Announcements

- No lecture Monday, 2/19
- No lecture Wednesday, 2/21
- Makeup lecture Friday, 2/23 – Gates371
- Also Friday 2/23: HW3 is due
- Project milestone due Monday, 2/26
Query Optimization

Three major components:

1. Search space  
   last week

2. Cardinality and cost estimation  
   last lecture

3. Plan enumeration algorithms  
   today
Paper Discussion

• How Good Are Query Optimizers, Really? VLDB’2015
Questions in the paper

• How good are cardinality estimators?

• How important are they for the optimizer?

• How large does the plan space need to be?
Cardinality Estimators

• Standard database benchmark: TPC-H

• They designed a new benchmark. Why?
Cardinality Estimators

• Standard database benchmark: TPC-H

• They designed a new benchmark. Why?

• Because TPC-H is synthetically generated, unrealistically uniform
Cardinality Estimators

What type of queries are in IMDB/JOB?
Cardinality Estimators

What type of queries are in IMDB/JOB?

• For CE: select * multijoin queries
• For runtime: replace * with min

[How good are they]
Cardinality Estimators

What type of queries are in IMDB/JOB?

• For CE: select * multijoin queries
• For runtime: replace * with min  Why?

• Materializing * is expensive…
• …and postgres does not push min down the plan
Single Table Estimation

<table>
<thead>
<tr>
<th></th>
<th>median</th>
<th>90th</th>
<th>95th</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>PostgreSQL</td>
<td>1.00</td>
<td>2.08</td>
<td>6.10</td>
<td>207</td>
</tr>
<tr>
<td>DBMS A</td>
<td>1.01</td>
<td>1.33</td>
<td>1.98</td>
<td>43.4</td>
</tr>
<tr>
<td>DBMS B</td>
<td>1.00</td>
<td>6.03</td>
<td>30.2</td>
<td>104000</td>
</tr>
<tr>
<td>DBMS C</td>
<td>1.06</td>
<td>1677</td>
<td>5367</td>
<td>20471</td>
</tr>
<tr>
<td>HyPer</td>
<td>1.02</td>
<td>4.47</td>
<td>8.00</td>
<td>2084</td>
</tr>
</tbody>
</table>

Table 1: Q-errors for base table selections
### Single Table Estimation

<table>
<thead>
<tr>
<th></th>
<th>median</th>
<th>90th</th>
<th>95th</th>
</tr>
</thead>
<tbody>
<tr>
<td>PostgreSQL</td>
<td>1.00</td>
<td>2.08</td>
<td>6.10</td>
</tr>
<tr>
<td>DBMS A</td>
<td>1.01</td>
<td>1.33</td>
<td><strong>1.98</strong></td>
</tr>
<tr>
<td>DBMS B</td>
<td>1.00</td>
<td>6.03</td>
<td>30.2</td>
</tr>
<tr>
<td>DBMS C</td>
<td>1.06</td>
<td>1677</td>
<td>5367</td>
</tr>
<tr>
<td>HyPer</td>
<td>1.02</td>
<td>4.47</td>
<td>8.00</td>
</tr>
</tbody>
</table>

**Table 1: Q-errors for base table selections**

What technique helped here? (conjectured)
**Single Table Estimation**

Table 1: Q-errors for Selections

<table>
<thead>
<tr>
<th></th>
<th>median</th>
<th>90th</th>
<th>95th</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PostgreSQL</td>
<td>1.00</td>
<td>2.08</td>
<td>6.10</td>
<td>207</td>
</tr>
<tr>
<td>DBMS A</td>
<td>1.01</td>
<td>1.33</td>
<td>1.98</td>
<td>43.4</td>
</tr>
<tr>
<td>DBMS B</td>
<td>1.00</td>
<td>6.03</td>
<td>30.2</td>
<td>104000</td>
</tr>
<tr>
<td>DBMS C</td>
<td>1.06</td>
<td>1677</td>
<td>5367</td>
<td>20471</td>
</tr>
<tr>
<td>HyPer</td>
<td>1.02</td>
<td>4.47</td>
<td>8.00</td>
<td>2084</td>
</tr>
</tbody>
</table>

What technique helped here? (conjectured)

- Sampling.
- E.g. Hyper: 1000 rows
# Single Table Estimation

<table>
<thead>
<tr>
<th></th>
<th>median</th>
<th>90th</th>
<th>95th</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>PostgreSQL</td>
<td>1.00</td>
<td>2.08</td>
<td>6.10</td>
<td>207</td>
</tr>
<tr>
<td>DBMS A</td>
<td>1.01</td>
<td>1.33</td>
<td>1.98</td>
<td>43.4</td>
</tr>
<tr>
<td>DBMS B</td>
<td>1.00</td>
<td>6.03</td>
<td>30.2</td>
<td>104000</td>
</tr>
<tr>
<td>DBMS C</td>
<td>1.06</td>
<td>1677</td>
<td>5367</td>
<td>20471</td>
</tr>
<tr>
<td>HyPer</td>
<td>1.02</td>
<td>4.47</td>
<td>8.00</td>
<td>2084</td>
</tr>
</tbody>
</table>

**Table 1: Q-errors for base table selections**
Single Table Estimation

<table>
<thead>
<tr>
<th>Database</th>
<th>Median</th>
<th>90th</th>
<th>95th</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>PostgreSQL</td>
<td>1.00</td>
<td>2.08</td>
<td>6.10</td>
<td>207</td>
</tr>
<tr>
<td>DBMS A</td>
<td>1.01</td>
<td>1.33</td>
<td>1.98</td>
<td>43.4</td>
</tr>
<tr>
<td>DBMS B</td>
<td>1.00</td>
<td>6.03</td>
<td>30.2</td>
<td>104000</td>
</tr>
<tr>
<td>DBMS C</td>
<td>1.06</td>
<td>1677</td>
<td>5367</td>
<td>20471</td>
</tr>
<tr>
<td>HyPer</td>
<td>1.02</td>
<td>4.47</td>
<td>8.00</td>
<td>2084</td>
</tr>
</tbody>
</table>

Table 1: Q-errors for base table selections

Why queries still lead to poor estimates?
How good are they

Why queries still lead to poor estimates?

<table>
<thead>
<tr>
<th></th>
<th>median</th>
<th>90th</th>
<th>95th</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>PostgreSQL</td>
<td>1.00</td>
<td>2.08</td>
<td>6.10</td>
<td>207</td>
</tr>
<tr>
<td>DBMS A</td>
<td>1.01</td>
<td>1.33</td>
<td>1.98</td>
<td>43.4</td>
</tr>
<tr>
<td>DBMS B</td>
<td>1.00</td>
<td>6.03</td>
<td>30.2</td>
<td>104000</td>
</tr>
<tr>
<td>DBMS C</td>
<td>1.06</td>
<td>1677</td>
<td>5367</td>
<td>20471</td>
</tr>
<tr>
<td>HyPer</td>
<td>1.02</td>
<td>4.47</td>
<td>8.00</td>
<td>2084</td>
</tr>
</tbody>
</table>

Table 1: Q-errors for half SELECTions

Low selectivity: $10^{-5} - 10^{-6}$
Single Table Estimation

• **1d Histograms:**
  – Good for single equality or range predicate
  – Poor for multiple predicates
  – Useless for LIKE

• **Samples:**
  – Good for multiple predicates, LIKE
  – Poor for low selectivity predicates
How good are they

Joins (0 to 6)

Figure 3: Quality of cardinality estimates for multi-join queries in comparison with the true cardinalities. Each boxplot summarizes the error distribution of all subexpressions with a particular size (over all queries in the workload)
Joins (0 to 6)

Figure 3: Quality of cardinality estimates for multi-join queries in comparison with the true cardinalities. Each boxplot summarizes the error distribution of all subexpressions with a particular size (over all queries in the workload)
Estimation of Joins

• Error increases exponentially with the number of joins
  – This was known from [Ioannidis’91]

• Underestimate, because of positive correlations
How good are they

TPC-H v.s. Real Data (IMDB)
[How good are they]

TPC-H v.s. Real Data (IMDB)

Huge errors

Perfect estimates
Impact of Mis-estimates

• Question: how much does a good/poor CE matter for the quality of a query plan
• How did they measure that?
Impact of Mis-estimates

• Question: how much does a good/poor CE matter for the quality of a query plan

• How did they measure that?
  – Inject into postgres other systems’ estimates – won’t discuss this
  – Inject into postgres true cardinalities; call it optimal plan, compare with regular plan

• Two configs of indexes: PK and PK+FK
Impact of Mis-estimates

PK indexes

Figure 6: Slowdown of queries using PostgreSQL estimates w.r.t. using true cardinalities (primary key indexes only)
Impact of Mis-estimates

PK indexes

Most queries: no slowdown w.r.t optimal

Figure 6: Slowdown of queries using PostgreSQL estimates w.r.t. using true cardinalities (primary key indexes only)
Impact of Mis-estimates

PK indexes

Most queries: no slowdown w.r.t optimal

"Better that optimal" how can that be?

Figure 6: Slowdown of queries using PostgreSQL estimates w.r.t. using true cardinalities (primary key indexes only)
Impact of Mis-estimates

PK indexes

Figure 6: Slowdown of queries using PostgreSQL estimates w.r.t. using true cardinalities (primary key indexes only)
Impact of Mis-estimates

PK indexes

Figure 6: Slowdown of queries using PostgreSQL estimates w.r.t. using true cardinalities (primary key indexes only)
Impact of Mis-estimates

PK indexes

Figure 6: Slowdown of queries using PostgreSQL estimates w.r.t. using true cardinalities (primary key indexes only)

Still some queries significantly slower. Why?
Impact of Mis-estimates

PK indexes

Figure 6: Slowdown of queries using PostgreSQL estimates w.r.t. using true cardinalities (primary key indexes only)
Impact of Mis-estimates

Indexes on PK only

- Low sensitivity to CE, because the “fact” table needs to be scanned anyway
- Plans most sensitive to CE errors:
  - Plans with nested-loop joins
  - Hash-table preallocation
- Discuss “robust query optimization”
Impact of Mis-estimates

FK/PK indexes

Figure 7: Slowdown of queries using PostgreSQL estimates w.r.t. using true cardinalities (different index configurations)
Impact of Mis-estimates

FK/PK indexes

Figure 7: Slowdown of queries using PostgreSQL estimates w.r.t. using true cardinalities (different index configurations)
Discussion

• When PK indexes only, optimizer chooses a good plan anyway; impact of CE is limited; confirmed by others too

• When indexes on PK+FK, performance improves, but sensitivity to CE is higher
Cardinalities to Cost

[How good are they]
Cardinalities to Cost

Postgres cost

[How good are they]
Cardinalities to Cost

Postgres cost

No I/O, keep only CPU

[How good are they]
Cardinalities to Cost

- Postgres cost
- No I/O, keep only CPU
- Their own simple formula

How good are they?
Cardinalities to Cost

• CE accounts for largest errors

• Cost models: both simple or complex are fine

Postgres cost
No I/O, keep only CPU
Their own simple formula
Query Optimization

Three major components:

1. Search space
2. Cardinality and cost estimation
3. Plan enumeration algorithms
Two Types of Optimizers

- Heuristic-based optimizers
  - Limited, used only by the simplest DBMS

- Cost-based optimizers (next)
  - Enumerate query plans, return the cheapest
Two Types of Plan Enumeration Algorithms

• Dynamic programming
  – Based on System R [Selinger 1979]
  – Join reordering algorithm

• Cascades optimizer
System R Optimizer

For each subquery $Q \subseteq \{R_1, \ldots, R_n\}$, compute best plan:

- **Step 1:** $Q = \{R_1\}, \{R_2\}, \ldots, \{R_n\}$

- **Step 2:** $Q = \{R_1, R_2\}, \{R_1, R_3\}, \ldots, \{R_{n-1}, R_n\}$

- ...

- **Step n:** $Q = \{R_1, \ldots, R_n\}$
Details

For each subquery $Q \subseteq \{R_1, \ldots, R_n\}$ store:

- Estimated Size($Q$)
- A best plan for $Q$: Plan($Q$)
- The cost of that plan: Cost($Q$)

One plan for each “interesting order”
Details

Step 1: single relations \{R_1\}, \{R_2\}, \ldots, \{R_n\}

- Consider all possible access paths:
  - Sequential scan, or
  - Index 1, or
  - Index 2, or
  - ...

- Keep optimal plan for each “interesting order”
Details

Step \( k = 2 \ldots n \):
For each \( Q = \{R_{i_1}, \ldots, R_{i_k}\} \)

- For each \( j=1,\ldots,k \):
  - Consider all plans of the form \( P = P_1 \bowtie P_2 \)
  - \( Cost(P) = Cost(\bowtie) + Cost(P_1) + Cost(P_2) \)
  - Keep the cheapest plan, or
  - Keep multiple plans, for “interesting orders”

Runtime: exponential in \( n \).
Mitigated by: no cartesian products, restricted plan shapes
Importance of the Plan Space

• Do we need to explore a large space, or should we pick a plan at random?

• Do we need bushy trees, or are left-, or right-, or zigzag-trees enough?

• Do we need dynamic programming, or is greedy enough?
[How good are they]

Figure 9: Cost distributions for 5 queries and different index configurations. The vertical green lines represent the cost of the optimal plan.
Figure 9: Cost distributions for 5 queries and different index configurations. The vertical green lines represent the cost of the optimal plan.
How good are they?

<table>
<thead>
<tr>
<th></th>
<th>PK indexes</th>
<th></th>
<th></th>
<th>PK + FK indexes</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>median</td>
<td>95%</td>
<td>max</td>
<td>median</td>
<td>95%</td>
<td>max</td>
</tr>
<tr>
<td>zig-zag</td>
<td>1.00</td>
<td>1.06</td>
<td>1.33</td>
<td>1.00</td>
<td>1.60</td>
<td>2.54</td>
</tr>
<tr>
<td>left-deep</td>
<td>1.00</td>
<td>1.14</td>
<td>1.63</td>
<td>1.06</td>
<td>2.49</td>
<td>4.50</td>
</tr>
<tr>
<td>right-deep</td>
<td>1.87</td>
<td>4.97</td>
<td>6.80</td>
<td>47.2</td>
<td>30931</td>
<td>738349</td>
</tr>
</tbody>
</table>

Table 2: Slowdown for restricted tree shapes in comparison to the optimal plan (true cardinalities)
Generally, not much worse than optimal…

<table>
<thead>
<tr>
<th></th>
<th>PK indexes</th>
<th></th>
<th>PK + FK indexes</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>median</td>
<td>95%</td>
<td>max</td>
<td>median</td>
<td>95%</td>
<td>max</td>
</tr>
<tr>
<td>zig-zag</td>
<td>1.00</td>
<td>1.06</td>
<td>1.33</td>
<td>1.00</td>
<td>1.60</td>
<td>2.54</td>
</tr>
<tr>
<td>left-deep</td>
<td>1.00</td>
<td>1.14</td>
<td>1.63</td>
<td>1.06</td>
<td>2.49</td>
<td>4.50</td>
</tr>
<tr>
<td>right-deep</td>
<td>1.87</td>
<td>4.97</td>
<td>6.80</td>
<td>47.2</td>
<td>30931</td>
<td>738349</td>
</tr>
</tbody>
</table>

Table 2: Slowdown for restricted tree shapes in comparison to the optimal plan (true cardinalities)
Generally, not much worse than optimal…

<table>
<thead>
<tr>
<th></th>
<th>PK indexes</th>
<th></th>
<th>PK + FK indexes</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>median</td>
<td>95%</td>
<td>max</td>
<td>median</td>
</tr>
<tr>
<td>zig-zag</td>
<td>1.00</td>
<td>1.06</td>
<td>1.33</td>
<td>1.00</td>
</tr>
<tr>
<td>left-deep</td>
<td>1.00</td>
<td>1.14</td>
<td>1.63</td>
<td>1.06</td>
</tr>
<tr>
<td>right-deep</td>
<td>1.87</td>
<td>4.97</td>
<td>6.80</td>
<td>47.2</td>
</tr>
</tbody>
</table>

Table 2: Slowdown for restricted tree shapes in comparison to the optimal plan (true cardinalities)

…except here. Right-deep plans prevent index joins.
### Table 3: Comparison of exhaustive dynamic programming with the Quickpick-1000 (best of 1000 random plans) and the Greedy Operator Ordering heuristics. All costs are normalized by the optimal plan of that index configuration

<table>
<thead>
<tr>
<th>Method</th>
<th>PK indexes</th>
<th>PK + FK indexes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PostgreSQL estimates</td>
<td>true cardinalities</td>
</tr>
<tr>
<td></td>
<td>median 95% max</td>
<td>median 95% max</td>
</tr>
<tr>
<td>Dynamic Programming</td>
<td>1.03 1.85 4.79</td>
<td>1.00 1.00 1.00</td>
</tr>
<tr>
<td>Quickpick-1000</td>
<td>1.05 2.19 7.29</td>
<td>1.00 1.07 1.14</td>
</tr>
<tr>
<td>Greedy Operator Ordering</td>
<td>1.19 2.29 2.36</td>
<td>1.19 1.64 1.97</td>
</tr>
</tbody>
</table>

How good are they?
Cascades Optimizer

• Extends join ordering to full rewrite

• Supported by some of the most advanced DBMS today: SQL Server, Cocroach Lab; (not sure about DuckDB)

• Mostly “insider knowledge”
Cascades Optimizer

• Main idea: apply optimization rules:
  \[ Q \rightarrow Q' \]

• But keep both Q and Q’

• “Memo” data structure: reuses subplans
The Memo

Initialize Memo w/ one (naïve) plan

select * from R, S, T 
where R.B=S.B and S.C=T.C and R.A = 3 and T.D = 5

R(A,B), S(B,C), T(C,D)
R(A,B), S(B,C), T(C,D)

The Memo

initialize Memo w/ one (naïve) plan

scan R

1

1 initialize Memo w/ one (naïve) plan

$\sigma_{A=3}$

$\sigma_{D=5}$

$\bowtie$

$R$

$S$

$T$

select * from R, S, T where R.B = S.B and S.C = T.C and R.A = 3 and T.D = 5
R(A,B), S(B,C), T(C,D)

The Memo

Initialize Memo w/ one (naïve) plan

select * from R, S, T
where R.B=S.B
and S.C=T.C
and R.A = 3
and T.D = 5

1. Scan R
2. Select[A=3]
R(A,B), S(B,C), T(C,D)

The Memo

Initialize Memo w/ one (naïve) plan

select * from R, S, T where R.B = S.B and S.C = T.C and R.A = 3 and T.D = 5

1. Scan R
2. Select[A=3] 1
3. Scan S
4. Join[B=B] 2, 3
5. Scan T
6. Join[C=C] 4, 5
7. Select[D=5] 6
The Memo

select *
from R, S, T
where R.B = S.B
and S.C = T.C
and R.A = 3
and T.D = 5

R(A,B), S(B,C), T(C,D)

1. Scan R
2. Select[A=3] 1
3. Scan S
4. Join[B=B] 2, 3
5. Scan T
6. Join[C=C] 4, 5
7. Select[D=5] 6
The Memo

R(A,B), S(B,C), T(C,D)

Apply an optimization rule

select *
from R, S, T
where R.B=S.B
and S.C=T.C
and R.A = 3
and T.D = 5

1. Scan R
2. Select[A=3] 1
3. Scan S
4. Join[B=B] 2, 3
5. Scan T
6. Join[C=C] 4, 5
7. Select[D=5] 6
The Memo

R(A,B), S(B,C), T(C,D)

Apply an optimization rule

1. Scan R
2. Select[A=3] 1
3. Scan S
4. Join[B=B] 2, 3
5. Scan T
6. Join[C=C] 4, 5
7. Select[D=5] 6

select *
from R, S, T
where R.B=S.B
and S.C=T.C
and R.A = 3
and T.D = 5

Apply an optimization rule
The Memo

select * 
from R, S, T 
where R.B=S.B 
and S.C=T.C 
and R.A = 3 
and T.D = 5

1. Scan R
2. Select[A=3] R
3. Scan S
5. Scan T
7. Select[D=5] R, S, T
8. Select[D=5] S, T

R(A,B), S(B,C), T(C,D)
The Memo

select *
from R, S, T
where R.B=S.B
and S.C=T.C
and R.A = 3
and T.D = 5

R(A,B), S(B,C), T(C,D)
The Memo

R(A,B), S(B,C), T(C,D)

1. Scan R
2. Select[A=3] 1
3. Scan S
4. Join[B=B] 2, 3
5. Scan T
6. Join[C=C] 4,5
7. Select[D=5] 6
8. Select[D=5] 5

select * from R, S, T where R.B=S.B and S.C=T.C and R.A = 3 and T.D = 5
The Memo

select *
from R, S, T
where R.B = S.B
and S.C = T.C
and R.A = 3
and T.D = 5

R(A,B), S(B,C), T(C,D)
R(A,B), S(B,C), T(C,D)

The Memo

1. Scan R
2. Select[A=3] 1
3. Scan S
4. Join[B=B] 2, 3
5. Scan T
6. Join[C=C] 4, 5
7. Select[D=5] 6
8. Select[D=5] 5

Apply another rule

select * from R, S, T where R.B=S.B and S.C=T.C and R.A = 3 and T.D = 5
The Memo

1. Scan R
2. Select[A=3] 1
3. Scan S
4. Join[B=B] 2, 3
5. Scan T
6. Join[C=C] 4,5
7. Select[D=5] 6
8. Select[D=5] 5
9. Join[C=C] 3, 8

select * from R, S, T where R.B=S.B and S.C=T.C and R.A = 3 and T.D = 5

R(A,B), S(B,C), T(C,D)
Conclusions

• Query optimizers: some of the most complex systems in use today

Query optimization is not rocket science. If you fail at query optimization, they send you to build rockets.

Anonymous