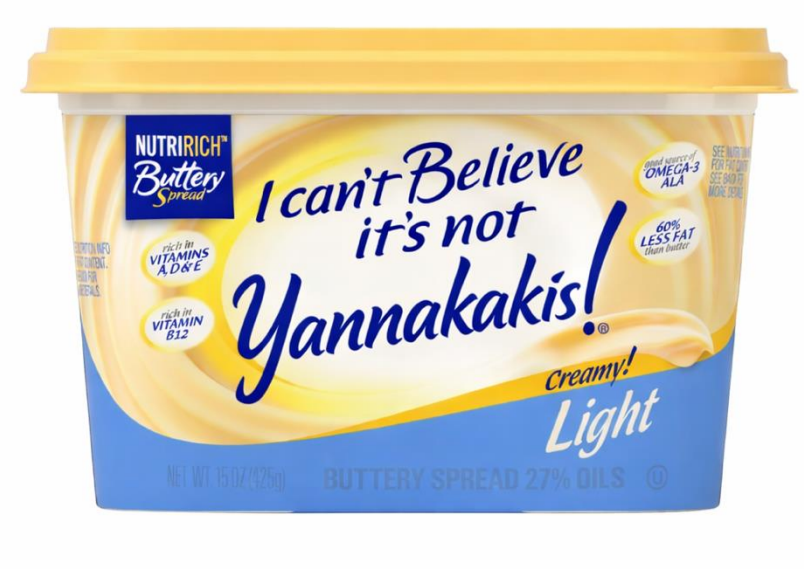
Pragmatic Bitmap Filters in Microsoft SQL Server

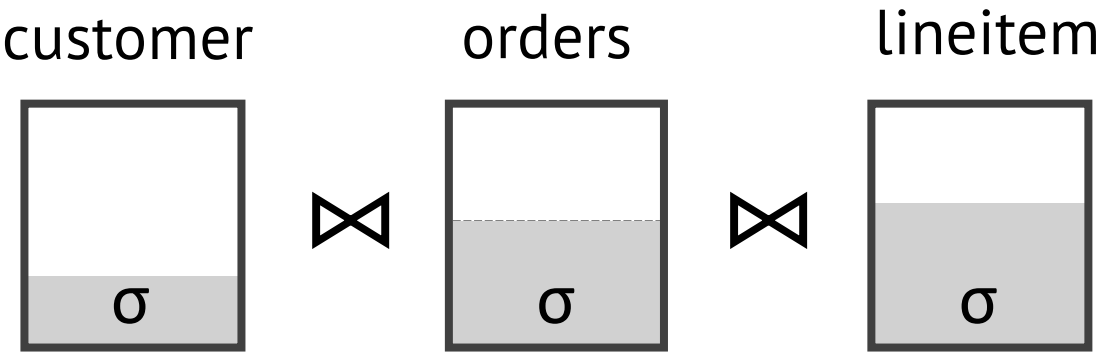
Hangdong Zhao, Yuanyuan Tian, Rana Alotaibi, Bailu Ding, Nicolas Bruno,
Jesús Camacho-Rodríguez, Vassilis Papadimos, Ernesto Cervantes Juárez,
Cesar Galindo-Legaria, Carlo Curino



Microsoft

Pre-filtering is **great**

Example: TPC-H Q3 100 GB

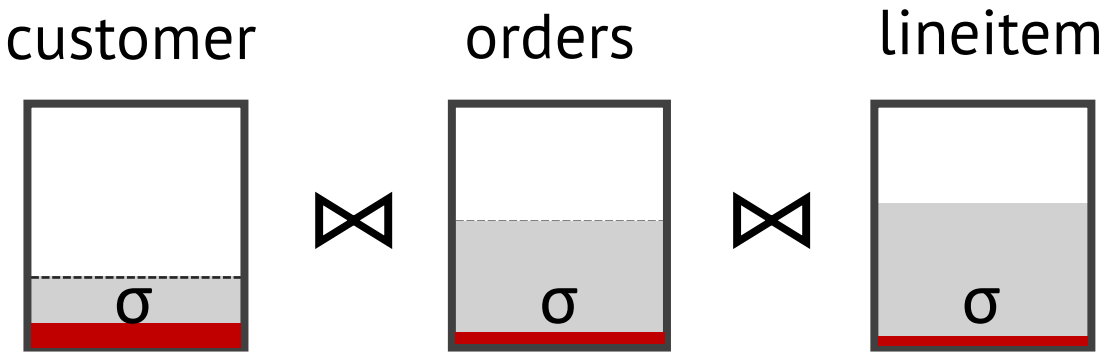


Rows participating in joins after local filtering

20% **49%** **53%**

Pre-filtering is **great**

Example: TPC-H Q3 100 GB



Rows participating in joins after local filtering

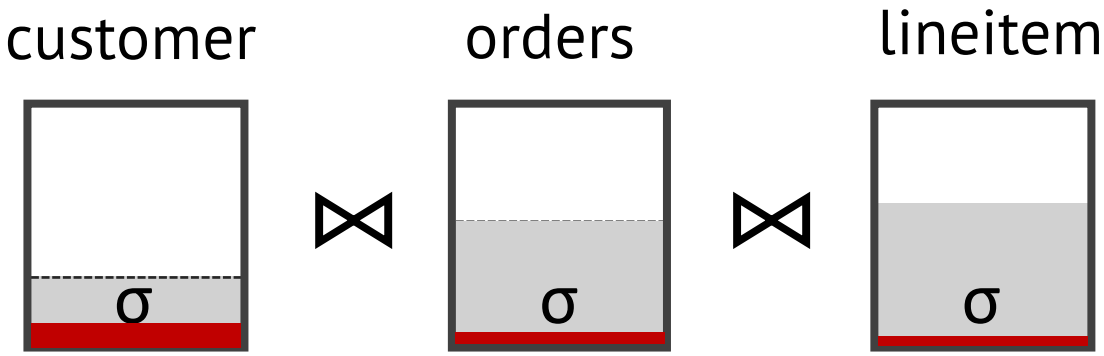
20%	**49%**	**53%**

Rows contributing to query results

5.6%	**<1%**	**<0.5%**

Pre-filtering is **great**

Example: TPC-H Q3 100 GB



Rows participating in joins after local filtering

20%	**49%**	**53%**

Rows contributing to query results

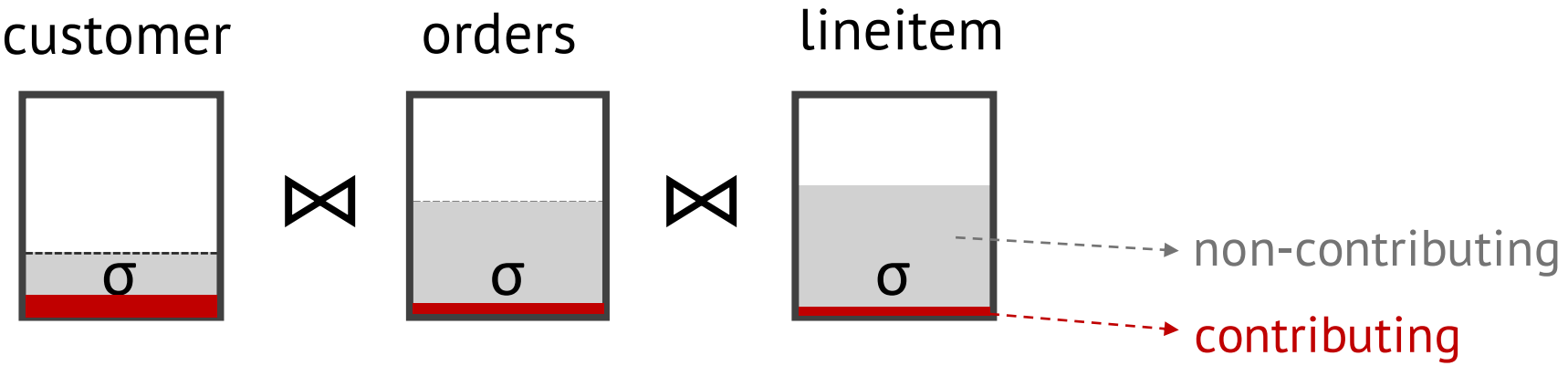
5.6%	**<1%**	**<0.5%**

4–100× size reduction!

most rows do not contribute to the join

Pre-filtering is **great**

Example: TPC-H Q3 100 GB



Rows participating in joins after local filtering

20% **49%** **53%**

Rows contributing to query results

5.6% **<1%** **<0.5%**

4–100× size reduction!

most rows do not contribute to the join

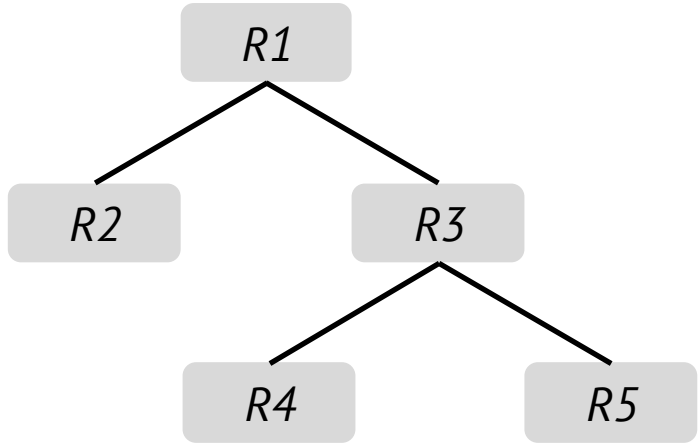
Pre-filtering is **great**

- Often orders-of-magnitude reduction of join input sizes (even in production)
- Yannakakis algorithm is the grand theory for best possible pre-filtering

Yannakakis VLDB'1981

Propagating semijoins first for acyclic joins

...whose join graph is essentially **a tree**

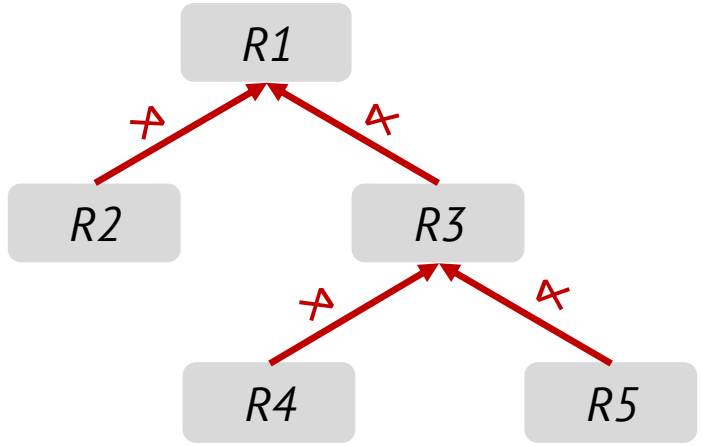


Yannakakis VLDB'1981

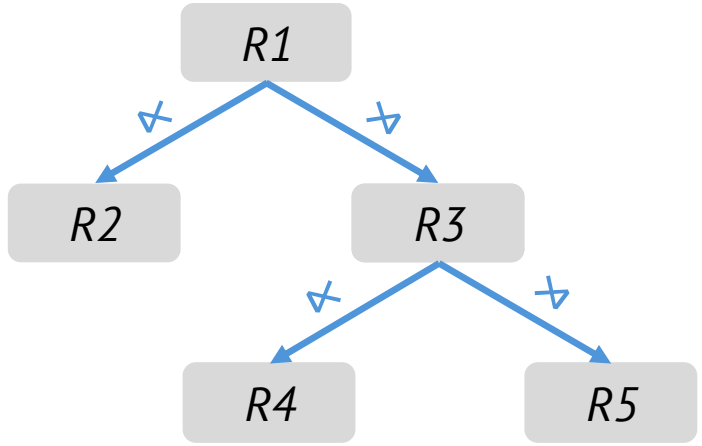
Propagating semijoins first for acyclic joins

Bottom-up pass

- Use semijoin to reduce parent table



Yannakakis VLDB'1981



Propagating semijoins first for acyclic joins

Bottom-up pass

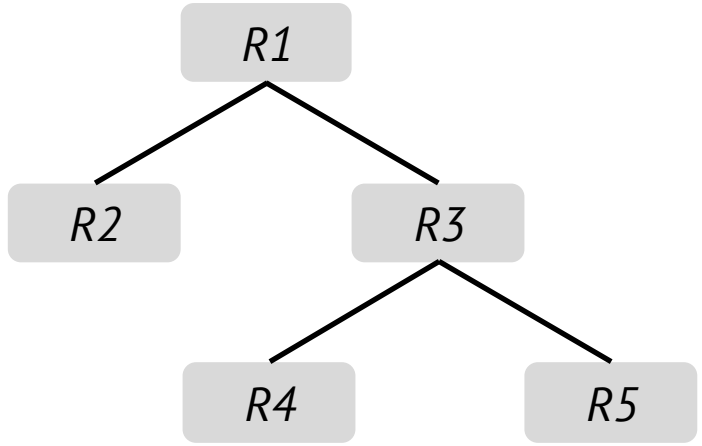
- Use semijoin to reduce parent table

Top-down pass

- Use semijoin to reduce child table

All non-contributing rows are filtered

Yannakakis VLDB'1981



Propagating semijoins first for acyclic joins

Bottom-up pass

- Use semijoin to reduce parent table

Top-down pass

- Use semijoin to reduce child table

Join the reduced tables

Instance-optimal for acyclic joins

Make Yannakakis Practical

For decades, the Yannakakis algorithm has been '**ignored**' by real database engines

Because it

Accepts **only** acyclic joins

Introduces **high overheads** for scans/semijoins/materialization of pre-filtered tables

Huge impacts on query optimization & execution

Make Yannakakis Practical

But recently, there is a renaissance….

A Common Theme
Replace Yannakakis' semijoins by **Bloom filters**

CIDR 2024

- Predicate Transfer: Efficient Pre-Filtering on Multi-Join Queries

VLDB 2024

- Robust Join Processing with Diamond Hardened Joins

SIGMOD 2025

- Debunking the Myth of Join-ordering: Toward Robust SQL Analytics
- Yannakakis+: Practical Acyclic Query Evaluation with Theoretical Guarantees
- Accelerate Distributed Joins with Predicate Transfer

VLDB 2025

- Parachute: Single-Pass Bi-Directional Information Passing
- Including Bloom Filters in Bottom-up Optimization
- Instance-Optimal Acyclic Join Processing Without Regret: Engineering the Yannakakis Algorithm in Column Stores

So… can I optimize SQL Server now?

I was too naïve :)

Why I give up optimizing SQL Server?

...SQL Server is way ahead of academia

Why I give up optimizing SQL Server?

...thanks to the great SQL Server engineers

Ciprian Clinciu
Campbell Fraser
Cesar Galindo-Legaria
Milind Joshi
Michal Nowakiewicz
Vassilis Papadimos
Andrew Richardson
Aleksandras Surna

Why I give up optimizing SQL Server?

SQL Server has been quietly generating—and executing—**instance-optimal** query plans for most of your SQL queries since 2014!

And not only that, it

Accept ~~only acyclic joins~~ **arbitrary join queries**

Bears no additional scans/semijoins/materialization of pre-filtered tables

Carefully controls ~~huge~~ impacts on query optimization/execution

Three Core Pieces

All that in an elegant design!

Batch-mode Hash Join
the only building block needed

Pull-based Execution
cascading bitmap pushdown

Cascades Optimizer
cost-based optimizations
considering bitmap filters

Three Core Pieces

All that in an elegant design!

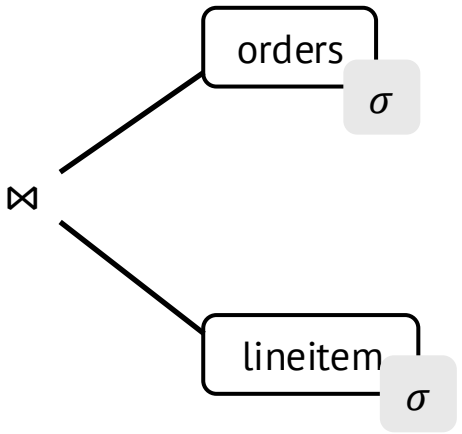
Batch-mode Hash Join
the only building block needed

Pull-based Execution
cascading bitmap pushdown

Cascades Optimizer
cost-based optimizations
considering bitmap filters

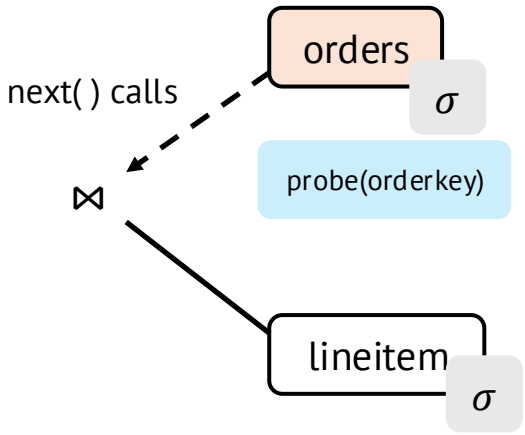
Batch-mode Hash Join

- pull-based execution of a batch-mode hash join



Batch-mode Hash Join

- pull-based execution of a batch-mode hash join

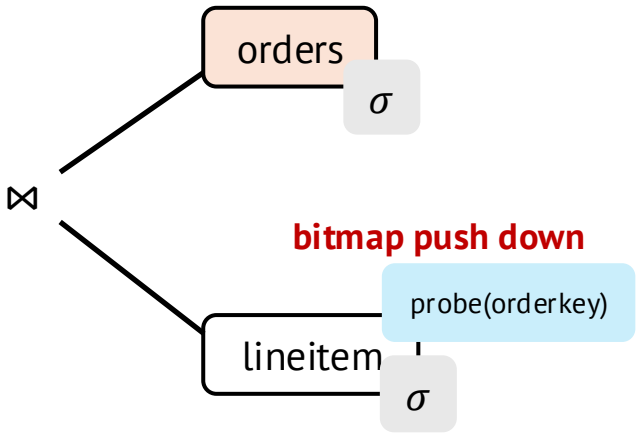


bitmap filter decisions

- Columns to create bitmap filters
- Bit-array or Bloom filters
- Sizes of bitmap filters

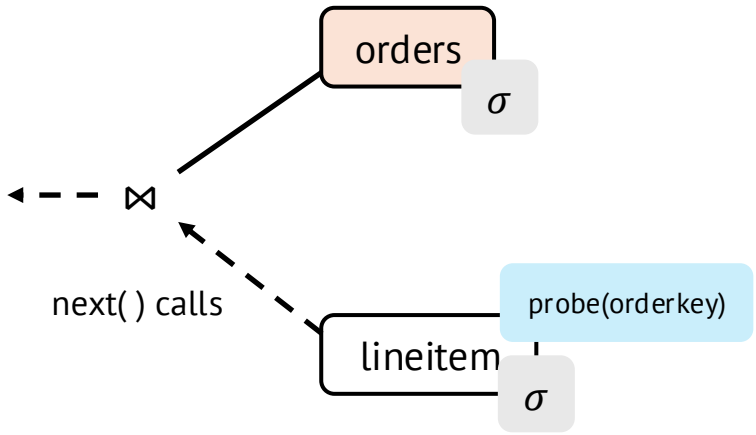
Batch-mode Hash Join

- pull-based execution of a batch-mode hash join



Batch-mode Hash Join

- pull-based execution of a batch-mode hash join



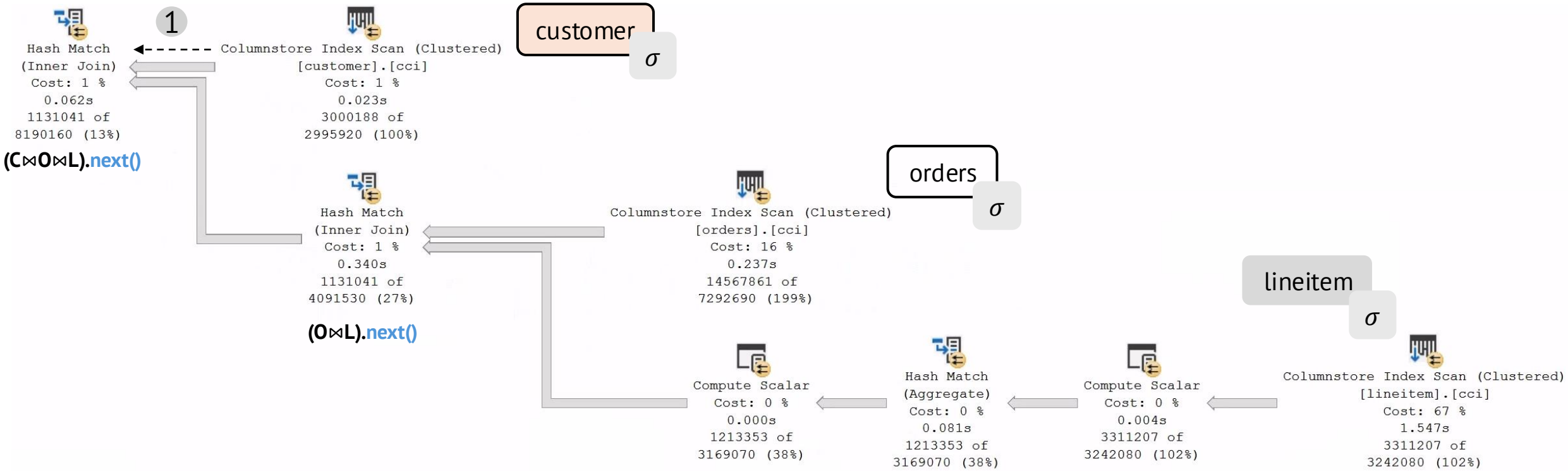
Three Core Pieces

Batch-mode Hash Join
the only building block needed

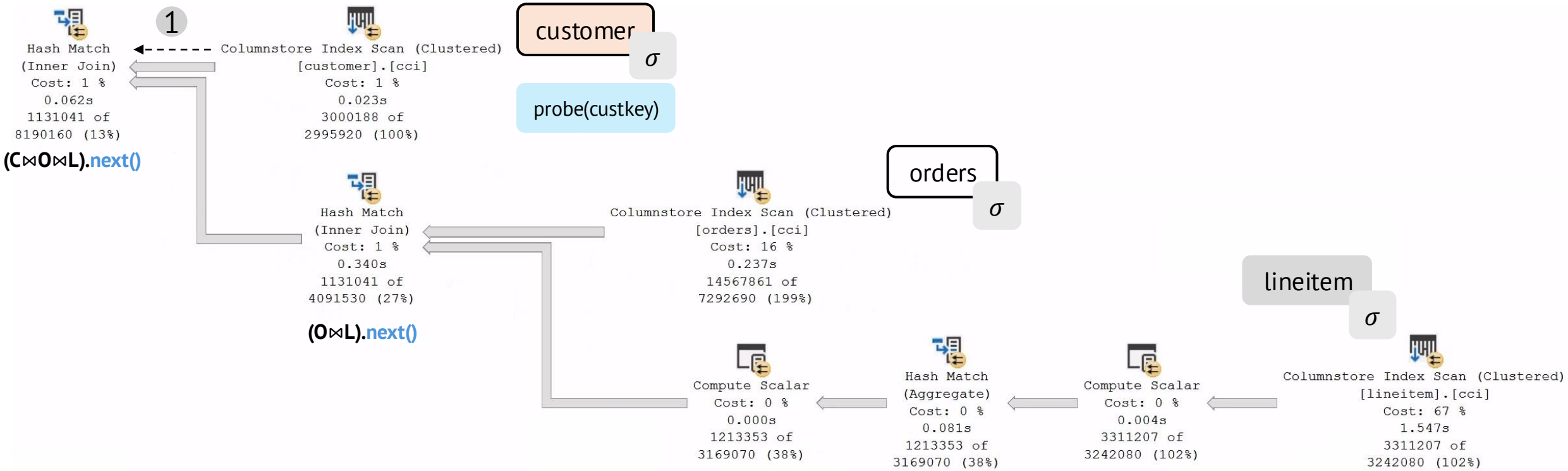
Pull-based Execution
cascading bitmap pushdown

Cascades Optimizer
cost-based optimizations
considering bitmap filters

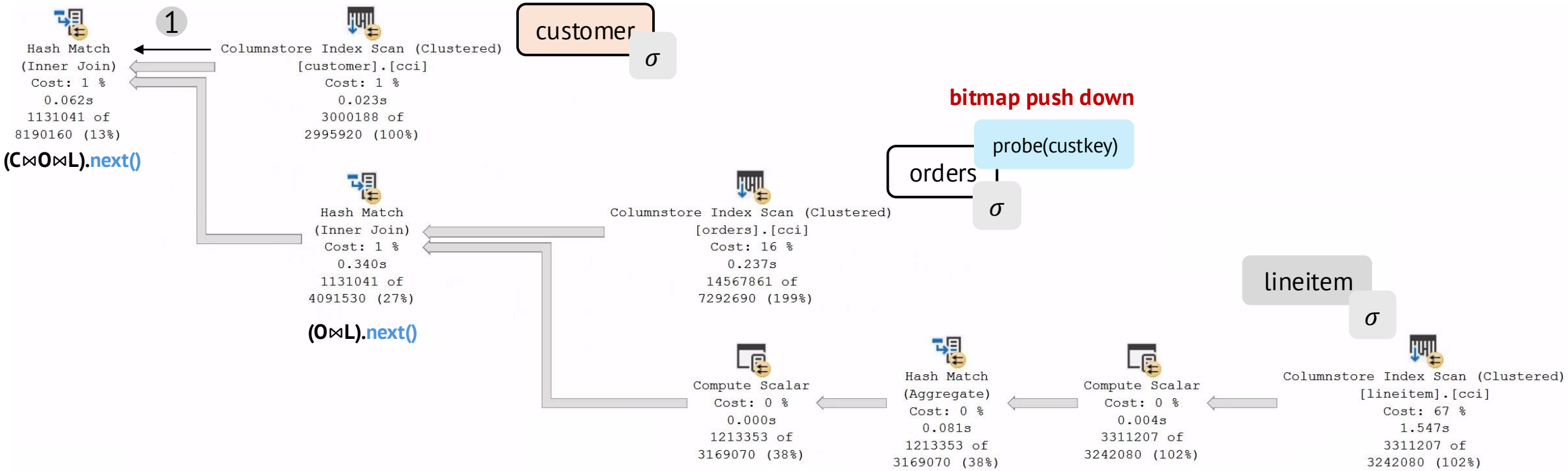
Composing Hash Join



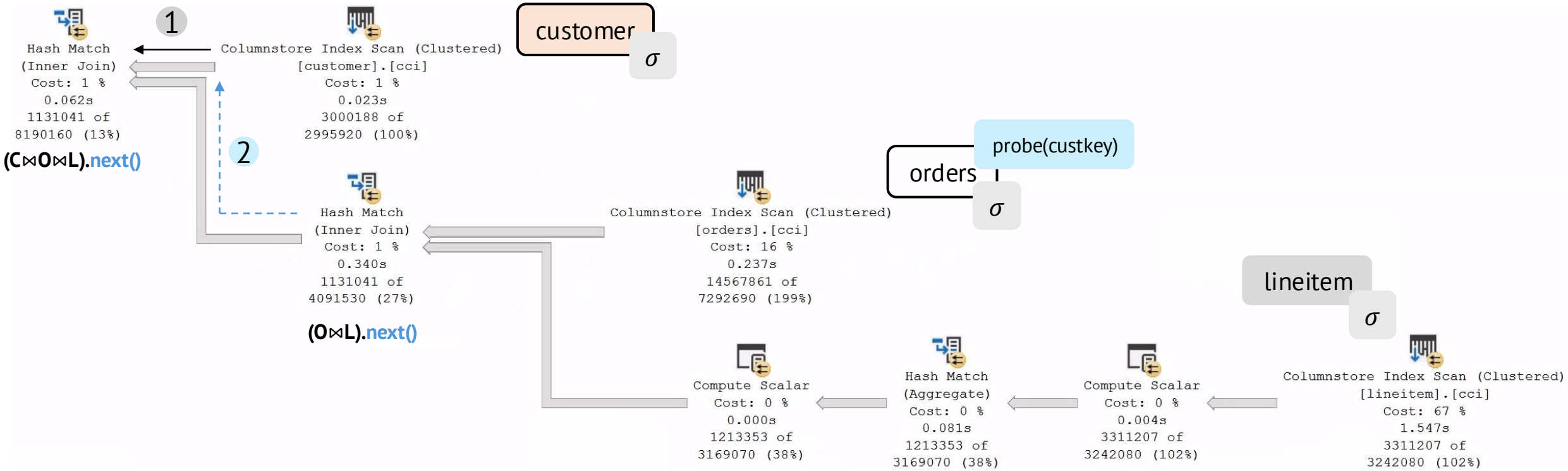
Composing Hash Join



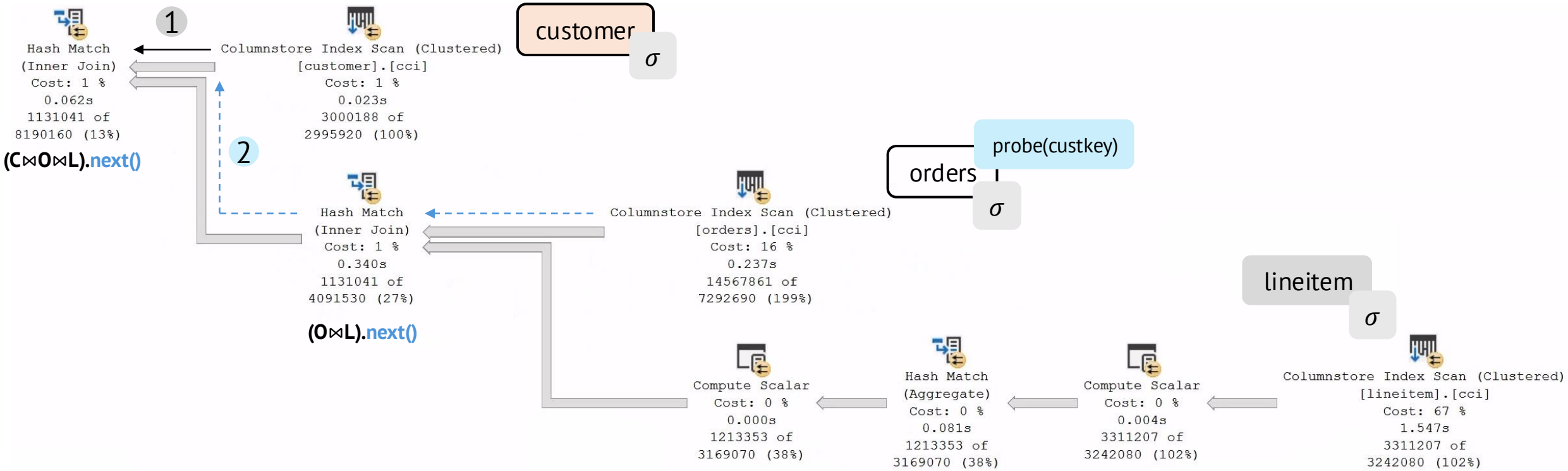
Composing Hash Join



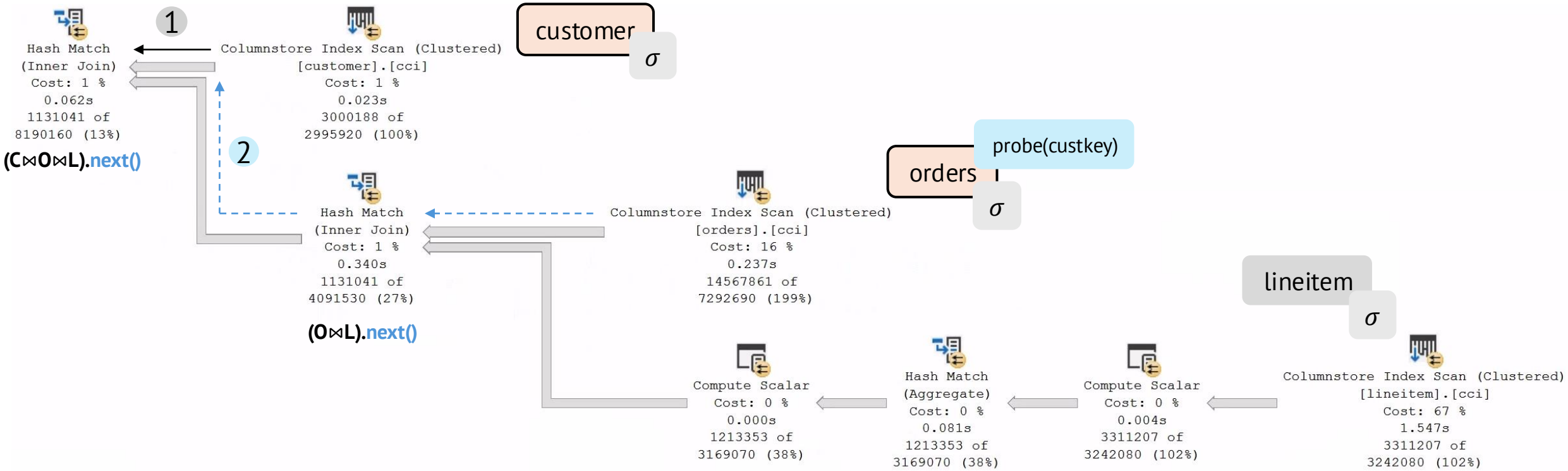
Composing Hash Join



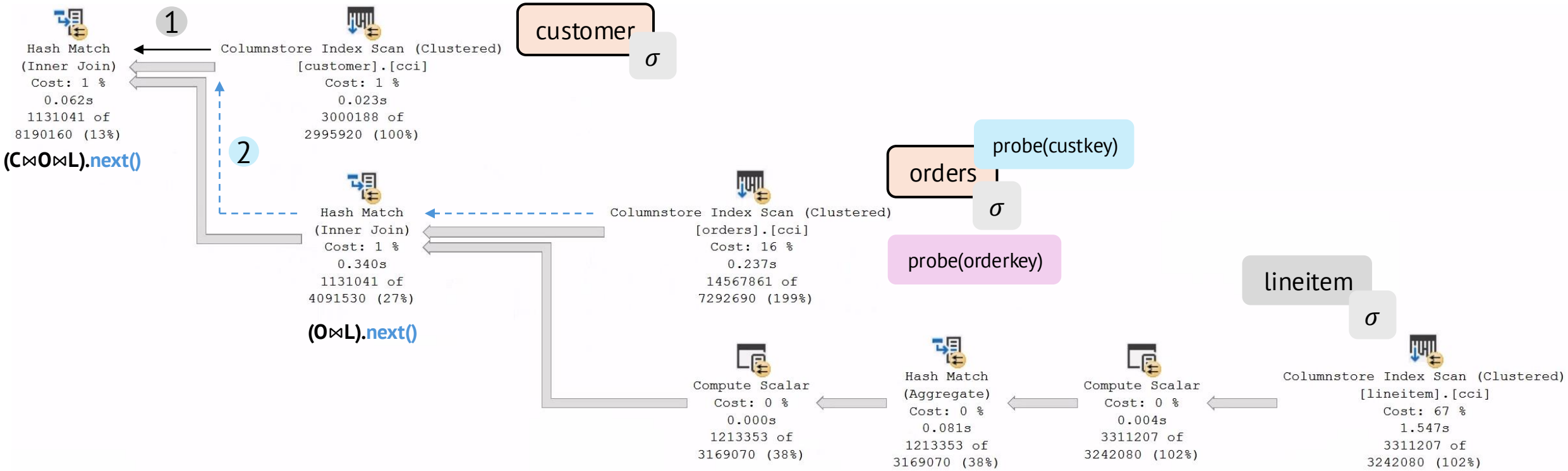
Composing Hash Join



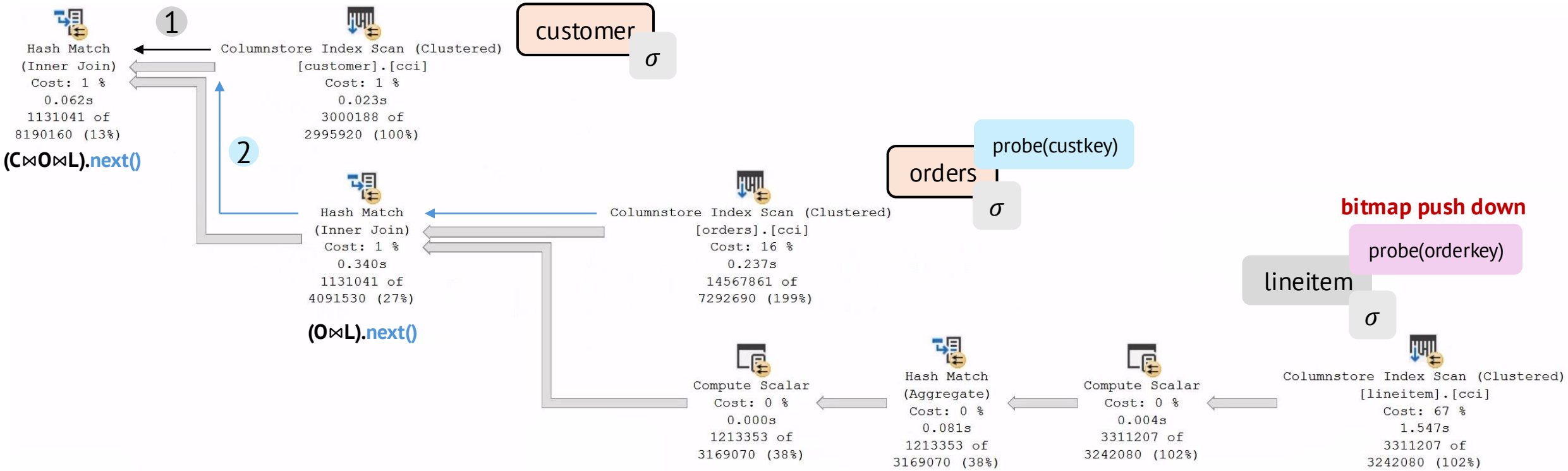
Composing Hash Join



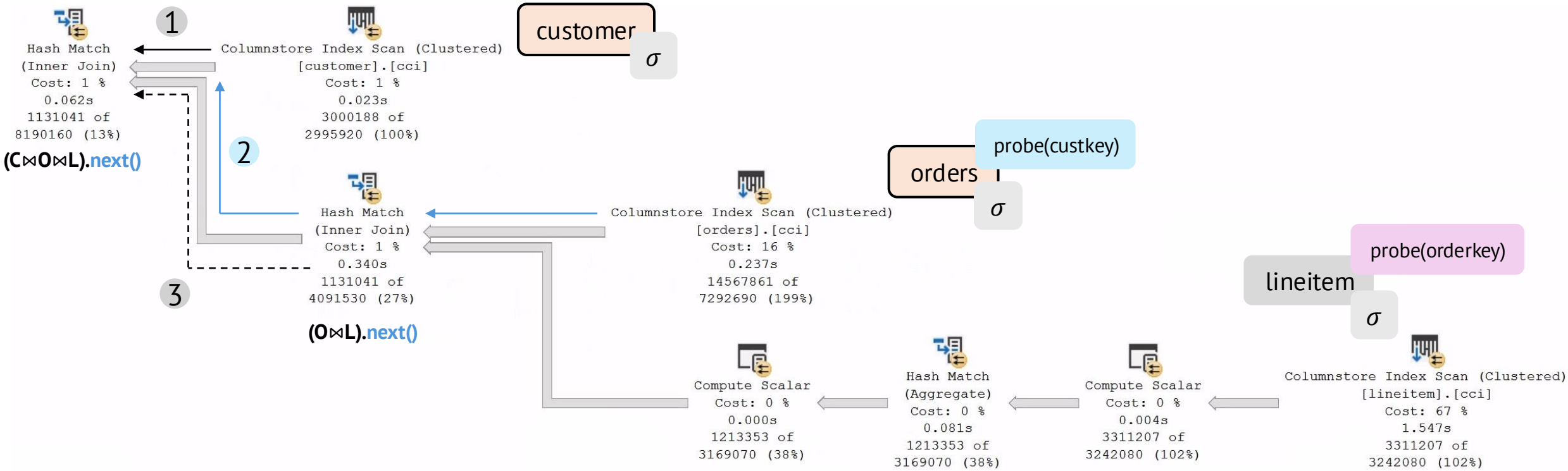
Composing Hash Join



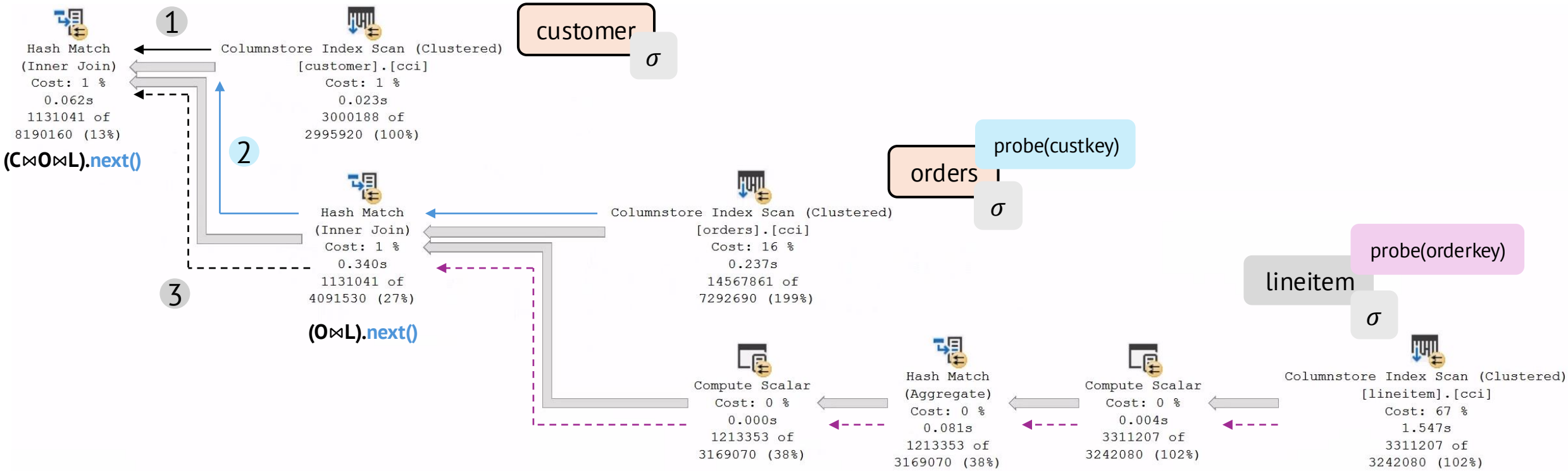
Composing Hash Join



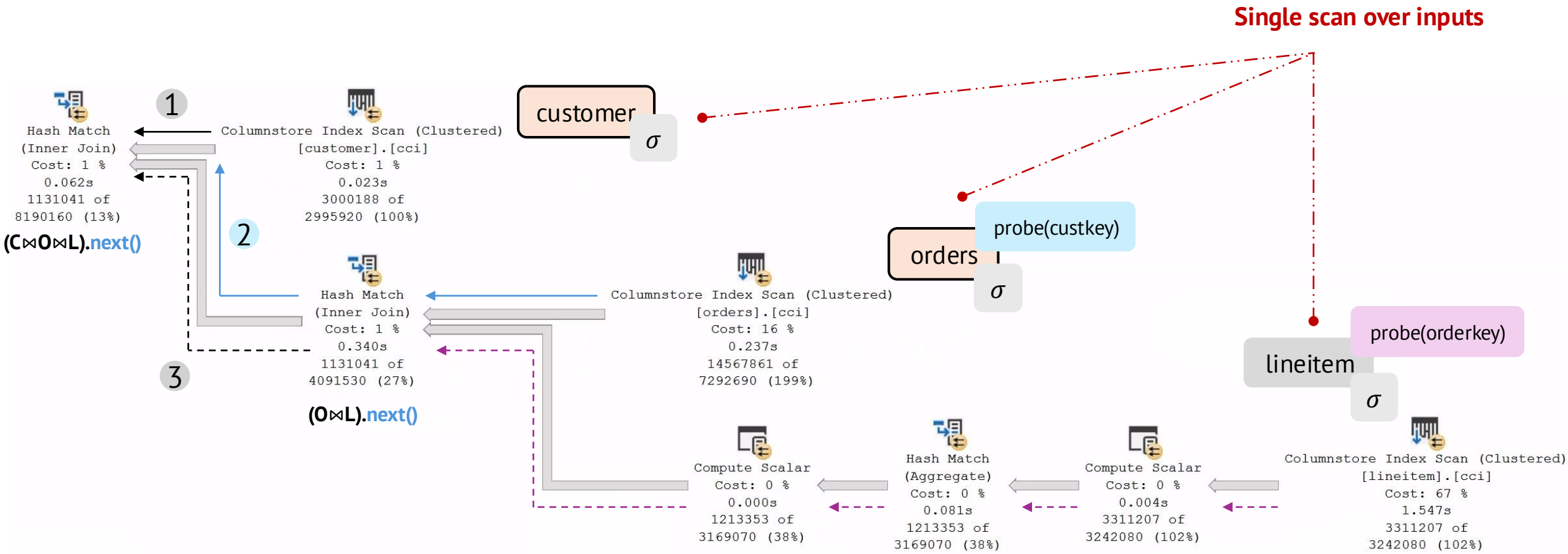
Composing Hash Join



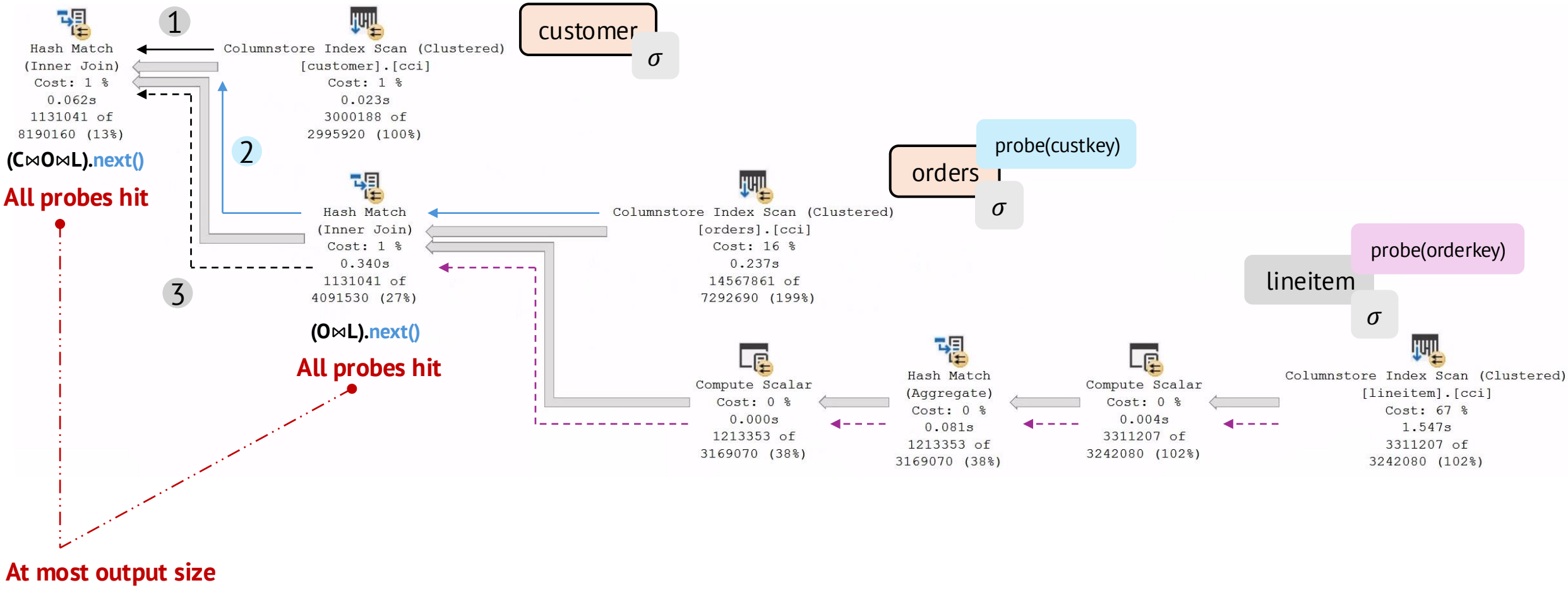
Composing Hash Join



Instance-optimal Query Execution



Instance-optimal Query Execution



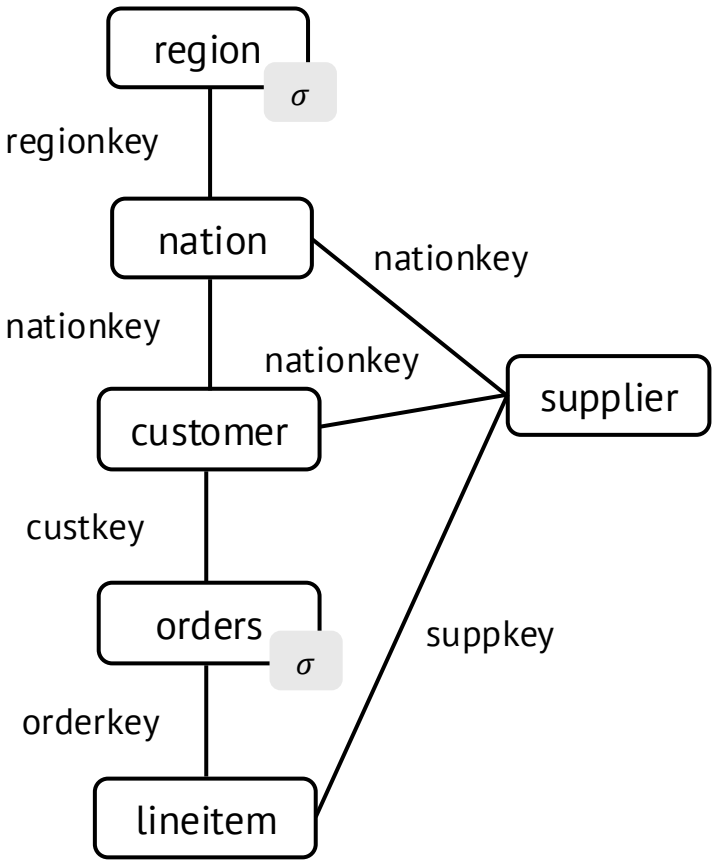
Three Core Pieces

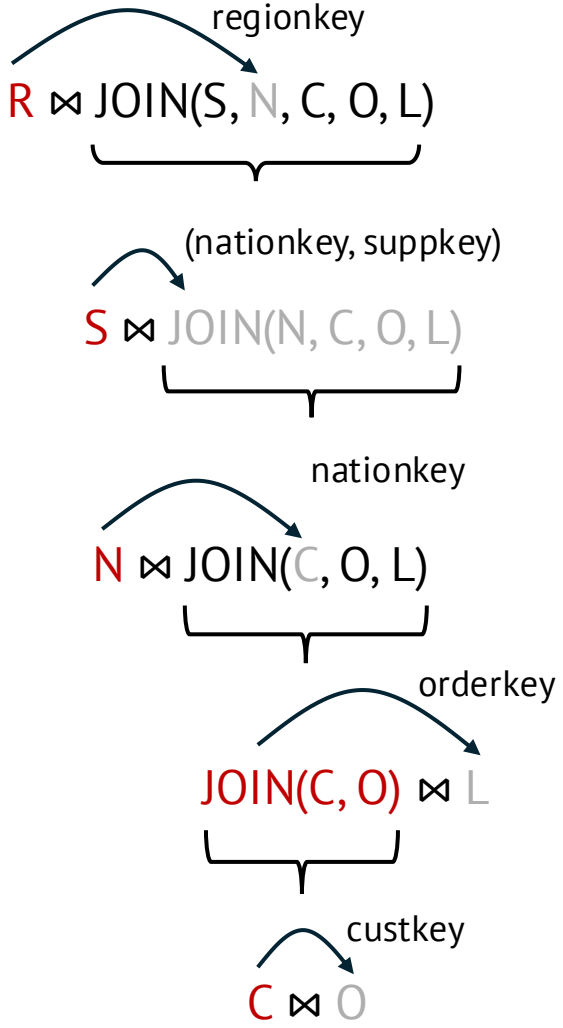
Batch-mode Hash Join
the only building block needed

Pull-based Execution
cascading bitmap pushdown

Cascades Optimizer
cost-based optimizations
considering bitmap filters

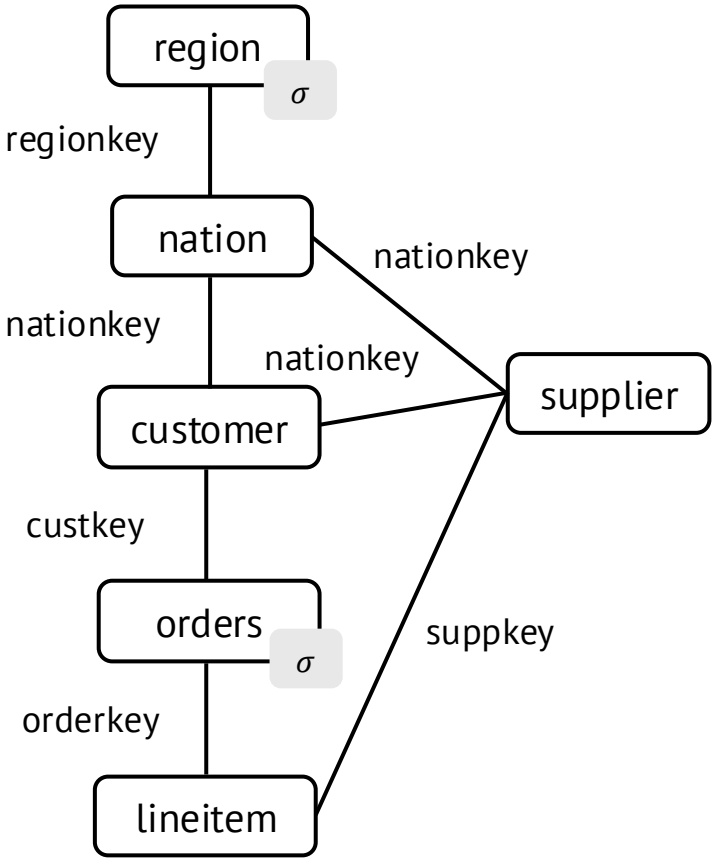
Pre-filtering for Arbitrary Queries

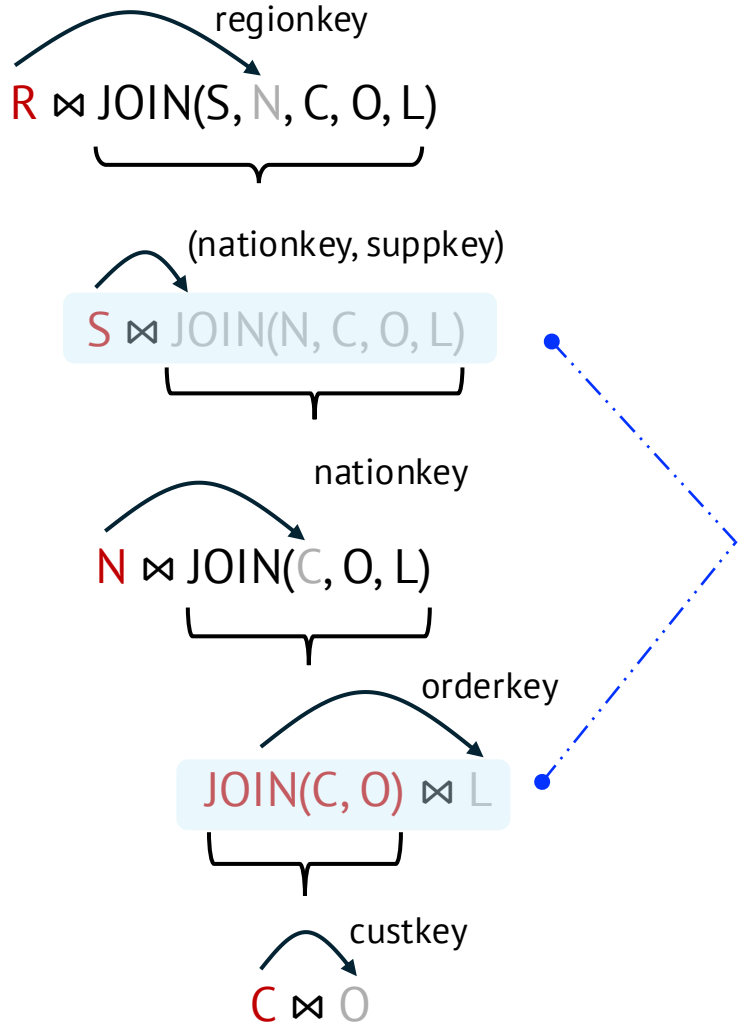




Example: TPC-H Q5 100 GB

Pre-filtering for Arbitrary Queries





Cost-based decisions on bitmap passing from & to any intermediate result

Experiments

- TPC-H 22 queries (100 GB)
- 7 queries become **>2x** faster

ID	Shape of join graph	Runtime speedup	Inst.-opt.?	#rows to join	SQL Srv. reads (%)	Yannak. reads (%)
1	No join	–	–	–	–	–
2	Line	**3.47×**	✓	81.1M	0.43%	0.21%
3	Line (C-O-L)	1.91×	✓	399.4M	**5.23%**	**1.25%**
4	A semijoin	1.92×	✓	385.1M	5.28%	5.14%
5▲	Cyclic join	**2.33×**	–	638.8M	**4.2%**	**1.21%**
6	No join	–	–	–	–	–
7▲	Line	**3.00×**	✓	348.4M	**4.15%**	**0.81%**
8▲	Snowflake	**2.86×**	✗	661.8M	**1.92%**	**0.16%**
9	Snowflake	1.47×	✗	832.2M	8.23%	8.22%
10	Line (N-C-O-L)	1.23×	✓	168.8M	12.51%	12.01%
11	Line	1.75×	✓	81.1M	8.01%	8.01%
12	Single join	1.21×	✓	153.2M	3.97%	3.95%
13	An outer join	–	–	–	–	–
14	Single join	1.14×	✓	27.5M	49.98%	49.98%
15	Single join	1.07×	✓	23.7M	95.78%	95.78%
16	Antijoin & join	1.03×	✓	83.0M	17.91%	17.91%
17	Correlate joins	**2.55×**	–	51.5M	1.22%	1.21%
18▲	Line (C-O-L)	1.22×	✗	1215.1M	49.39%	49.39%
19▲	Single join	1.82×	✓	25.8M	0.43%	0.39%
20	Correlate joins	**2.01×**	–	172.3M	0.2%	0.07%
21	Correlate joins	**2.36×**	–	379.4M	**21.43%**	**19.24%**
22	An antijoin	1.27×	✓	155.8M	**17.41%**	**15.94%**

Table 1: TPC-H (SF=100) results. ✓ (×) mark instance-optimal plans (not) chosen; – are inapplicable cases; ▲ denotes optimizer slowdown. Last three columns show total rows (in millions) for all joins and % after pre-filtering of SQL Server and Yannakakis.

Experiments

- TPC-H 22 queries (100 GB)
- 7 queries become **>2x** faster
- **12** queries are instance-optimal

- 3 queries are **not** using inst.-opt plan
  - Skip futile bitmap filters
  - Use bitmaps on intermediate results

ID	Shape of join graph	Runtime speedup	Inst.-opt.?	#rows to join	SQL Srv. reads (%)	Yannak. reads (%)
1	No join	–	–	–	–	–
2	Line	**3.47×**	✓	81.1M	0.43%	0.21%
3	Line (C-O-L)	1.91×	✓	399.4M	**5.23%**	**1.25%**
4	A semijoin	1.92×	✓	385.1M	5.28%	5.14%
5▲	Cyclic join	**2.33×**	–	638.8M	**4.2%**	**1.21%**
6	No join	–	–	–	–	–
7▲	Line	**3.00×**	✓	348.4M	**4.15%**	**0.81%**
8▲	Snowflake	**2.86×**	✗	661.8M	**1.92%**	**0.16%**
9	Snowflake	1.47×	✗	832.2M	8.23%	8.22%
10	Line (N-C-O-L)	1.23×	✓	168.8M	12.51%	12.01%
11	Line	1.75×	✓	81.1M	8.01%	8.01%
12	Single join	1.21×	✓	153.2M	3.97%	3.95%
13	An outer join	–	–	–	–	–
14	Single join	1.14×	✓	27.5M	49.98%	49.98%
15	Single join	1.07×	✓	23.7M	95.78%	95.78%
16	Antijoin & join	1.03×	✓	83.0M	17.91%	17.91%
17	Correlate joins	**2.55×**	–	51.5M	1.22%	1.21%
18▲	Line (C-O-L)	1.22×	✗	1215.1M	49.39%	49.39%
19▲	Single join	1.82×	✓	25.8M	0.43%	0.39%
20	Correlate joins	**2.01×**	–	172.3M	0.2%	0.07%
21	Correlate joins	**2.36×**	–	379.4M	**21.43%**	**19.24%**
22	An antijoin	1.27×	✓	155.8M	**17.41%**	**15.94%**

Table 1: TPC-H (SF=100) results. ✓ (✗) mark instance-optimal plans (not) chosen; – are inapplicable cases; ▲ denotes optimizer slowdown. Last three columns show total rows (in millions) for all joins and % after pre-filtering of SQL Server and Yannakakis.

Experiments

- TPC-H 22 queries (100 GB)
- 7 queries become **>2x** faster
- **12** queries are instance-optimal

- 3 queries are **not** using inst.-opt plan
  - Skip futile bitmap filters
  - Use bitmaps on intermediate results

- 5 have **>1% gaps (but <4%)** from Yannakakis
  - SQL Server already gets most pre-filtering benefits

ID	Shape of join graph	Runtime speedup	Inst.-opt.?	#rows to join	SQL Srv. reads (%)	Yannak. reads (%)
1	No join	–	–	–	–	–
2	Line	**3.47×**	✓	81.1M	0.43%	0.21%
3	Line (C-O-L)	1.91×	✓	399.4M	**5.23%**	**1.25%**
4	A semijoin	1.92×	✓	385.1M	5.28%	5.14%
5▲	Cyclic join	**2.33×**	–	638.8M	**4.2%**	**1.21%**
6	No join	–	–	–	–	–
7▲	Line	**3.00×**	✓	348.4M	**4.15%**	**0.81%**
8▲	Snowflake	**2.86×**	✗	661.8M	**1.92%**	**0.16%**
9	Snowflake	1.47×	✗	832.2M	8.23%	8.22%
10	Line (N-C-O-L)	1.23×	✓	168.8M	12.51%	12.01%
11	Line	1.75×	✓	81.1M	8.01%	8.01%
12	Single join	1.21×	✓	153.2M	3.97%	3.95%
13	An outer join	–	–	–	–	–
14	Single join	1.14×	✓	27.5M	49.98%	49.98%
15	Single join	1.07×	✓	23.7M	95.78%	95.78%
16	Antijoin & join	1.03×	✓	83.0M	17.91%	17.91%
17	Correlate joins	**2.55×**	–	51.5M	1.22%	1.21%
18▲	Line (C-O-L)	1.22×	✗	1215.1M	49.39%	49.39%
19▲	Single join	1.82×	✓	25.8M	0.43%	0.39%
20	Correlate joins	**2.01×**	–	172.3M	0.2%	0.07%
21	Correlate joins	**2.36×**	–	379.4M	**21.43%**	**19.24%**
22	An antijoin	1.27×	✓	155.8M	**17.41%**	**15.94%**

Table 1: TPC-H (SF=100) results. ✓ (×) mark instance-optimal plans (not) chosen; – are inapplicable cases; ▲ denotes optimizer slowdown. Last three columns show total rows (in millions) for all joins and % after pre-filtering of SQL Server and Yannakakis.

Future Directions

- Accurate bitmap-aware Cardinality Estimations & Cost Models
- Robust Query Plans for Complex SQL Workloads
- Pre-filtering Solutions for Modern Data Platforms & Analytics

Thank you

Check out our papers for

- Formal proof of instance optimality for SQL Server
- Careful costing to generate optimal plans
- Robustness and semi-robustness guarantees come as by-product
- A better presentation