Lecture 23: Adaptive Query Optimization & Cost Models

Today's Agenda

Recap

Cascades Framework

- Optimization tasks as data structures.
- Rules to place **property enforcers** (*e.g.*, sorting order).
- Ordering of transformations by priority.
- Predicates are first class citizens (same as logical/physical operators).

Today's Agenda

- Adaptive Query Optimization
- Techniques for Adaptive Query Optimization
  - Modify Future Invocations
  - Replan Current Invocation
  - Plan Pivot Points
- Cost Models
- Cost Estimation

Adaptive Query Optimization

Observation

- The query optimizers that we have talked about so far all generate a plan for a query **before** the DBMS executes a query.
- The best plan for a query can change as the database evolves over time.
  - Physical design changes.
  - Data modifications.
  - Prepared statement parameters.
  - Statistics updates.

Bad Query Plans

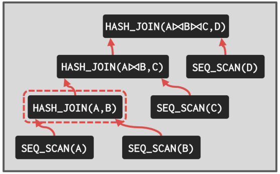
- The most common problem in a query plan is incorrect join orderings.
  - This occurs because of inaccurate **cardinality estimates** that propagate up the plan.
- If the DBMS can detect how bad a query plan is, then it can decide to **adapt** the plan rather than continuing with the current sub-optimal 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 = 'XXX'
   AND D.val = 'YYY';
```

Original Plan



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 summarization** 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.
- Reference

Adaptive Query Optimization

- Approach 1: Modify Future Invocations
- Approach 2: Replan Current Invocation
- Approach 3: Plan Pivot Points

Modify Future Invocations

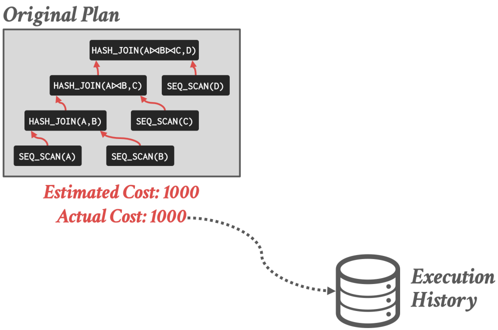
Modify Future Invocations

- 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

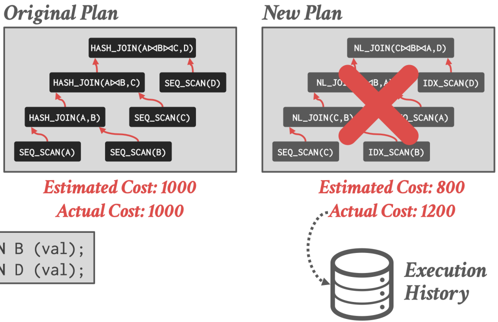
Reversion-Based Plan Correction

- 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



Reversion-Based Plan Correction

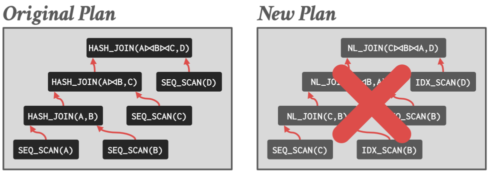



```
CREATE INDEX idx_b_val ON B (val);
CREATE INDEX idx_d_val ON D (val);
```

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. Reference

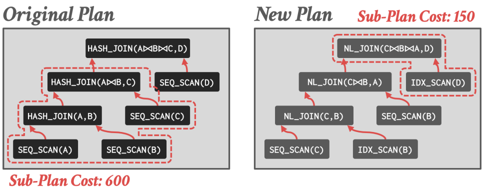
Microsoft – Plan Stitching




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

```
DROP INDEX idx_b_val;
```

Microsoft – Plan Stitching




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

```
DROP INDEX idx_b_val;
```

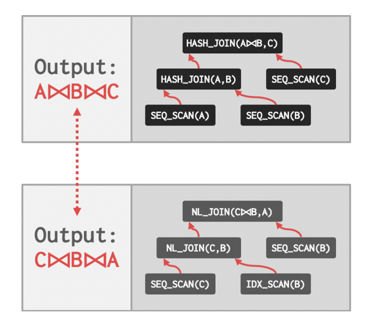
Microsoft – Plan Stitching


```
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.



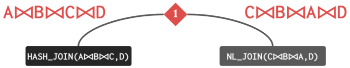
Encoding Search Space

- Generate a graph that contains all possible sub-plans.
- Add operators to indicate alternative paths through the plan.

Encoding Search Space

Generate a graph that contains all possible sub-plans.

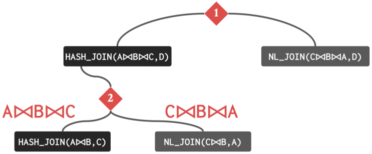
Add OR operators to indicate alternative paths through the plan.



Encoding Search Space

Generate a graph that contains all possible sub-plans.

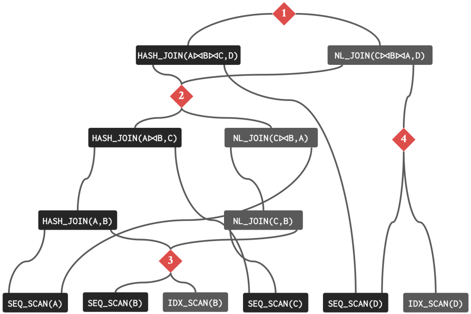
Add OR operators to indicate alternative paths through the plan.



Encoding Search Space

Generate a graph that contains all possible sub-plans.

Add OR operators to indicate alternative paths through the plan.



Constructing Stitched Plans

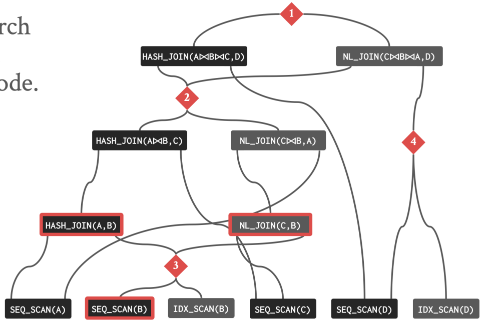
- Perform bottom-up search that selects the cheapest sub-plan for each OR node.

Constructing Stitched Plans

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

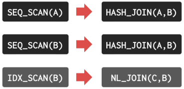
SEQ_SCAN(A) → HASH_JOIN(A,B)

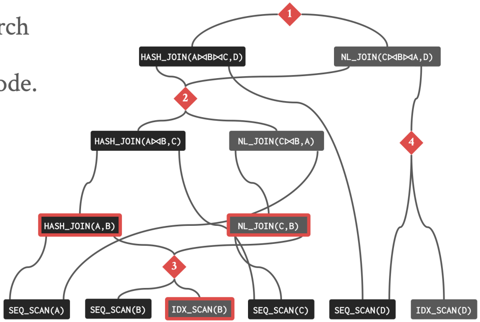
SEQ_SCAN(B) → HASH_JOIN(A,B)



Constructing Stitched Plans

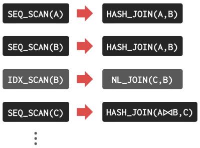
Perform bottom-up search that selects the cheapest sub-plan for each **OR** node.

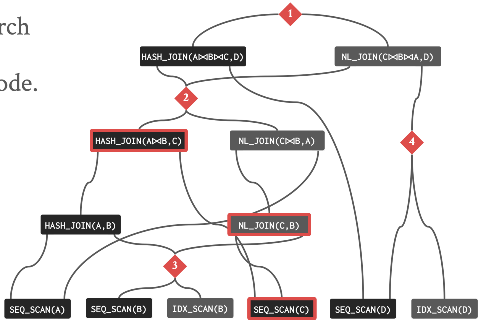




Constructing Stitched Plans

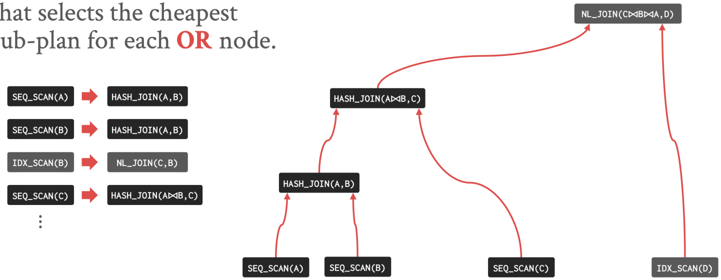
Perform bottom-up search that selects the cheapest sub-plan for each **OR** node.





Constructing Stitched Plans

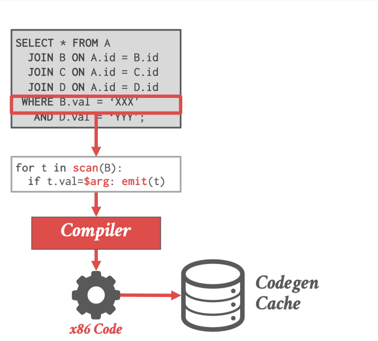
Perform bottom-up search that selects the cheapest sub-plan for each **OR** node.



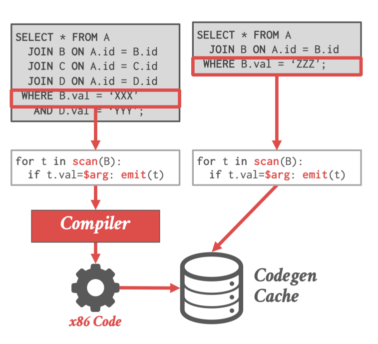
REDSHIFT – Codegen Stitching

- 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.

REDSHIFT – Codegen Stitching



REDSHIFT – Codegen Stitching



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.
- Reference

Replan Current Invocation

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

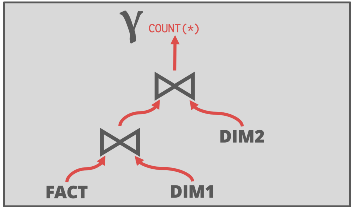
QUICKSTEP – Lookahead Info Passing

```sql
--- Star Schema
CREATE TABLE fact(          --- Fact Table
  id INT PRIMARY KEY,
  dim1_id INT REFERENCES dim1 (id),
  dim2_id INT REFERENCES dim2 (id)
);
CREATE TABLE dim1 (         --- Dimension Tables
  id INT, val VARCHAR
);
CREATE TABLE dim2 (
  id INT, val VARCHAR
);
SELECT COUNT(*) FROM fact AS f
  JOIN dim1 ON f.dim1_id = dim1.id
  JOIN dim2 ON f.dim2_id = dim2.id
```

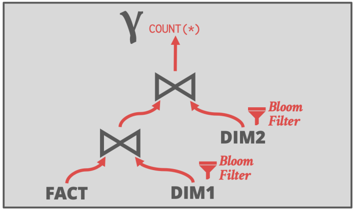
QUICKSTEP – Lookahead Info Passing

- 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.
- Reference

QUICKSTEP – Lookahead Info Passing



QUICKSTEP – Lookahead Info Passing



Plan Pivot Points

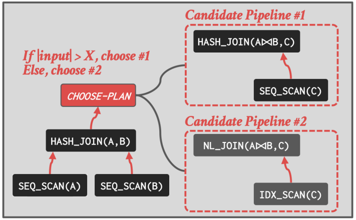
Plan Pivot Points

- 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

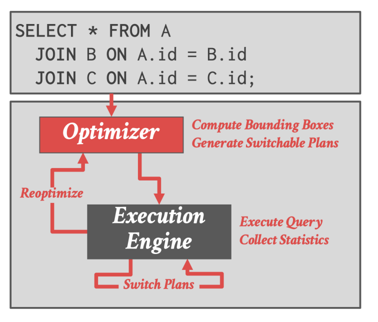
- Generate multiple sub-plans per pipeline in the query.
- Add a choose-plan operator that allows the DBMS to select which plan to execute at runtime.
- First introduced as part of the Volcano project in the 1980s.
- Reference

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



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.
- Reference



Cost Models

Cost-based Query Planning

- Generate an estimate of the cost of executing a particular query plan for the current state of the database.
  - Estimates are only meaningful internally.
- This is independent of the **search strategies** that we talked about.

Cost Model Components

- **Choice 1: Physical Costs**
  - Predict CPU cycles, I/O, cache misses, RAM consumption, pre-fetching, etc. . .
  - Depends heavily on hardware.
- **Choice 2: Logical Costs**
  - Estimate result sizes per operator (*e.g.*, join operator).
  - Independent of the operator algorithm.
  - Need estimations for operator result sizes.
- **Choice 3: Algorithmic Costs**
  - Complexity of the operator algorithm implementation (*e.g.*, hash join vs. nested loop join).

Disk-Based DBMS: Cost Model

- The number of disk accesses will always dominate the execution time of a query.
  - CPU costs are negligible.
  - Have to consider sequential vs. random I/O.
- This is easier to model if the DBMS has full control over buffer management.
  - We will know the replacement strategy, pinning, and assume exclusive access to disk.

Postgres

- Uses a combination of CPU and I/O costs that are weighted by "magic" constant factors.
- Default settings are obviously for a disk-resident database without a lot of memory:
  - Processing a tuple in memory is $400\times$ faster than reading a tuple from disk.
  - Sequential I/O is $4\times$ faster than random I/O.

IBM DB2

- Database characteristics in system catalogs
- Hardware environment (microbenchmarks)
- Storage device characteristics (microbenchmarks)
- Communications bandwidth (distributed only)
- Memory resources (buffer pools, sort heaps)
- Concurrency Environment
  - Average number of users
  - Isolation level / blocking
  - Number of available locks
- Reference

In-Memory DBMS: Cost Model

- No I/O costs, but now we have to account for CPU and memory access costs.
- Memory cost is more difficult because the DBMS has no control over **CPU cache management**.
  - Unknown replacement strategy, no pinning, shared caches, non-uniform memory access.
- The number of tuples processed per operator is a reasonable estimate for the CPU cost.

Smallbase

- Two-phase model that automatically generates hardware costs from a logical model.
- **Phase 1: Identify Execution Primitives**
  - List of ops that the DBMS does when executing a query
  - Example: evaluating predicate, index probe, sorting.
- **Phase 2: Microbenchmark**
  - On start-up, profile ops to compute CPU/memory costs
  - These measurements are used in formulas that compute operator cost based on table size.

Selectivity

- The **selectivity** of an operator is the percentage of data accessed for a predicate.
  - Modeled as probability of whether a predicate on any given tuple will be satisfied.
- The DBMS estimates selectivities using:
  - Domain Constraints
  - Precomputed Statistics (Zone Maps)
  - Histograms / Approximations
  - Sampling

Observation

- The number of tuples processed per operator depends on three factors:
  - The access methods available per table
  - The distribution of values in the database's attributes
  - The predicates used in the query
- Simple queries are easy to estimate. More complex queries are not.

Cost Estimation

Approximations

- Maintaining exact statistics about the database is expensive and slow.
- Use approximate data structures called **sketches** to generate error-bounded estimates.
  - Count Distinct
  - Quantiles
  - Frequent Items
  - Tuple Sketch
- **Example:** Yahoo! Sketching Library

Sampling

- Another approximation technique
- Execute a predicate on a random sample of the target data set.
- The number of tuples to examine depends on the size of the table.
- **Approach 1: Maintain Read-Only Copy**
  - Periodically refresh to maintain accuracy.
- **Approach 2: Sample Real Tables**
  - Use `READ UNCOMMITTED` isolation.
  - May read multiple versions of same logical tuple.

Result Cardinality

- The number of tuples that will be generated per operator is computed from its selectivity multiplied by the number of tuples in its input.
  - **Assumption 1: Uniform Data**
    - The distribution of values (except for the heavy hitters) is the same.
  - **Assumption 2: Independent Predicates**
    - The predicates on attributes are independent
  - **Assumption 3: Inclusion Principle**
    - The domain of join keys overlap such that each key in the inner relation will also exist in the outer table.

Correlated Attributes

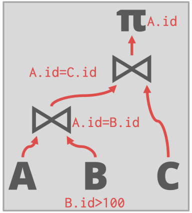
- Consider a database of automobiles:
  - Number of Makes = 10, Number of Models = 100
- And the following query:
  - (make="Honda" *AND* model="Accord")
- With the independence and uniformity assumptions, the selectivity is:
  - $1/10 \times 1/100 = 0.001$
- But since only Honda makes Accords the real selectivity is $1/100 = 0.01$

Column Group Statistics

- The DBMS can track statistics for groups of attributes together rather than just treating them all as independent variables.
  - Mostly supported in commercial systems.
  - Requires the DBA to declare manually.

Estimation Problem

```
SELECT A.id
  FROM A, B, C
 WHERE A.id = B.id
   AND A.id = C.id
   AND B.id > 100
```



Compute the cardinality of base tables

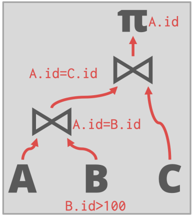
$\mathbf{A} \longrightarrow |\mathbf{A}|$

$\mathbf{B}.\mathrm{id}>100 \longrightarrow |\mathbf{B}| \times \mathit{sel}(\mathbf{B}.\mathrm{id}>100)$

$\mathbf{C} \longrightarrow |\mathbf{C}|$

Estimation Problem

```
SELECT A.id
  FROM A, B, C
 WHERE A.id = B.id
   AND A.id = C.id
   AND B.id > 100
```



Compute the cardinality of base tables

$\mathbf{A} \longrightarrow |\mathbf{A}|$

$\mathbf{B}.\mathrm{id}>100 \longrightarrow |\mathbf{B}| \times \textit{\textbf{sel}}(\mathbf{B}.\mathrm{id}>100)$

$\mathbf{C} \longrightarrow |\mathbf{C}|$

Compute the cardinality of join results

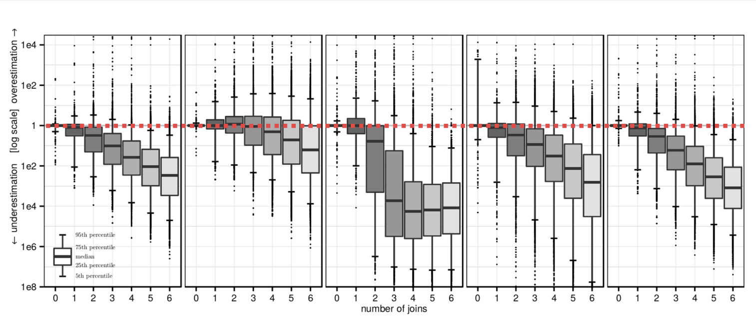
$$\mathbf{A} \bowtie \mathbf{B} = (|\mathbf{A}| \times |\mathbf{B}|) \,/\, \textit{\textbf{max}}(\textit{\textbf{sel}}(\mathbf{A}.\mathrm{id}=\mathbf{B}.\mathrm{id}),\ \textit{\textbf{sel}}(\mathbf{B}.\mathrm{id}>100))$$

$$(\mathbf{A} \bowtie \mathbf{B}) \bowtie \mathbf{C} = (|\mathbf{A}| \times |\mathbf{B}| \times |\mathbf{C}|) \,/\, \textit{\textbf{max}}(\textit{\textbf{sel}}(\mathbf{A}.\mathrm{id}=\mathbf{B}.\mathrm{id}),\ \textit{\textbf{sel}}(\mathbf{B}.\mathrm{id}>100),\ \textit{\textbf{sel}}(\mathbf{A}.\mathrm{id}=\mathbf{C}.\mathrm{id}))$$

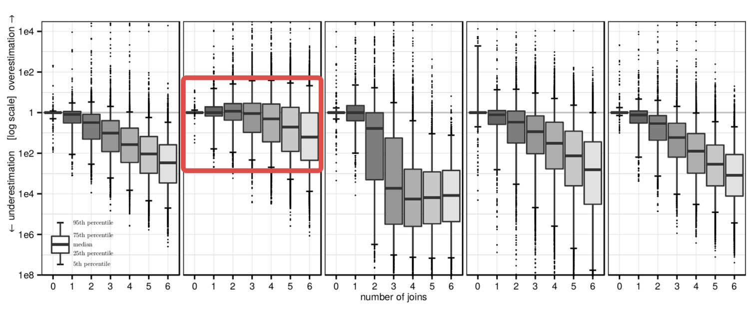
Estimator Quality

- Evaluate the correctness of cardinality estimates generated by DBMS optimizers as the number of joins increases.
  - Let each DBMS perform its stats collection.
  - Extract measurements from query plan.
- Compared five DBMSs using 100k queries.
- Reference

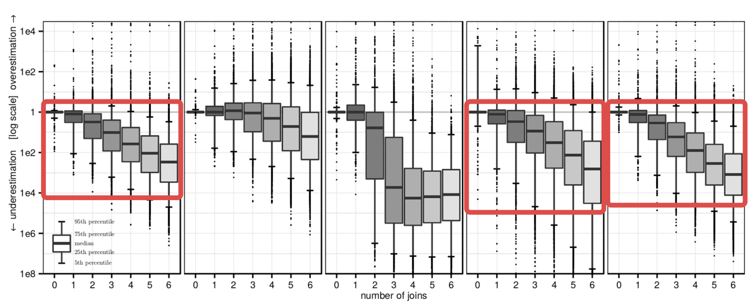
Estimator Quality



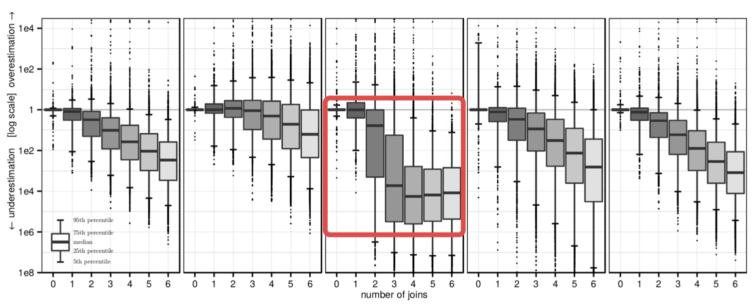
Estimator Quality



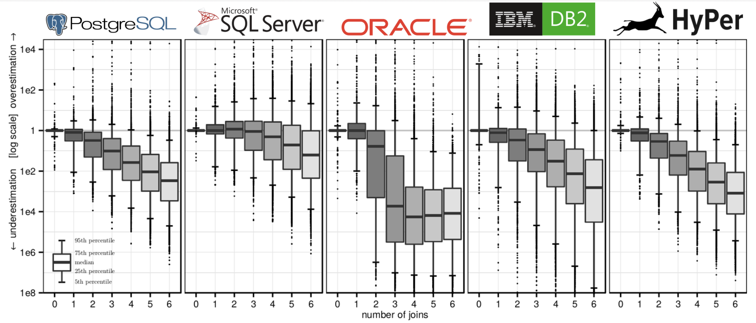
Estimator Quality



Estimator Quality

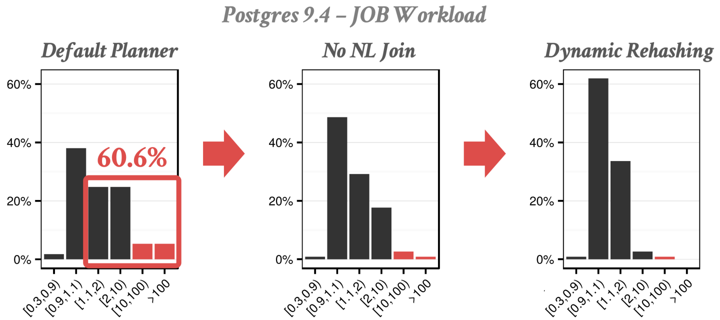


Estimator Quality



Execution Slowdown

- Slowdown compared to using true cardinalities



Slowdown compared to using true cardinalities

Lessons Learned

- Query opt is more important than a fast engine
  - Cost-based join ordering is necessary
- Cardinality estimates are routinely wrong
  - Try to use operators that do not rely on estimates
- Hash joins + seq scans are a robust exec model
  - The more indexes that are available, the more brittle the plans become (but also faster on average)
- Working on accurate models is a waste of time
  - Better to improve cardinality estimation instead

Conclusion

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
- Using number of tuples processed is a reasonable cost model for in-memory DBMSs.
  - But computing this is non-trivial.
- A combination of sampling + sketches allows the DBMS to achieve accurate estimations.

Next Class

- User-defined functions.