Query feedback for correcting cardinality estimation

Maxence Ahlouche

DALIBO

HADAS - Traitement de Données et de Connaissances à Grande Echelle

Query optimization 101

Goal: find the cheapest way to retrieve data

Query

```
SELECT * FROM Actors
WHERE nationality = 'French'
```

Statistics

histogram and **most common values** for data distribution

Plan space enumeration
Dynamic programming to find all possible plans

→ List of execution plans
* Sequential scan
* Index scan

Selectivity estimations
Estimate cardinality of intermediate results

→ Selectivity of each predicate
'price > 100'
20% of input rows match

Cost estimation
Evaluate impact on CPU usage (O_{cpu}) and I/O access (O_{io})

→ Execution plan
Index scan

Cardinality: number of rows in a (temporary) table

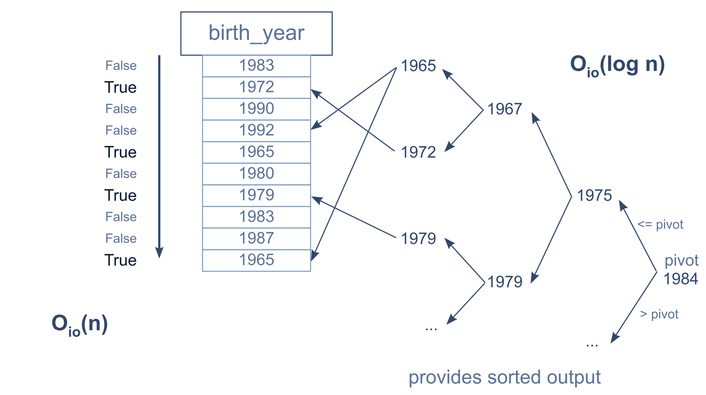
Selectivity: percentage of rows matching a predicate

cardinality(output) = cardinality(input) * selectivity(predicates)

Data access operators are not equivalent

```
SELECT * FROM Actor
WHERE birth_year < 1980;
```

Table scan | Index scan

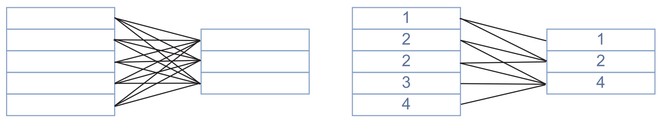


provides sorted output

Neither are **join operators**

```
SELECT Actor.name, Country.name
FROM Actor JOIN Country
ON Actor.nationality = Country.id;
```

Nested loop join | Merge join (requires sorted input)



provides sorted output

$O_{io}(n * m)$ | $O_{io}(n + m)$

$O_{cpu}(n * m)$ | $O_{cpu}(n + m)$

Impact on **execution plans costs**

```
SELECT Actor.name, Country.name
FROM Actor JOIN Country
ON Actor.nationality = Country.id
WHERE Country.id IN (1, 2, 3)
ORDER BY nationality;
```

Plan A

Table scan	Table scan
table: Actor	**table**: Country
predicate: nat IN (1,2,3)	**predicate**: id IN (1, 2, 3)
selectivity: 0.01	**selectivity**: 0.015
$O_{io}(n)$	$O_{io}(n)$
10k rows	3 rows

→ Nested loop join
predicate: Actor.nationality = Country.id
$O_{cpu}(n * m)$
→ 10k rows
→ Sort
column: Actor.nationality
$O_{cpu}(n * log\ n)$
→ output data

I/O operations: card(A) + card(C)

CPU operations: 10k * 3 + 10k * log 10k

Plan B

Index scan	Index scan
column: Actor.nat	**column**: Country.id
predicate: nat IN (1,2,3)	**predicate**: id IN (1, 2, 3)
selectivity: 0.05	**selectivity**: 0.015
O_{io}(log n)	O_{io}(log n)
10k rows, sorted	3 rows, sorted

→ Merge join
predicate: Actor.nationality = Country.id
$O_{cpu}(n + m)$
→ output data

I/O operations: log card(A) + log card(C)

CPU operations: 10k + 3

Same output data, drastic performance improvement

Sensitivity to cardinality estimates

How **bad cardinality estimates** happen

```
SELECT * FROM Actor          ← card(Actor) = 1M
JOIN Role ON [...] JOIN Movie ON [...]
WHERE Movie.country = 1      ← selectivity(p1) = 0.01
AND Actor.nationality = 1    ← selectivity(p2) = 0.01
```

strong correlation between p1 and p2 but most systems assume **data independence**

estimate output cardinality of the query:

$card(query)_{est} = card(Actor) * sel(p1) * sel(p2) = 100$
valid for uncorrelated data

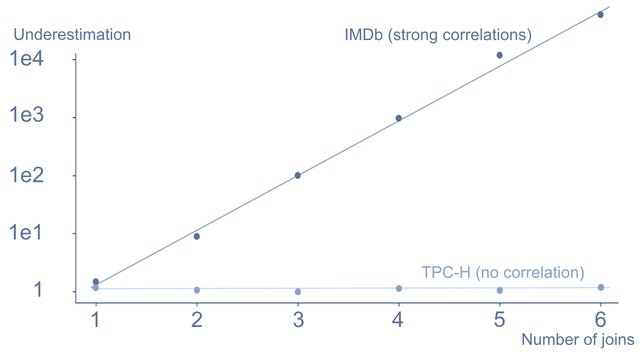
$card(query)_{real} = card(Actor) * sel(p1) = 10k$

underestimation factor: 100

Cardinality errors **propagate** through joins

underestimation = $card_{real}$ / $card_{est}$

Using a complex query with **many joins** and **lots of correlation** in data

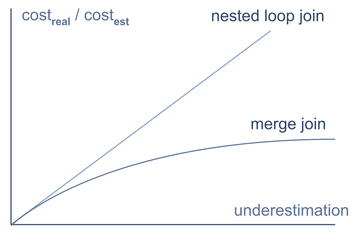


Notice the **log scale**

Operators sensitivity to cardinality errors

$card(A)_{est} = 100$, $card(B)_{est} = 1$
$card(A)_{real} = 100$, $card(B)_{real} = 100$
underestimation = 100

A nested loop join B O(n * m)	A merge join B O(n + m)
estimated cost = 100 * 1 = **100** (planner's choice)	estimated cost = 100 + 1 = **101**
actual cost = 100 * 1 * 100 = **10k**	actual cost = 100 + 1 * 100 = **200** (sensible choice)



Operators have different sensitivity to bad estimates

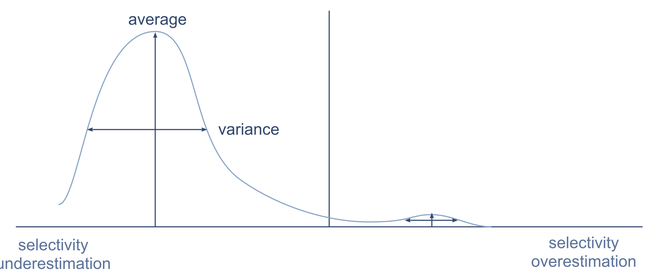
Query planners don't take them into account

Query feedback to the rescue

Patterns in estimation errors

Predicate: (columnA, operator, {columnB, value})

Hypothesis: predicates involving two correlated columns will show patterns in selectivity error



When relevant, adjust selectivity estimation

Relevance criteria:
- estimation error is always (±ε) on the same side
- variance is low

Possible error measurements:
- $| sel_{real} - sel_{est} |$
- sel_{real} / sel_{est} for underestimation
  sel_{est} / sel_{real} for overestimation

Estimation error stats from **query feedback**

Selectivity estimation is executed thousands of times per second: calculations need to be fast

For each predicate, store:
- number of executions n
- sum of selectivity errors $sum(e_i)$
- sum of squared errors $sum(e_i 2)$

Separate over- and under-estimation stats: finer distribution approximation, better decisions

Which allow to compute:
- $average(e) = 1/n * sum(e_i)$
- $variance(e) = avg(e)2 - 1/n * sum(e_i 2)$

Benefits:
- no need to store every single error
- fast online stat update: no need to precompute stats

Bonus:
A monitoring tool can track evolution of selectivity error on a column and detect stale data distribution statistics

Future work

- Evaluate **impact on performance** for OLTP and OLAP workloads
- Extend stats collection and selectivity adjustment to **pairs of (columnA, operator, value) predicates**
- **Automate refreshing the data distribution statistics** when they are detected stale

Bibliography

- **PostgreSQL source code and documentation**
  https://www.postgresql.org/docs/current/static/
  https://git.postgresql.org/gitweb/?p=postgresql.git
- Graefe, Goetz 1993
  Query Evaluation Techniques for Large Databases
  ACM Computing Surveys 25(2): 73–169
- Leis, Viktor, Andrey Gubichev, Atanas Mirchev, et al. 2015
  How Good Are Query Optimizers, Really?
  Proceedings of the VLDB Endowment 9(3): 204–215
- Wentao Wu, Yun Chi, Shenghuo Zhu, et al. 2013
  Predicting Query Execution Time: Are Optimizer Cost Models Really Unusable?
  IEEE 29th International Conference on Data Engineering Pp. 1081–1092

LIG