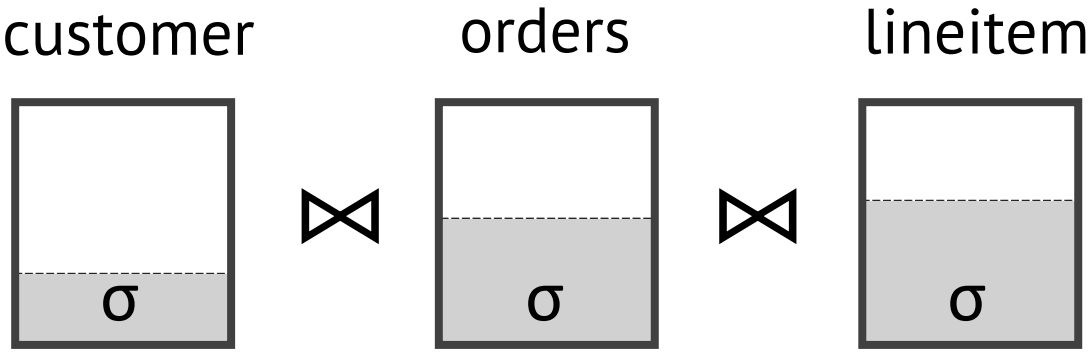
Pragmatic Bitmap Filters in Microsoft SQL Server

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 Techniques

TPC-H Q3 100 GB

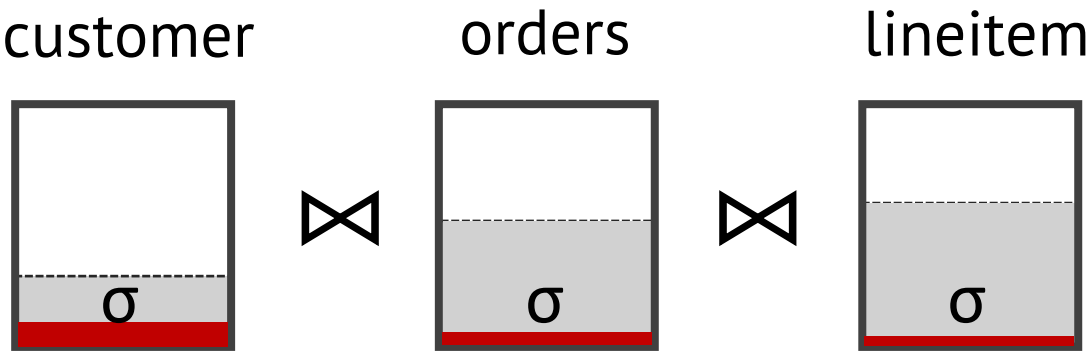


Rows participating in joins after local filtering

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

Pre-filtering Techniques

TPC-H Q3 100 GB



Rows participating in joins after local filtering

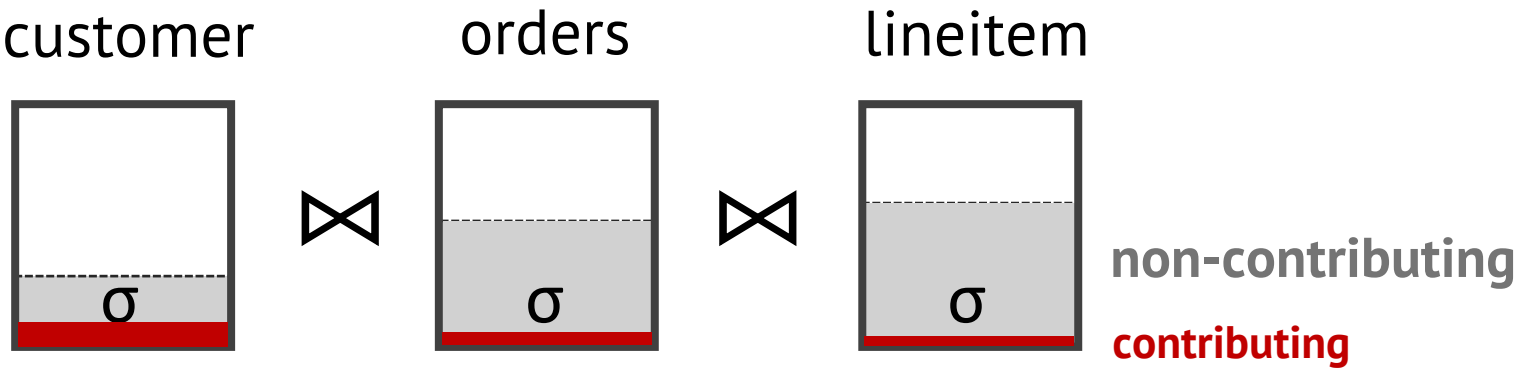
20% **49%** **53%**

Rows contributing to query results

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

Pre-filtering Techniques

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 Potentials

- orders-of-magnitude reduction of join input table sizes (total # of rows in all joining tables)

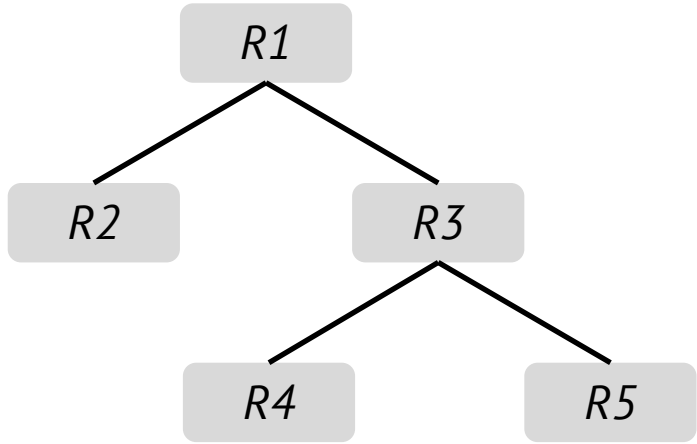
- Yannakakis algorithm is the grand theory for pre-filtering (or sideways information passing)

TPC-H	pre-filtering potentials
Q2	559
Q3	80
Q4	20
Q5	56
Q7	122
Q8	669
Q9	12
Q10	8
Q11	25
Q12	25
Q13	1
Q14	2
Q15	1
Q16	6
Q17	931
Q18	16827
Q19	232
Q20	1489
Q21	23
Q22	2
Geometric Mean	**44**

Yannakakis VLDB'1981

Propagating semijoins first for acyclic joins

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

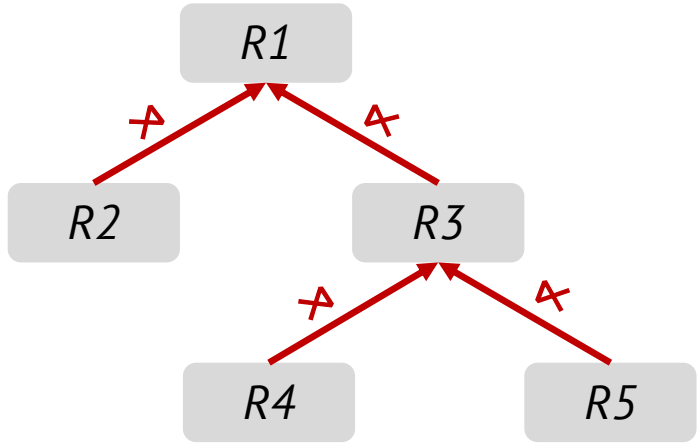


Yannakakis VLDB'1981

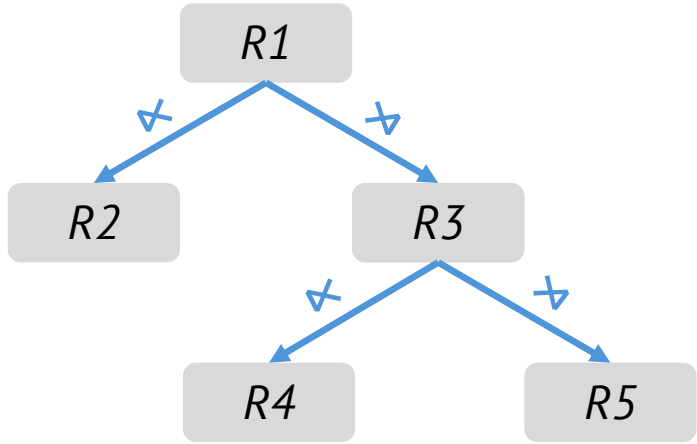
Propagating semijoins first for acyclic joins

Bottom-up pass

- Use semijoin to reduce parent table



Yannakakis VLDB'1981



Propagating semijoins first for acyclic joins

Bottom-up pass

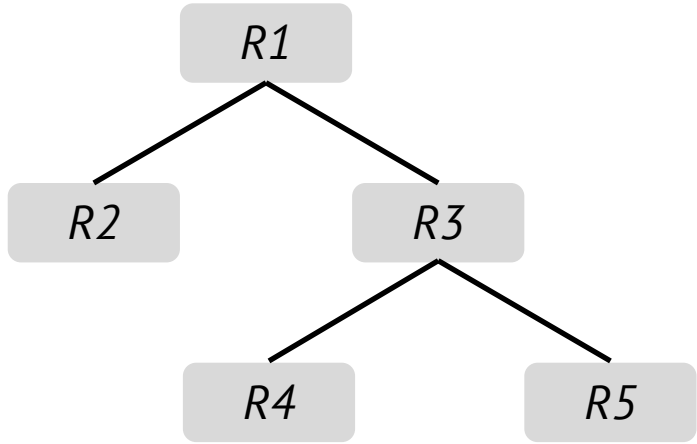
- Use semijoin to reduce parent table

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**

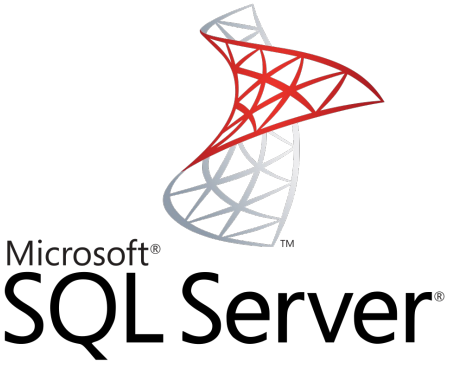
CIDR 2024
- Predicate Transfer: Efficient Pre-Filtering on Multi-Join Queries

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

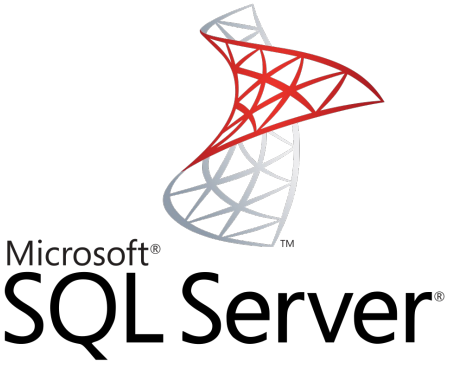
VLDB 2025
- Parachute: Single-Pass Bi-Directional Information Passing
- Including Bloom Filters in Bottom-up Optimization
- Instance-Optimal Acyclic Join Processing Without Regret: Engineering the Yannakakis Algorithm in Column Stores

SQL Server's Approach



...SQL Server is way ahead of academia

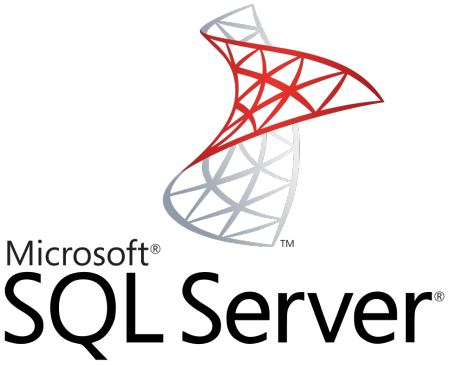
SQL Server's Approach



...thanks to the great SQL Server engineers

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

SQL Server's Approach



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

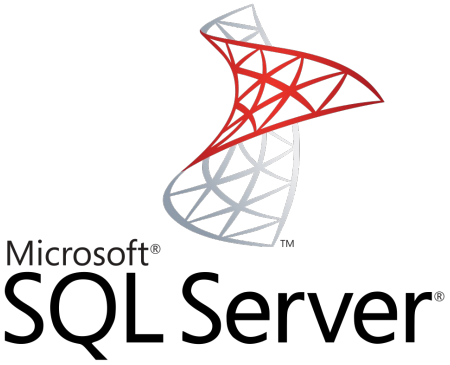
And not only that, it

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

SQL Server's Approach



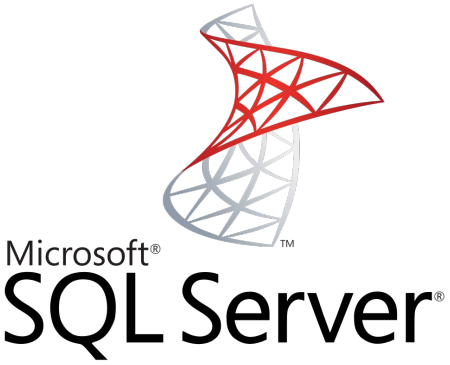
All that in an elegant design!

Batch-mode Hash Join
the only building block needed

Pull-based Execution
cascading bitmap pushdown

Cascades Optimizer
cost-based optimizations
considering bitmap filters

SQL Server's Approach



All that in an elegant design!

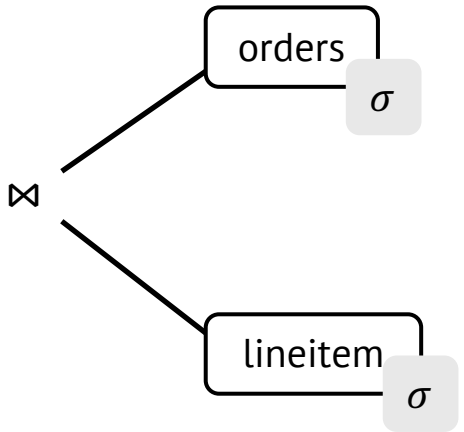
Batch-mode Hash Join
the only building block needed

Pull-based Execution
cascading bitmap pushdown

Cascades Optimizer
cost-based optimizations
considering bitmap filters

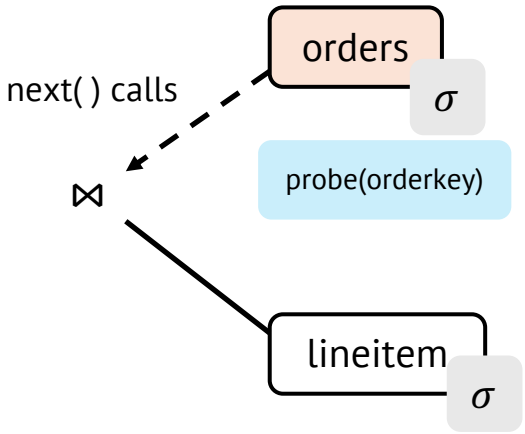
Batch-mode Hash Join

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



Batch-mode Hash Join

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

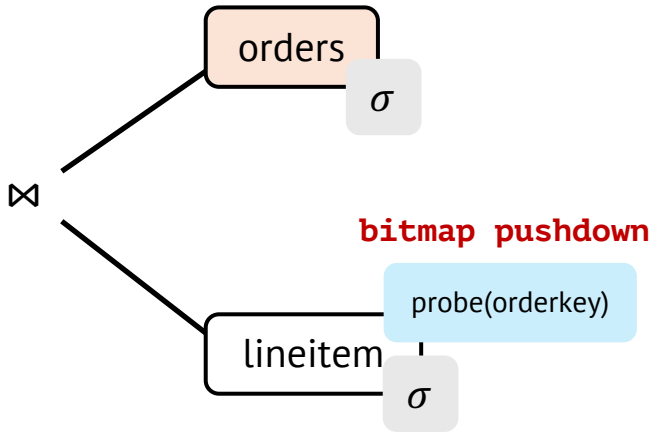


Key bitmap filter decisions

- Columns to create bitmap filters
- Bit-array, Bloom filters, or zonemaps
- Capacity and sizes of bitmaps

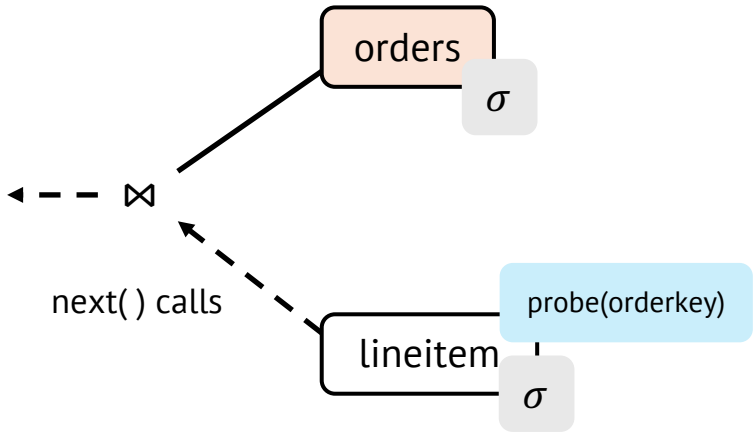
Batch-mode Hash Join

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

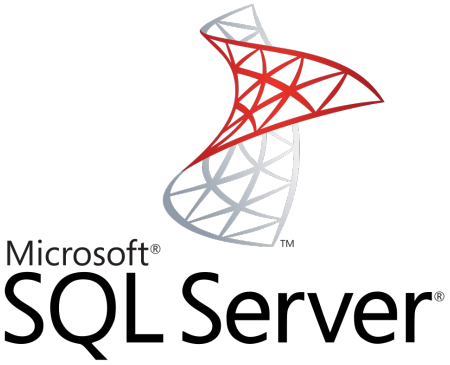


Batch-mode Hash Join

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



SQL Server's Approach

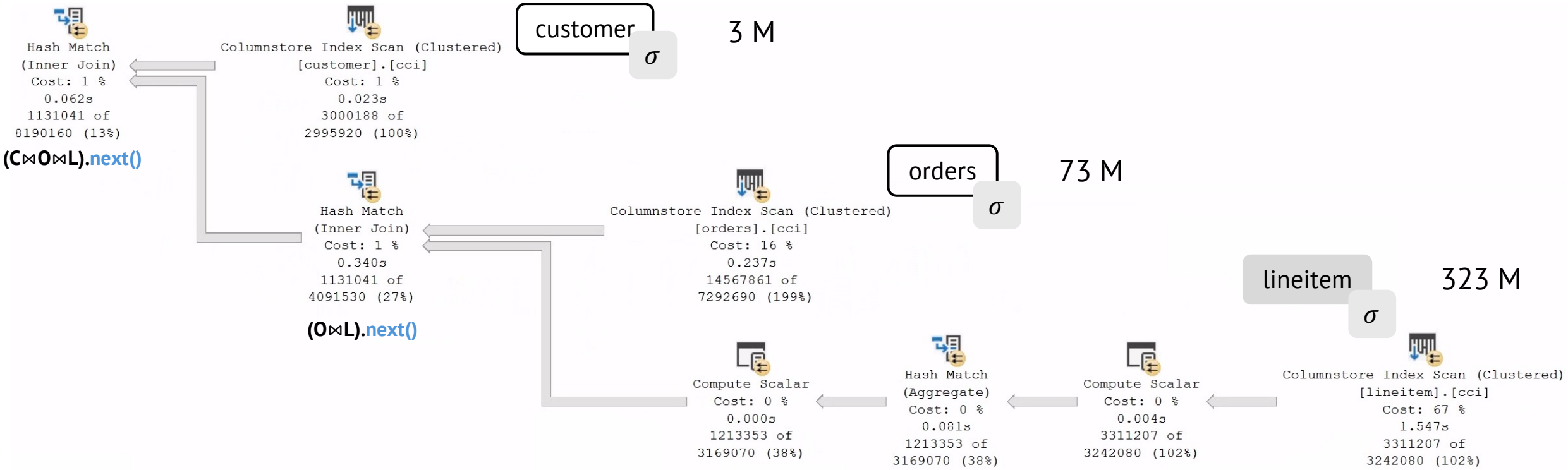


Batch-mode Hash Join
the only building block needed

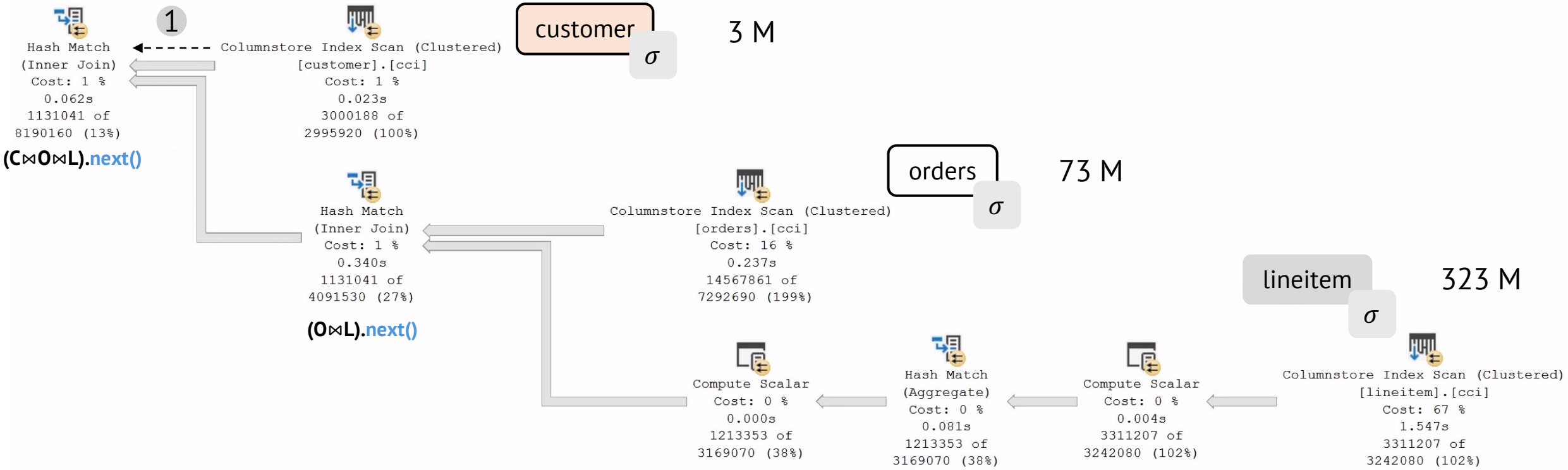
Pull-based Execution
cascading bitmap pushdown

Cascades Optimizer
cost-based optimizations
considering bitmap filters

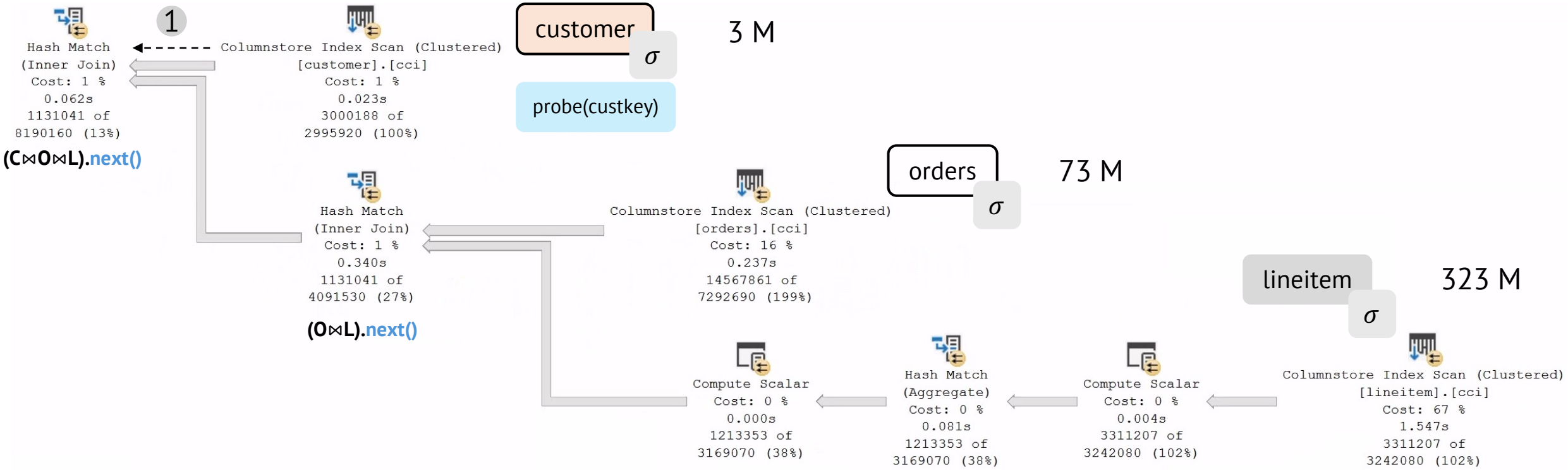
Composing Hash Join



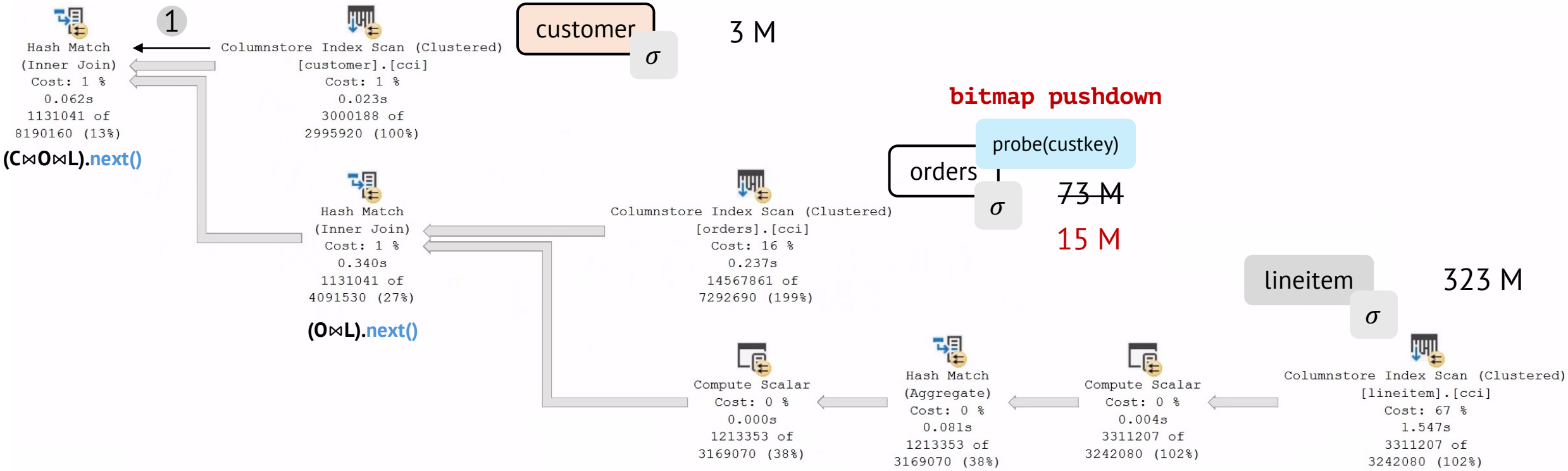
Composing Hash Join



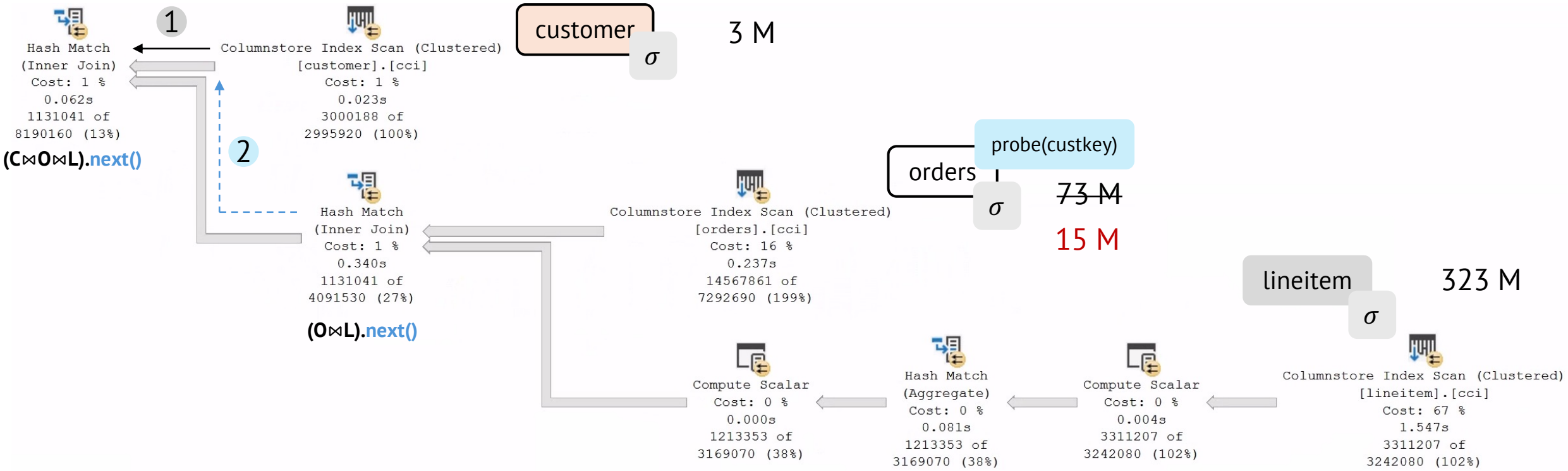
Composing Hash Join



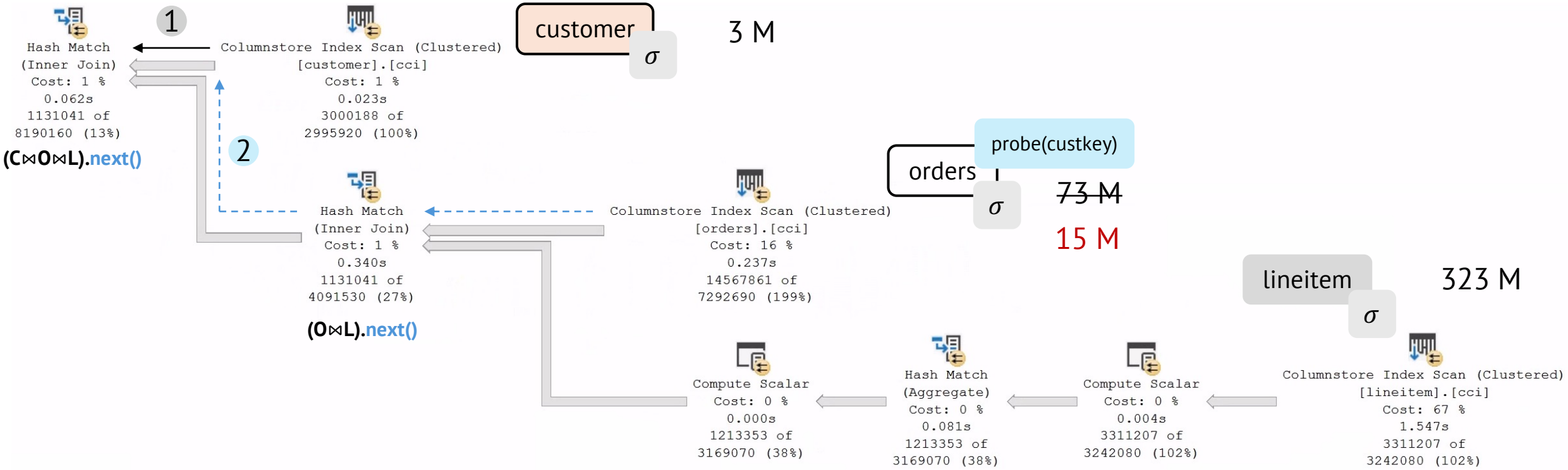
Composing Hash Join



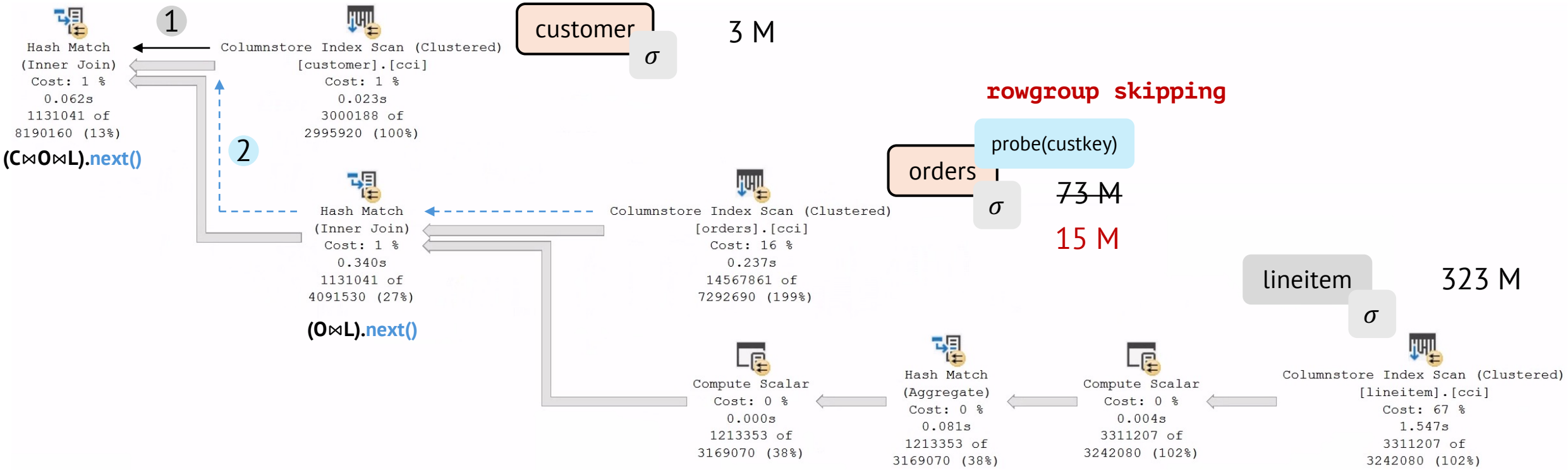
Composing Hash Join



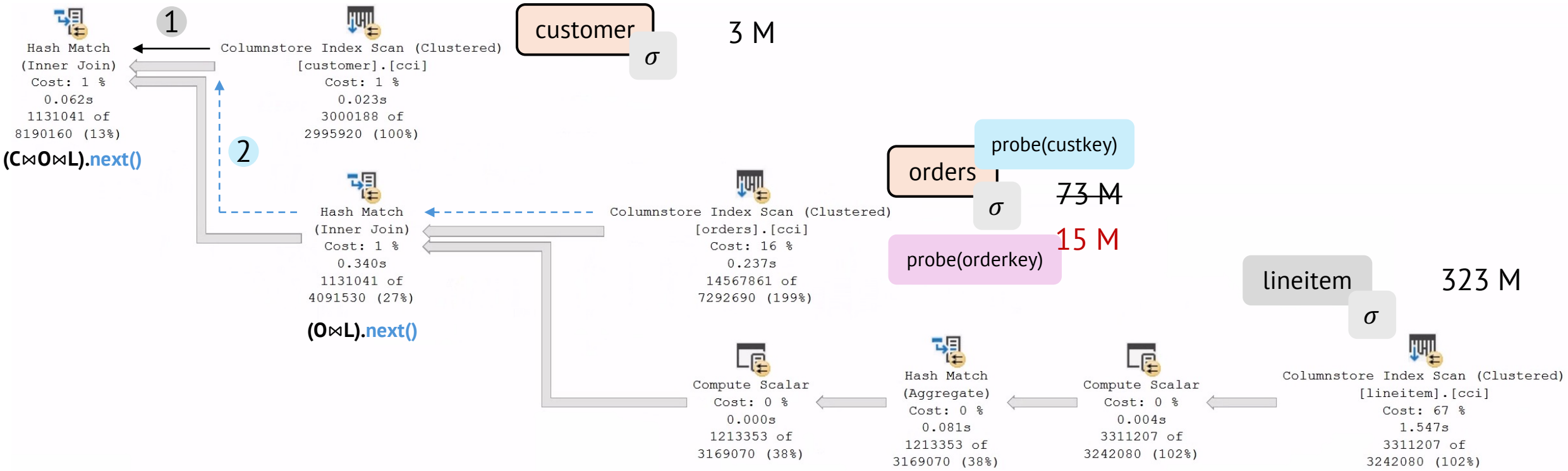
Composing Hash Join



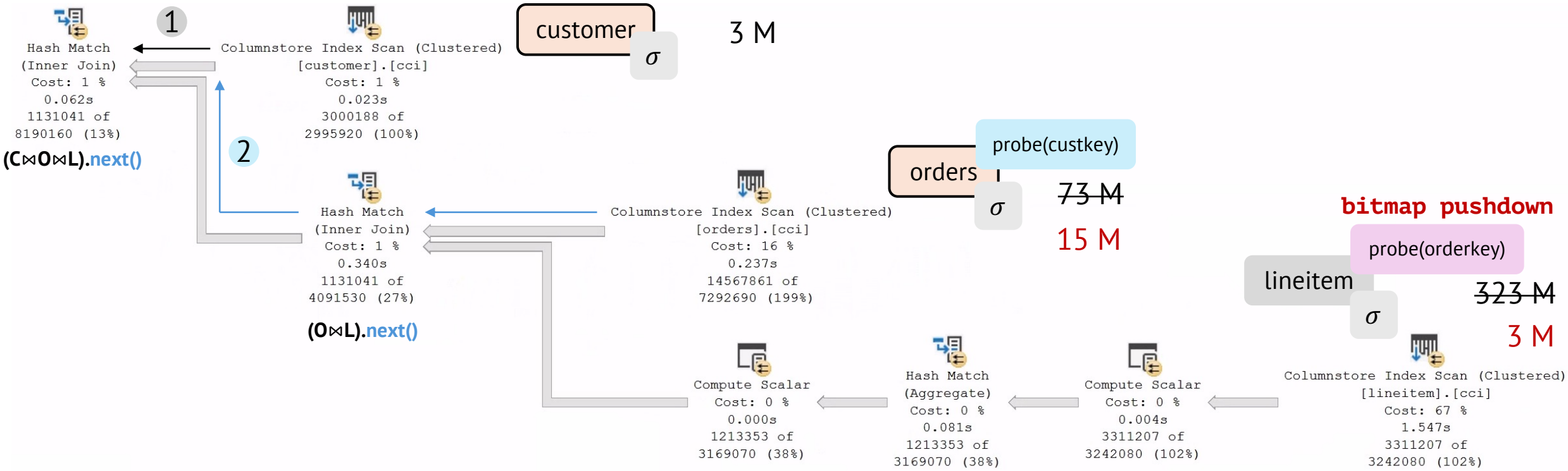
Composing Hash Join



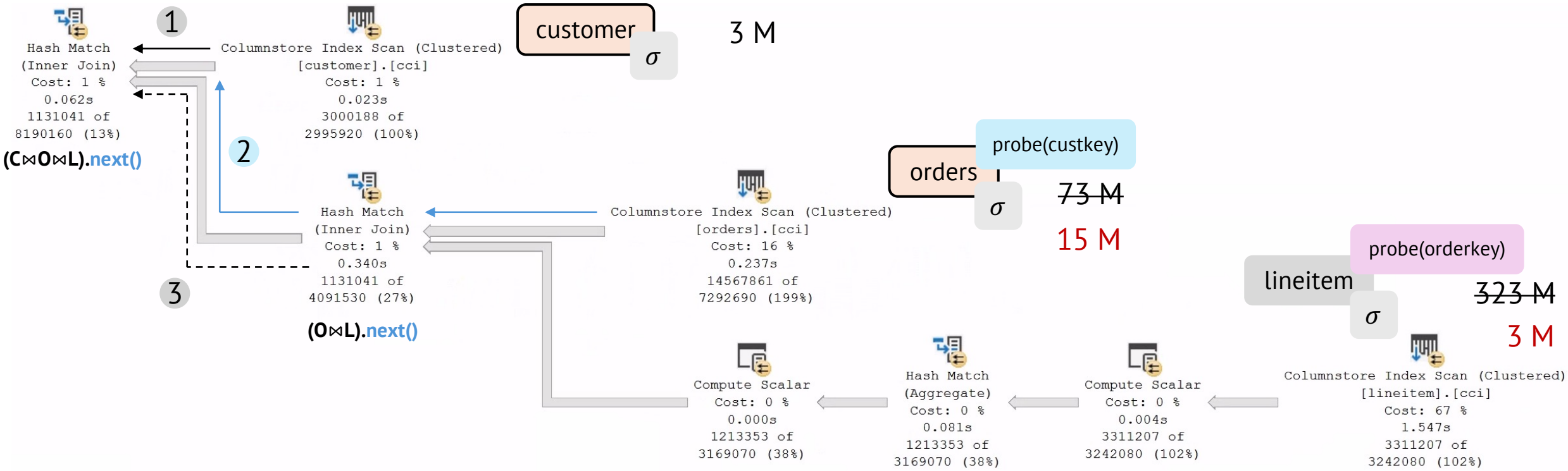
Composing Hash Join



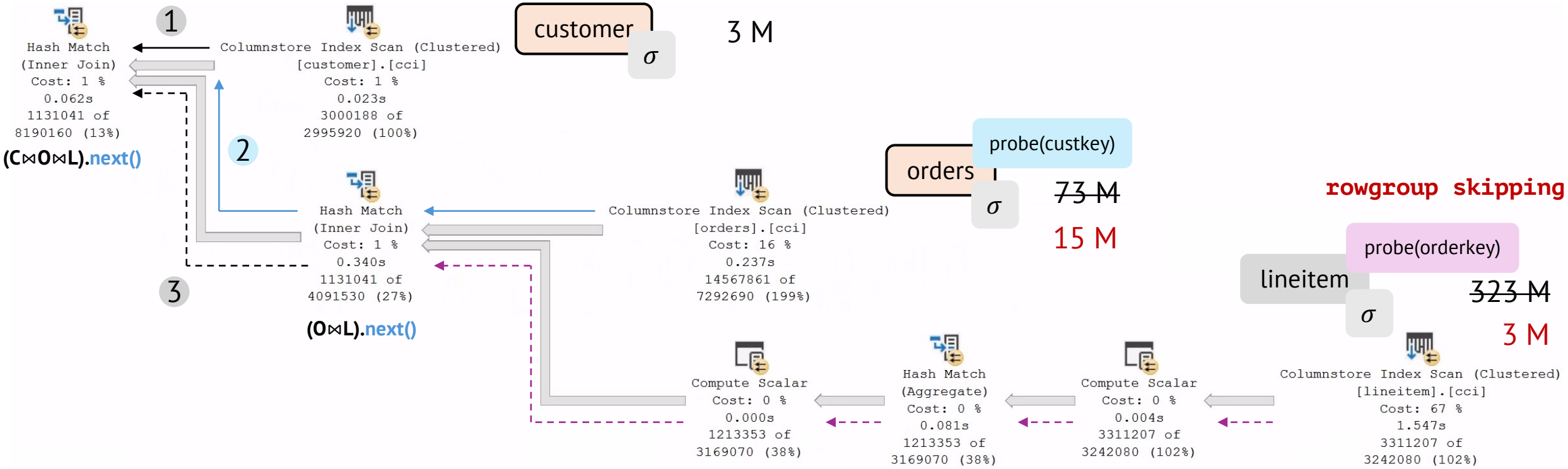
Composing Hash Join



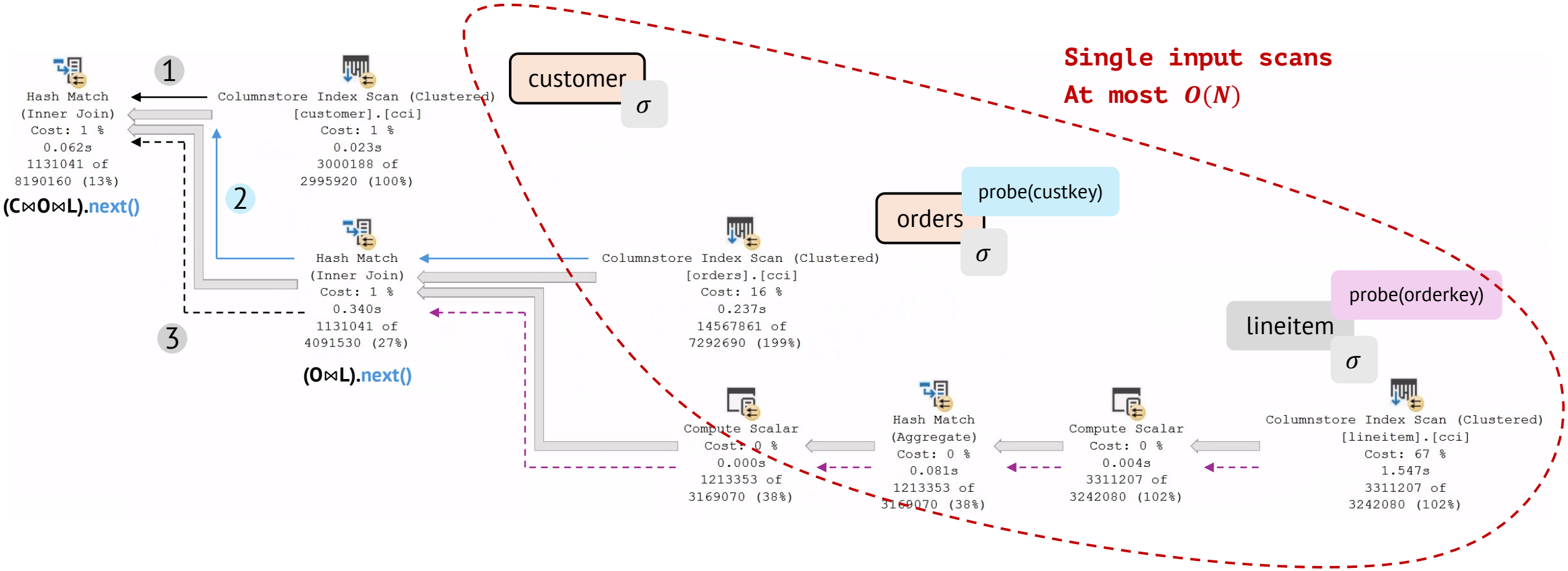
Composing Hash Join



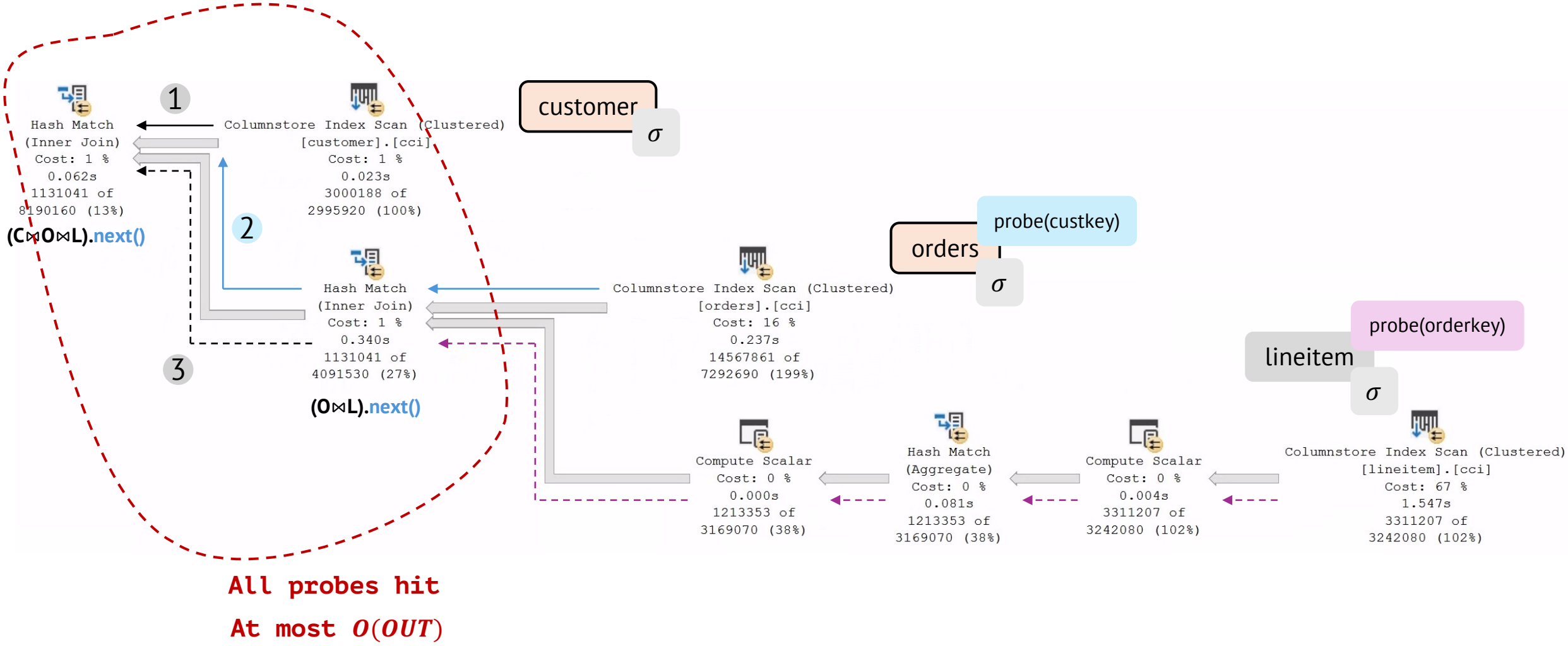
Composing Hash Join



Instance-optimal Query Execution

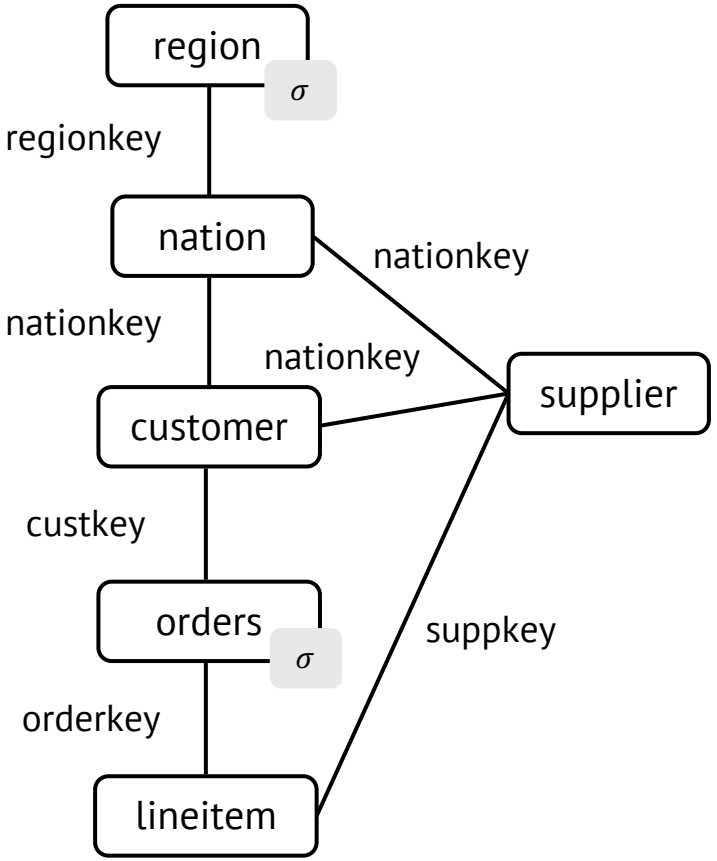


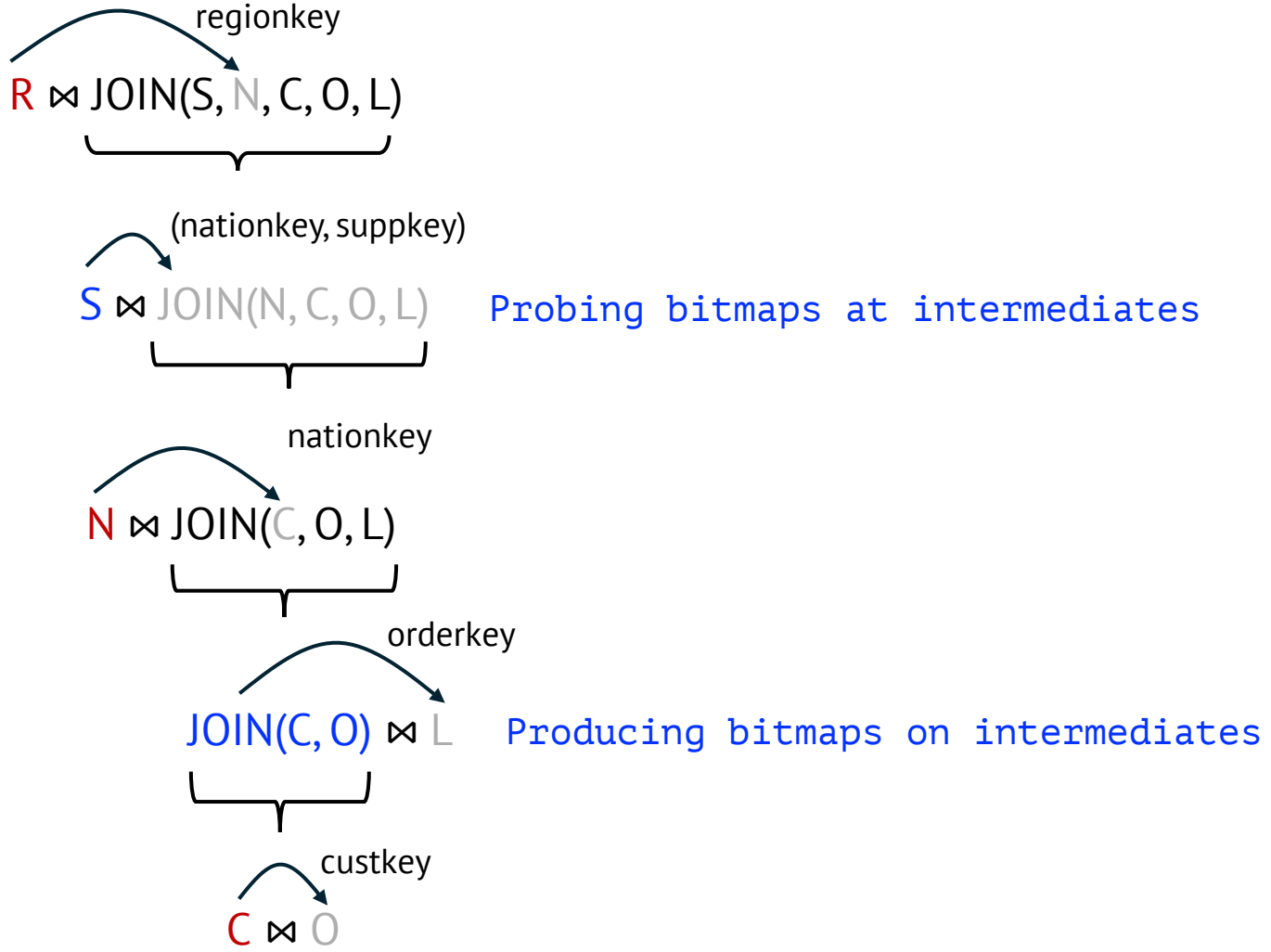
Instance-optimal Query Execution



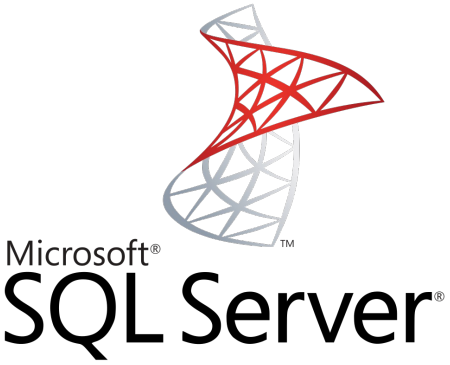
Example: TPC-H Q5 100 GB

Pre-filtering for Arbitrary Queries





SQL Server's Approach



Batch-mode Hash Join
the only building block needed

Pull-based Execution
cascading bitmap pushdown

Cascades Optimizer
cost-based optimizations
considering bitmap filters

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.

Thank you

- Formal proof of instance optimality
- Costing decisions to trade-off optimal plans
- Robustness guarantees that come as by-product
- And more…

I Can't Believe It's Not Yannakakis: Pragmatic Bitmap Filters in Microsoft SQL Server

Hangdong Zhao
Gray Systems Lab, Microsoft
USA

Yuanyuan Tian
Gray Systems Lab, Microsoft
USA

Rana Alotaibi
KACST
Saudi Arabia

Bailu Ding
Microsoft Research
USA

Nicolas Bruno
Gray Systems Lab, Microsoft
USA

Jesús Camacho-Rodríguez
SQL DW, Microsoft
USA

Vassilis Papadimos
SQL Server, Microsoft
USA

Ernesto Cervantes Juárez
SQL DW, Microsoft
USA

Cesar Galindo-Legaria
SQL DW, Microsoft
USA

Carlo Curino
Gray Systems Lab, Microsoft
USA

ABSTRACT

The quest for optimal join processing has reignited interest in the Yannakakis algorithm, as researchers seek to realize its theoretical ideal in practice via bitmap filters instead of expensive semijoins. While this academic pursuit may seem distant from industrial practice, our investigation into production databases led to a startling discovery: over the last decade, Microsoft SQL Server has built an infrastructure for bitmap pre-filtering that subsumes the very spirit of Yannakakis! This is not a story of academia leading industry; but rather of industry practice, guided by pragmatic optimization, outpacing academic endeavors. This paper dissects this discovery. As a crucial contribution, we prove how SQL Server's bitmap filters, pull-based execution, and Cascades optimizer conspire to not only consider, but often generate, instance-optimal plans, when it truly minimizes the estimated cost! Moreover, its rich plan search space reveals novel, largely overlooked pre-filtering opportunities on intermediate results, which approach strong semi-robust runtime for arbitrary join graphs. Instead of a verdict, this paper is an invitation: by exposing a system design that is long-hidden, we point our community towards a challenging yet promising research terrain.

1 INTRODUCTION

In 1981, Yannakakis proposed a seminal join algorithm [39] (see Sec. 2.1) that, in its essence, established for any acyclic join query, a set of *instance-optimal* query plans, for which no asymptotically faster alternatives can be hoped for. Despite this breakthrough, the algorithm has been largely sidelined by database engines as it bears an additional (costly) semijoin phase before any join execution.

A new renaissance has recently begun. When researchers [38] came to the realization that, similar to Bloom joins [18, 45], bitmap (or Bloom) filters can achieve near-equivalent, but much cheaper, pre-filtering as opposed to semijoins, the strong promise of Yannakakis re-gained appeal. A surge of academic pursuits (see Sec. 2.2) have since sought to retrofit its theoretical guarantees into modern databases via bitmap filters, showing they can not only accelerate joins, but also mitigate the negative impact of poor join orders [44].

Our mission was set to bridge the gap between recent academic insights and production engines. To this purpose, we dived into SQL Server, on which bitmap filters have been deployed since 2012. Our investigation led to a surprising discovery: **SQL Server, through a decade of continual refinement, has already captured Yannakakis as part of its design!** More shockingly, SQL Server unveils in a cost-based framework novel pre-filtering opportunities that were largely overlooked in the existing work.

Given the widespread adoption of Bloom filters and semijoins in industrial engines [1, 2, 22], the "Yannakakis effect" may potentially appear in other modern databases incorporating bitmap filters. This paper, however, presents a first formal analysis of this phenomenon, centered around its concrete implementation in SQL Server.

We show how SQL Server elegantly blends bitmap filters into a pull-based execution (Sec. 3). Its hash join creates bitmaps alongside hash tables and pushes them into the probe-side subplan to drop mismatches early. In its pull-based execution, chaining such hash joins naturally cascades bitmaps through multi-joins before recursive `next()` calls for hash probing. We prove such execution paradigm being instance-optimal, meaning that **SQL Server implicitly considers all Yannakakis-style plans for acyclic joins!**

Since full instance-optimality is provably impossible for arbitrary join queries, we propose a generalized notion of performance *semi-robustness* (Sec. 3.4), where runtime depends only on a small number of critical intermediates, rather than on every intermediate result. By considering bitmap filters to be built from and applied to intermediate join results, a capability largely absent in prior pre-filtering work, we show that **SQL Server's execution model can achieve strong semi-robust guarantees even on cyclic joins**.

While recent works often treat bitmap filters as a heuristic add-on, **SQL Server considers them directly in the costing framework of its Cascades optimizer** [13, 20] (Sec. 4). During planning,

16th Annual Conference on Innovative Data Systems Research (CIDR '26). January 18-21, Chaminade, USA