Vectorized Query Execution

The journey of vectorizing the merge joiner

presented by Yahor Yuzefovich

Agenda

1. What is a merge join algorithm
2. Outline of the implementation in the volcano model
3. Vectorizing the merge joiner
    a. single column equality
    b. multi column equality
    c. output population
    d. more optimizations

Credits: this presentation is largely based on George Utsin’s presentation

SELECT t1.k, t1.v, t2.v FROM t1 INNER MERGE JOIN t2 ON t1.k = t2.k

t1

k	v
1	A
2	C
2	D
4	H
5	J
6	L

t2

k	v
1	B
2	E
2	F
2	G
3	I
5	K

Output

SELECT t1.k, t1.v, t2.v FROM t1 INNER MERGE JOIN t2 ON t1.k = t2.k

t1

k	v
1	A
2	C
2	D
4	H
5	J
6	L

t2

k	v
1	B
2	E
2	F
2	G
3	I
5	K

Output

1	A	B
2	C	E
2	C	F
2	C	G
2	D	E
2	D	F
2	D	G
5	J	K

SELECT t1.k, t1.v, t2.v FROM t1 INNER MERGE JOIN t2 ON t1.k = t2.k

t1

1	A
2	C
2	D
4	H
5	J
6	L

t2

1	B
2	E
2	F
2	G
3	I
5	K

Output

SELECT t1.k, t1.v, t2.v FROM t1 INNER MERGE JOIN t2 ON t1.k = t2.k

t1

1	A
2	C
2	D
4	H
5	J
6	L

t2

1	B
2	E
2	F
2	G
3	I
5	K

Output

1	A	B

SELECT t1.k, t1.v, t2.v FROM t1 INNER MERGE JOIN t2 ON t1.k = t2.k

t1

1	A
2	C
2	D
4	H
5	J
6	L

t2

1	B
2	E
2	F
2	G
3	I
5	K

Output

1	A	B

SELECT t1.k, t1.v, t2.v FROM t1 INNER MERGE JOIN t2 ON t1.k = t2.k

t1

1	A
2	C
2	D
4	H
5	J
6	L

t2

1	B
2	E
2	F
2	G
3	I
5	K

Output

1	A	B
2	C	D

SELECT t1.k, t1.v, t2.v FROM t1 INNER MERGE JOIN t2 ON t1.k = t2.k

t1

1	A
2	C
2	D
4	H
5	J
6	L

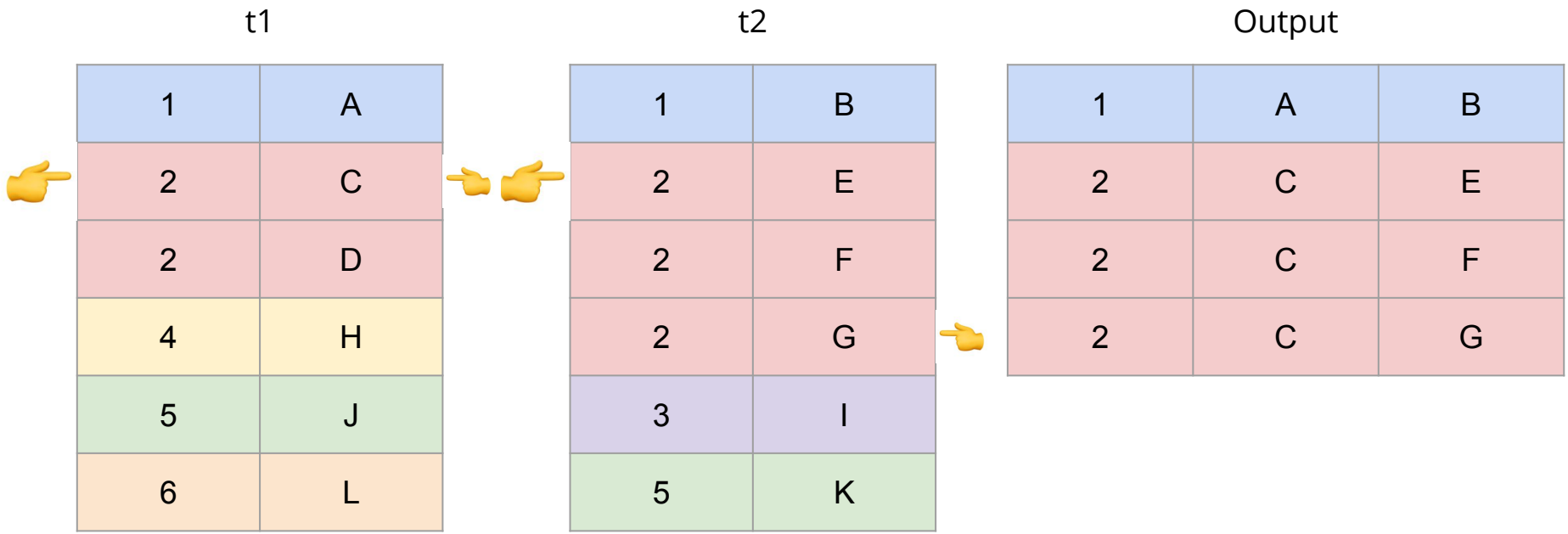
t2

1	B
2	E
2	F
2	G
3	I
5	K

Output

1	A	B
2	C	D
2	C	F

SELECT t1.k, t1.v, t2.v FROM t1 INNER MERGE JOIN t2 ON t1.k = t2.k



SELECT t1.k, t1.v, t2.v FROM t1 INNER MERGE JOIN t2 ON t1.k = t2.k

t1

1	A
2	C
2	D
4	H
5	J
6	L

t2

1	B
2	E
2	F
2	G
3	I
5	K

Output

1	A	B
2	C	E
2	C	F
2	C	G
2	D	E

SELECT t1.k, t1.v, t2.v FROM t1 INNER MERGE JOIN t2 ON t1.k = t2.k

t1

1	A
2	C
2	D
4	H
5	J
6	L

t2

1	B
2	E
2	F
2	G
3	I
5	K

Output

1	A	B
2	C	E
2	C	F
2	C	G
2	D	E

SELECT t1.k, t1.v, t2.v FROM t1 INNER MERGE JOIN t2 ON t1.k = t2.k

t1

1	A
2	C
2	D
4	H
5	J
6	L

t2

1	B
2	E
2	F
2	G
3	I
5	K

Output

1	A	B
2	C	E
2	C	F
2	C	G
2	D	E
2	D	F

SELECT t1.k, t1.v, t2.v FROM t1 INNER MERGE JOIN t2 ON t1.k = t2.k

t1

1	A
2	C
2	D
4	H
5	J
6	L

t2

1	B
2	E
2	F
2	G
3	I
5	K

Output

1	A	B
2	C	E
2	C	F
2	C	G
2	D	E
2	D	F
2	D	G

SELECT t1.k, t1.v, t2.v FROM t1 INNER MERGE JOIN t2 ON t1.k = t2.k

t1

1	A
2	C
2	D
4	H
5	J
6	L

t2

1	B
2	E
2	F
2	G
3	I
5	K

Output

1	A	B
2	C	E
2	C	F
2	C	G
2	D	E
2	D	F
2	D	G

SELECT t1.k, t1.v, t2.v FROM t1 INNER MERGE JOIN t2 ON t1.k = t2.k

t1

1	A
2	C
2	D
4	H
5	J
6	L

t2

1	B
2	E
2	F
2	G
3	I
5	K

Output

1	A	B
2	C	E
2	C	F
2	C	G
2	D	E
2	D	F
2	D	G

SELECT t1.k, t1.v, t2.v FROM t1 INNER MERGE JOIN t2 ON t1.k = t2.k

t1

1	A
2	C
2	D
4	H
5	J
6	L

t2

1	B
2	E
2	F
2	G
3	I
5	K

Output

1	A	B
2	C	E
2	C	F
2	C	G
2	D	E
2	D	F
2	D	G

SELECT t1.k, t1.v, t2.v FROM t1 INNER MERGE JOIN t2 ON t1.k = t2.k

t1

1	A
2	C
2	D
4	H
5	J
6	L

t2

1	B
2	E
2	F
2	G
3	I
5	K

Output

1	A	B
2	C	E
2	C	F
2	C	G
2	D	E
2	D	F
2	D	G
5	J	K

SELECT t1.k, t1.v, t2.v FROM t1 INNER MERGE JOIN t2 ON t1.k = t2.k

t1

1	A
2	C
2	D
4	H
5	J
6	L

t2

1	B
2	E
2	F
2	G
3	I
5	K

Output

1	A	B
2	C	E
2	C	F
2	C	G
2	D	E
2	D	F
2	D	G
5	J	K

SELECT t1.k, t1.v, t2.v FROM t1 INNER MERGE JOIN t2 ON t1.k = t2.k

t1

1	A
2	C
2	D
4	H
5	J
6	L

t2

1	B
2	E
2	F
2	G
3	I
5	K

Output

1	A	B
2	C	E
2	C	F
2	C	G
2	D	E
2	D	F
2	D	G
5	J	K

Agenda

1. What is a merge join algorithm
2. Outline of the implementation in the volcano model
3. Vectorizing the merge joiner
    a. single column equality
    b. multi column equality
    c. output population
    d. more optimizations

Merge joiner outline

1. consume both inputs a group at a time (all rows identical on equality columns)
    a. this requires row-at-a-time comparison against the “head” of the group
2. maintaining the cursors in each group, render an output row
    a. copying rows from both sides one-at-a-time

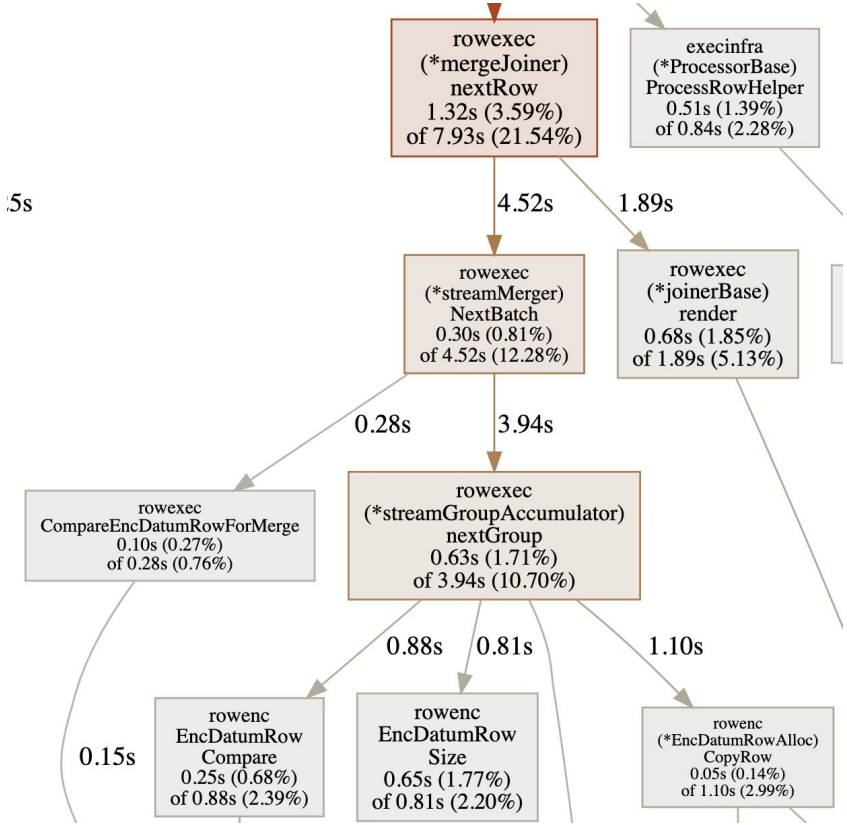
Merge joiner, finding groups

```
for {
    row, _ := s.src.Next()
    if row == nil {
        s.srcConsumed = true
        return s.curGroup, nil
    }

    if len(s.curGroup) == 0 {
        if s.curGroup == nil {
            s.curGroup = make([]rowenc.EncDatumRow, 0, 64)
        }
        s.curGroup = append(s.curGroup, row)
        continue
    }

    cmp, _ := s.curGroup[0].Compare(s.types, &s.datumAlloc, s.ordering, evalCtx, row)
    if cmp == 0 {
        s.curGroup = append(s.curGroup, row)
    } else {
        n := len(s.curGroup)
        ret := s.curGroup[:n:n]
        s.curGroup = s.curGroup[:0]
        s.leftoverRow = row
        return ret, nil
    }
}
```

```
for _, c := range ordering {
    cmp, _ := r[c.ColIdx].Compare(types[c.ColIdx]
    if cmp != 0 {
        if c.Direction == encoding.Descending {
            cmp = -cmp
        }
        return cmp, nil
    }
}
return 0, nil
```

```
// Datum represents a SQL value.
type Datum interface {
    TypedExpr

    // Compare returns -1 if the receiver is less than other, 0 if receiver is
    // equal to other and +1 if receiver is greater than other.
    Compare(ctx *EvalContext, other Datum) int
```

Merge joiner, finding groups



Merge joiner, rendering output

```
jb.combinedRow = jb.combinedRow[:len(lrow)+len(rrow)]
copy(jb.combinedRow, lrow)
copy(jb.combinedRow[len(lrow):], rrow)
return jb.combinedRow, nil
```

```
BenchmarkMergeJoiner/InputSize=4-8          200000      7543 ns/op      8.48 MB/s
BenchmarkMergeJoiner/InputSize=16-8         200000     10644 ns/op     24.05 MB/s
BenchmarkMergeJoiner/InputSize=256-8         20000     81155 ns/op     50.47 MB/s
BenchmarkMergeJoiner/InputSize=4096-8         2000   1177922 ns/op     55.64 MB/s
BenchmarkMergeJoiner/InputSize=65536-8         100  19149506 ns/op     54.76 MB/s
```

```
BenchmarkMergeJoiner/InputSize=4-8          200000      7543 ns/op      8.48 MB/s
BenchmarkMergeJoiner/InputSize=16-8         200000     10644 ns/op     24.05 MB/s
BenchmarkMergeJoiner/InputSize=256-8         20000     81155 ns/op     50.47 MB/s
BenchmarkMergeJoiner/InputSize=4096-8         2000   1177922 ns/op     55.64 MB/s
BenchmarkMergeJoiner/InputSize=65536-8         100  19149506 ns/op     54.76 MB/s
```

Agenda

1. What is a merge join algorithm
2. Outline of the implementation in the volcano model
3. Vectorizing the merge joiner
    a. single column equality
    b. multi column equality
    c. output population
    d. more optimizations

Vectorized execution engine

Inspired by MonetDB/X100 paper

Operate on batches of up to 1024 rows at once

Avoid row-at-a-time operations

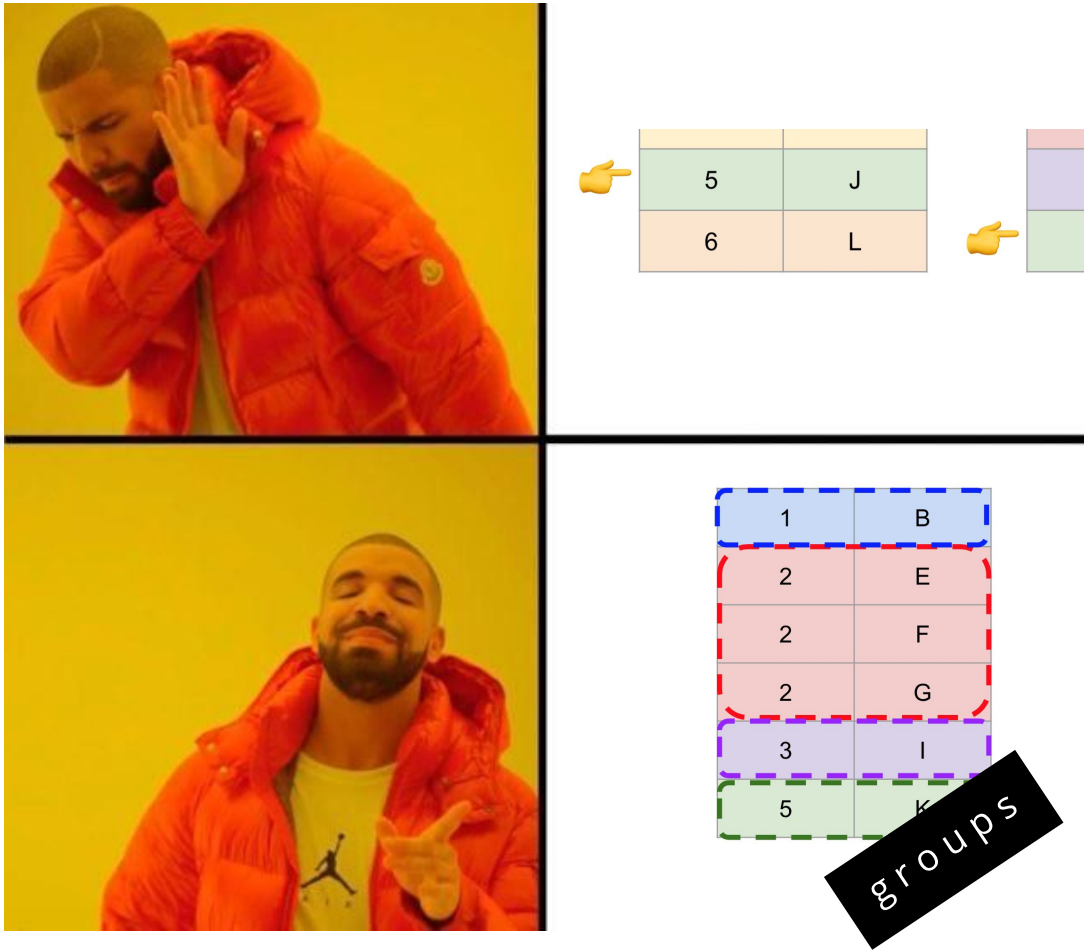
Doesn’t use vectorized (SIMD) instructions (yet)

What does it mean to “vectorize”?

Instead of performing comparisons “row by row”, do it “column by column”

What does it mean to “vectorize”?

Instead of performing comparisons “row by row”, do it “column by column”



SELECT t1.k, t1.v, t2.v FROM t1 INNER MERGE JOIN t2 ON t1.k = t2.k

t1

1	A
2	C
2	D
4	H
5	J
6	L

t2

1	B
2	E
2	F
2	G
3	I
5	K

Output

SELECT t1.k, t1.v, t2.v FROM t1 INNER MERGE JOIN t2 ON t1.k = t2.k

t1

1	A
2	C
2	D
4	H
5	J
6	L

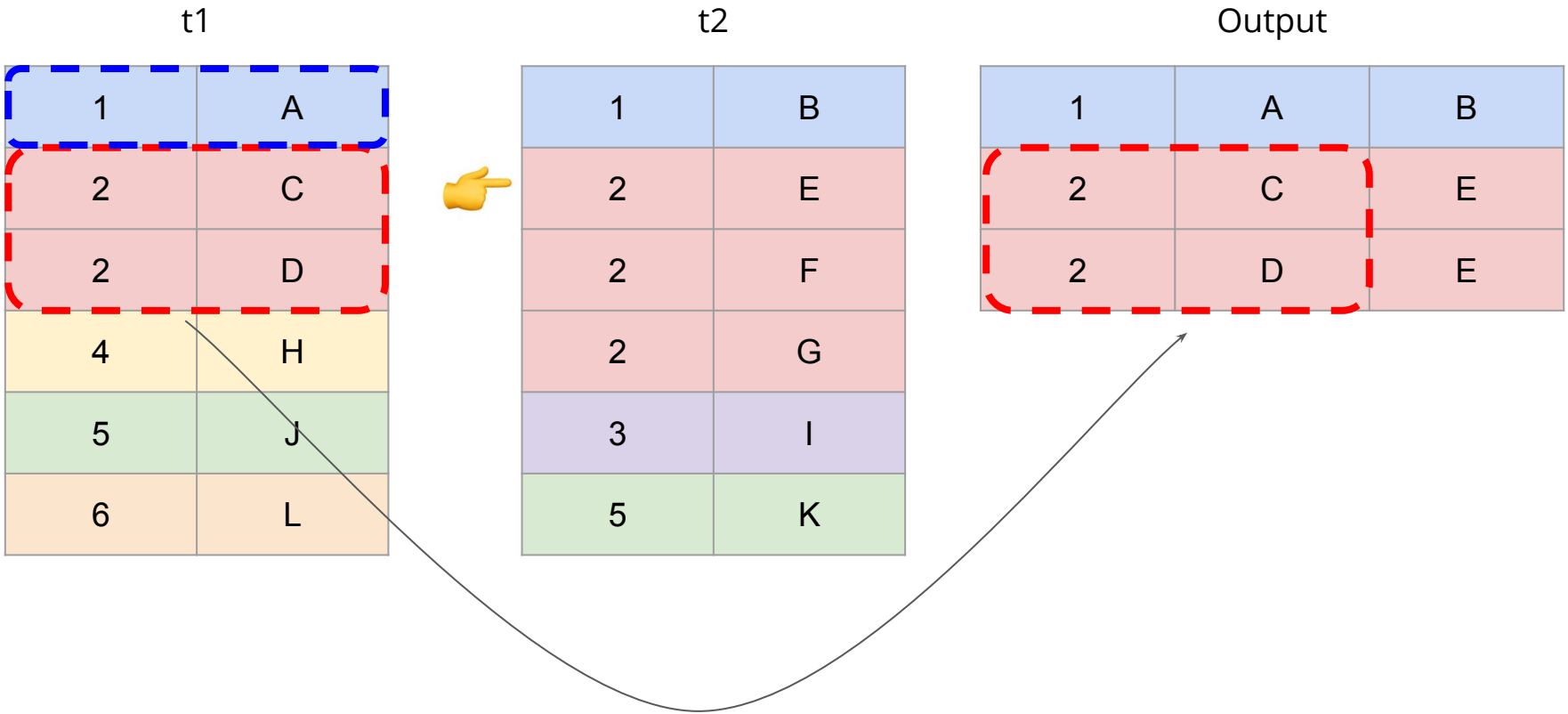
t2

1	B
2	E
2	F
2	G
3	I
5	K

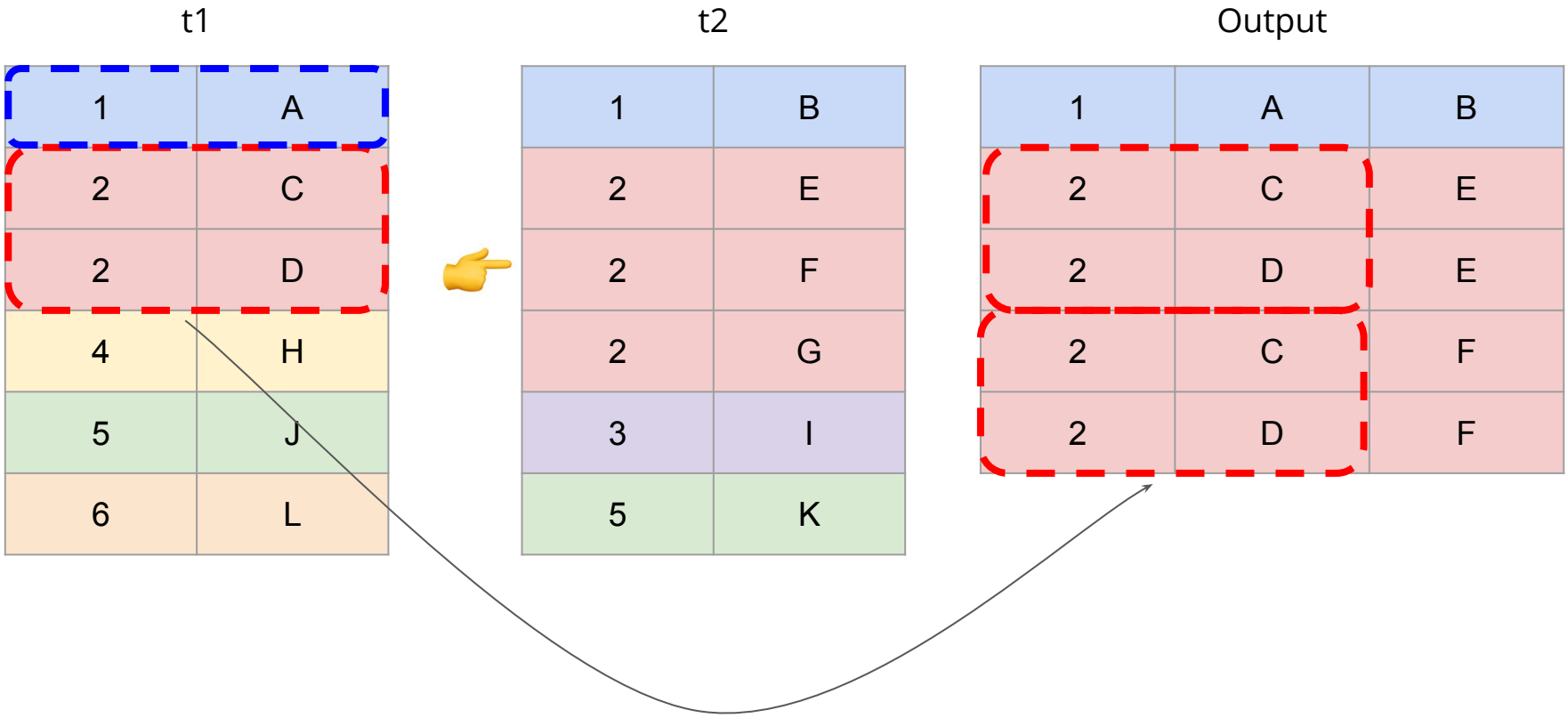
Output

1	A	B

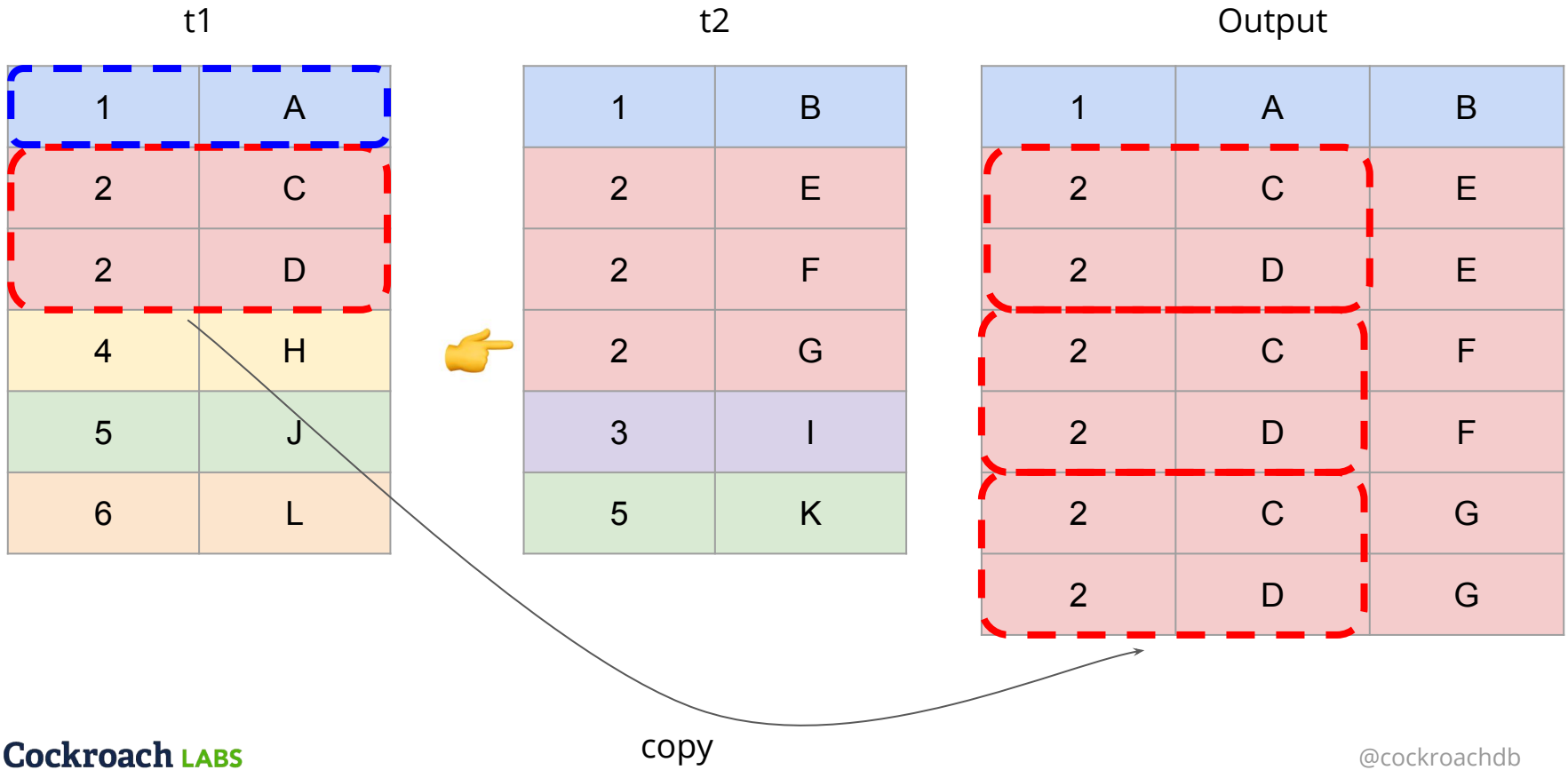
SELECT t1.k, t1.v, t2.v FROM t1 INNER MERGE JOIN t2 ON t1.k = t2.k



SELECT t1.k, t1.v, t2.v FROM t1 INNER MERGE JOIN t2 ON t1.k = t2.k



SELECT t1.k, t1.v, t2.v FROM t1 INNER MERGE JOIN t2 ON t1.k = t2.k



SELECT t1.k, t1.v, t2.v FROM t1 INNER MERGE JOIN t2 ON t1.k = t2.k

t1

1	A
2	C
2	D
4	H
5	J
6	L

t2

1	B
2	E
2	F
2	G
3	I
5	K

Output

1	A	B
2	C	E
2	D	E
2	C	F
2	D	F
2	C	G
2	D	G

SELECT t1.k, t1.v, t2.v FROM t1 INNER MERGE JOIN t2 ON t1.k = t2.k

t1

1	A
2	C
2	D
4	H
5	J
6	L

t2

1	B
2	E
2	F
2	G
3	I
5	K

Output

1	A	B
2	C	E
2	D	E
2	C	F
2	D	F
2	C	G
2	D	G

SELECT t1.k, t1.v, t2.v FROM t1 INNER MERGE JOIN t2 ON t1.k = t2.k

t1

1	A
2	C
2	D
4	H
5	J
6	L

t2

1	B
2	E
2	F
2	G
3	I
5	K

Output

1	A	B
2	C	E
2	D	E
2	C	F
2	D	F
2	C	G
2	D	G
5	J	K

SELECT t1.k, t1.v, t2.v FROM t1 INNER MERGE JOIN t2 ON t1.k = t2.k

t1

1	A
2	C
2	D
4	H
5	J
6	L

t2

1	B
2	E
2	F
2	G
3	I
5	K

Output

1	A	B
2	C	E
2	D	E
2	C	F
2	D	F
2	C	G
2	D	G
5	J	K

Before:

```
BenchmarkMergeJoiner/InputSize=4-8          200000          7543 ns/op         8.48 MB/s
BenchmarkMergeJoiner/InputSize=16-8         200000         10644 ns/op        24.05 MB/s
BenchmarkMergeJoiner/InputSize=256-8         20000         81155 ns/op        50.47 MB/s
BenchmarkMergeJoiner/InputSize=4096-8         2000       1177922 ns/op        55.64 MB/s
BenchmarkMergeJoiner/InputSize=65536-8         100      19149506 ns/op        54.76 MB/s
```

After:

```
kMergeJoiner/rows=1024-8          5000          246903 ns/op         265.43 MB/s
kMergeJoiner/rows=4096-8          2000          818305 ns/op         320.35 MB/s
kMergeJoiner/rows=16384-8          500         3081277 ns/op         340.31 MB/s
kMergeJoiner/rows=1048576-8         10       191384634 ns/op         350.65 MB/s
```

Before:

```
BenchmarkMergeJoiner/InputSize=4-8          200000          7543 ns/op         8.48 MB/s
BenchmarkMergeJoiner/InputSize=16-8         200000         10644 ns/op        24.05 MB/s
BenchmarkMergeJoin                                                            50.47 MB/s
BenchmarkMergeJoin                                                            55.64 MB/s
BenchmarkMergeJoin                                                            54.76 MB/s
```

After:

```
kMergeJoiner/rows=                                                           265.43 MB/s
kMergeJoiner/rows=                                                           320.35 MB/s
kMergeJoiner/rows=                                                           340.31 MB/s
kMergeJoiner/rows=                                                           350.65 MB/s
```

Agenda

1. What is a merge join algorithm
2. Outline of the implementation in the volcano model
3. Vectorizing the merge joiner
    a. single column equality
    b. multi column equality
    c. output population
    d. more optimizations

But what about multi column equality?

But what about multi column equality?

Remember, we can only look at one column per input at a time.

Multi Column Equality: The input

1	2	
1	4	
1	4	
2	3	
2	4	
3	5	
3	10	
3	11	
4	6	
6	7	

1	2	
1	3	
2	3	
2	3	
2	9	
3	6	
3	7	
5	6	
5	7	
6	7	

Multi Column Equality: The input

1	2	
1	4	
1	4	
2	3	
2	4	
3	5	
3	10	
3	11	
4	6	
6	7	

1	2	
1	3	
2	3	
2	3	
2	9	
3	6	
3	7	
5	6	
5	7	
6	7	

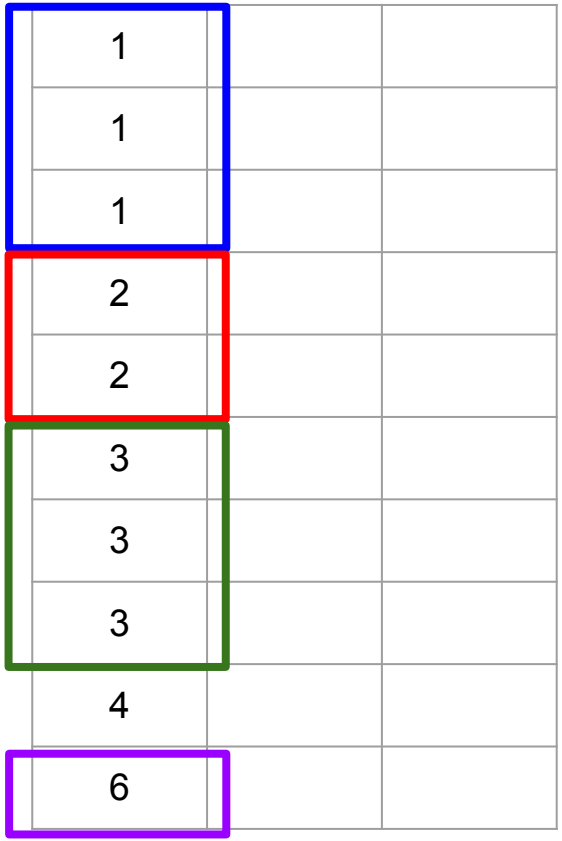
Multi Column Equality: First Column

1		
1		
1		
2		
2		
3		
3		
3		
4		
6		

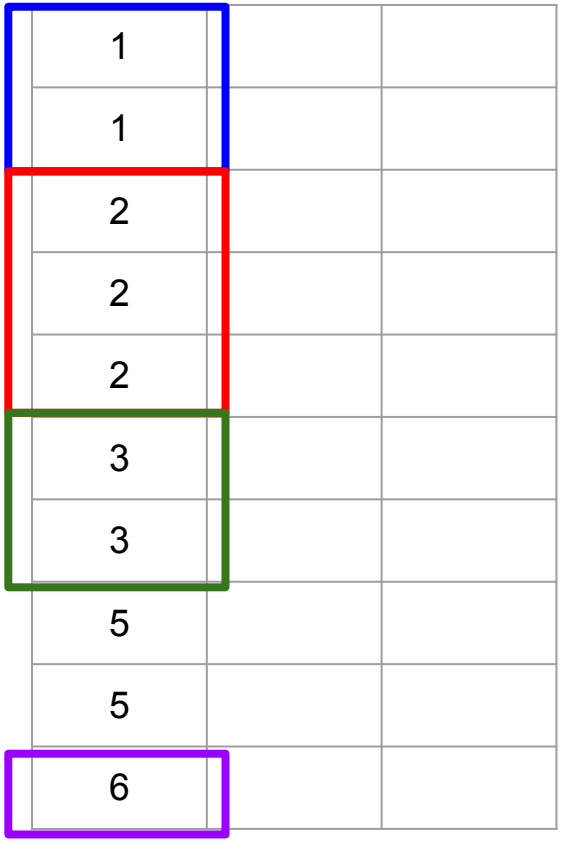
Start with no groups...

1		
1		
2		
2		
2		
3		
3		
5		
5		
6		

Multi Column Equality: First Column



Add groups as we go along



Multi Column Equality: Second Column

	2	
	4	
	4	
	3	
	4	
	5	
	10	
	11	
	6	
	7	

Start with no groups...

	2	
	3	
	3	
	3	
	9	
	6	
	7	
	6	
	7	
	7	

Multi Column Equality: Second Column

	2	
	4	
	4	
	3	
	4	
	5	
	10	
	11	
	6	
	7	

	2	
	3	
	3	
	3	
	9	
	6	
	7	
	6	
	7	
	7	

Vectorizing Multi Column Equality

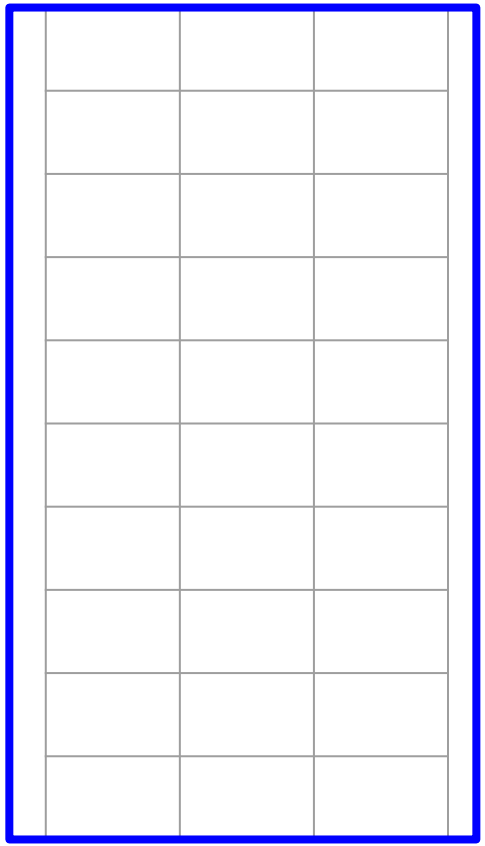
0. The input

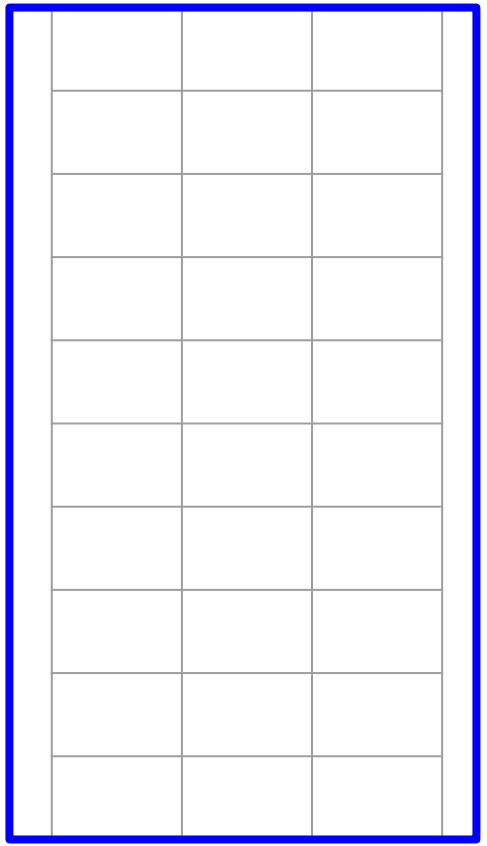
1	2	
1	4	
1	4	
2	3	
2	4	
3	5	
3	10	
3	11	
4	6	
6	7	

1	2	
1	3	
2	3	
2	5	
2	9	
3	6	
3	7	
5	6	
5	7	
6	7	

Vectorizing Multi Column Equality

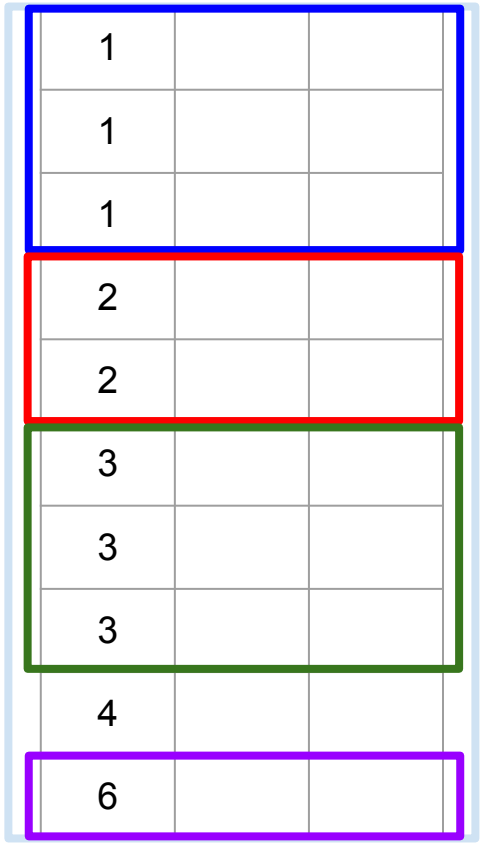
1. Assume the maximal cross product before we begin by creating one group pair

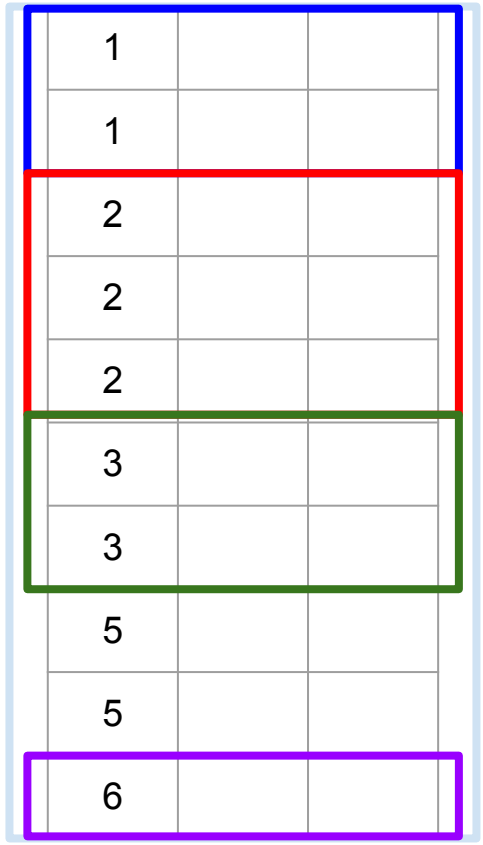




Vectorizing Multi Column Equality

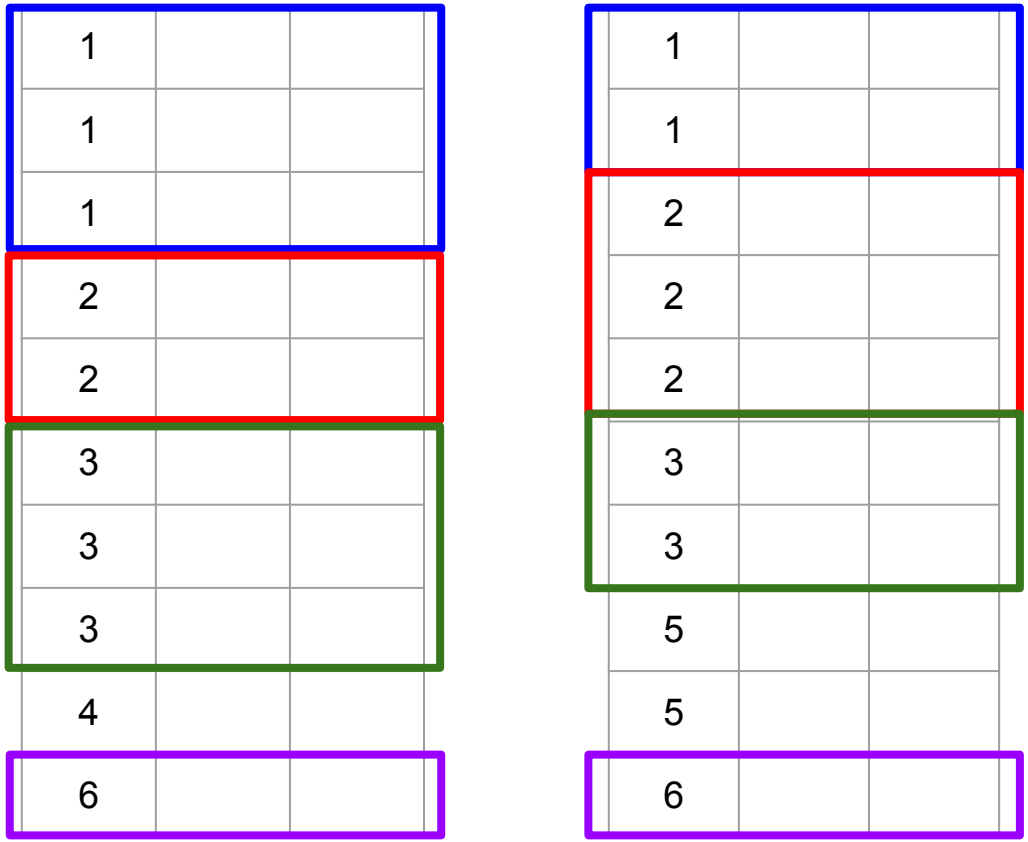
2. Break up the original group pair into new groups based on the first equality column





Vectorizing Multi Column Equality

2. Break up the original group pair into new groups based on the first equality column



Vectorizing Multi Column Equality

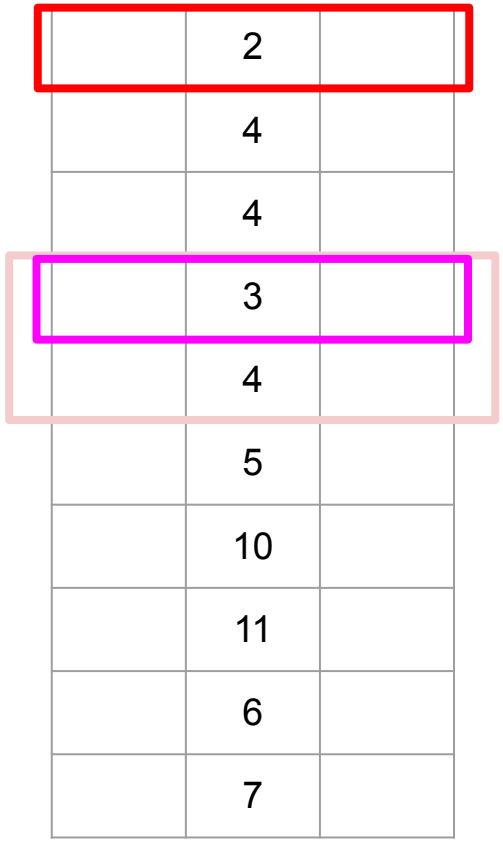
3. Break up each of the groups from the first column based on the second equality column

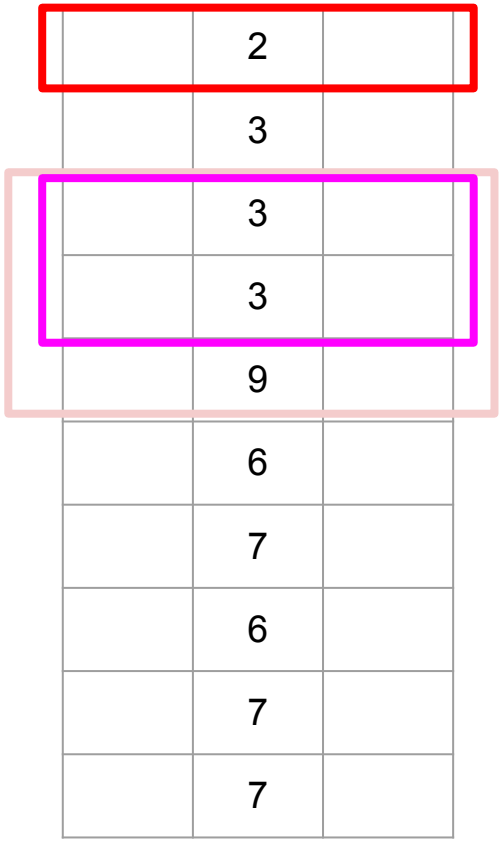
	2	
	4	
	4	
	3	
	4	
	5	
	10	
	11	
	6	
	7	

	2	
	3	
	3	
	3	
	9	
	6	
	7	
	6	
	7	
	7	

Vectorizing Multi Column Equality

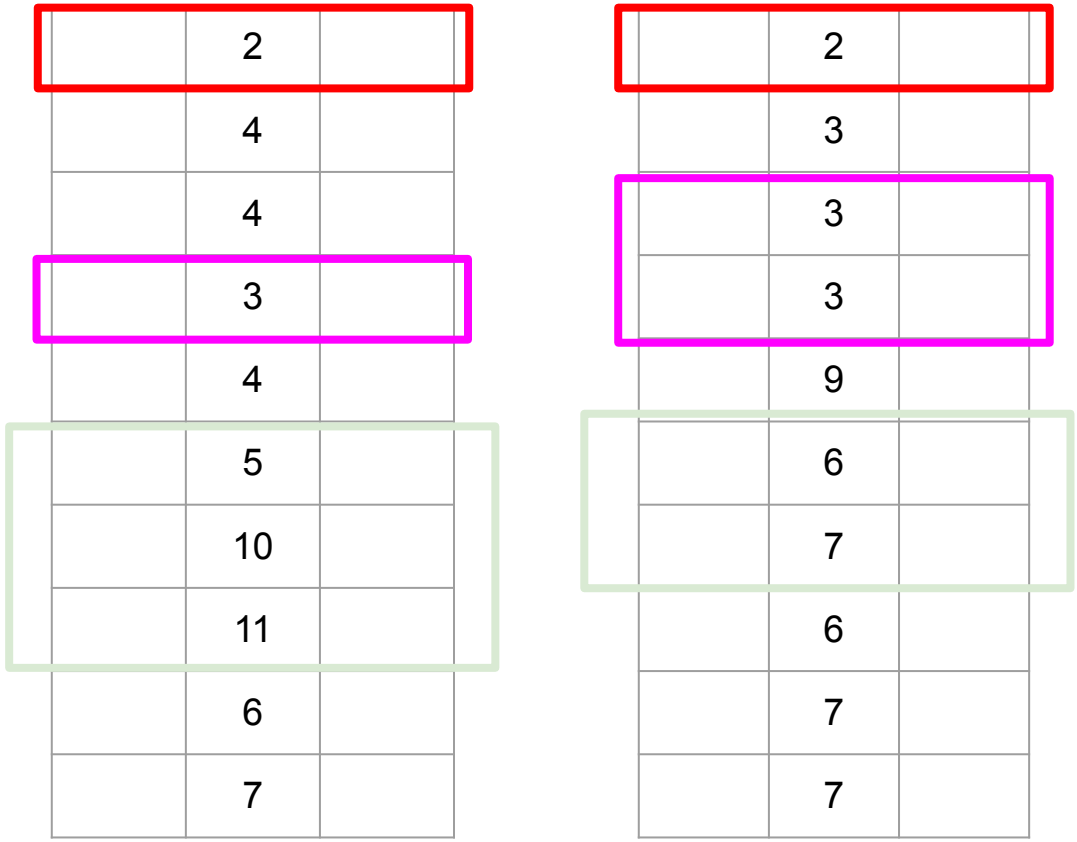
3. Break up each of the groups from the first column based on the second equality column





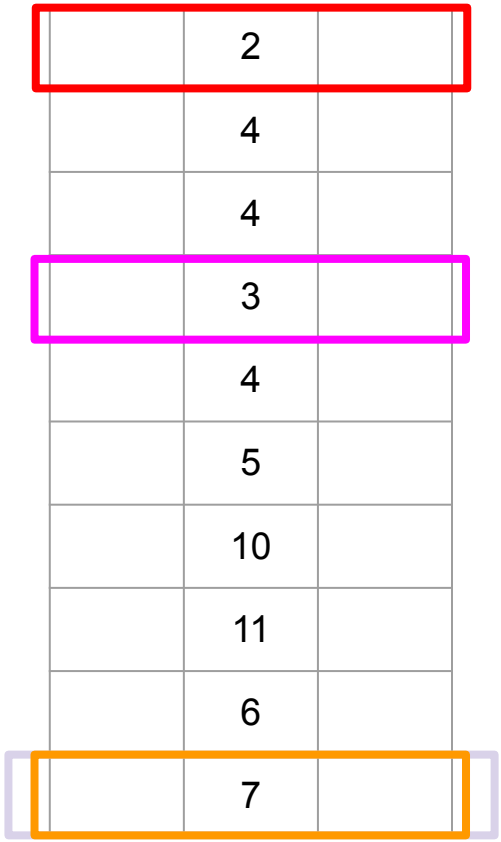
Vectorizing Multi Column Equality

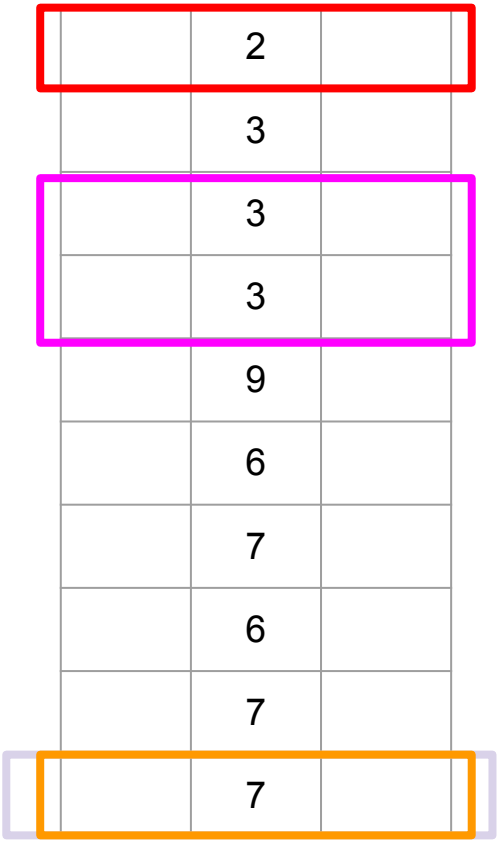
3. Break up each of the groups from the first column based on the second equality column



Vectorizing Multi Column Equality

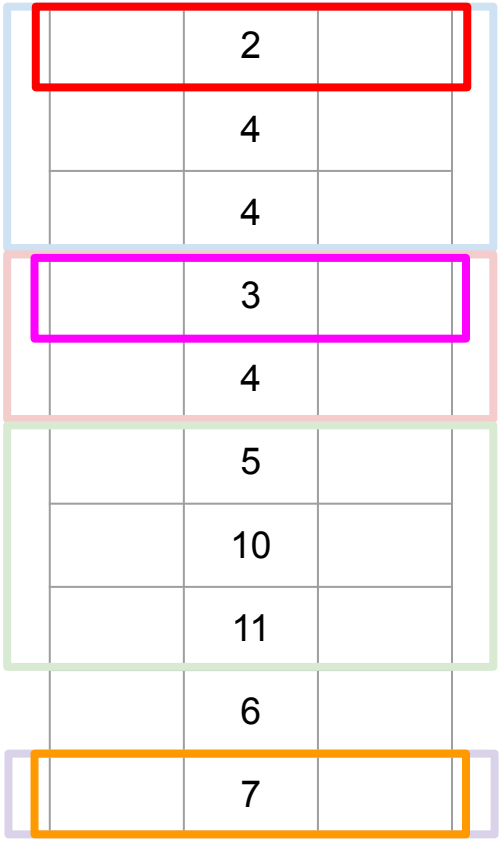
3. Break up each of the groups from the first column based on the second equality column

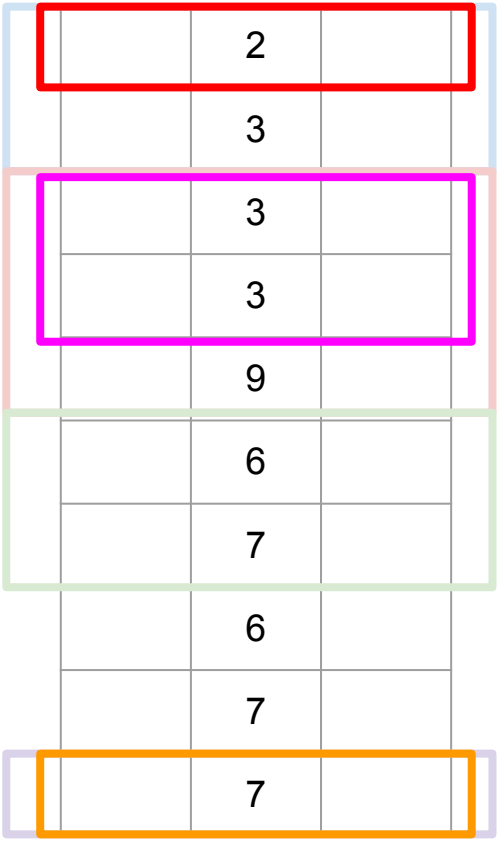




Vectorizing Multi Column Equality

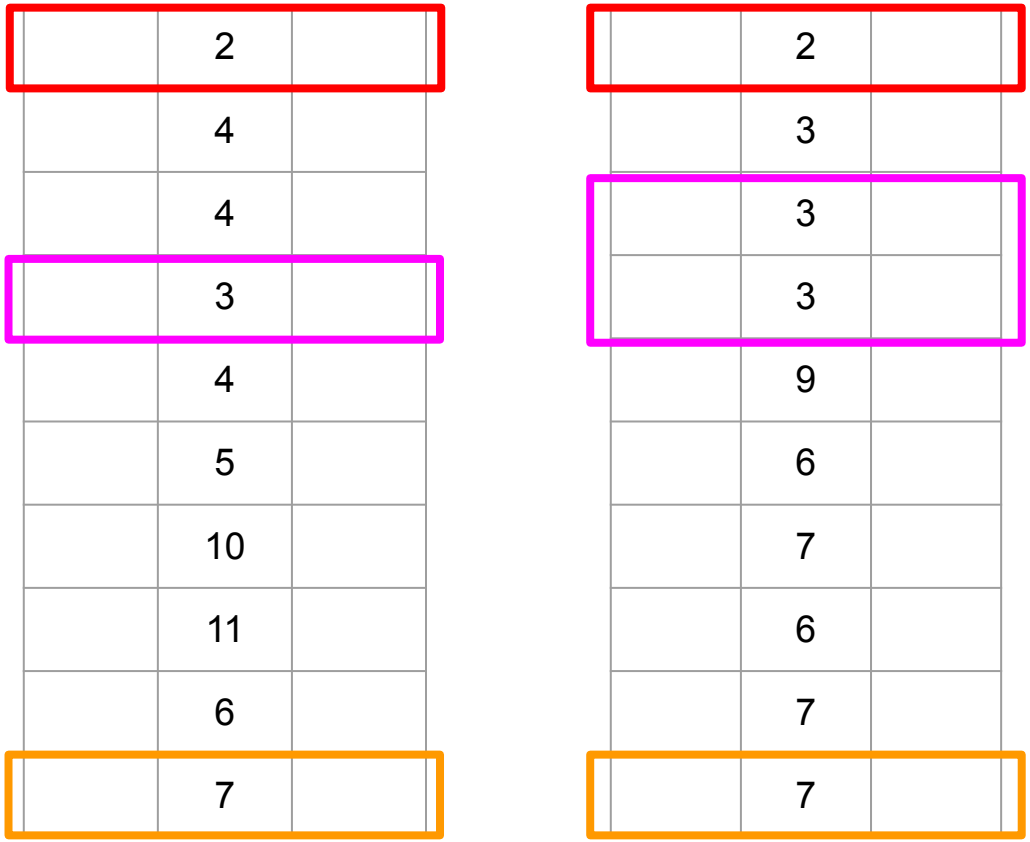
3. Break up each of the groups from the first column based on the second equality column





Vectorizing Multi Column Equality

3. Break up each of the groups from the first column based on the second equality column



Vectorizing Multi Column Equality

4. And so on until you have used all the equality columns

1	2	
	4	
	4	
2	3	
	4	
	5	
	10	
	11	
	6	
6	7	

1	2	
	3	
2	3	
2	3	
	9	
	6	
	7	
	6	
	7	
6	7	

Agenda

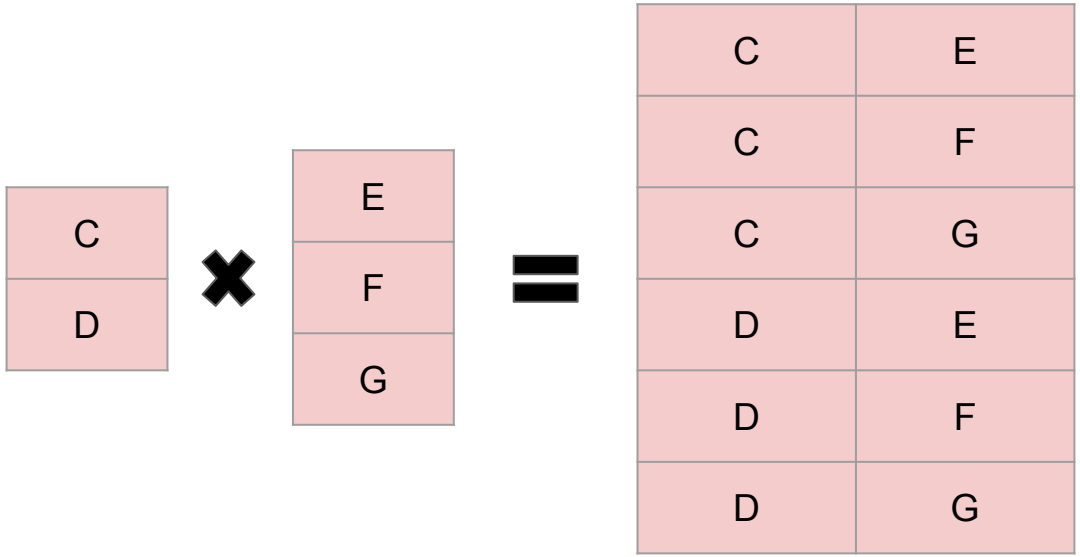
1. What is a merge join algorithm
2. Outline of the implementation in the volcano model
3. Vectorizing the merge joiner
    a. single column equality
    b. multi column equality
    c. output population
    d. more optimizations

Vectorizing the cross product

We can do better with a critical observation

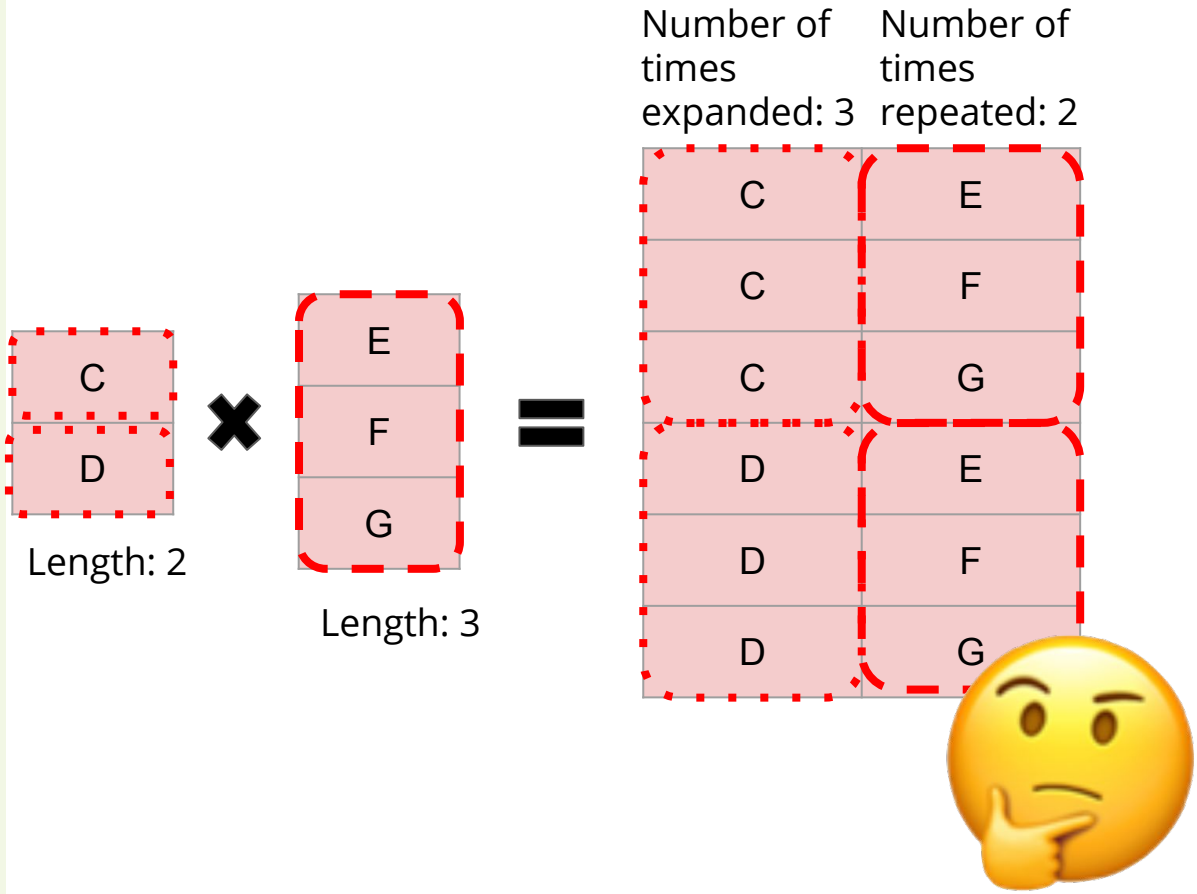
Vectorizing the cross product

This is the cross product we wish to build



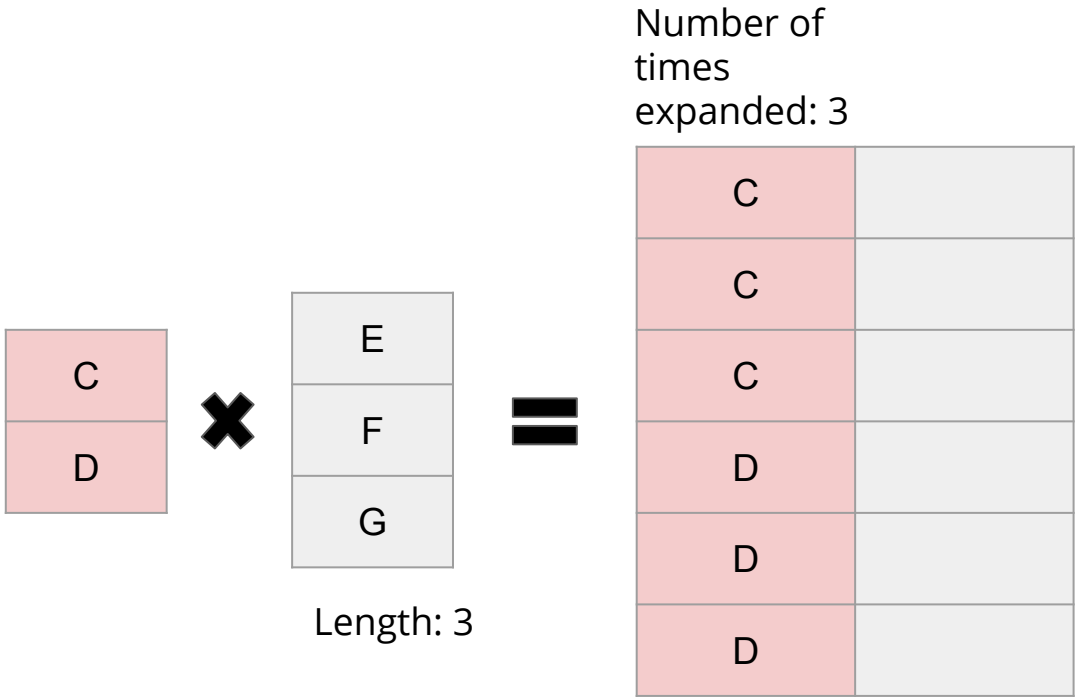
Vectorizing the cross product

Notice how our input columns have the cross product cardinality information encoded in the other's length.



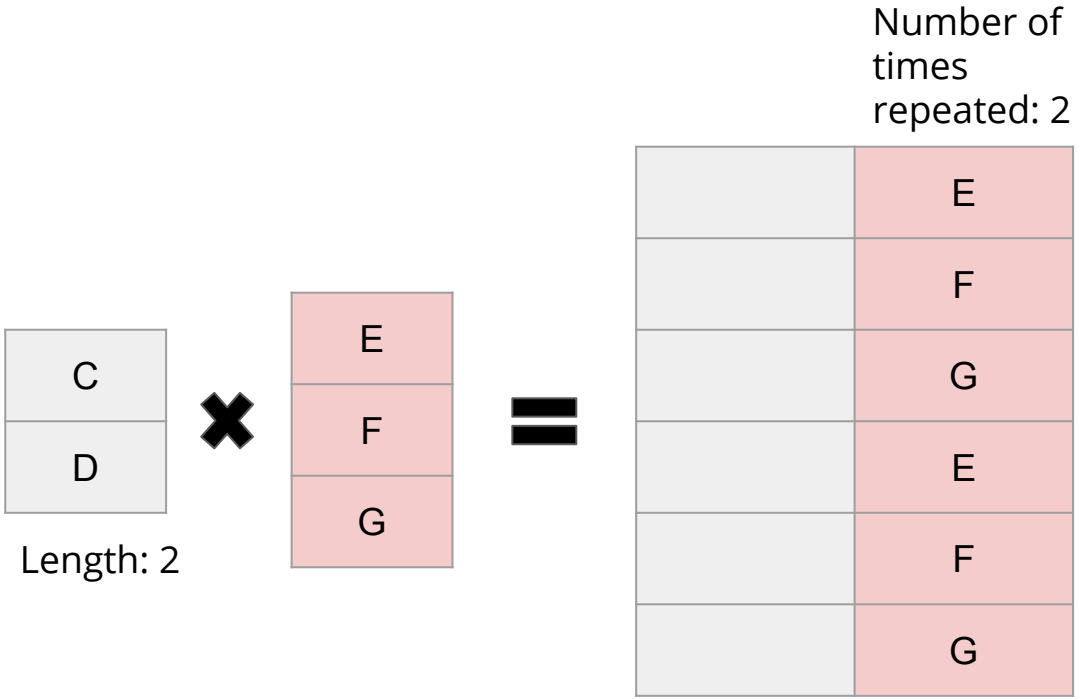
Vectorizing the cross product

Thus we can “expand” the left output all at once, and ignore the right output (for the time being)



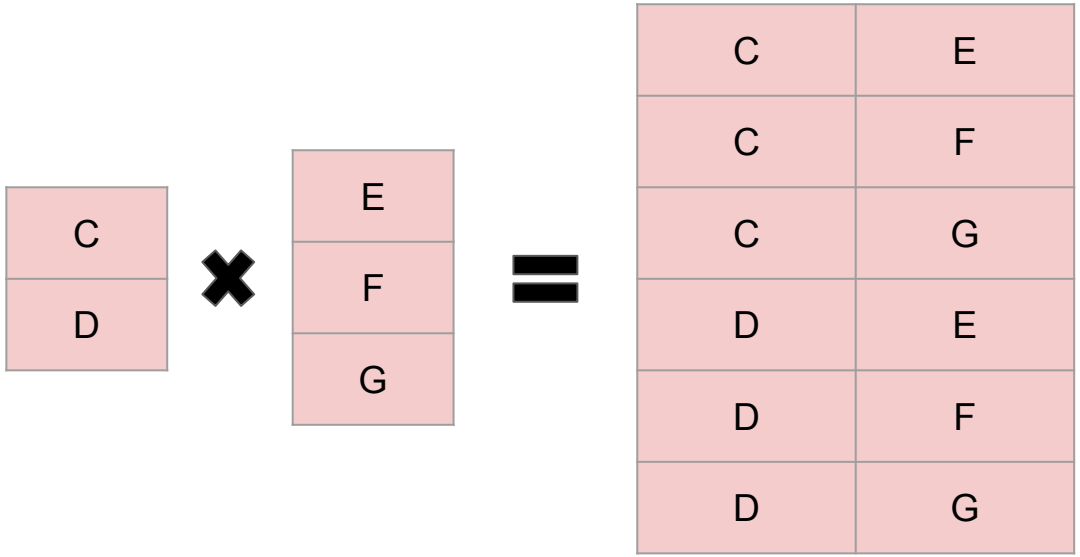
Vectorizing the cross product

Similarly with the right output.



Vectorizing the cross product

Thus we have built our cross product a column at a time!



Before:

```
kMergeJoiner/rows=1024-8          5000          246903 ns/op          265.43 MB/s
kMergeJoiner/rows=4096-8          2000          818305 ns/op          320.35 MB/s
kMergeJoiner/rows=16384-8          500         3081277 ns/op          340.31 MB/s
kMergeJoiner/rows=1048576-8         10       191384634 ns/op          350.65 MB/s
```

After:

```
/rows=1024-8                     20000           66913 ns/op          979.42 MB/s
/rows=4096-8                      5000          265602 ns/op          986.98 MB/s
/rows=16384-8                     2000         1042486 ns/op         1005.84 MB/s
/rows=1048576-8                     20        65653445 ns/op         1022.17 MB/s
```

Before:

```
kMergeJoiner/rows=1024-8          5000          246903 ns/op          265.43 MB/s
kMergeJoiner/rows=4096-8          2000          818305 ns/op          320.35 MB/s
kMergeJoiner/rows=16384-8          500         3081277 ns/op          340.31 MB/s
kMergeJoiner/rows=1048576-8                              ns/op          350.65 MB/s
```

After:

```
/rows=1024-8                                             ns/op          979.42 MB/s
/rows=4096-8                                             ns/op          986.98 MB/s
/rows=16384-8                                            ns/op         1005.84 MB/s
/rows=1048576-8                                          ns/op         1022.17 MB/s
```

Agenda

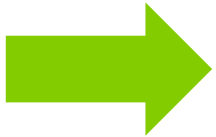
1. What is a merge join algorithm
2. Outline of the implementation in the volcano model
3. Vectorizing the merge joiner
    a. single column equality
    b. multi column equality
    c. output population
    d. more optimizations

Making things faster with:

Assignment instead of `copy`

Typically, it's faster to use `copy` than to loop through and assign a range of values

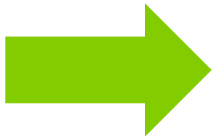
```
for i := range src {
    dest[i] = src[i]
}
```



```
copy(dest, src)
```

But if it's one element, it's faster to perform a length check and assign instead of `copy`

```
copy(dest, src)
```

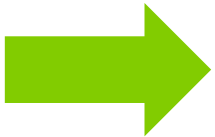


```
if len(src) == 1 {
    dest[0] = src[0]
} else {
    copy(dest, src)
}
```

Making things faster with:

Bounds Check Elimination

```
for i := range src {
    dest[i] = src[i]*2
}
```



```
dest = dest[:len(src)]
for i := range src {
    dest[i] = src[i]*2
}
```

https://www.ardanlabs.com/blog/2018/04/bounds-check-elimination-in-go.html

Making things faster with:

Templating

Making things faster with:

Templating

```
for i := range src {
    if len(src) > 5 {
        dest[i] = src[i]
    } else {
        dest[i] = src[foo[i]]
    }
}
```

```
if len(src) > 5 {
    for i := range src {
        dest[i] = src[i]
    }
} else {
    for i := range src {
        dest[i] = src[foo[i]]
    }
}
```

Making things faster with:

Templating

```
for i := range src {
    if len(src) > 5 {
        dest[i] = src[i]
    } else {
        dest[i] = src[foo[i]]
    }
}
```

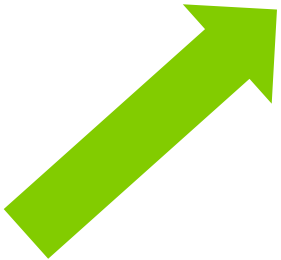
```
if len(src) > 5 {
    for i := range src {
        dest[i] = src[i]
    }
} else {
    for i := range src {
        dest[i] = src[foo[i]]
    }
}
```

Repeated code!

Making things faster with:

Templating

```
for i := range src {
    if len(src) > 5 {
        dest[i] = src[i]
    } else {
        dest[i] = src[foo[i]]
    }
}
```



```
if len(src) > 5 {
    _FOR_LOOP(false)
} else {
    _FOR_LOOP(true)
}
```

```
if len(src) > 5 {
    for i := range src {
        dest[i] = src[i]
    }
} else {
    for i := range src {
        dest[i] = src[foo[i]]
    }
}
```

```
name                                  old speed      new speed      delta
MergeJoiner/rows=1024-8               1.46GB/s ± 1%  1.86GB/s ± 0%  +27.28%
MergeJoiner/rows=4096-8               1.53GB/s ± 0%  1.97GB/s ± 0%  +28.46%
MergeJoiner/rows=16384-8              1.60GB/s ± 1%  2.06GB/s ± 1%  +29.20%
MergeJoiner/rows=1048576-8            1.67GB/s ± 1%  2.18GB/s ± 1%  +30.60%
```

slaps roof of merge joiner

name	new speed	delta
MergeJoiner/rows=1024-8	1.86GB/s ± 0%	+27.28%
MergeJoiner/rows=4096-8	1.97GB/s ± 0%	+28.46%
MergeJoiner/rows=16384-	2.06GB/s ± 1%	+29.20%
MergeJoiner/rows=104857	2.18GB/s ± 1%	+30.60%

Templating, why

- ❏ different data type support
- ❏ different join type support
- ❏ null handling

```
pkg/sql/colexec/mergejoiner_fullouter.eg.go 48220
pkg/sql/colexec/mergejoiner_exceptall.eg.go 42074
pkg/sql/colexec/mergejoiner_leftouter.eg.go 40407
pkg/sql/colexec/mergejoiner_leftanti.eg.go 40268
pkg/sql/colexec/mergejoiner_rightouter.eg.go 40125
pkg/sql/colexec/mergejoiner_intersectall.eg.go 33455
pkg/sql/colexec/mergejoiner_inner.eg.go 32307
pkg/sql/colexec/mergejoiner_leftsemi.eg.go 32142
pkg/sql/colexec/mergejoiner_test.go 2233
pkg/sql/colexec/mergejoiner_tmpl.go 1678
pkg/sql/colexec/mergejoiner.go 783
pkg/sql/colexec/mergejoiner_util.go 219
pkg/sql/colexec/execgen/cmd/execgen/mergejoiner_gen.go 201
```

Conclusion

- vectorized operators can offer orders of magnitude improvements in the performance
- vectorized execution in CockroachDB on real queries achieves up to 4x speed up (limited by the scan speed)
- “vectorizing” an operator requires change of the perspective

Check out our excellent blog!

- How We Built a Vectorized Execution Engine by Alfonso Subiotto Marques and Rafi Shamim
- Vectorizing the Merge Joiner by George Utsin

Questions?

cockroa.ch/earlycareers

github.com/cockroachdb

www.cockroachlabs.com