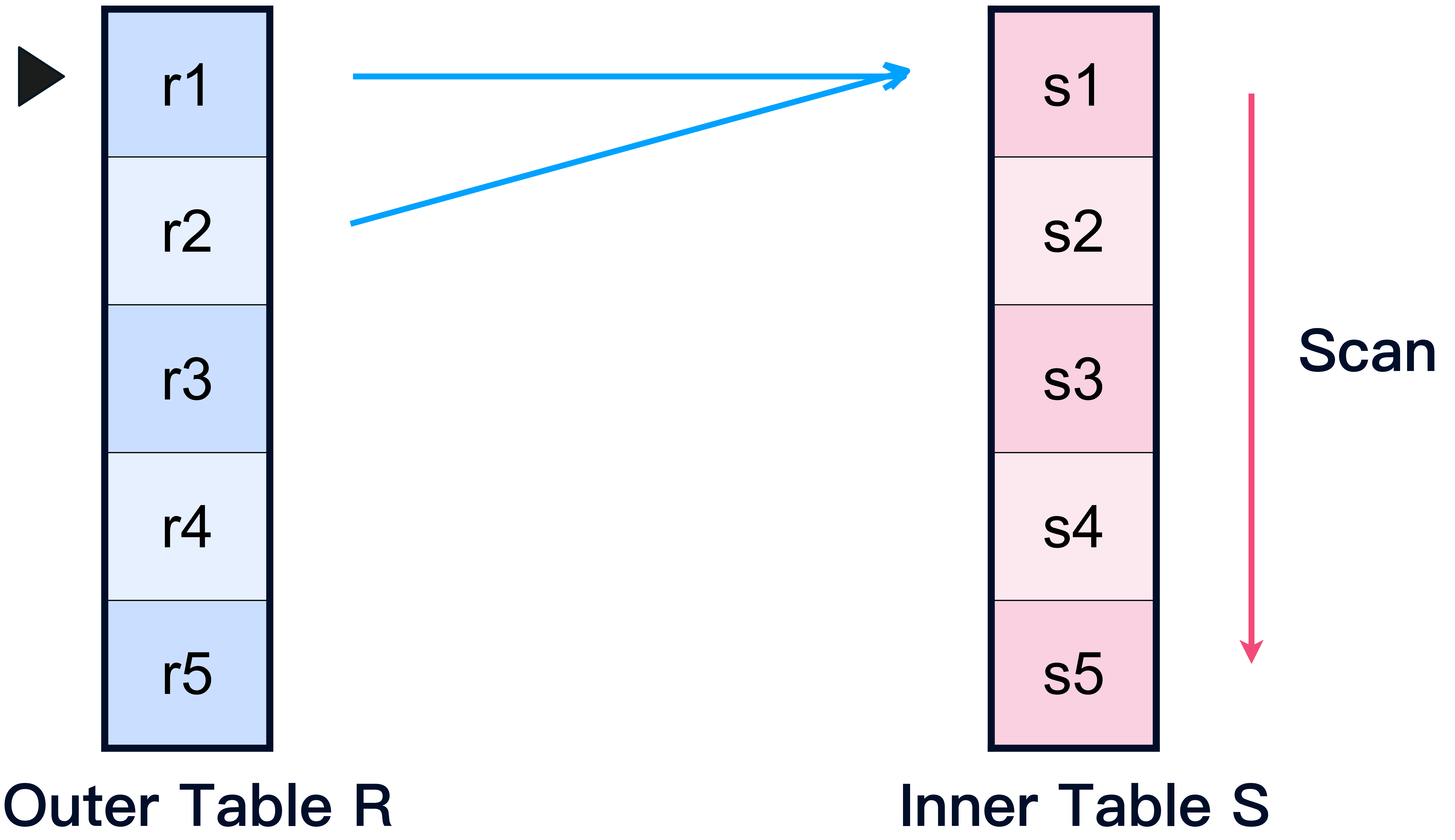
Content

Part 1

Introduction to Hash Join

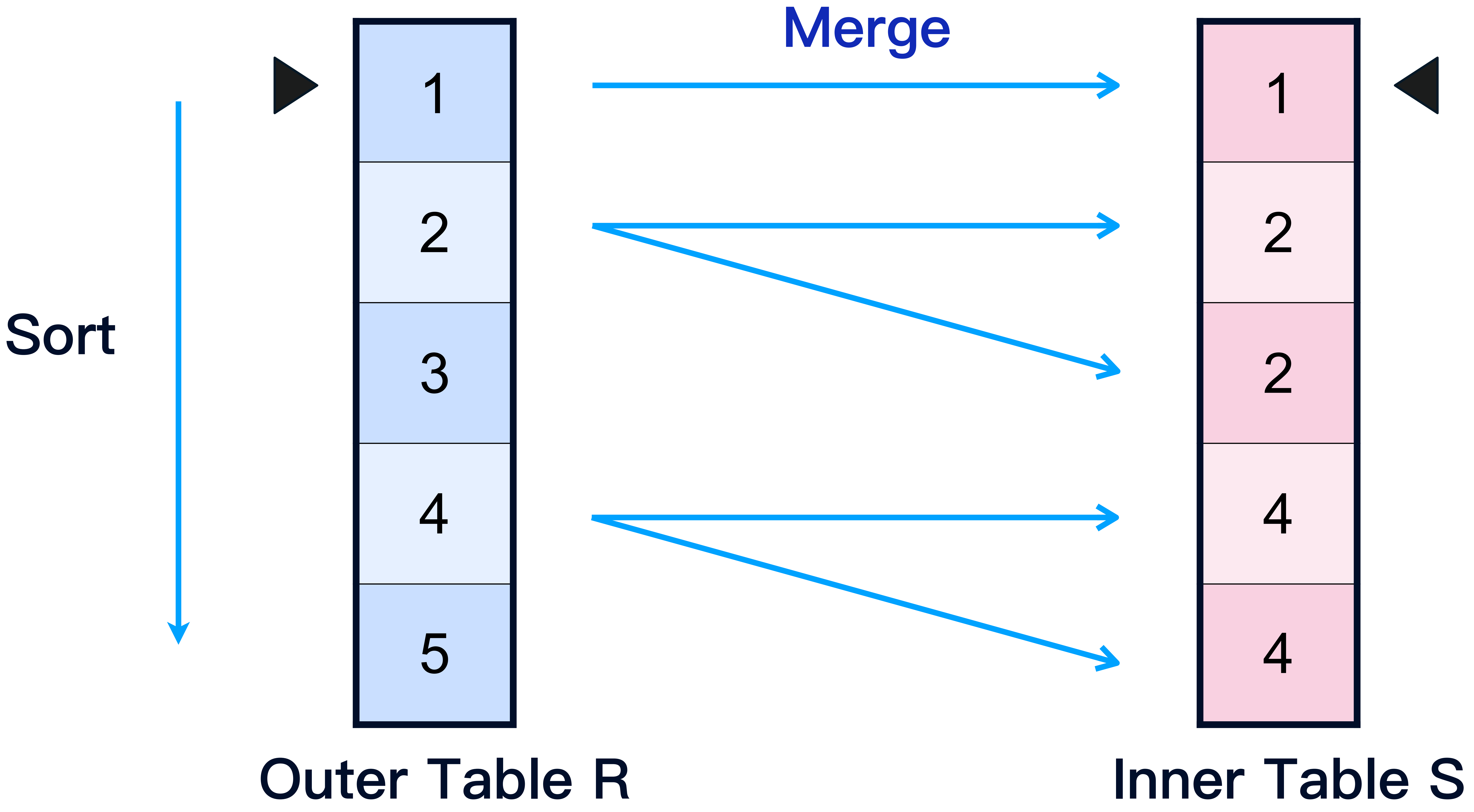
Join Type

Approach #1: Nested Loop Join (R⋈S)



Join Type

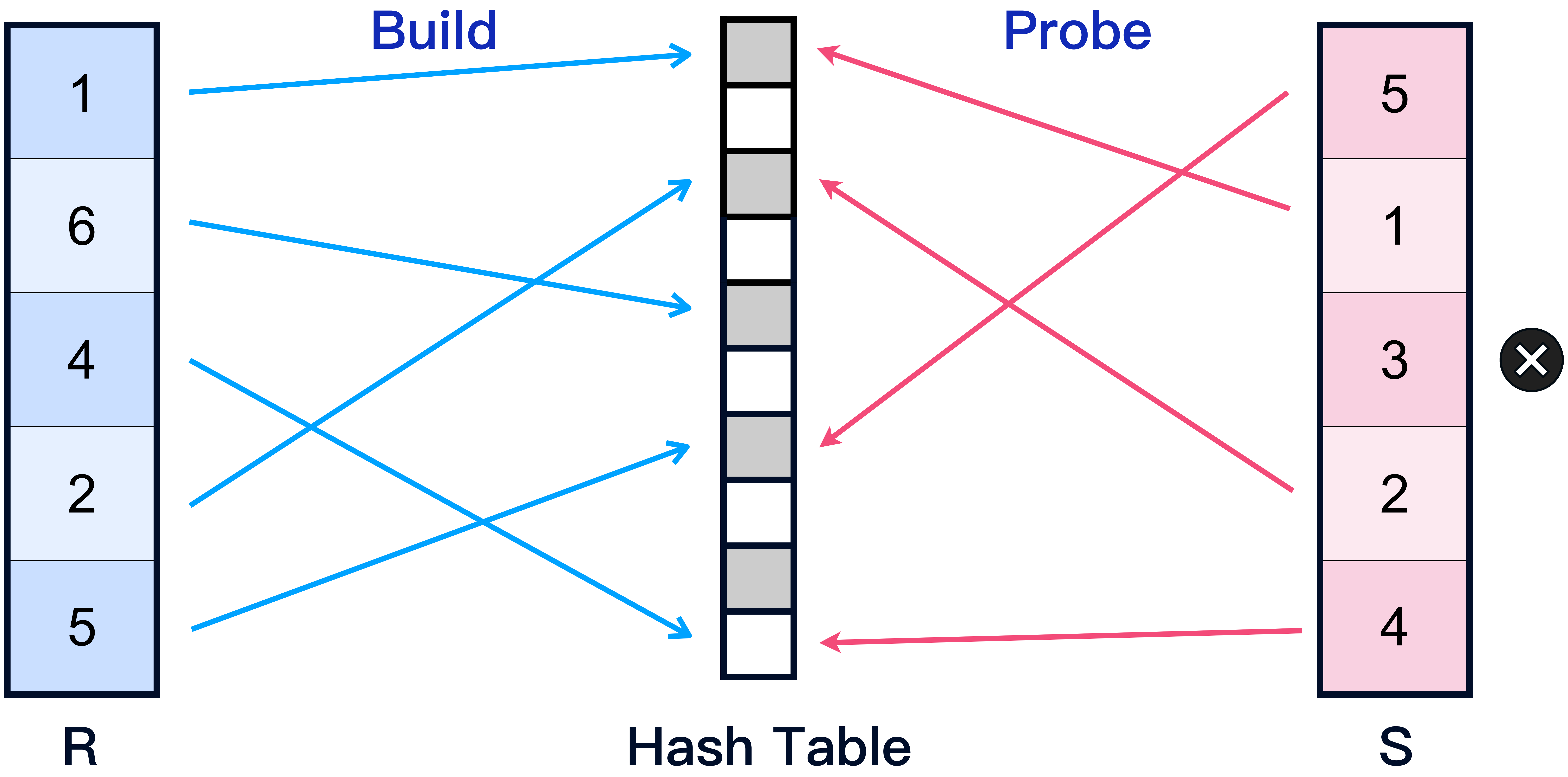
Approach #2: Sort Merge Join (R⋈S)



Join Type

Databend

Approach #3: Hash Join (R⋈S)



Hash Join

Databend

Phase #1: Partition(optional)

Phase #2: Build

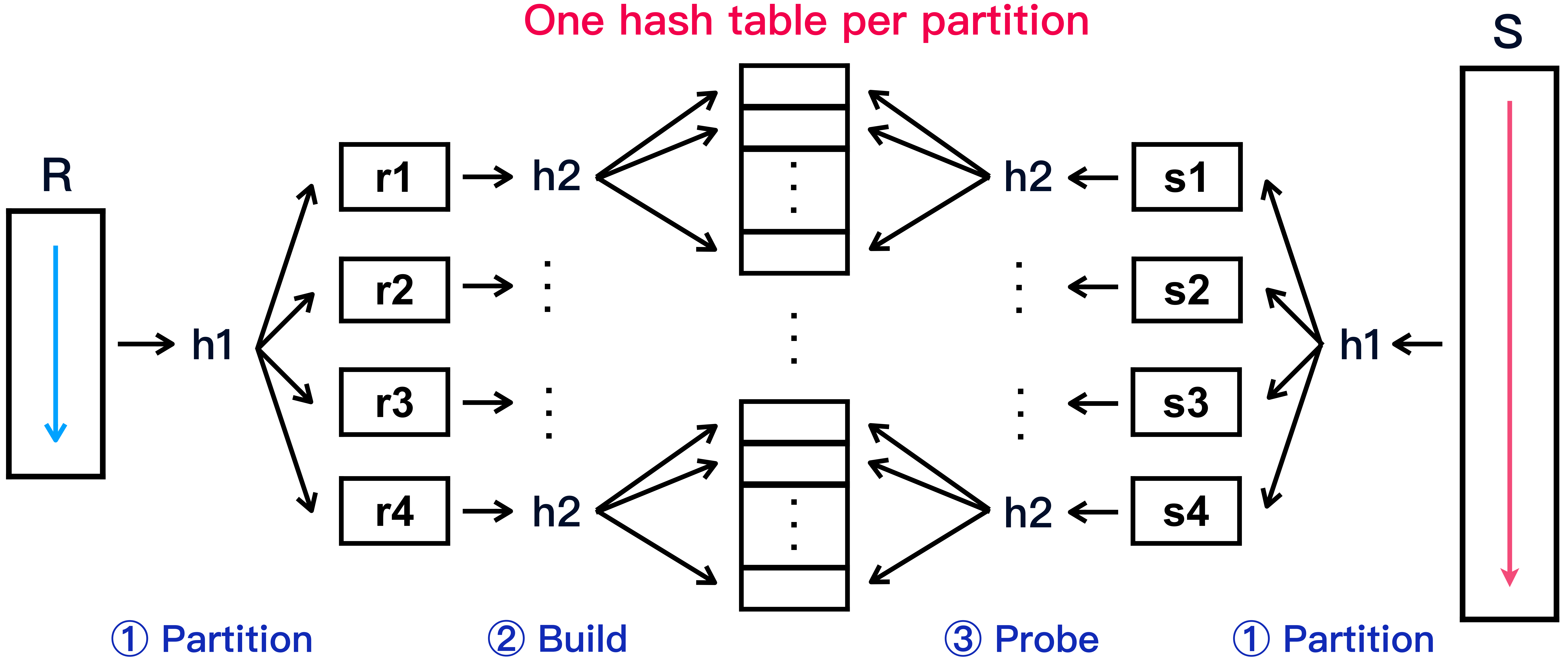
Phase #3: Probe

Hash Join

Databend

Phase #1: Partition(optional)

Why partition ?



Hash Join (R⋈S)

Databend

Phase #1: Partition(optional)

Approach #1: Non–Blocking Partitioning

→ Only scan the relation once

Approach #2: Blocking Partitioning

→ Scan the relation multiple times (generate histograms)

Hash Join (R⋈S)

Phase #2: Build

#1: Hash Function

→ Murmur, Crc, …

#2: Collision

→ Linear Probe Hasing

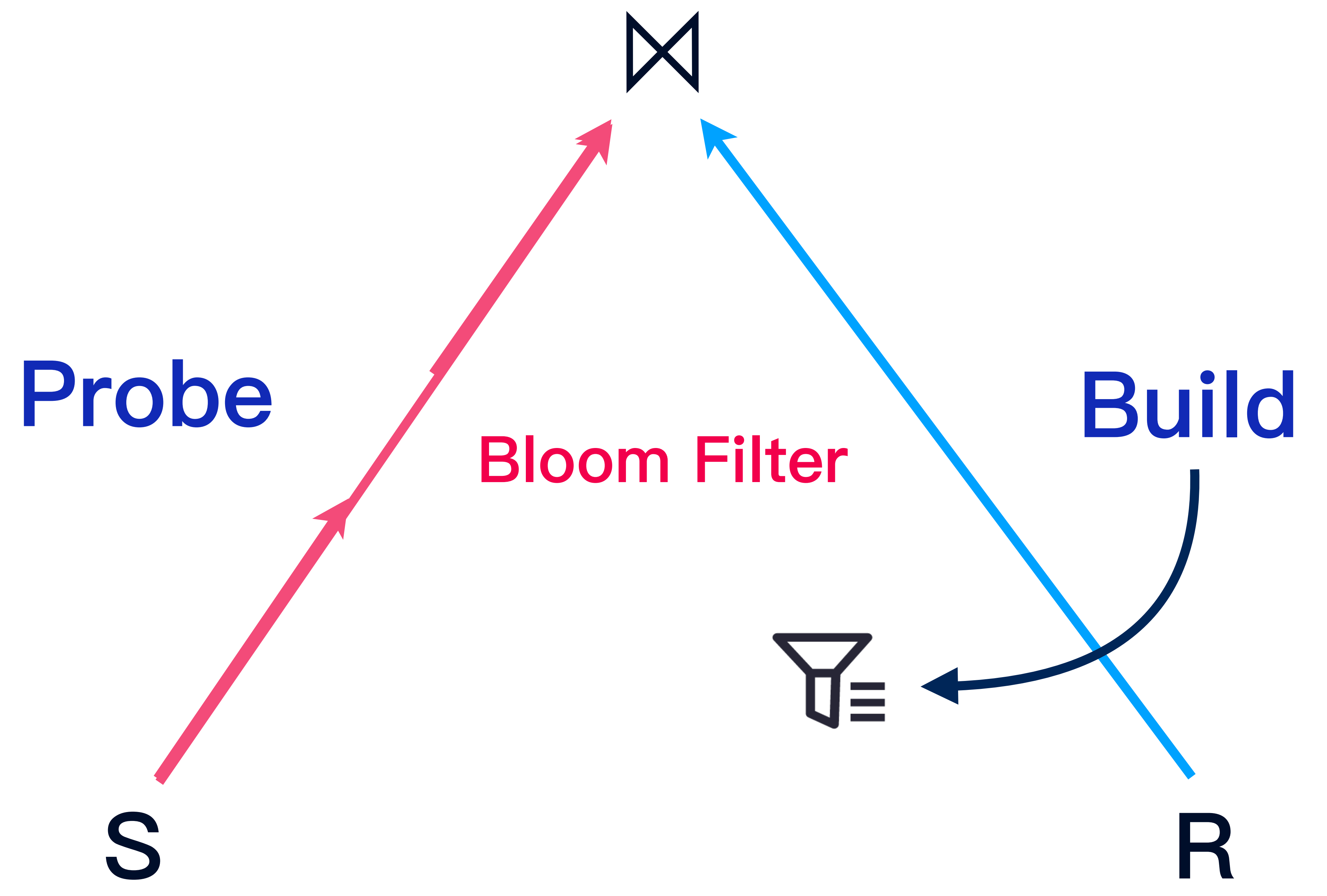
→ Chained Hashing

Hash Join (R⋈S)

Phase #3: Probe

Bloom Filter

→ Check the filter first



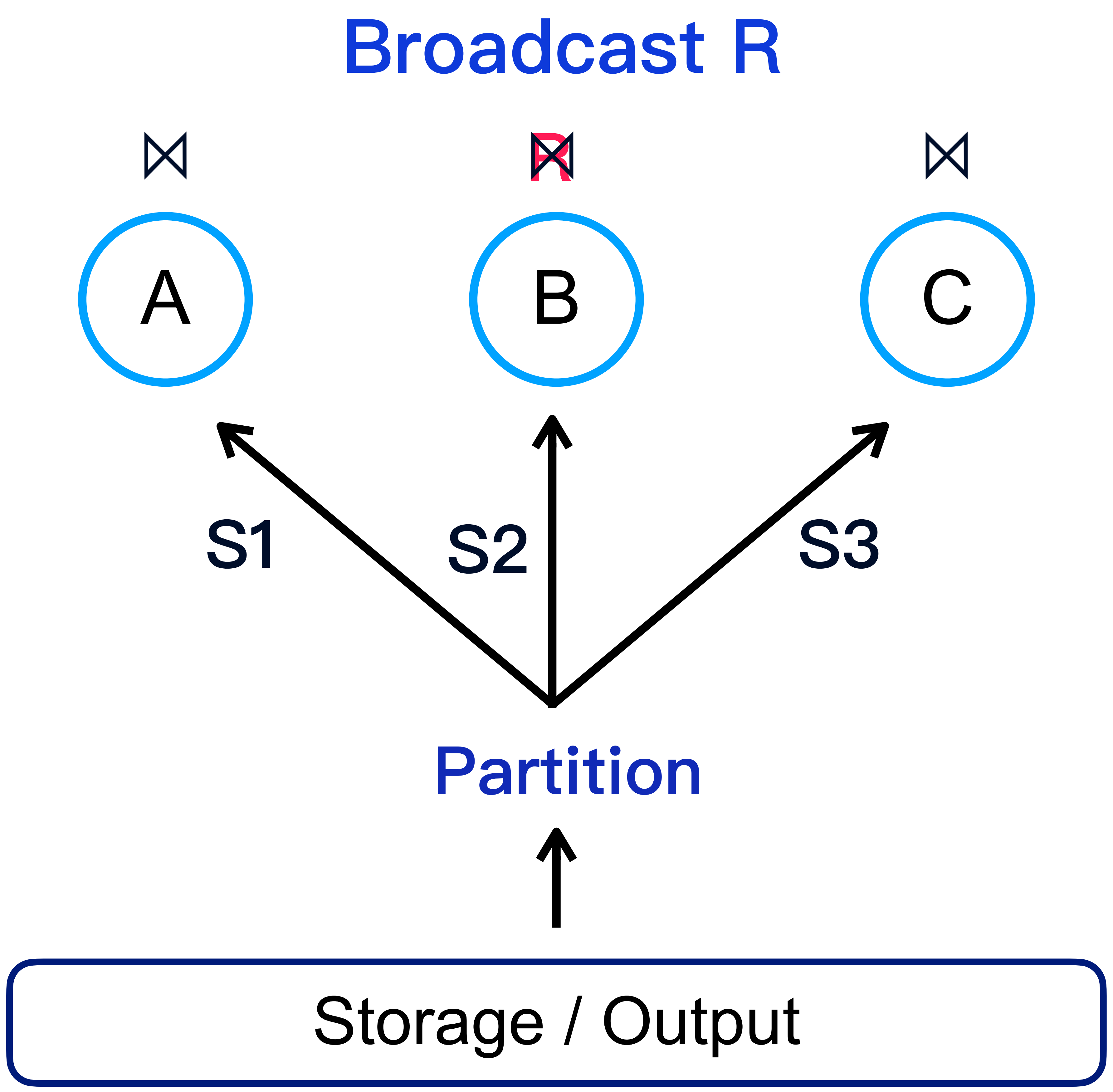
Databend

Distributed Join (R⋈S)

Approach #1: Broadcast Join

Approach #2: Shuffle Join

→ Join Key –> Hash Bucket



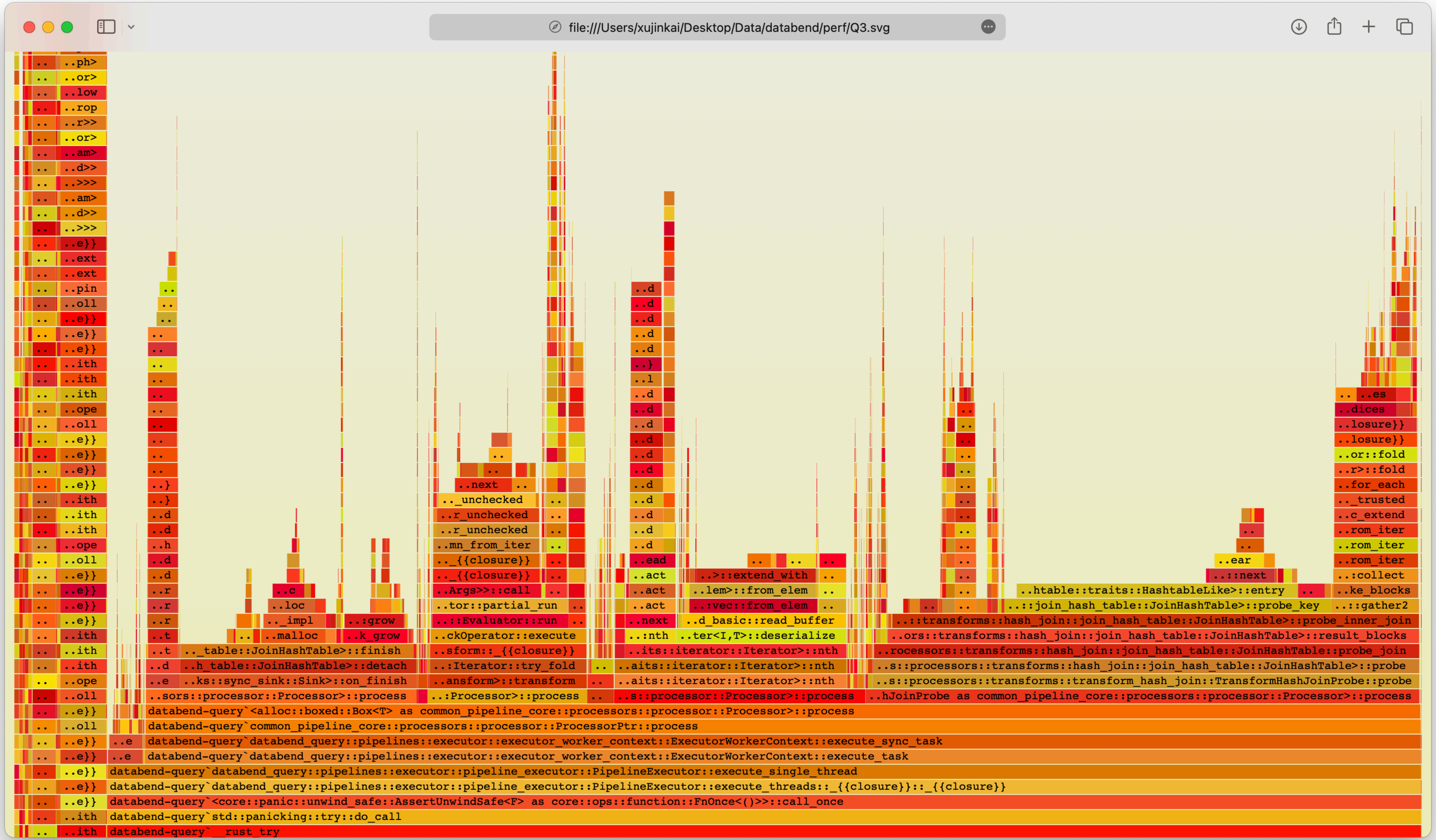
Databend

Part 2

New Hash Table and parallel finalize

Old Hash Table

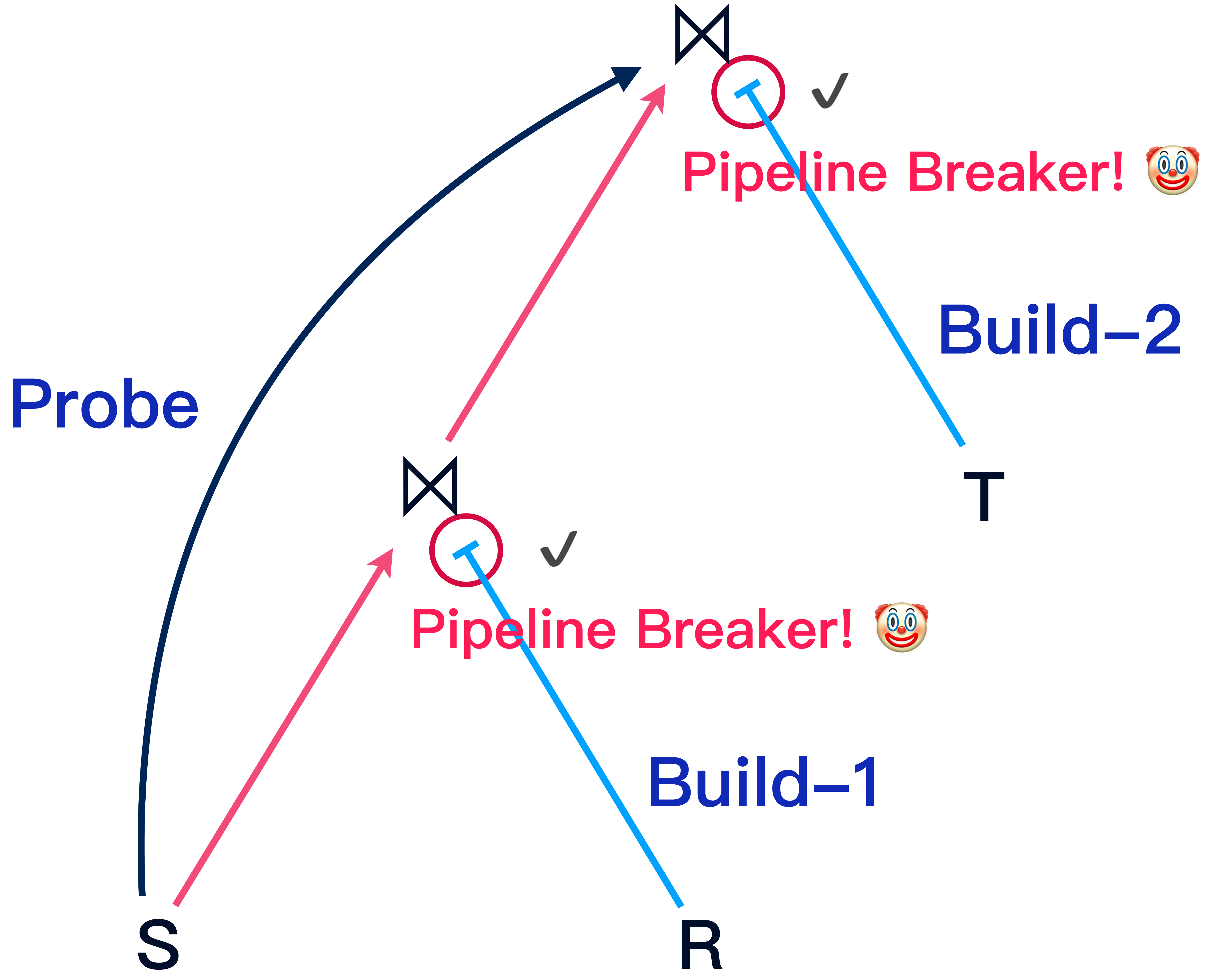
Databend



Hash Join Pipeline Model

Pipeline Breaker

The Pipeline needs to **wait for** Build–1 and Build–2



How to parallelize?

Databend

Problems

#1: Multi–Threading Build

→ ~~Lock~~ 🔐

#2: Resize Hash Table

→ ~~Lock~~ 🔐

→ ~~Grow and Copy~~ 😫

Lock Free! 🚀

No Grow! 🚀

New Has

```
hashtable.atomic_pointers = unsafe {
    std::mem::transmute::<*mut u64, *mut AtomicU64>(src: hashtable.pointers.as_mut_ptr())
};
```

Databend

Code

Insert

```rust
pub fn insert(&mut self, key: K, raw_entry_ptr: *mut RawEntry<K>) {
    let index: usize = key.hash() as usize & self.hash_mask;
    // # Safety
    // `index` is less than the capacity of hash table.
    let mut head: u64 = unsafe { (*self.atomic_pointers.add(count: index)).load(order: Ordering::Relaxed) };
    loop {
        let res: Result<u64, u64> = unsafe {
            (*self.atomic_pointers.add(count: index)).compare_exchange_weak(
                current: head,
                new: raw_entry_ptr as u64,
                success: Ordering::SeqCst,
                failure: Ordering::SeqCst,
            )
        };
        match res {
            Ok(_) => break,
            Err(x: u64) => head = x,
        };
    }
    unsafe { (*raw_entry_ptr).next = head };
}
```

Code

Probe

Databend

```rust
fn probe_hash_table(
    &self,
    key_ref: &Self::Key,
    vec_ptr: *mut RowPtr,
    mut occupied: usize,
    capacity: usize,
) -> (usize, u64) {
    let index: usize = key_ref.fast_hash() as usize & self.hash_mask;
    let origin: usize = occupied;
    let mut raw_entry_ptr: u64 = self.pointers[index];
    loop {
        if raw_entry_ptr == 0 || occupied >= capacity {
            break;
        }
        let raw_entry: &StringRawEntry = unsafe { &*(raw_entry_ptr as *mut StringRawEntry) };
        // Compare `early` and the length of the string, the size of `early` is 4.
        let min_len: usize = std::cmp::min(v1: STRING_EARLY_SIZE, v2: key_ref.len());
        if raw_entry.length as usize == key_ref.len()
            && key_ref[0..min_len] == raw_entry.early[0..min_len]
        {
            let key: &[u8] = unsafe {
                std::slice::from_raw_parts(
                    data: raw_entry.key as *const u8,
                    len: raw_entry.length as usize,
                )
            };
            if key == key_ref {
                // # Safety
                // occupied is less than the capacity of vec_ptr.
                unsafe {
                    std::ptr::copy_nonoverlapping(
                        src: &raw_entry.row_ptr as *const RowPtr,
                        dst: vec_ptr.add(count: occupied),
                        count: 1,
                    )
                };
                occupied += 1;
            }
        }
        raw_entry_ptr = raw_entry.next;
    }
    if occupied > origin {
        (occupied - origin, raw_entry_ptr)
```

Databend

Part 3

Benchmark

Databend

Benchmark

TPC-H: SF=30

→ CPU: M1 Pro 10core

Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11
—	149%	208%	530%	157%	—	568%	156%	293%	160%	125%

Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
112%	184%	308%	107%	136%	149%	191%	101%	258%	271%	721%

Thank you !

Q&A