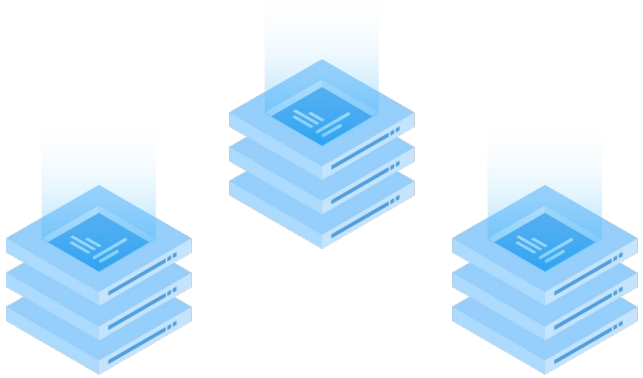
CockroachDB Vectorized Engine

Presented by Yuanjia Zhang

PingCAP TiDB



PingCAP

- Introduction to Vectorized Execution
- CockroachDB VectorWise Overview
- Vectorized Join Algorithm in CockroachDB

PingCAP

Part I - Intro to Vectorized Execution

What's vectorized execution

Consider the Example:

A function that selects every row whose color is green and has four tires.

```
bool sel_eq_row(r row) {
    return r[0].String() == "green" && r[1].Int() == 4;
}

vec<row> sel_eq_rows(vec<row> rows) {
    vec<row> res;
    for (r : rows) {
        if sel_eq_row(r) {
            res.append(r);
        }
    }
    return res;
}
```

```
vec<int> sel_eq_string(vec<string> col) {
    vec<int> res;
    for i := 0; i < len(col); i++ {
        if col[i] == "green" {
            res.append(i)
        }
    }
    return res;
}

vec<int> sel_eq_int(vec<int> col, vec<int> sel) {
    vec<int> res;
    for (i : sel) {
        if col[i] == 4 {
            res.append(i)
        }
    }
    return res;
}
```

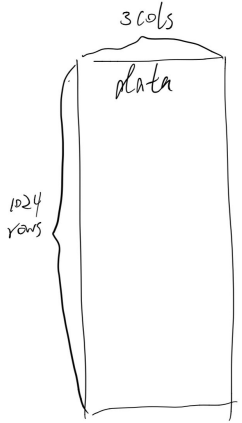
Two constraints, it can:

1. only work on one data type,
2. must process multiple tuples.

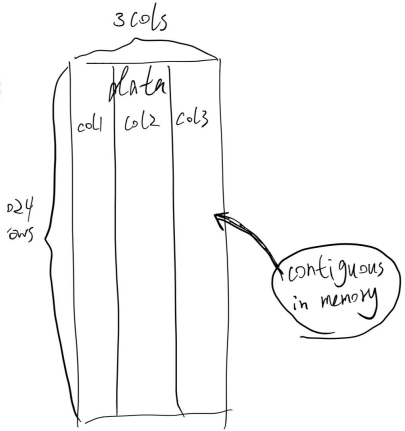
Why is it fast

Advantages compared with traditional Vocalno-Model:

- lower interpretation overhead,
- higher cache hit-ratio.




```
bool sel_eq_row(r row) {
    return r[0].String() == "green" && r[1].Int() == 4;
}

vec<row> sel_eq_rows(vec<row> rows) {
    vec<row> res;
    for (r : rows) {
        if sel_eq_row(r) {
            res.append(r);
        }
    }
    return res;
}
```



Why is it fast

The instruction throughput of a CPU depending on:

- the amount of independent instructions the CPU can detect,
- the number of branches can be predicated,
- the cache hit-ratio of the memory loads and stores.

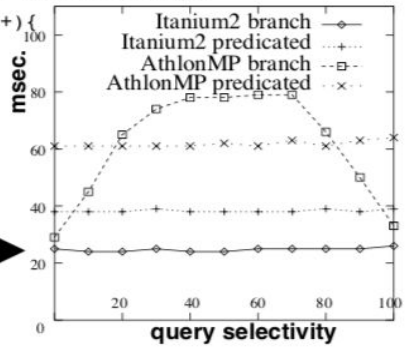
out-of-order execution + instruction pipelining

```
F(A[0]),G(A[0]), F(A[1]),G(A[1]),..  F(A[n]),G(A[n])
into:
F(A[0]),F(A[1]),F(A[2]), G(A[0]),G(A[1]),G(A[2]), F(A[3]),..
```

Instr. No. \ Clock cycle	1	2	3	4	5	6	7
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX

branch predication

```
int sel_lt_int_col_int_val(int n, int* res, int* in, int V) {
  for(int i=0,j=0; i<n; i++){
    /* branch version */
    if (src[i] < V)
      out[j++] = i;
    /* predicated version */
    bool b = (src[i] < V);
    out[j] = i;
    j += b;
  }
  return j;
}
```

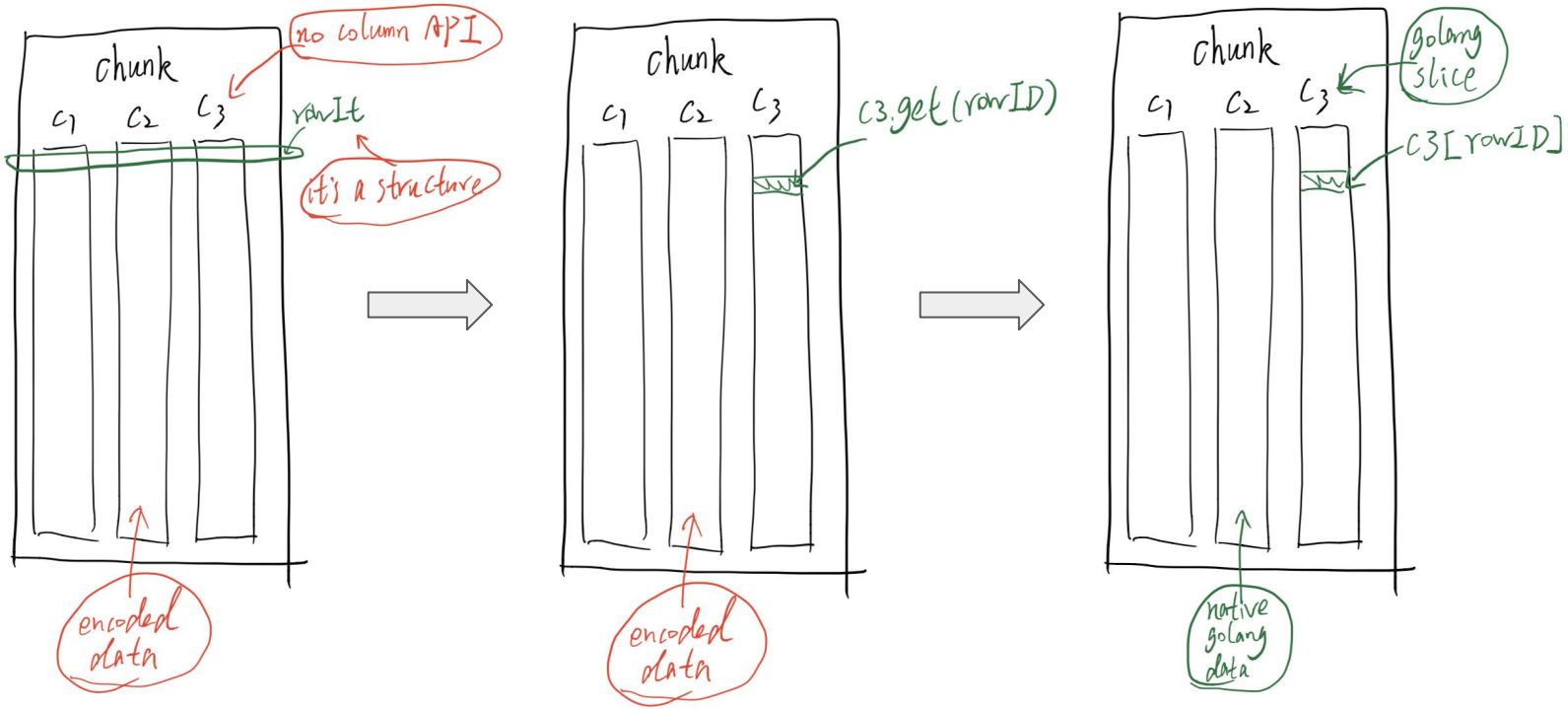


References:

- MonetDB/X100: Hyper-Pipelining Query Execution
- Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask

Examples 1

Chunk iteration in TiDB



Examples 1

```go
func (r Row) GetInt64(colIdx int) int64 {
    col := r.c.columns[colIdx]
    return *(*int64)(unsafe.Pointer(&col.data[r.idx*8]))
}
```

```go
func BenchmarkIterateChunkIterator(b *testing.B) {
    chk := newChunk( elemLen...: 8)
    for i := 0; i < 1024; i++ {
        chk.AppendInt64( colIdx: 0, int64(i))
    }
    iter := NewIterator4Chunk(chk)

    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        sum := int64(0)
        for row := iter.Begin(); row != iter.End(); row = iter.Next() {
            sum += row.GetInt64( colIdx: 0)
        }
    }
}
```

```go
func (c *column) GetInt64(rowID int) int64 {
    return *(*int64)(unsafe.Pointer(&c.data[rowID*8]))
}
```

```go
func BenchmarkIterateColumn(b *testing.B) {
    chk := newChunk( elemLen...: 8)
    for i := 0; i < 1024; i++ {
        chk.AppendInt64( colIdx: 0, int64(i))
    }
    col := chk.columns[0]

    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        sum := int64(0)
        for row := 0; row < 1024; row++ {
            sum += col.GetInt64(row)
        }
    }
}
```

```go
func BenchmarkIterateSlice(b *testing.B) {
    xs := make([]int64, 0, 1024)
    for i := 0; i < 1024; i++ {
        xs = append(xs, int64(i))
    }

    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        sum := int64(0)
        for row := 0; row < 1024; row ++ {
            sum += xs[row]
        }
    }
}
```

```
pkg: github.com/pingcap/tidb/util/chunk
BenchmarkIterateChunkIterator-12          1000000              1391 ns/op
BenchmarkIterateColumn-12                 5000000               389 ns/op
BenchmarkIterateSlice-12                  5000000               286 ns/op
PASS
ok      github.com/pingcap/tidb/util/chunk      5.480s
```

Examples 2

Loop Optimization
in CockroachDB

```
/*
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]]
    }
}
*/
```

```
func loopWithCheck(x int) int {
    sum := 0
    for i := 0; i < 1024; i++ {
        if x > 512 {
            sum += 2
        } else {
            sum += 1
        }
    }
    return sum
}

func BenchmarkLoopWithCheck(b *testing.B) {
    for i := 0; i < b.N; i++ {
        loopWithCheck( x: 511)
        loopWithCheck( x: 513)
    }
}
```

```
func loopWithoutCheck(x int) int {
    sum := 0
    if x > 512 {
        for i := 0; i < 1024; i++ {
            sum += 2
        }
    } else {
        for i := 0; i < 1024; i++ {
            sum += 1
        }
    }
    return sum
}

func BenchmarkLoopWithoutCheck(b *testing.B) {
    for i := 0; i < b.N; i++ {
        loopWithoutCheck( x: 511)
        loopWithoutCheck( x: 513)
    }
}
```

```
pkg: lab/vec_lab/loop_with_if
BenchmarkLoopWithCheck-12                1000000              1131 ns/op
BenchmarkLoopWithoutCheck-12             3000000               499 ns/op
PASS
ok      lab/vec_lab/loop_with_if        3.155s
```

PingCAP

Part II - CockroachDB VectorWise Overview

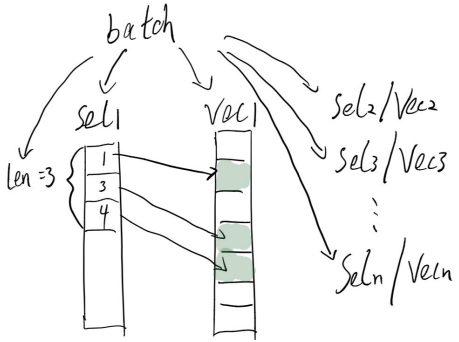
Batch, Vec and Sel

Vectorized structures in CockroachDB

```
type Vec interface {
	Type() types.T

	Bool() []bool
	Int64() []int64
	Bytes() [][]byte
	Decimal() []apd.Decimal

	Append(AppendArgs)
	Copy(CopyArgs)
	Slice(colType types.T,
		start uint64, end uint64) Vec

	MaybeHasNulls() bool
	Nulls() *Nulls
	SetNulls(*Nulls)
	...
}
```

```
func (m *memColumn) NullAt64(i uint64) bool {
	intIdx := i >> 6
	return ((m.nulls[intIdx] >> (i % 64)) & 1) == 1
}

func (m *memColumn) Bool() []bool {
	return m.col.([]bool)
}

func (m *memColumn) Int8() []int8 {
	return m.col.([]int8)
}

func (m *memColumn) Int16() []int16 {
	return m.col.([]int16)
}

func (m *memColumn) Int32() []int32 {
	return m.col.([]int32)
}

func (m *memColumn) Copy(args CopyArgs) {
	if args.Nils != nil && args.Sel64 == nil {
		panic( v: "Nils set without Sel64")
	}
```

```
type Batch interface {
	Width() int
	AppendCol(types.T)

	ColVec(i int) Vec
	ColVecs() []Vec

	Length() uint16
	Selection() []uint16

	...
}
```



Computation on Vec

```go
func (m *memColumn) CopyWithSelInt16(vec Vec,
    sel []uint16, nSel uint16, colType types.T) {
    m.UnsetNulls()

    switch colType {
    case types.Bool:
        toCol := m.Bool()
        fromCol := vec.Bool()

        if vec.HasNulls() {
            for i := uint16(0); i < nSel; i++ {
                if vec.NullAt(sel[i]) {
                    m.SetNull(i)
                } else {
                    toCol[i] = fromCol[sel[i]]
                }
            }
        } else {
            for i := uint16(0); i < nSel; i++ {
                toCol[i] = fromCol[sel[i]]
            }
        }
    case types.Bytes:...
    }
}
```

loop optimization

```go
func (p *selLTInt8Int8Op) Next() coldata.Batch {
    for {
        batch := p.input.Next()
        if batch.Length() == 0 {
            return batch
        }

        col1 := batch.ColVec(p.col1Idx).Int8()[:coldata.BatchSize]
        col2 := batch.ColVec(p.col2Idx).Int8()[:coldata.BatchSize]
        n := batch.Length()

        var idx uint16
        if sel := batch.Selection(); sel != nil {
            sel = sel[:n]
            for _, i := range sel {
                var cmp bool
                cmp = col1[i] < col2[i]
                if cmp {
                    sel[idx] = i
                    idx++
                }
            }
        } else {
            batch.SetSelection(true)
            sel := batch.Selection()
            for i := uint16(0); i < n; i++ {
                var cmp bool
                cmp = col1[i] < col2[i]
                if cmp {
                    sel[idx] = i
                    idx++
                }
            }
        }
        if idx > 0 {
            batch.SetLength(idx)
            return batch
        }
    }
}
```

update Sel

update Len

Templating

Do not repeat yourself:

- optimization
- multiple types

```
{{ define forLoop }}
for i := range src {

        dest[i] = src[_SRC_INDEX]

}

{{ end }}

if len(src) > 5 {

        _FOR_LOOP(false)

} else {

        _FOR_LOOP(true)

}
```

```
func GetSelectionOperator(
    ct sqlbase.ColumnType,
    cmpOp tree.ComparisonOperator,
    input Operator,
    col1Idx int,
    col2Idx int,
) (Operator, error) {
    switch t := conv.FromColumnType(ct); t {
    {{range $typ, $overloads := .}}
    case types.{{$typ}}:
        switch cmpOp {
        {{range $overloads}}
        case tree.{{.Name}}:
            return &{{template "opName" .}}{
                input:    input,
                col1Idx: col1Idx,
                col2Idx: col2Idx,
            }, nil
        {{end}}
        default:
            return nil, errors.Errorf("unhandled comparison operator: %s", cmpOp)
        }
    {{end}}
    default:
        return nil, errors.Errorf("unhandled type: %s", t)
    }
}
```

```
func GetSelectionOperator(
    ct sqlbase.ColumnType,
    cmpOp tree.ComparisonOperator,
    input Operator,
    col1Idx int,
    col2Idx int,
) (Operator, error) {
    switch t := conv.FromColumnType(ct); t {
    case types.Bool:...
    case types.Bytes:...
    case types.Decimal:...
    case types.Int8:...
    case types.Int16:...
    case types.Int32:...
    case types.Int64:...
    case types.Float32:...
    case types.Float64:
        switch cmpOp {
        case tree.EQ:...
        case tree.NE:...
        case tree.LT:...
        case tree.LE:...
        case tree.GT:...
        case tree.GE:
            return &selGEFloat64Float64Op{
                input:    input,
                col1Idx: col1Idx,
                col2Idx: col2Idx,
            }, nil
        default:...
        }
    default:...
    }
}
```

Drawbacks

- Only a few expressions support vectorized computation.
- The whole vectorized plan will be rejected if there is unsupported expression.
- No parallel execution.

```
func planProjectionOperators(
    ctx *tree.EvalContext, expr tree.TypedExpr, columnTypes []semtypes.T, input exec.Operator,
) (op exec.Operator, resultIdx int, ct []semtypes.T, err error) {
    resultIdx = -1
    switch t := expr.(type) {
    case *tree.IndexedVar:...
    case *tree.ComparisonExpr:...
    case *tree.BinaryExpr:...
    case tree.Datum:...
    default:
        return op: nil, resultIdx, ct: nil, errors.Errorf("unhandled projection expression type: %s", reflect.TypeOf(t))
    }
}
```

For example, this query cannot be converted to a vectorized plan:

SELECT order_id FROM order_table WHERE money < 100 AND MONTH(date) = 10;

PingCAP

Part III - Vectorized Join Algorithm in CockroachDB

Vectorized Merge Join

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

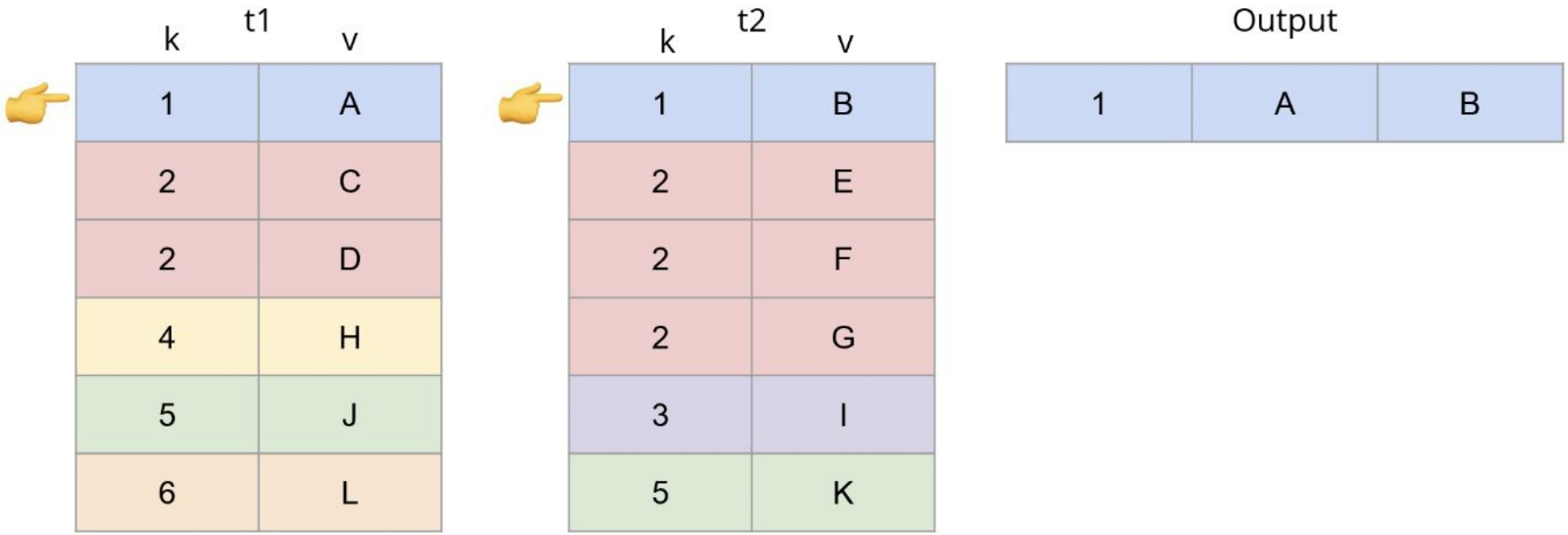
k	v
1	B
2	E
2	F
2	G
3	I
5	K

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

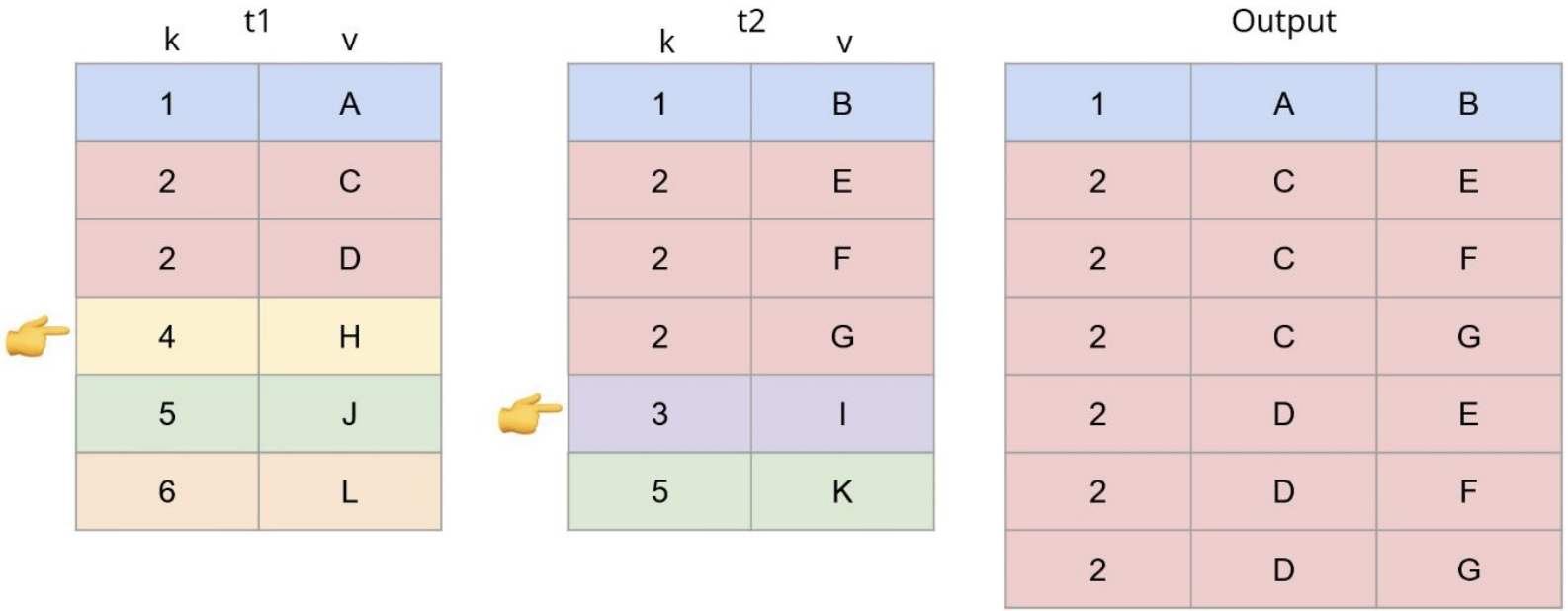
Vectorized Merge Join

The traditional merge join algorithm



Vectorized Merge Join

The traditional merge join algorithm



Vectorized Merge Join

The traditional merge join algorithm

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

Vectorized Merge Join

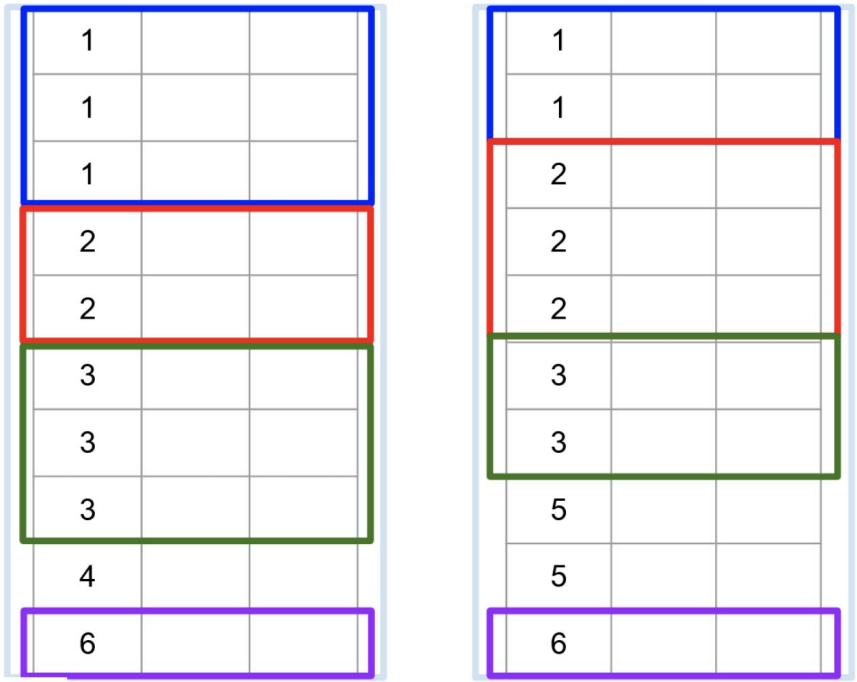
The vectorized probing phase

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	

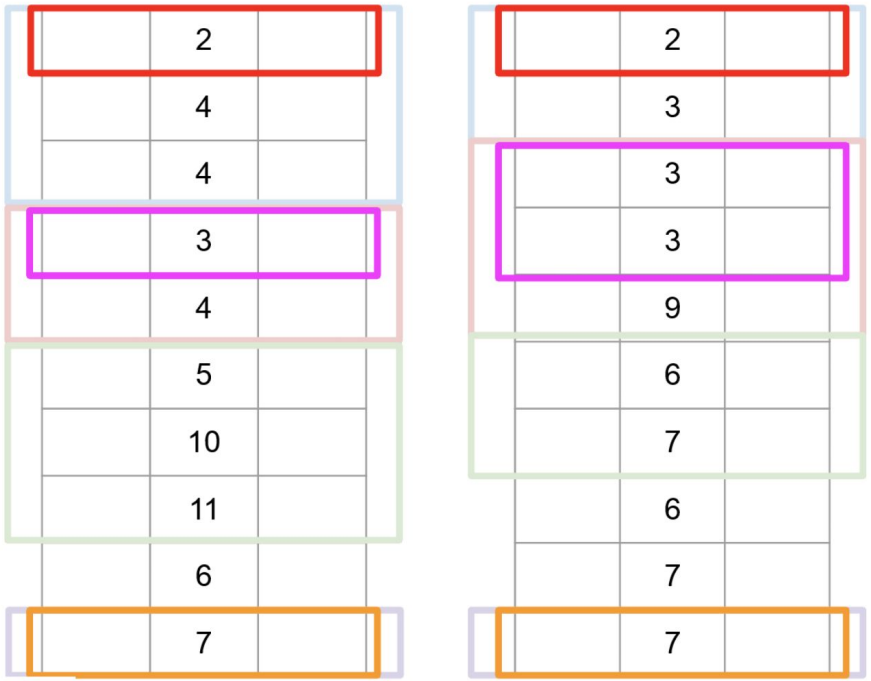
Vectorized Merge Join

The vectorized probing phase



Vectorized Merge Join

The vectorized probing phase



Vectorized Merge Join

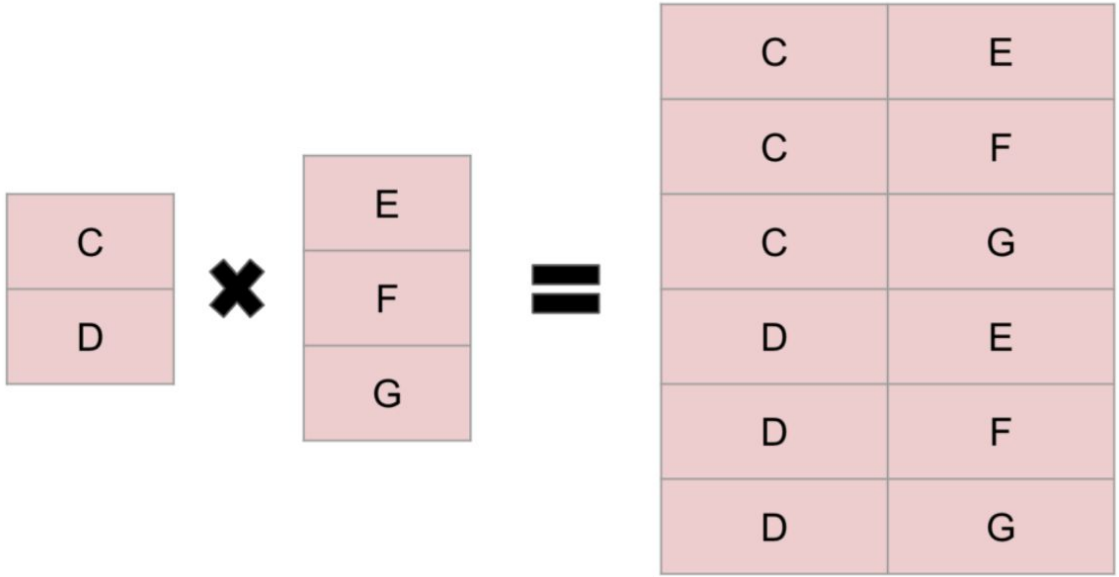
The vectorized probing phase

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	

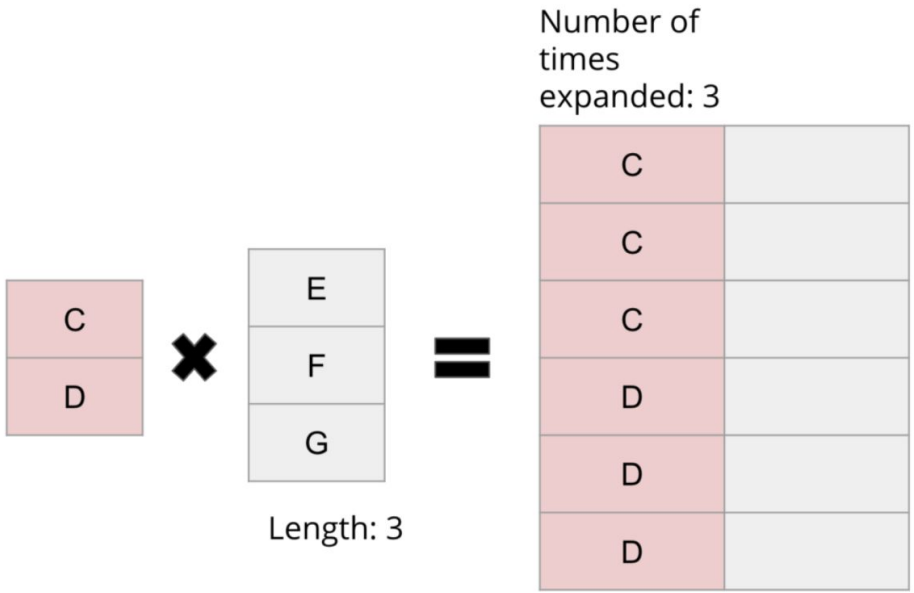
Vectorized Merge Join

The vectorized matrializing phase



Vectorized Merge Join

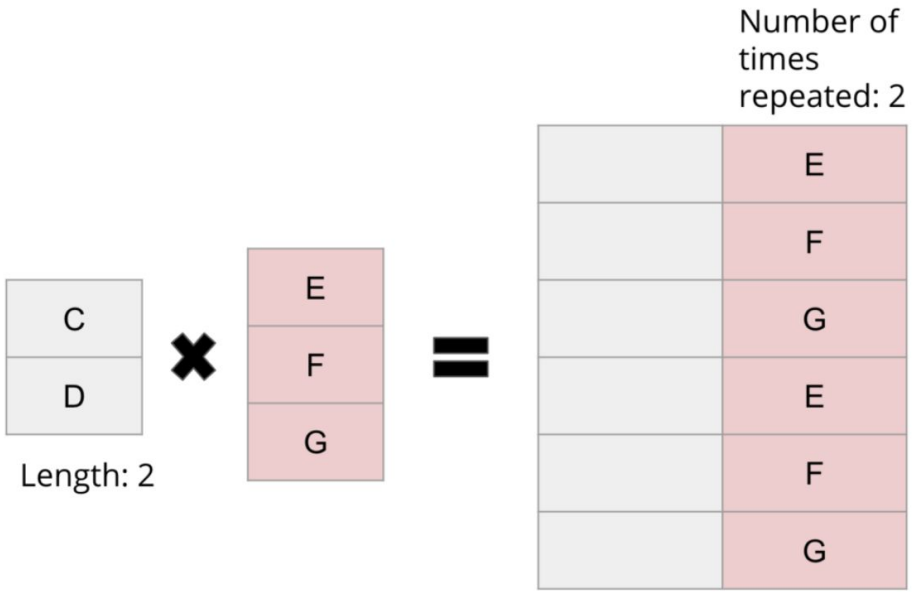
The vectorized matrializing phase



Left cross product of join

Vectorized Merge Join

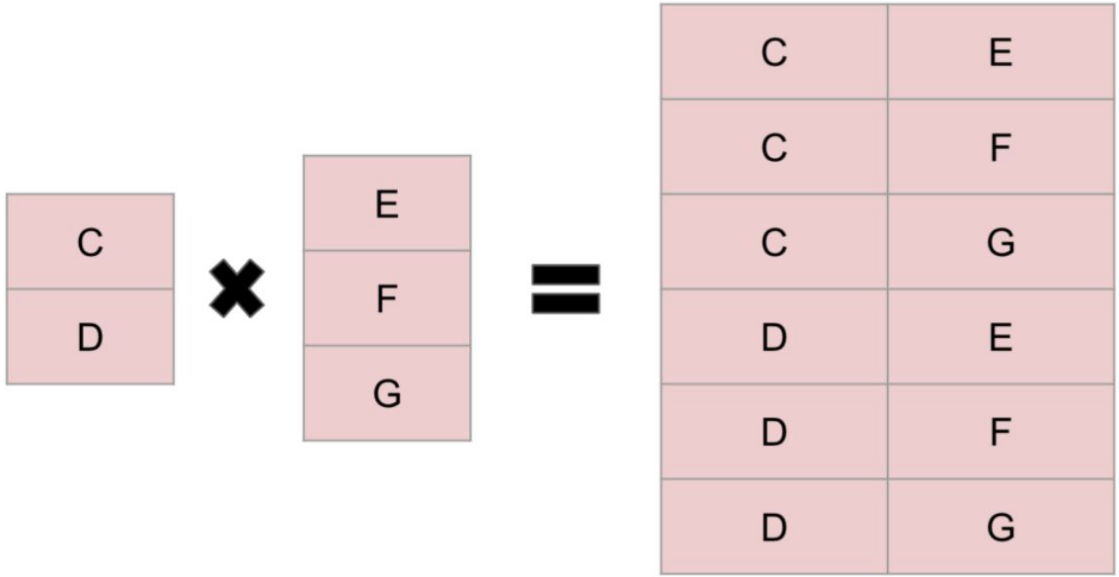
The vectorized matrializing phase



Right cross product of join

Vectorized Merge Join

The final result



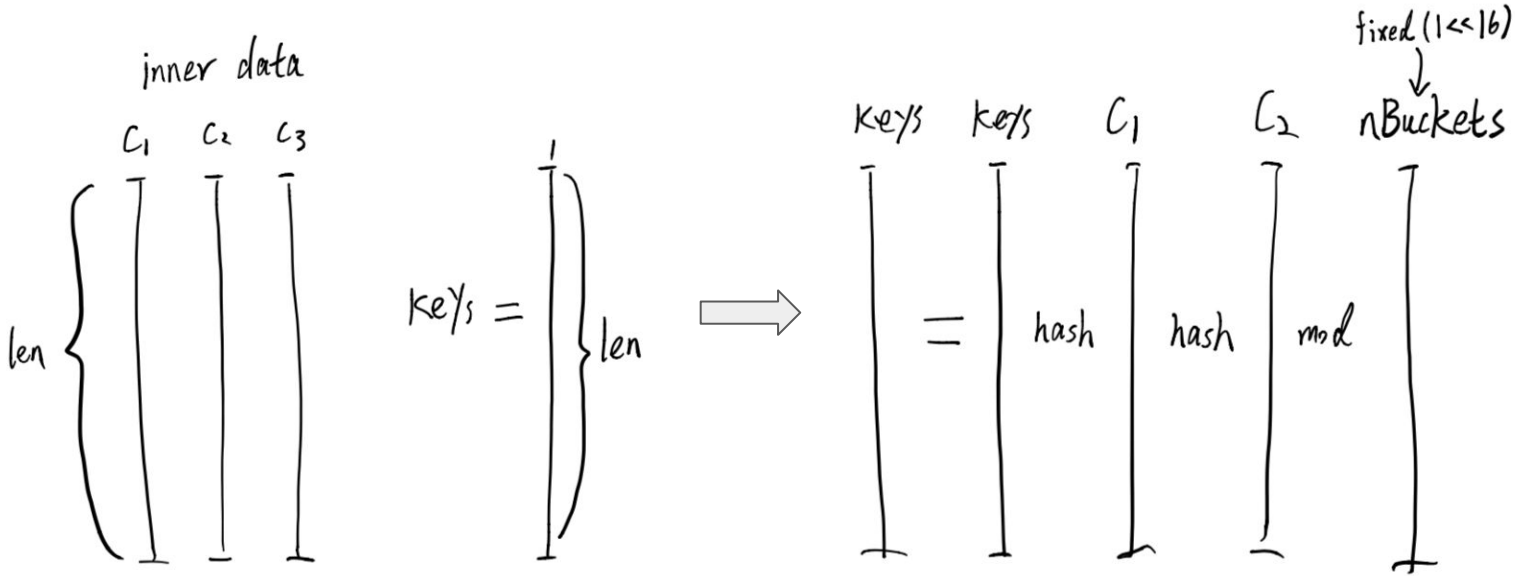
Vectorized Many-To-One Hash Join

SELECT * FROM inner, outer WHERE inner.c1 = outer.c1 AND inner.c2 = outer.c2

- Many(outer rows)-To-One(inner row)

Vectorized Many-To-One Hash Join

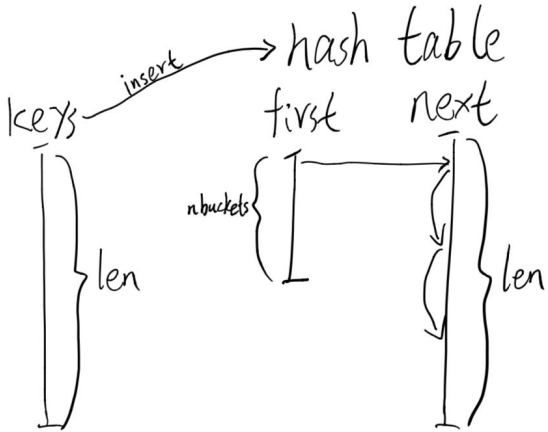
Build phase: hash key computation



Vectorized Many-To-One Hash Join

Build phase: hash table maintaining

- first[i]: the first element(inner RowID) in the bucket i;
- next[i]: the next RowID in the same bucket where row i is;



first[1] = 2

next[2] = 5

next[5] = 7

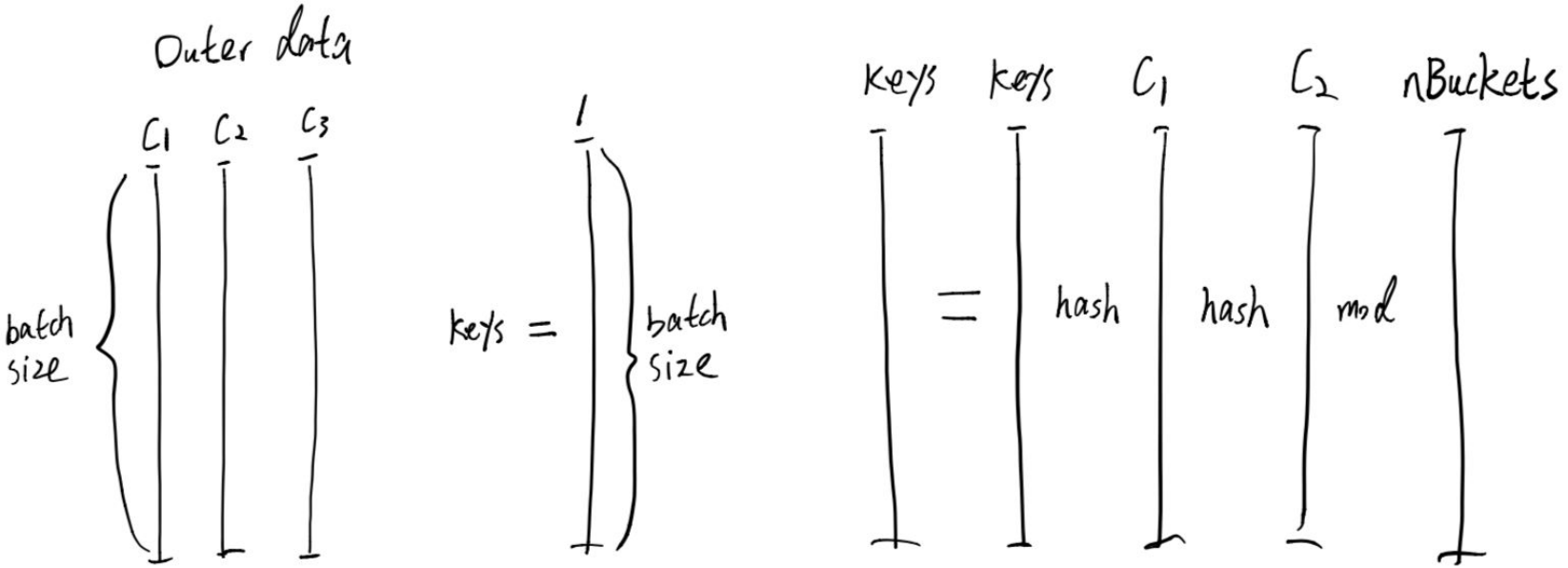
next[7] = 0

bucket[1] = {2, 5, 7}

```
type hashTable struct {
	first     []uint64
	next      []uint64
	same      []uint64
	visited   []bool
	head      []bool
	vals      []coldata.Vec
	valTypes  []types.T
	valCols   []uint32
	keyCols   []uint32
	outCols   []uint32
	outTypes  []types.T
	keys      []coldata.Vec
	buckets   []uint64
	groupID   []uint64
	toCheck   []uint16
	headID    []uint64
	differs   []bool
	...
}
```

Vectorized Many-To-One Hash Join

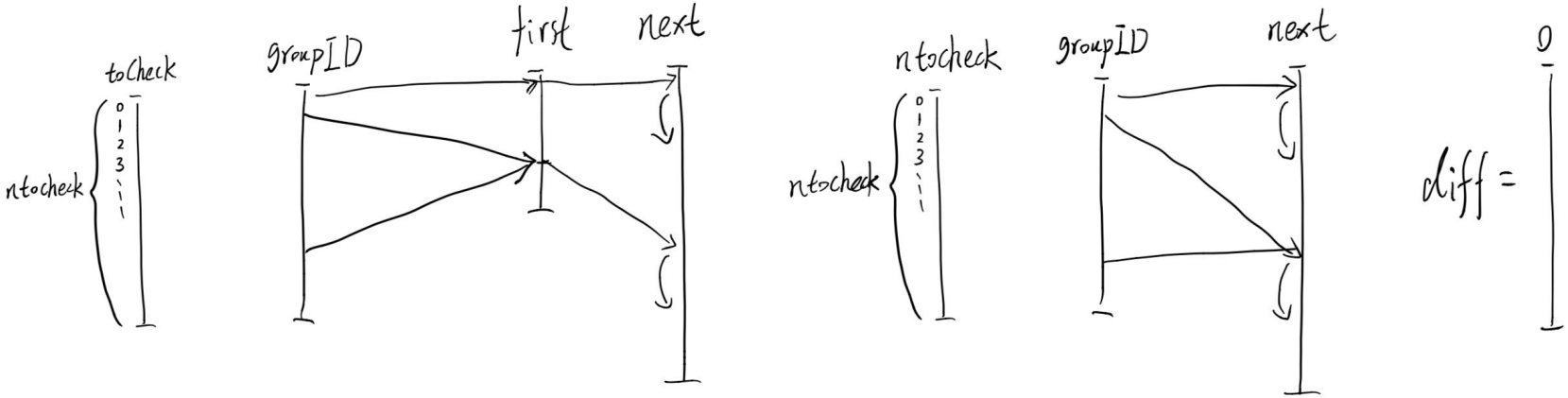
Probing phase: hash key computation



Vectorized Many-To-One Hash Join

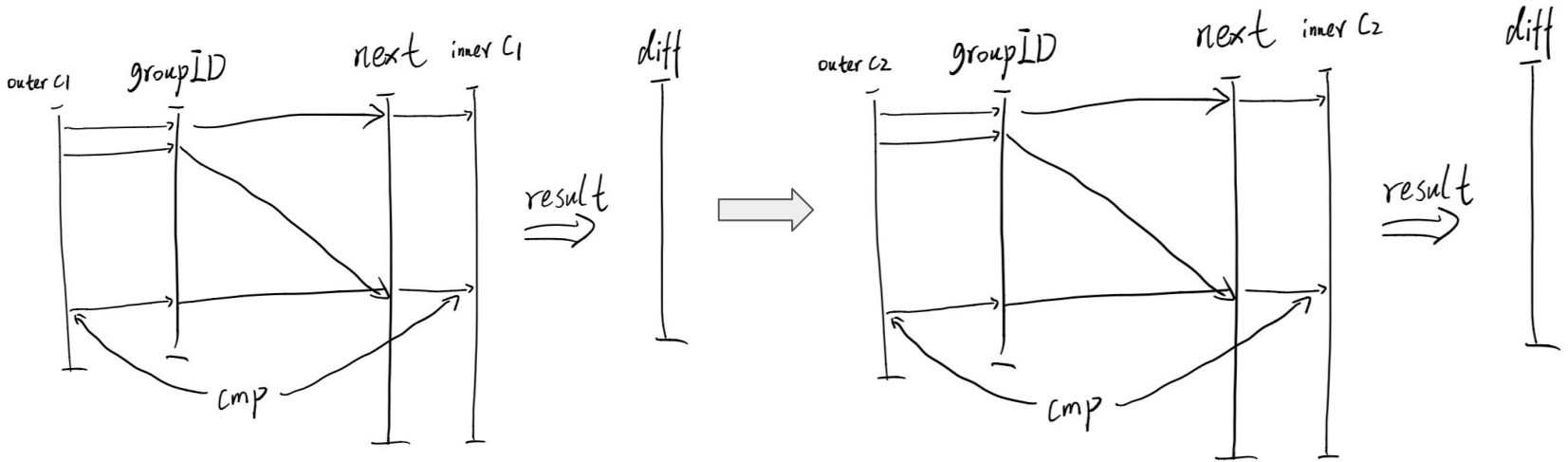
Probing phase: init

- toCheck: outer rowIDs which has not found the matched row in the hash table;
- groupID[i]: the next inner rowID should be compared with the outer row i;
- diff[i]: if outer row i is different with its corresponding inner row in this round of iteration;



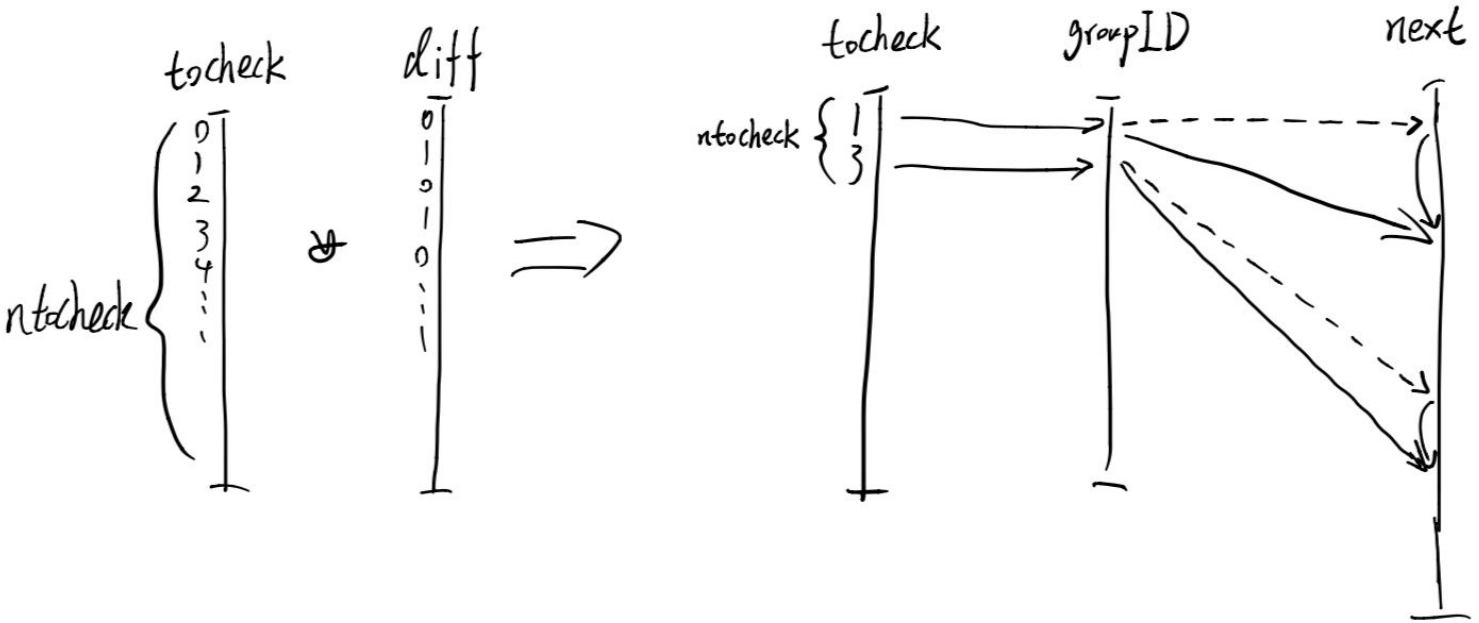
Vectorized Many-To-One Hash Join

Probing phase: checking columns



Vectorized Many-To-One Hash Join

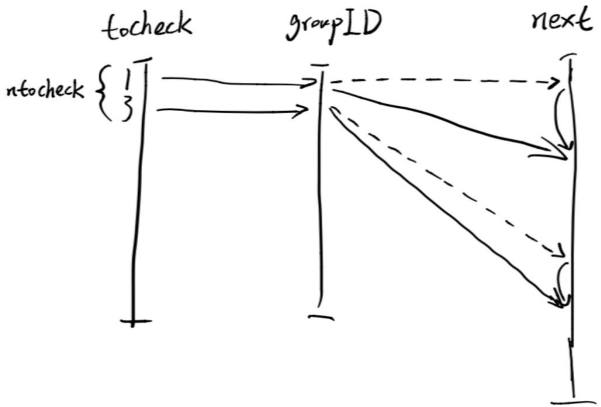
Probing phase: finishing this round of iteration



Vectorized Many-To-One Hash Join

Probing phase: beginning next round

- reset diff[0:nToCheck] to 0;
- beginning next round;
- until:
  - nToCheck == 0: all outer rows have found their matched inner rows;
  - all elements in groupID is 0: no more inner rows we have to check;



Vectorized Many-To-One Hash Join

congregating and matrializing phase

PingCAP

F & Q

Thank You !