Lecture #21

Carnegie Mellon University

ADVANCED DATABASE SYSTEMS

Optimizer Implementation (Part III)

@Andy_Pavlo // 15-721 // Spring 2020

OBSERVATION

The best plan for a query can change as the database evolves over time.

→ Physical design changes.
→ Data modifications.
→ Prepared statement parameters.
→ Statistics updates.

The query optimizers that we have talked about so far all generate a plan for a query <u>before</u> the DBMS executes a query.

BAD QUERY PLANS

The most common problem in a query plan is incorrect join orderings.

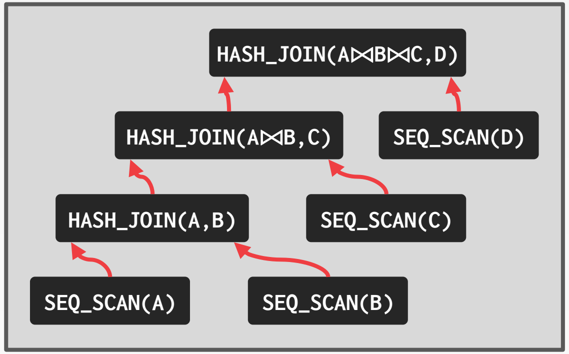
→ This occurs because of inaccurate cardinality estimations that propagate up the plan.

If the DBMS can detect how bad a query plan is, then it can decide to <u>adapt</u> the plan rather than continuing with the current sub-optimal plan.

BAD QUERY PLANS

```
SELECT * FROM A
  JOIN B ON A.id = B.id
  JOIN C ON A.id = C.id
  JOIN D ON A.id = D.id
 WHERE B.val = 'WuTang'
   AND D.val = 'Clan';
```

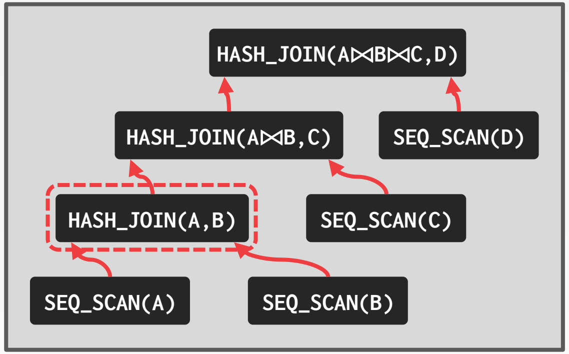
Original Plan



BAD QUERY PLANS

```
SELECT * FROM A
  JOIN B ON A.id = B.id
  JOIN C ON A.id = C.id
  JOIN D ON A.id = D.id
 WHERE B.val = 'WuTang'
   AND D.val = 'Clan';
```

Original Plan

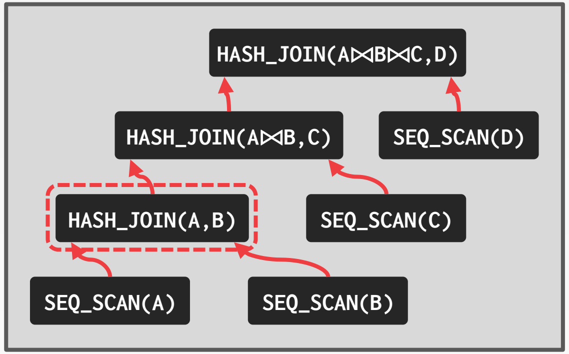


Estimated Cardinality: 1000
Actual Cardinality: 100000

BAD QUERY PLANS

```
SELECT * FROM A
  JOIN B ON A.id = B.id
  JOIN C ON A.id = C.id
  JOIN D ON A.id = D.id
 WHERE B.val = 'WuTang'
   AND D.val = 'Clan';
```

Original Plan



Estimated Cardinality: 1000
Actual Cardinality: 100000

If the optimizer knew the true cardinality, would it have picked the same the join ordering, join algorithms, or access methods?

WHY GOOD PLANS GO BAD

Estimating the execution behavior of a plan to determine its quality relative to other plans.

These estimations are based on a static summarizations of the contents of the database and its operating environment:

→ Statistical Models / Histograms / Sampling
→ Hardware Performance
→ Concurrent Operations

ADAPTIVE QUERY OPTIMIZATION

Modify the execution behavior of a query by generating multiple plans for it:

→ Individual complete plans.
→ Embed multiple sub-plans at materialization points.

Use information collected during query execution to improve the quality of these plans.

→ Can use this data for planning one query or merge the it back into the DBMS's statistics catalog.

ADAPTIVE QUERY PROCESSING IN THE LOOKING GLASS
CIDR 2005

ADAPTIVE QUERY OPTIMIZATION

Approach #1: Modify Future Invocations

Approach #2: Replan Current Invocation

Approach #3: Plan Pivot Points

MODIFY FUTURE INVOCATIONS

The DBMS monitors the behavior of a query during execution and uses this information to improve subsequent invocations.

Approach #1: Plan Correction

Approach #2: Feedback Loop

REVERSION-BASED PLAN CORRECTION

The DBMS tracks the history of query invocations:

→ Cost Estimations
→ Query Plan
→ Runtime Metrics

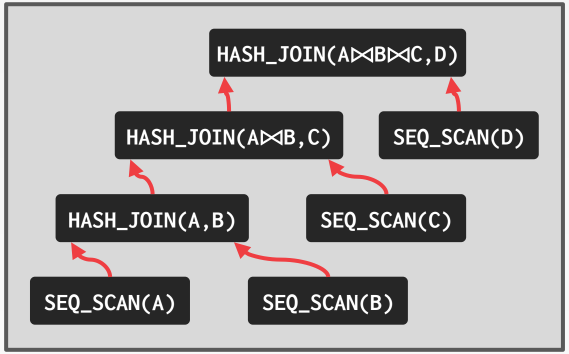
If the DBMS generates a new plan for a query, then check whether that plan performs worse than the previous plan.

→ If it regresses, then switch back to the cheaper plans.

REVERSION-BASED PLAN CORRECTION

```
SELECT * FROM A
  JOIN B ON A.id = B.id
  JOIN C ON A.id = C.id
  JOIN D ON A.id = D.id
 WHERE B.val = 'WuTang'
   AND D.val = 'Clan';
```

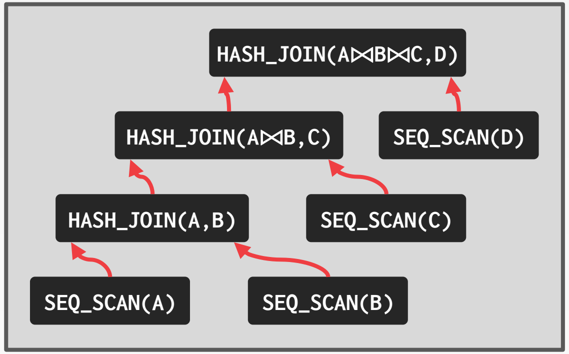
Original Plan



REVERSION-BASED PLAN CORRECTION

```
SELECT * FROM A
  JOIN B ON A.id = B.id
  JOIN C ON A.id = C.id
  JOIN D ON A.id = D.id
 WHERE B.val = 'WuTang'
   AND D.val = 'Clan';
```

Original Plan



Estimated Cost: 1000

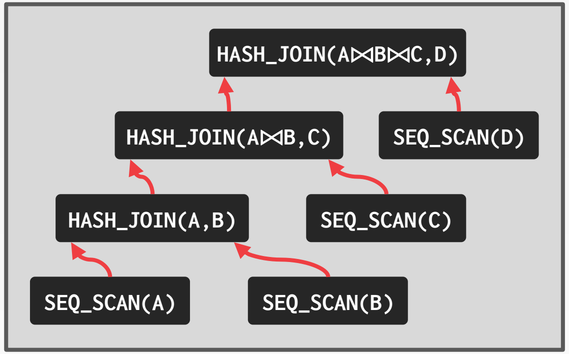
Actual Cost: 1000

Execution History

REVERSION-BASED PLAN CORRECTION

```
SELECT * FROM A
  JOIN B ON A.id = B.id
  JOIN C ON A.id = C.id
  JOIN D ON A.id = D.id
 WHERE B.val = 'WuTang'
   AND D.val = 'Clan';
```

Original Plan



Estimated Cost: 1000

Actual Cost: 1000

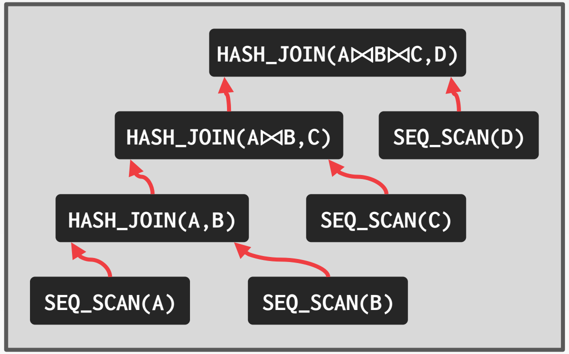
```
CREATE INDEX idx_b_val ON B (val);
CREATE INDEX idx_d_val ON D (val);
```

Execution History

REVERSION-BASED PLAN CORRECTION

```
SELECT * FROM A
  JOIN B ON A.id = B.id
  JOIN C ON A.id = C.id
  JOIN D ON A.id = D.id
 WHERE B.val = 'WuTang'
   AND D.val = 'Clan';
```

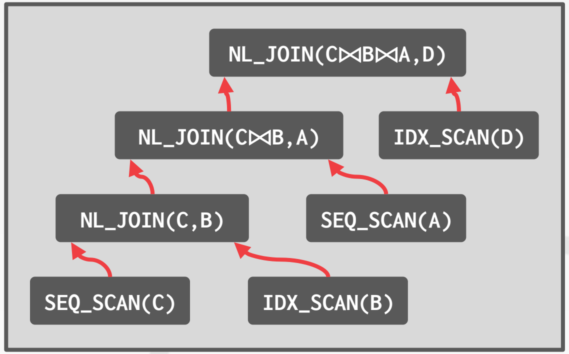
Original Plan



Estimated Cost: 1000

Actual Cost: 1000

New Plan



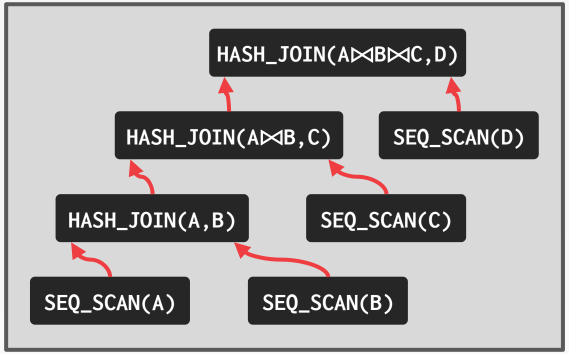

```
CREATE INDEX idx_b_val ON B (val);
CREATE INDEX idx_d_val ON D (val);
```

Execution History

REVERSION-BASED PLAN CORRECTION

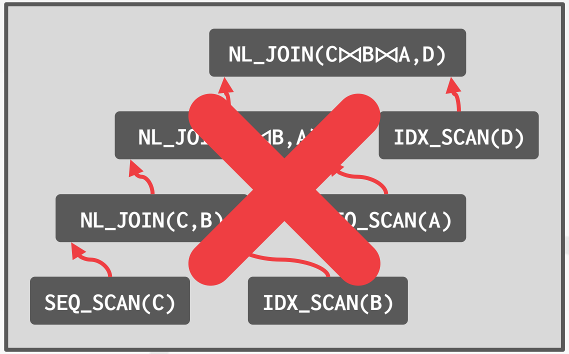
```
SELECT * FROM A
  JOIN B ON A.id = B.id
  JOIN C ON A.id = C.id
  JOIN D ON A.id = D.id
 WHERE B.val = 'WuTang'
   AND D.val = 'Clan';
```

Original Plan



Estimated Cost: 1000
Actual Cost: 1000

New Plan



Estimated Cost: 800
Actual Cost: 1200


```
CREATE INDEX idx_b_val ON B (val);
CREATE INDEX idx_d_val ON D (val);
```

MICROSOFT – PLAN STITCHING

Combine useful sub-plans from queries to create potentially better plans.

→ Sub-plans do not need to be from the same query.

→ Can still use sub-plans even if overall plan becomes invalid after a physical design change.

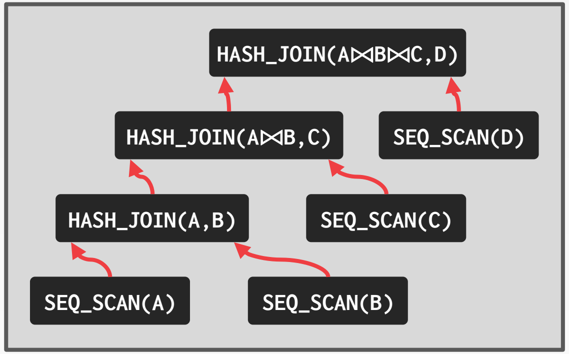
Uses a dynamic programming search (bottom-up) that is not guaranteed to find a better plan.

PLAN STITCH: HARNESSING THE BEST OF MANY PLANS
VLDB 2018

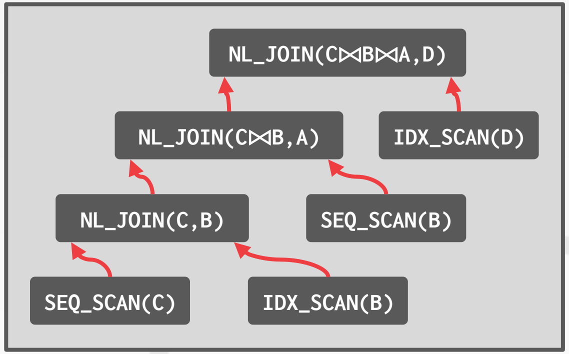
MICROSOFT – PLAN STITCHING

```sql
SELECT * FROM A
  JOIN B ON A.id = B.id
  JOIN C ON A.id = C.id
  JOIN D ON A.id = D.id
 WHERE B.val = 'WuTang'
   AND D.val = 'Clan';
```

Original Plan



New Plan

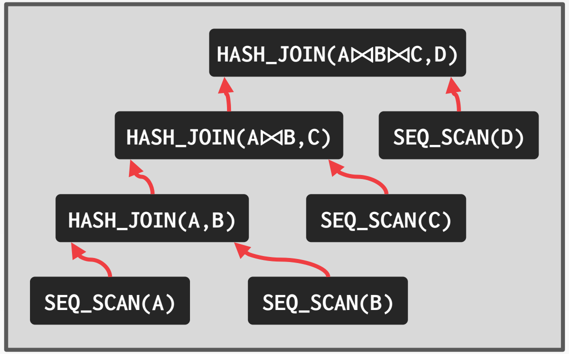



```sql
CREATE INDEX idx_b_val ON B (val);
CREATE INDEX idx_d_val ON D (val);
```

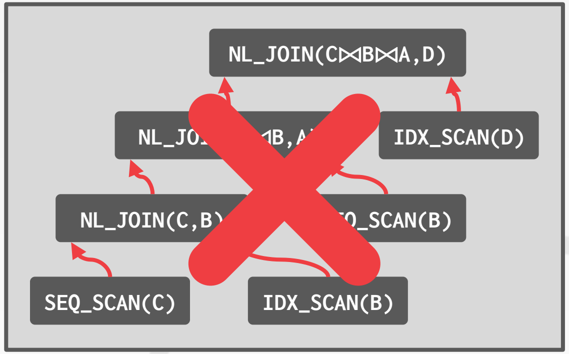
MICROSOFT – PLAN STITCHING

```sql
SELECT * FROM A
  JOIN B ON A.id = B.id
  JOIN C ON A.id = C.id
  JOIN D ON A.id = D.id
 WHERE B.val = 'WuTang'
   AND D.val = 'Clan';
```

Original Plan



New Plan



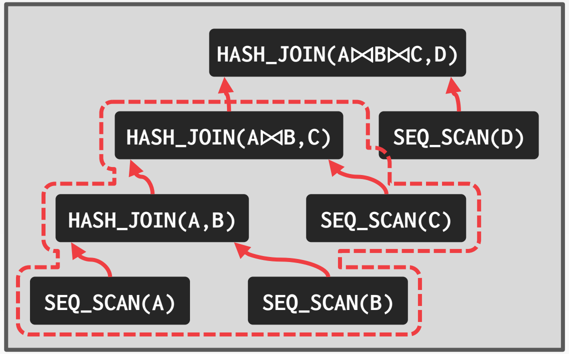

```sql
CREATE INDEX idx_b_val ON B (val);
CREATE INDEX idx_d_val ON D (val);
```

```sql
DROP INDEX idx_b_val;
```

MICROSOFT – PLAN STITCHING

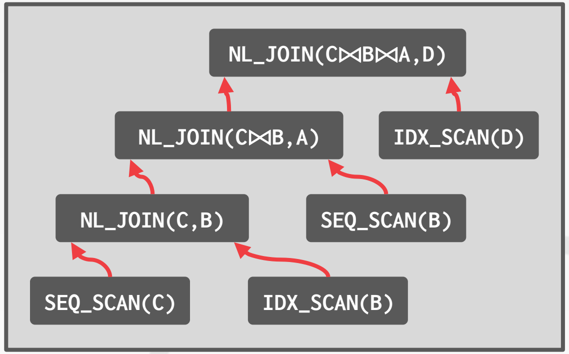
```
SELECT * FROM A
  JOIN B ON A.id = B.id
  JOIN C ON A.id = C.id
  JOIN D ON A.id = D.id
 WHERE B.val = 'WuTang'
   AND D.val = 'Clan';
```

Original Plan



Sub-Plan Cost: 600

New Plan



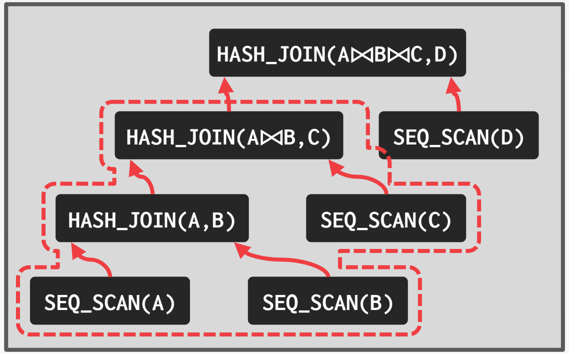

```
CREATE INDEX idx_b_val ON B (val);
CREATE INDEX idx_d_val ON D (val);
```

```
DROP INDEX idx_b_val;
```

MICROSOFT – PLAN STITCHING

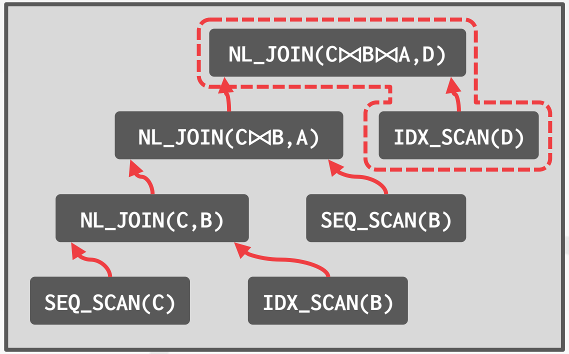
```
SELECT * FROM A
  JOIN B ON A.id = B.id
  JOIN C ON A.id = C.id
  JOIN D ON A.id = D.id
 WHERE B.val = 'WuTang'
   AND D.val = 'Clan';
```

Original Plan



Sub-Plan Cost: 600

New Plan *Sub-Plan Cost: 150*




```
CREATE INDEX idx_b_val ON B (val);
CREATE INDEX idx_d_val ON D (val);
```

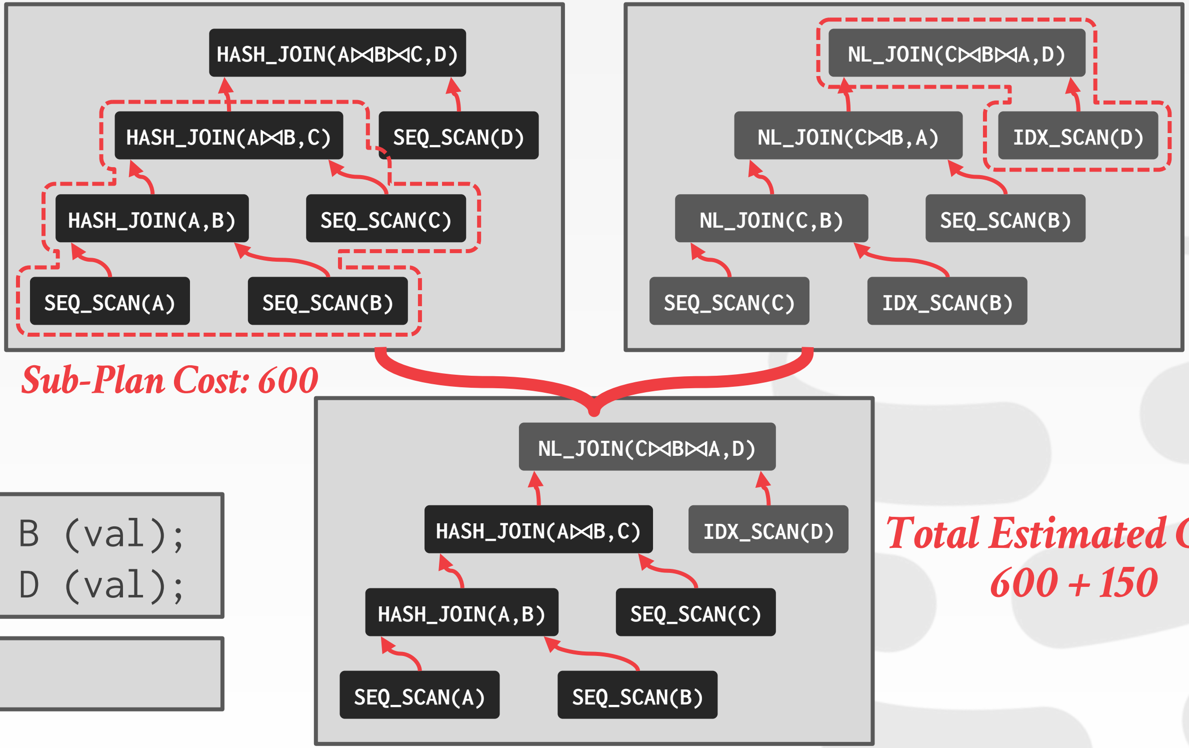
```
DROP INDEX idx_b_val;
```

MICROSOFT – PLAN STITCHING

```
SELECT * FROM A
  JOIN B ON A.id = B.id
  JOIN C ON A.id = C.id
  JOIN D ON A.id = D.id
 WHERE B.val = 'WuTang'
   AND D.val = 'Clan';
```

Original Plan

New Plan *Sub-Plan Cost: 150*



Total Estimated Cost: 600 + 150

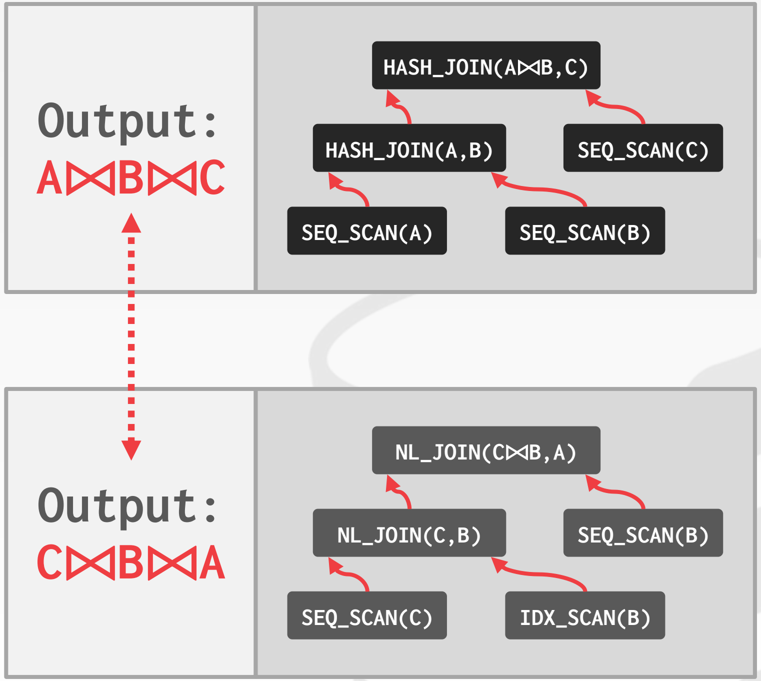
```
CREATE INDEX idx_b_val ON B (val);
CREATE INDEX idx_d_val ON D (val);
```

```
DROP INDEX idx_b_val;
```

IDENTIFYING EQUIVALENT SUBPLANS

Sub-plans are equivalent if they have the same logical expression and required physical properties.

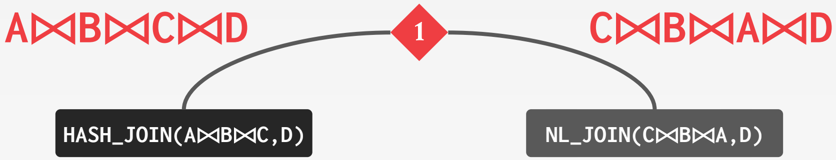
Use simple heuristic that prunes any subplans that never be equivalent (e.g., access different tables) and then matches based on comparing expression trees.



ENCODING SEARCH SPACE

Generate a graph that contains all possible sub-plans.

Add OR operators to indicate alternative paths through the plan.

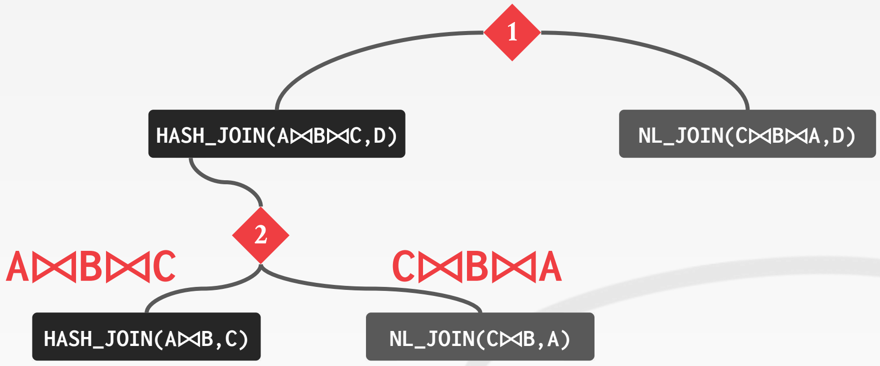


Source: Bailu Ding

ENCODING SEARCH SPACE

Generate a graph that contains all possible sub-plans.

Add OR operators to indicate alternative paths through the plan.

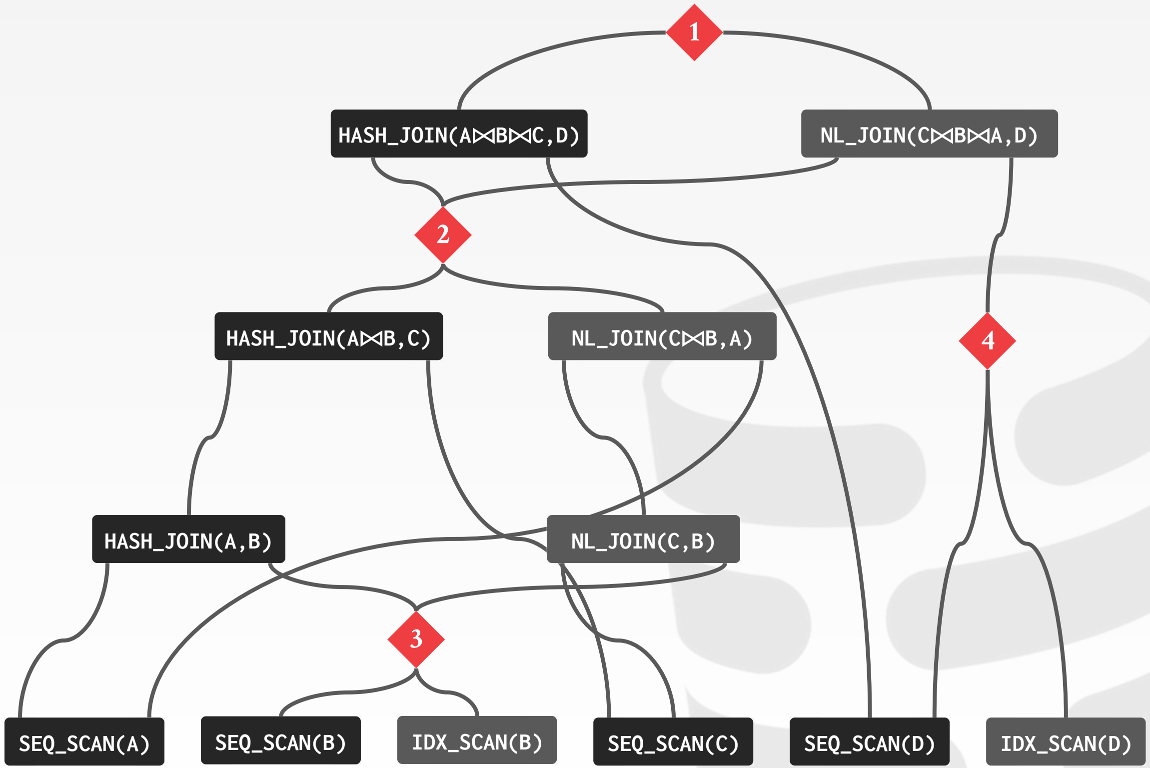


Source: Bailu Ding

ENCODING SEARCH SPACE

Generate a graph that contains all possible sub-plans.

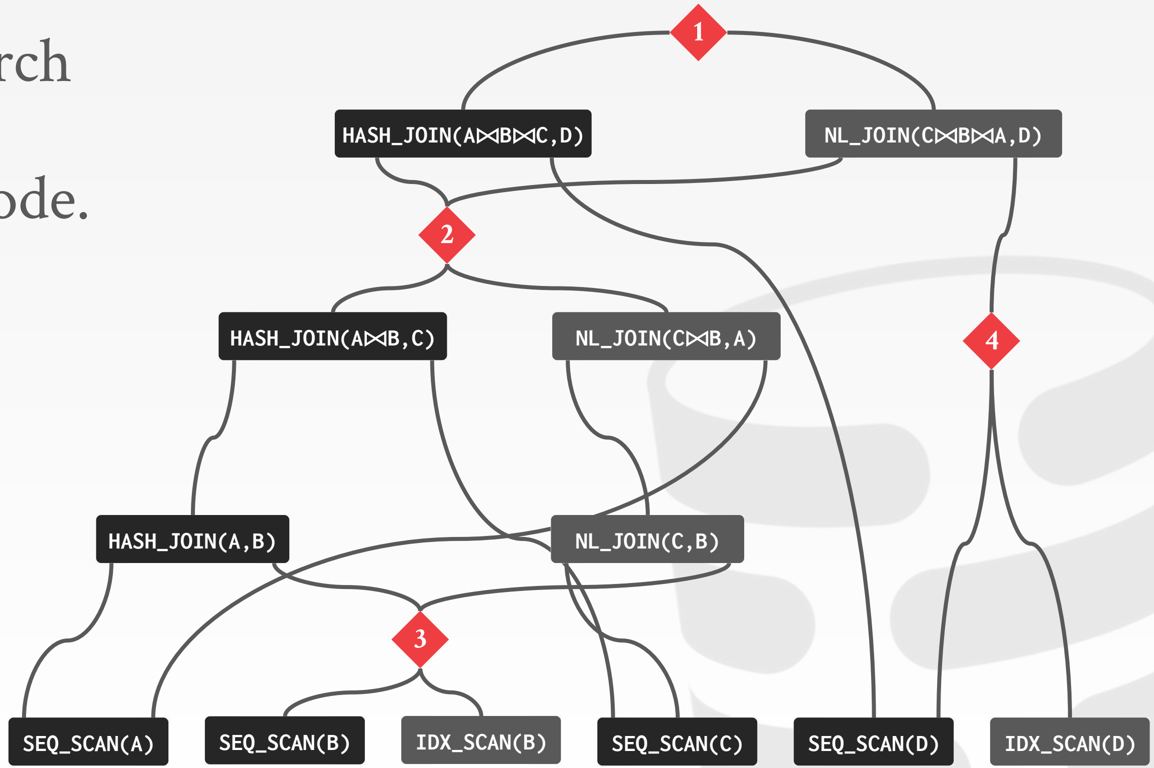
Add OR operators to indicate alternative paths through the plan.



Source: Bailu Ding

CONSTRUCTING STITCHED PLANS

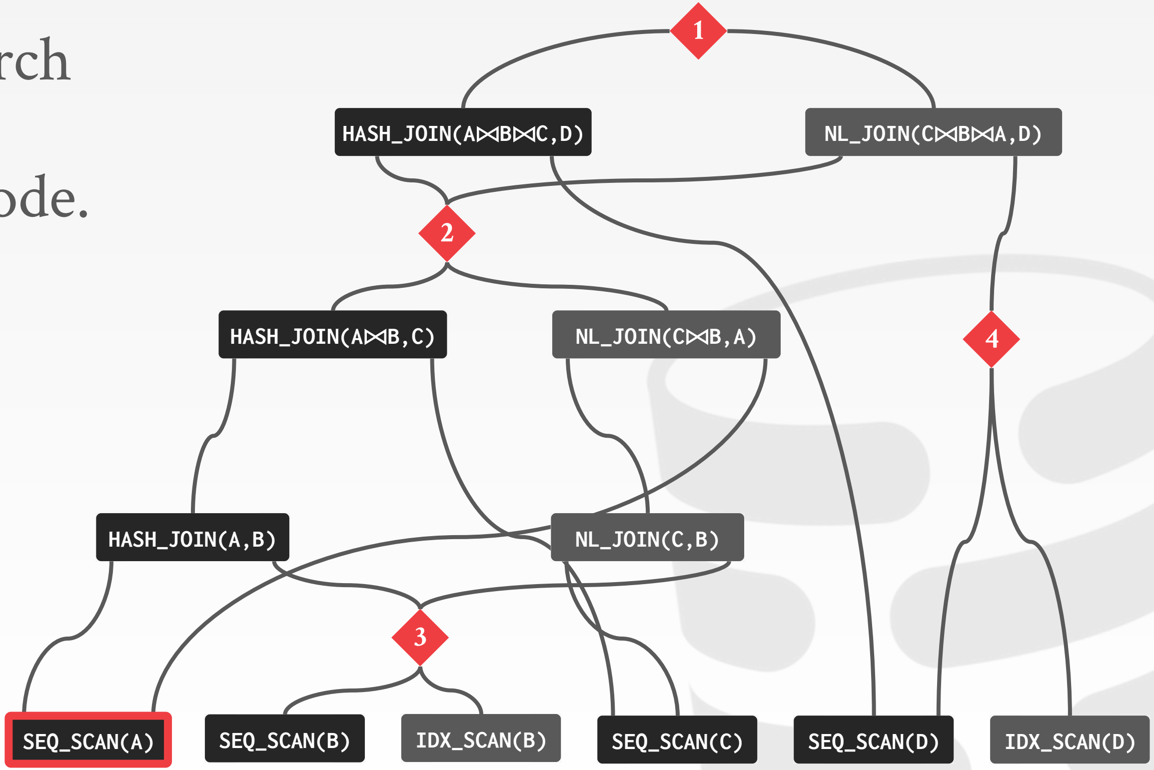
Perform bottom-up search that selects the cheapest sub-plan for each **OR** node.



Source: Bailu Ding

CONSTRUCTING STITCHED PLANS

Perform bottom-up search that selects the cheapest sub-plan for each **OR** node.



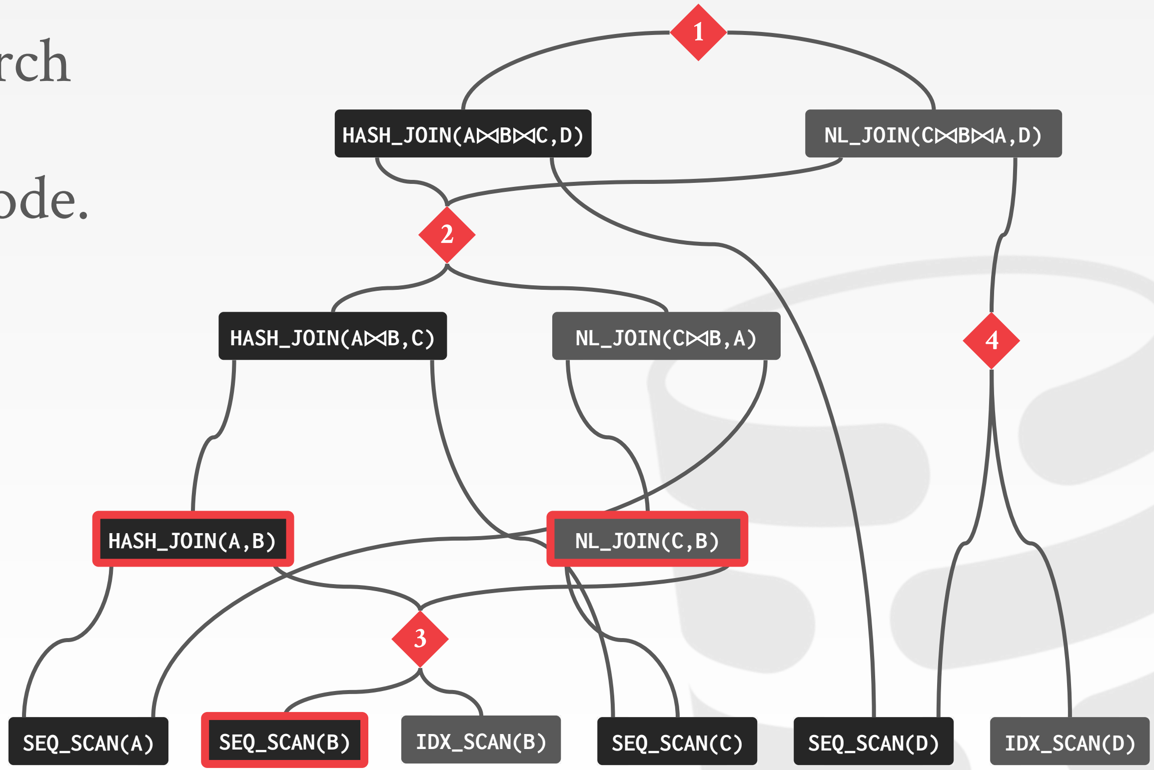
Source: Bailu Ding

CONSTRUCTING STITCHED PLANS

Perform bottom-up search that selects the cheapest sub-plan for each **OR** node.

SEQ_SCAN(A) → HASH_JOIN(A,B)

SEQ_SCAN(B) → HASH_JOIN(A,B)

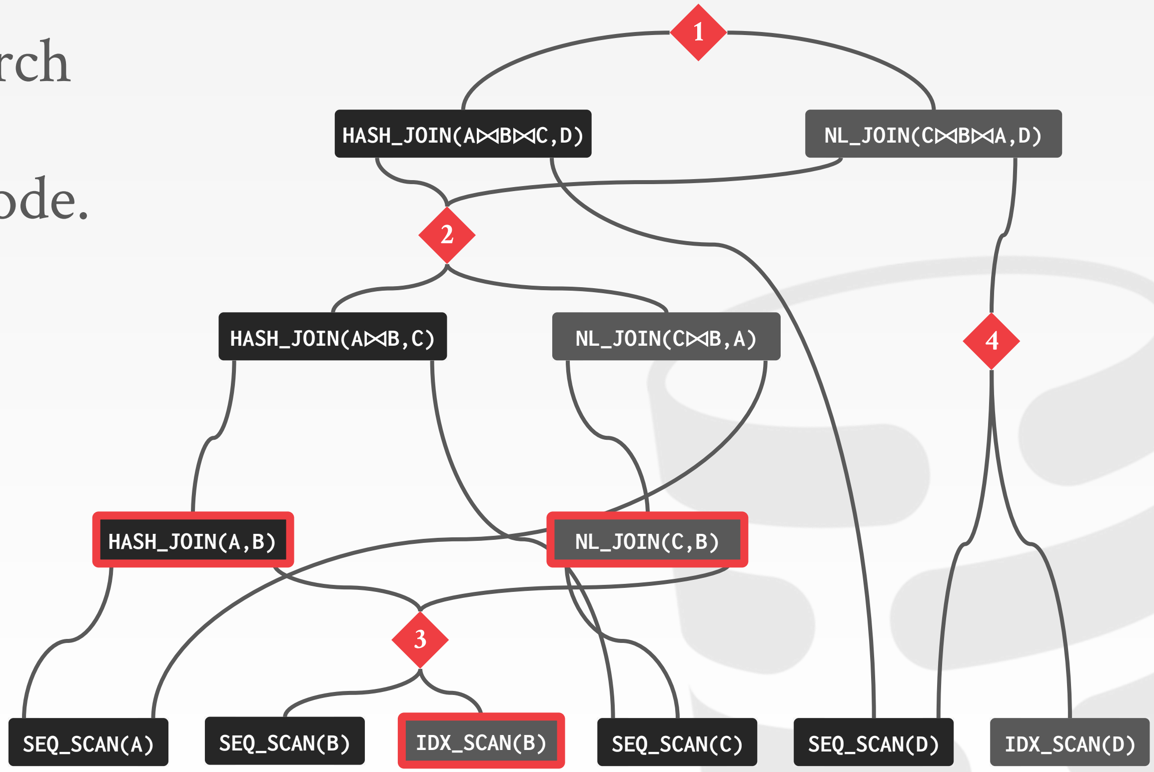


Source: Bailu Ding

CONSTRUCTING STITCHED PLANS

Perform bottom-up search that selects the cheapest sub-plan for each **OR** node.

SEQ_SCAN(A)	→	HASH_JOIN(A,B)
SEQ_SCAN(B)	→	HASH_JOIN(A,B)
IDX_SCAN(B)	→	NL_JOIN(C,B)

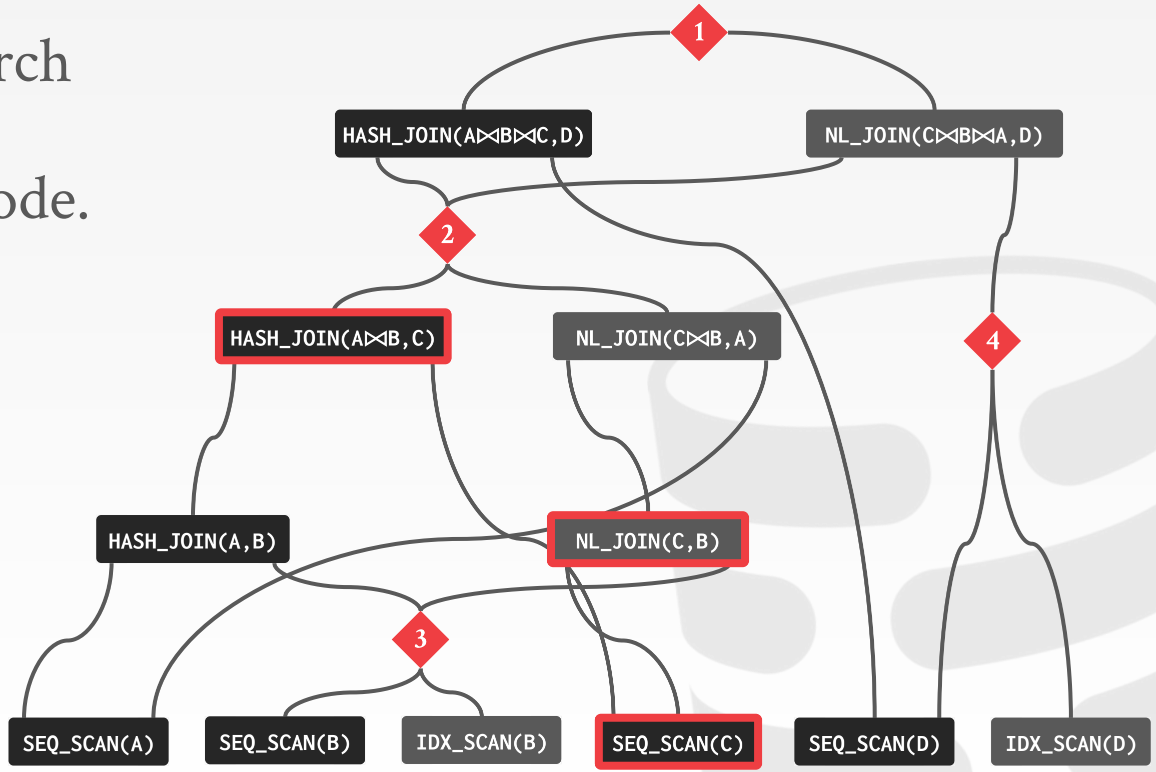


Source: Bailu Ding

CONSTRUCTING STITCHED PLANS

Perform bottom-up search that selects the cheapest sub-plan for each **OR** node.

SEQ_SCAN(A) → HASH_JOIN(A,B)
SEQ_SCAN(B) → HASH_JOIN(A,B)
IDX_SCAN(B) → NL_JOIN(C,B)
SEQ_SCAN(C) → HASH_JOIN(A⋈B,C)
⋮



Source: Bailu Ding

CONSTRUCTING STITCHED PLANS

Perform bottom-up search that selects the cheapest sub-plan for each **OR** node.

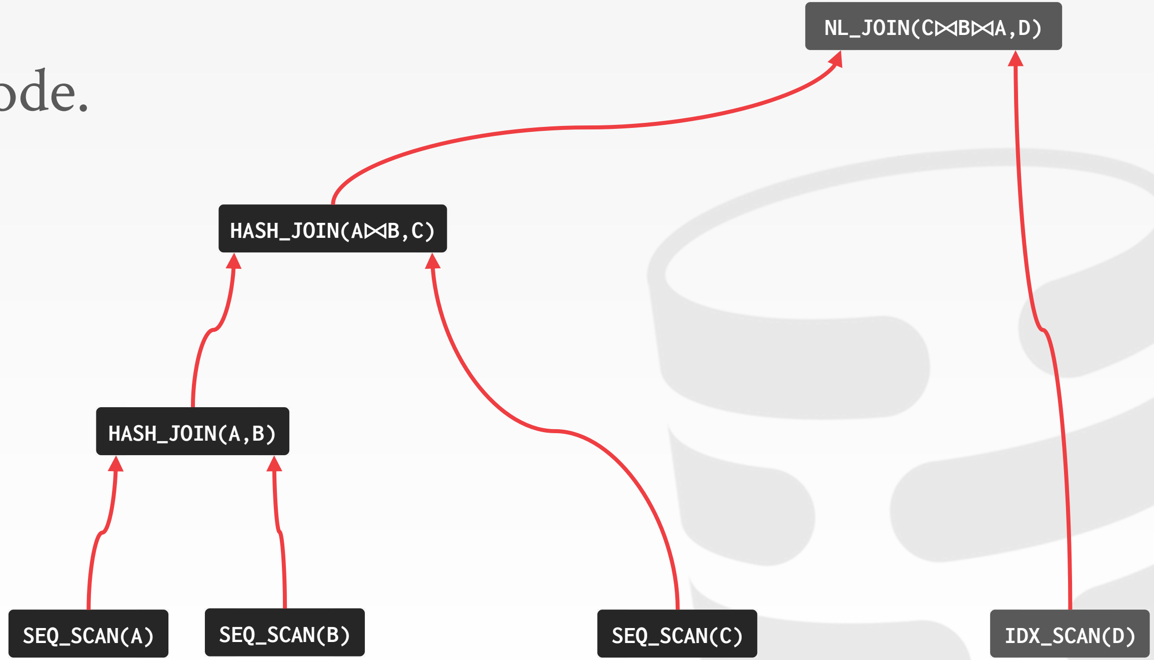
SEQ_SCAN(A) → HASH_JOIN(A,B)

SEQ_SCAN(B) → HASH_JOIN(A,B)

IDX_SCAN(B) → NL_JOIN(C,B)

SEQ_SCAN(C) → HASH_JOIN(A⋈B,C)

⋮



Source: [Bailu Ding]

REDSHIFT – CODEGEN STITCHING

```
SELECT * FROM A
  JOIN B ON A.id = B.id
  JOIN C ON A.id = C.id
  JOIN D ON A.id = D.id
 WHERE B.val = 'WuTang'
   AND D.val = 'Clan';
```

```
for t in scan(B):
  if t.val=$arg: emit(t)
```

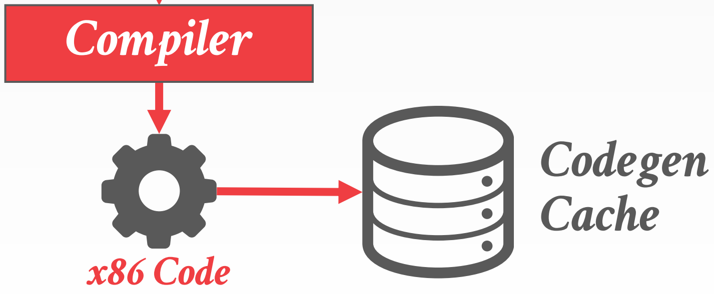
Redshift is a transpilation-based codegen engine.

To avoid the compilation cost for every query, the DBMS caches subplans and then combines them at runtime for new queries.

REDSHIFT – CODEGEN STITCHING

```
SELECT * FROM A
  JOIN B ON A.id = B.id
  JOIN C ON A.id = C.id
  JOIN D ON A.id = D.id
 WHERE B.val = 'WuTang'
   AND D.val = 'Clan';
```

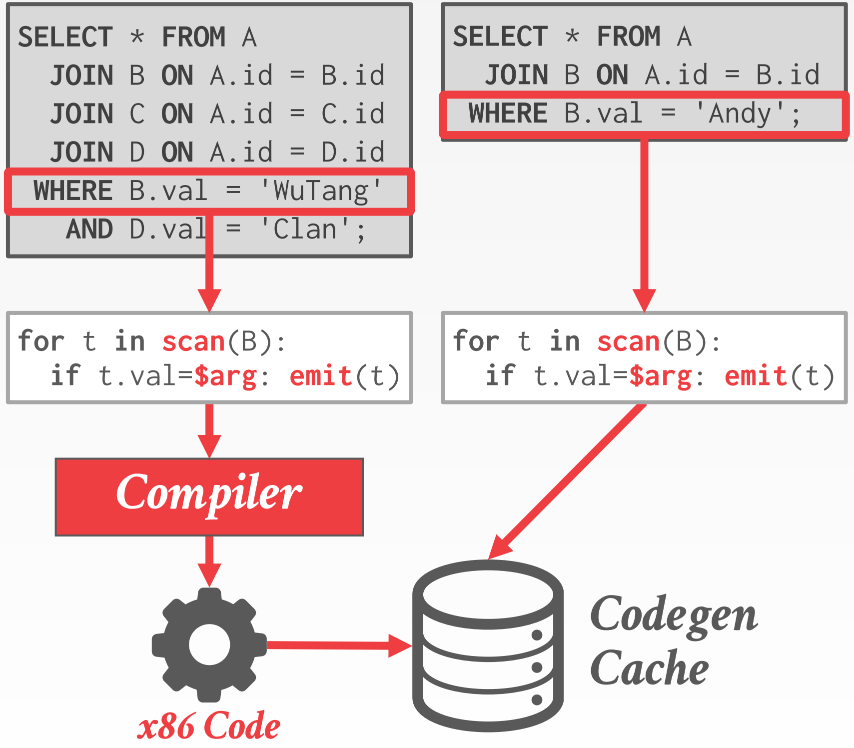
```
for t in scan(B):
  if t.val=$arg: emit(t)
```



Redshift is a transpilation-based codegen engine.

To avoid the compilation cost for every query, the DBMS caches subplans and then combines them at runtime for new queries.

REDSHIFT – CODEGEN STITCHING



Redshift is a transpilation-based codegen engine.

To avoid the compilation cost for every query, the DBMS caches subplans and then combines them at runtime for new queries.

IBM DB2 – LEARNING OPTIMIZER

Update table statistics as the DBMS scans a table during normal query processing.

Check whether the optimizer's estimates match what it encounters in the real data and incrementally updates them.

LEO - DB2'S LEARNING OPTIMIZER
VLDB 2001

REPLAN CURRENT INVOCATION

If the DBMS determines that the observed execution behavior of a plan is far from its estimated behavior, them it can halt execution and generate a new plan for the query.

Approach #1: Start-Over from Scratch

Approach #2: Keep Intermediate Results

QUICKSTEP – LOOKAHEAD INFO PASSING

```
CREATE TABLE fact (
  id INT PRIMARY KEY,
  dim1_id INT
   ↳REFERENCES dim1 (id),
  dim2_id INT,
   ↳REFERENCES dim2 (id)
);
```

```
CREATE TABLE dim1 (
  id INT, val VARCHAR
);
CREATE TABLE dim2 (
  id INT, val VARCHAR
);
```

First compute Bloom filters on dimension tables.

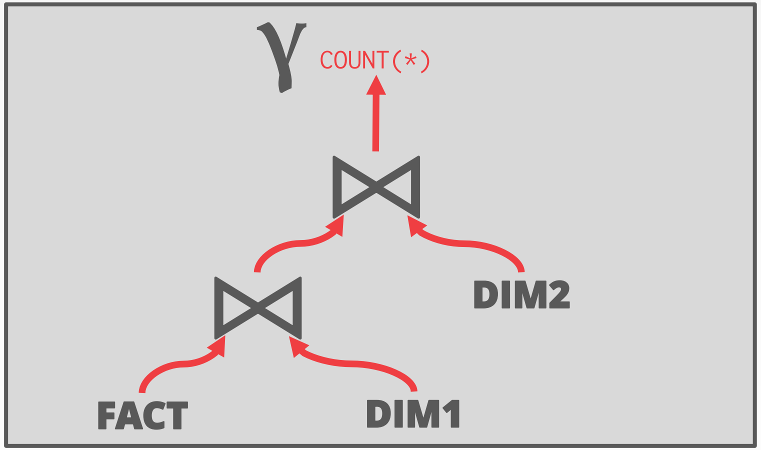
Probe these filters using fact table tuples to determine the ordering of the joins.

Only supports left-deep join trees on star schemas.

LOOKING AHEAD MAKES QUERY PLANS ROBUST
VLDB 2017

QUICKSTEP – LOOKAHEAD INFO PASSING

```sql
SELECT COUNT(*) FROM fact AS f
  JOIN dim1 ON f.dim1_id = dim1.id
  JOIN dim2 ON f.dim2_id = dim2.id
```



First compute Bloom filters on dimension tables.

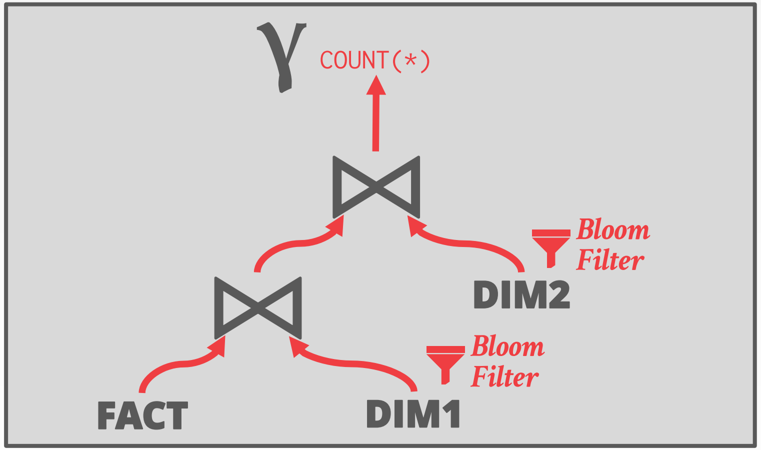
Probe these filters using fact table tuples to determine the ordering of the joins.

Only supports left-deep join trees on star schemas.

LOOKING AHEAD MAKES QUERY PLANS ROBUST
VLDB 2017

QUICKSTEP – LOOKAHEAD INFO PASSING

```
SELECT COUNT(*) FROM fact AS f
  JOIN dim1 ON f.dim1_id = dim1.id
  JOIN dim2 ON f.dim2_id = dim2.id
```



First compute Bloom filters on dimension tables.

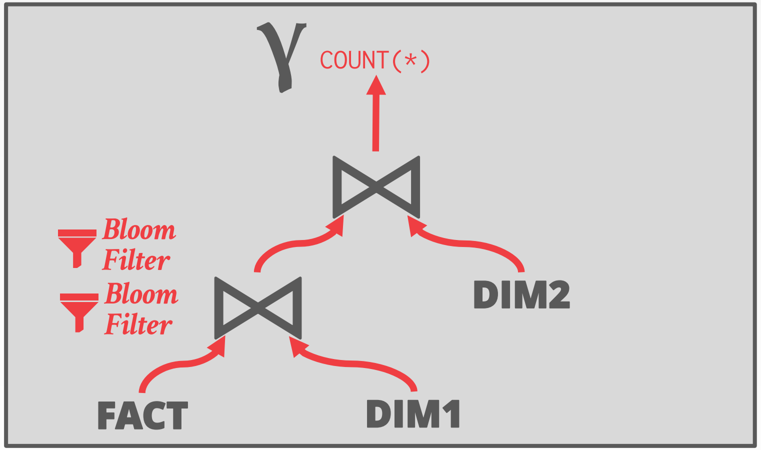
Probe these filters using fact table tuples to determine the ordering of the joins.

Only supports left-deep join trees on star schemas.

LOOKING AHEAD MAKES QUERY PLANS ROBUST
VLDB 2017

QUICKSTEP – LOOKAHEAD INFO PASSING

```
SELECT COUNT(*) FROM fact AS f
  JOIN dim1 ON f.dim1_id = dim1.id
  JOIN dim2 ON f.dim2_id = dim2.id
```



First compute Bloom filters on dimension tables.

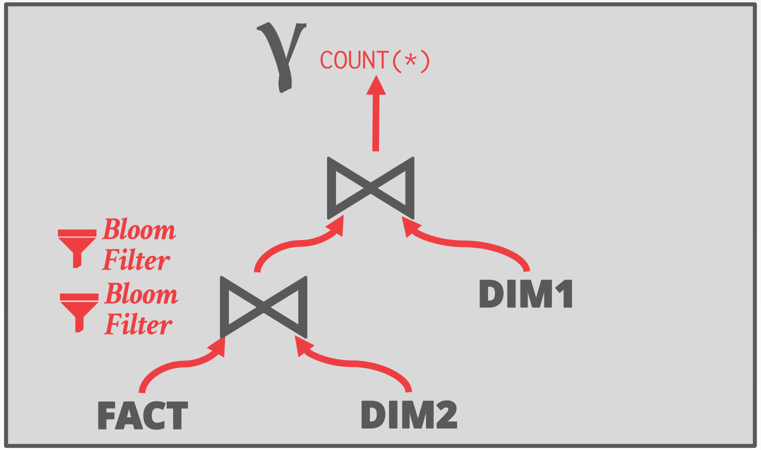
Probe these filters using fact table tuples to determine the ordering of the joins.

Only supports left-deep join trees on star schemas.

LOOKING AHEAD MAKES QUERY PLANS ROBUST
VLDB 2017

QUICKSTEP – LOOKAHEAD INFO PASSING

```
SELECT COUNT(*) FROM fact AS f
  JOIN dim1 ON f.dim1_id = dim1.id
  JOIN dim2 ON f.dim2_id = dim2.id
```



First compute Bloom filters on dimension tables.

Probe these filters using fact table tuples to determine the ordering of the joins.

Only supports left-deep join trees on star schemas.

LOOKING AHEAD MAKES QUERY PLANS ROBUST
VLDB 2017

PLAN PIVOT POINTS

The optimizer embeds alternative sub-plans at materialization points in the query plan.

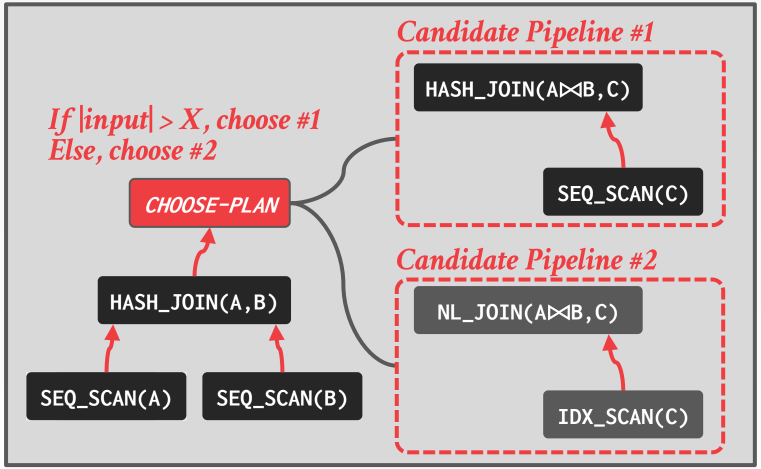
The plan includes "pivot" points that guides the DBMS towards a path in the plan based on the observed statistics.

Approach #1: Parametric Optimization

Approach #2: Proactive Reoptimization

PARAMETRIC OPTIMIZATION

```sql
SELECT * FROM A
  JOIN B ON A.id = B.id
  JOIN C ON A.id = C.id;
```



Generate multiple sub-plans per pipeline in the query.

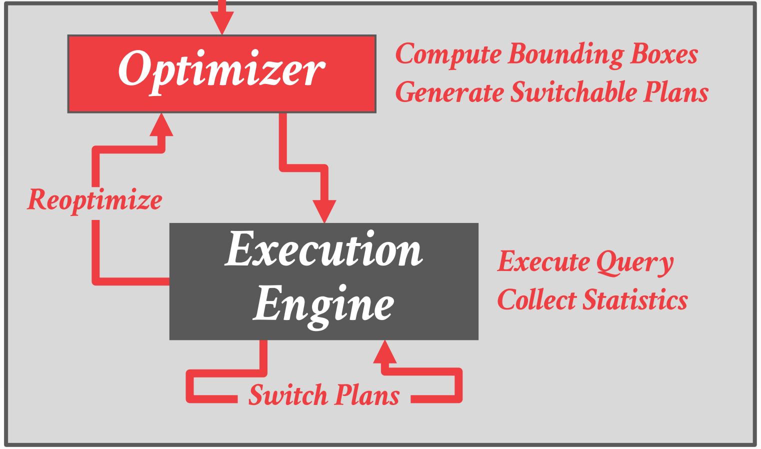
Add a ***choose-plan*** operator that allows the DBMS to select which plan to execute at runtime.

First introduced as part of the Volcano project in the 1980s.

DYNAMIC QUERY EVALUATION PLANS
SIGMOD RECORD 1989

PROACTIVE REOPTIMIZATION

```
SELECT * FROM A
  JOIN B ON A.id = B.id
  JOIN C ON A.id = C.id;
```



Generate multiple sub-plans within a single pipeline.

Use a ***switch*** operator to choose between different sub-plans during execution in the pipeline.

Computes bounding boxes to indicate the uncertainty of estimates used in plan.

PROACTIVE RE-OPTIMIZATION
SIGMOD 2005

PARTING THOUGHTS

The "plan-first execute-second" approach to query planning is notoriously error prone.

Optimizers should work with the execution engine to provide alternative plan strategies and receive feedback.

Adaptive techniques now appear in many of the major commercial DBMSs
→ DB2, Oracle, MSSQL, TeraData

NEXT CLASS

Let's understand how these cost models work and why they are so bad.