Query Optimizer Implementation Part 3
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 before 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 adapt the plan rather than continuing with the current sub-optimal plan.
BAD QUERY PLANS

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

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';

Estimated Cardinality: 1000
Actual Cardinality: 100000
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
Choice #1: Static Optimization
→ Select the best plan prior to execution.
→ Plan quality is dependent on cost model accuracy.
→ Can amortize over executions with prepared statements.

Choice #2: Dynamic Optimization
→ Select operator plans on-the-fly as queries execute.
→ Will have re-optimize for multiple executions.
→ Difficult to implement/debug (non-deterministic)

Choice #3: Adaptive Optimization
→ Compile using a static algorithm.
→ If the estimate errors > threshold, change or re-optimize.
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 OPTIMIZATION

Approach #1: Modify Future Invocations

Approach #2: Replan Current Invocation

Approach #3: Plan Pivot Points
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
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

Original Plan

```
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';
```

Estimated Cost: 1000
Actual Cost: 1000
REVERSION-BASED PLAN CORRECTION

Original Plan

```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';
```

Estimated Cost: 1000
Actual Cost: 1000
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';

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

Original Plan

Estimated Cost: 1000
Actual Cost: 1000

Execution History
REVERSION-BASED PLAN CORRECTION

**Original Plan**

```
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';
```

**Estimated Cost:** 1000
**Actual Cost:** 1000

**New Plan**

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

**Estimated Cost:** 800
**Actual Cost:** 1200

Execution History
REVERSION-BASED PLAN CORRECTION

**Original Plan**

```
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';
```

**Estimated Cost:** 1000  
**Actual Cost:** 1000

**New Plan**

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

**Estimated Cost:** 800  
**Actual Cost:** 1200

Execution History
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.
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';

CREATE INDEX idx_b_val ON B (val);
CREATE INDEX idx_d_val ON D (val);
**Microsoft: Plan Stitching**

**Original Plan**

```
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';
```

**New Plan**

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

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

Original Plan

Sub-Plan Cost: 600

New Plan  
Sub-Plan Cost: 150
**MICROSOFT: PLAN STITCHING**

**Original Plan**

```
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';
```

**New Plan**

```
SUBPLAN COST: 600
```

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

**Total Estimated Cost: 600 + 150**
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 \textbf{OR} operators to indicate alternative paths through the plan.

Source: Bailu Ding
Generate a graph that contains all possible sub-plans.

Add OR operators to indicate alternative paths through the plan.

Source: Bailu Ding
ENCODE SEARCH SPACE

Generate a graph that contains all possible sub-plans.

Add **OR** operators to indicate alternative paths through the plan.

Source: Bailu Ding
Generate a graph that contains all possible sub-plans.

Add 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.
CONSTRUCTING STITCHED PLANS

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

Source: Bailu Ding
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 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.
REPLAN CURRENT INVOCATION

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

Approach #1: Start-Over from Scratch
Approach #2: Keep Intermediate Results
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.
Quickstep: Lookahead Info Passing

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.

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.
Prove these filters using fact table tuples to determine the ordering of the joins.
Only supports left-deep join trees on star schemas.
QUICKSTEP: LOOKAHEAD INFO PASSING

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.
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**

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.
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 Reoptimization

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.

```
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.
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.
PLAN STABILITY

Choice #1: Hints
→ Allow the DBA to provide hints to the optimizer.

Choice #2: Fixed Optimizer Versions
→ Set the optimizer version number and migrate queries one-by-one to the new optimizer.

Choice #3: Backwards-Compatible Plans
→ Save query plan from old version and provide it to the new DBMS.
PLAN STABILITY

Choice #1: Hints
→ Allow the DBA to provide hints to the optimizer.

Choice #2: Fixed Optimizer Versions
→ Set the optimizer version number and migrate queries one by one to the new optimizer.

Choice #3: Backwards-Compatible Plans
→ Save query plan from old version and provide it to the new DBMS.

```
SELECT * FROM table1 AS t1
JOIN table2 AS t2 ON (t1.key = t2.key)
JOIN table3 AS t3 ON (t2.key = t3.key);
```
PLAN STABILITY

Choice #1: Hints
→ Allow the DBA to provide hints to the optimizer.

Choice #2: Fixed Optimizer Versions
→ Set the optimizer version number and migrate queries one by one.

Choice #3: Backwards Compatible Plans
→ Save query plan from old version and provide it to the new DBMS.
PLAN STABILITY

Choice #1: Hints
→ Allow the DBA to provide hints to the optimizer.

Choice #2: Fixed Optimizer Versions
→ Set the optimizer version number and migrate queries one by one to the new DBMS.

Choice #3: Backwards Compatible Plans
→ Save query plan from old version and provide it to the new DBMS.

```sql
SELECT * FROM dbo.tableA AS a
INNER JOIN dbo.tableB AS b ON b.ID = a.ID
```

Use PLAN N'<?xml version="1.0" encoding="utf-16"?>
  <BatchSequence>
    <Batch>
      <Statements>
        <StmtSimple StatementCompId="1" StatementEstRows="1" StatementId="1"
          StatementOptmLevel="FULL" StatementOptmEarlyAbortReason="GoodEnoughPlanFound"
          CardinalityEstimationMode="Version" StatementSubTreeCost="0.00657068"
          StatementText="SELECT * FROM dbo.tableA a INNER JOIN dbo.tableB b ON b.ID = a.ID"
          StatementType="SELECT"
          QueryHash="#x126A18B217E558B" QueryPlanHash="0x3700F7E43143DF3"
          RetrievedFromCache="false" SecurityPolicyApplied="false"
          StatementSetOptions ANSI_NULLS="true" ANSI_PADDING="true"
          ANSI_WARNINGS="true" ARITHABORT="true" CONCAT_NULL_YIELDS_NULL="true"
          NUMERIC_ROUNDABORT="false" QUOTED_IDENTIFIER="true" />
        <QueryPlan CachedPlanSize="16" CompileTime="0" CompileCPU="0"
          CompileMemory="152">
          <MemoryGrantInfo SerialRequiredMemory="0" SerialDesiredMemory="0="/>
          <OptimizerHardwareDependentProperties
            EstimatedAvailableMemoryGrant="10560000" EstimatedPagesCached="31680000"
            EstimatedAvailableDegreeOfParallelism="8" MaxCompileMemory="36453648"/>
          <RelOp AvgRowsSize="15" EstimatedCPU="4.18E-06" EstimateID="0"
            EstimateRewinds="0" EstimateExecutionMode="Row"
            EstimateRows="1" LogicalOp="Inner Join"
            NodeId="0" Parallel="false"
            PhysicalOp="Nested Loops" EstimatedTotalSubtreeCost="0.00657068" />
      </Batch>
    </Statements>
  </BatchSequence>
</ShowPlanXML>
```
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
Let's understand how these cost models work and why they are so bad.