Query Execution

Remy Wang

May 2026

Review how projection and filter work in relational algebra.

The first interesting operation is group by-aggregate.

We can implement it by first sorting, then summing over adjacent elements.

x	y
1	1
1	2
1	3
2	1
2	2

Because the table is sorted by x, each group is together, and we can sum over them.

A popular sorting algorithm is merge sort. Let's review. The key step in merge sort is merging two sorted lists.

A: **1**, 3, 5
B: **2**, 4, 6
out:

A: ~~1~~, **3**, 5
B: **2**, 4, 6
out: 1

A: ~~1~~, **3**, 5
B: ~~2~~, **4**, 6
out: 1, 2

A: ~~1~~, ~~3~~, **5**
B: ~~2~~, **4**, 6
out: 1, 2, 3

A: ~~1~~, ~~3~~, **5**
B: ~~2~~, ~~4~~, **6**
out: 1, 2, 3, 4

A: ~~1~~, ~~3~~, ~~5~~
B: ~~2~~, ~~4~~, **6**
out: 1, 2, 3, 4, 5

A: ~~1~~, ~~3~~, ~~5~~
B: ~~2~~, ~~4~~, ~~6~~
out: 1, 2, 3, 4, 5, 6

Using the merge operation, we can then implement sort recursively:

1. Break the input into two parts
2. Recursively sort each part
3. Merge them together

input:

[3, 1, 4, 1, 5, 9, 2, 6]

split:

[3, 1, 4, 1] [5, 9, 2, 6]

split:

[3, 1] [4, 1] [5, 9] [2, 6]

split:

[3] [1] [4] [1] [5] [9] [2] [6]

merge:

[1, 3] [1, 4] [5, 9] [2, 6]

merge:

[1, 1, 3, 4] [2, 5, 6, 9]

merge:

[1, 1, 2, 3, 4, 5, 6, 9]

Sorting can also be used to implement join.

A: **1**, 3, 5, 7
B: **2**, 3, 5, 6
out:

A: 1, **3**, 5, 7
B: **2**, 3, 5, 6
out:

A: 1, **3**, 5, 7
B: 2, **3**, 5, 6
out:

A: 1, 3, **5**, 7
B: 2, 3, **5**, 6
out: 3

A: 1, 3, 5, **7**
B: 2, 3, 5, **6**
out: 3, 5

A: 1, 3, 5, **7**
B: 2, 3, 5, 6
out: 3, 5

What if there are duplicates?

A: (3, a), (3, b), (5, c)
B: (3, x), (3, y), (5, z)

(letters distinguish rows with the same key)

Naively advancing both pointers on a match:

out: (3, a, x), (3, b, y), (5, c, z)

But we want *all* matching pairs — there should be 5.

When keys match, output the cross product of the *runs* of equal keys on both sides.

run in A: (3, a), (3, b)
run in B: (3, x), (3, y)
out: (3, a, x), (3, a, y), (3, b, x), (3, b, y)

Then advance past both runs.

A: (3, a), (3, b), **(5, c)**
B: (3, x), (3, y), **(5, z)**
out: . . . , (5, c, z)

Instead of explicitly detecting runs and computing cartesian product, we can also implement this by iterating over the matches *on the fly*.

Suppose we have found the first match:

A: **(3, a)**, (3, b), (3, c), (5, d)
B: **(3, x)**, (3, y), (3, z), (5, u)
out: (3, a, x)

We'll keep iterating until one side no longer matches:

A: **(3, a)**, (3, b), (3, c), (5, d)
B: (3, x), **(3, y)**, (3, z), (5, u)
out: (3, a, x), (3, a, y)

We'll keep iterating until one side no longer matches:

A: **(3, a)**, (3, b), (3, c), (5, d)
B: (3, x), (3, y), **(3, z)**, (5, u)
out: (3, a, x), (3, a, y), (3, a, z)

Then we *rewind* and take a step in the other table:

A: (3, a), **(3, b)**, (3, c), (5, d)
B: **(3, x)**, (3, y), (3, z), (5, u)
out: (3, a, x), (3, a, y), (3, a, z), (3, b, x)

Then keep iterating again:

A: (3, a), **(3, b)**, (3, c), (5, d)
B: (3, x), **(3, y)**, (3, z), (5, u)
out: (3, a, x), (3, a, y), (3, a, z), (3, b, x), (3, b, y)

Then keep iterating again:

A: (3, a), **(3, b)**, (3, c), (5, d)
B: (3, x), (3, y), **(3, z)**, (5, u)
out: (3, a, x), (3, a, y), (3, a, z), (3, b, x), (3, b, y), (3, b, z)

Rewind and step in A again:

A: (3, a), (3, b), **(3, c)**, (5, d)
B: **(3, x)**, (3, y), (3, z), (5, u)
out: (3, a, x), (3, a, y), (3, a, z), (3, b, x), (3, b, y), (3, b, z), (3, c, x)

Rewind and step in A again:

A: (3, a), (3, b), **(3, c)**, (5, d)
B: (3, x), **(3, y)**, (3, z), (5, u)
out: (3, a, x), (3, a, y), (3, a, z), (3, b, x), (3, b, y), (3, b, z), (3, c, x), (3, c, y)

This continues until we find the last match for the current value:

A: (3, a), (3, b), **(3, c)**, (5, d)
B: (3, x), (3, y), **(3, z)**, (5, u)
out: (3, a, x), (3, a, y), (3, a, z), (3, b, x), (3, b, y), (3, b, z), (3, c, x), (3, c, y), (3, c, z)

After which we move on to find the next matching value:

A: (3, a), (3, b), (3, c), **(5, d)**
B: (3, x), (3, y), (3, z), **(5, u)**
out: (3, a, x), (3, a, y), (3, a, z), (3, b, x), (3, b, y), (3, b, z), (3, c, x), (3, c, y), (3, c, z), (5, d, u)

We can also implement group-by with a *map* (dictionary in Python).

For example, to group by `x` and sum over `y`:

```
d = {}

for (x, y) in t:
  if x not in d:
    d[x] = y
  else:
    d[x] += y
```

Map can also be used to implement join:

```
d = {}

for (k, a) in A:
  if k not in d:
    d[k] = [a]
  else:
    d[k].append(a)

for (k, b) in B:
  if k in d:
    for a in d[k]:
      yield (k, a, b)
```

Under the hood, maps are implemented with *hash functions*. A (good) hash function has two important properties:

- The same input *must* be mapped to the same output
- Two different inputs are unlikely to be mapped to the same output

We can implement a map by using hash functions to determine the slot for each item based on the key:

Suppose `h(k) = k mod 10` with 10 slots. Insert 23, 91, 47:

0	1	2	3	4	5	6	7	8	9
	91		23				47		

A *hash collision* is when the hash function returns the same result for two different inputs. There are several methods to resolve collisions.

Chaining uses a linked list to store colliding items.

Insert 33 and 13 — both hash to slot 3:

0	1	2	3	4	5	6	7	8	9
	91		23→33→13				47		

Linear probing places an item in the immediate next available slot.

Starting from the same initial table, insert 33 (h=3): slot 3 is taken, so try slot 4.

0	1	2	3	4	5	6	7	8	9
	91		23	33			47		

An issue with linear probing is that, collisions can create clusters, which in turn makes further collisions more likely.

Continuing, insert 13 (h=3): slots 3, 4 taken, so it lands at slot 5. Then insert 4 (h=4): slots 4, 5 taken, so it lands at slot 6.

0	1	2	3	4	5	6	7	8	9
	91		23	33	13	4	47		

The cluster now spans slots 3–6, and any future key hashing into that range will extend it further.

To solve this, an alternative is *quadratic probing*, which makes increasingly larger skips for each item.

Using offsets $+1$, $+4$, $+9$, ..., starting from slots 3 and 4 occupied, insert 13 (h=3): $+1 \rightarrow$ slot 4 (taken), $+4 \rightarrow$ slot 7.

0	1	2	3	4	5	6	7	8	9
			23	33			13		

13 jumps over slots 5 and 6, avoiding the cluster.

In practice, however, linear probing performs better than the other two methods due to cache locality.