L9
Query Execution & Optimization

Eugene Wu

Fall 2015

Steps for a New Application

Requirements
- what are you going to build?

Conceptual Database Design
- pen-and-pencil description

Logical Design
- formal database schema

Schema Refinement
- fix potential problems, normalization

Physical Database Design — Optimization
- optimize for speed/storage

App/Security Design
- prevent security problems

Recall

Relational algebra
- equivalence: multiple stmts for same query
- some statements (much) faster than others

Which is faster?

a. $\sigma_{v=1}(R \times T)$

b. $\sigma_{v=1}(\sigma_{v=1}(R) \times T)$

|R| = |T| = 10 pages. 100? 1M?

unique values in R = 1. 100? 1M? ← selectivity!

Overview of Query Optimization

SQL → query plan

How plans are executed

Some implementations of operators

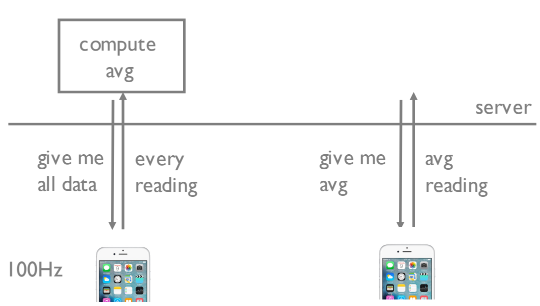
Cost estimation of a plan
- Selectivity

System R dynamic programming

All ideas from System R's "Selinger Optimizer" 1979

iPhones as a database

"avg acceleration over the past hour"



SQL → Query Plan

SELECT a FROM R	$\pi_a(R)$	π_a — R
SELECT a FROM R WHERE a > 10	$\pi_a(\sigma_{a>10}(R))$	π_a — $\sigma_{a>10}$ — R
SELECT a FROM R JOIN S ON R.b = S.b	$\pi_a(\bowtie_b(R,S))$	π_a — $\bowtie_b$ — R, S

Query Evaluation

Push vs Pull?

Push
- Operators are input-driven
- As operator (say reading input table) gets data, push it to parent operator.

Pull
- Operators are demand-driven
- If parent says "give me next result", then do the work

Are cursors push or pull?

Query Evaluation

Naïve execution (operator at a time)
- read R
- filter a> 10 and write out
- read and project a
- Cost: B + M + M

SELECT a
FROM R
WHERE a > 10

π_a
|
$\sigma_{a>10}$
|
R

B # *data* pages
M # pages matched in WHERE clause

Could we do better?

Query Evaluation

Pipelined exec (tuple/page at a time)
- read first page of R, pass to σ
- filter a > 10 and pass to π
- project a
- (all operators run concurrently)
- Cost: B

Note: can't pipeline some operators!
e.g., sort, some joins, aggregates
why?

SELECT a
FROM R
WHERE a > 10

π_a
|
$\sigma_{a>10}$
|
R

B # *data* pages
M # pages matched in WHERE clause

Query Evaluation

What if R is indexed?
- Hash index
  - Not appropriate
- B+Tree index
  - use a> 10 to find initial data page
  - scan leaf data pages
  - Cost: $\log_F B + M$

SELECT a
FROM R
WHERE a > 10

π_a
|
$\sigma_{a>10}$
|
R

B # *data* pages
M # pages matched in WHERE clause

Access Paths

Choice of how to access input data is called the Access Path
- file scan or
- index + matching condition (e.g., a > 10)

Access Paths

Sequential Scan
- doesn't accept any matching conditions

Hash index search key <a,b,c>
- accepts conjunction of equality conditions on *all* search keys
- e.g., a=1 and b = 5 and c = 5
- will (a = 1 and b = 5) work? why?

Tree index search key <a,b,c>
- accepts conjunction of terms of *prefix* of search keys
- e.g., a > 1 and b = 5 and c < 5
- will (a > 1 and b = 5) work?
- will (a > 1 and c > 9) work?

How to pick Access Paths?

Selectivity

ratio of # outputs satisfying predicates vs # inputs

0.01 means 1 output tuple for every 100 input tuples

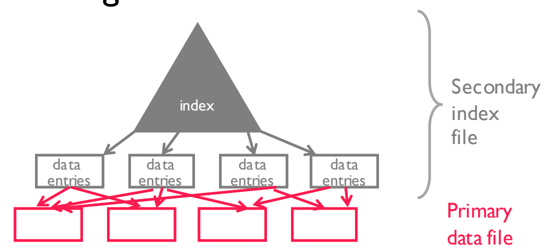
Assume

attribute selectivity is independent

if selectivity(a=1) = 0.1, selectivity(b>3) = 0.6

selectivity(a=1 and b>3) = 0.1*0.6 = 0.06

High level index structure



What is a data entry?

actual data record

<search key value, rid>

<search key value, rid_list>

How to pick Access Paths?

Hash index on <a, b, c>

a = 1, b = 1, c = 1 how to estimate selectivity?

1. pre-compute attribute statistics by scanning data
   e.g., a has 100 values, b has 200 values, c has 1 value
   selectivity = 1 / (100 * 200 * 1)
2. How many distinct values does hash index have?
   e.g., 1000 distinct values in hash index
3. make a number up
   "default estimate" is the fancy term

System Catalog Keeps Statistics

System R

NCARD	"relation cardinality" # tuples in relation
TCARD	# pages relation occupies
ICARD	# keys (distinct values) in index
NINDX	pages occupied by index

min and max keys in indexes

Statistics were expensive in 1979!

Super elegant: catalog stored in relations too!

What Optimization Options Do We Have?

Access Path ✔

Predicate push-down

Join implementation

Join ordering

In general, depends on operator implementations. So let's take a look

Predicate Push Down

SELECT a
FROM R
WHERE a > 10

(a): π_a — $\sigma_{a>10}$ — R

(b): $\sigma_{a>10}$ — π_a — R

Which is faster if B+ Tree index: (a) or (b)?

(a) $\log_F(B)$ + M pages

(b) B pages

It's a Good Idea, especially when we look at Joins

Projection with DISTINCT clause

need to deduplicate e.g., π_{rating} Sailors

Two basic approaches
- Sort: fundamental database operation
  - sort on rating, remove dups on scan of sorted data
- Hash:
  - partition into N buckets
  - sort each bucket and remove dups

Index on projected fields
- scan the index pages, avoid reading data

The Join

Core database operation
- join of 100 tables common in enterprise apps

Join algorithms is a large area of research
- e.g., distributed, temporal, geographic, multi-dim, range, sensors, graphs, etc
- Discuss three basic joins
  - nested loops, indexed nested loops, hash join

Best join implementation depends on the query, the data, the indices, hardware, etc

Datasets

```
from collections import defaultdict
from random import randint

M = 10000
N = 1000
outer = [ [randint(0, 1000), randint(0, 1000)]
          for i in xrange(M)]
inner = [ [randint(0, 1000), randint(0, 1000)]
          for i in xrange(N)]

# outer ⋈1 inner
# outer JOIN inner ON outer.1 = inner.1
```

Nested Loops Join

```
for row in outer:
    for irow in inner:
        if row[0] == irow[0]:    # could be any check
            yield (row, irow)
```

Very flexible
- Equality check can be replaced with any condition
- Incremental algorithm
- Cost: M + MN

Is this the same as a cross product?

Indexed Nested Loops Join

```
for row in outer:
    for irow in index.get(row[0], []):
        yield (row, irow)
```

Slightly less flexible
- Only supports conditions that the index supports
- Assume hash index, 50 tuples per page, equality join on <u>sid</u>, 5% of outer tuples match
- Cost: M + (50M * 0.05 * 1 page)

Sort Merge Join

Sort outer and inner tables on join key
- Cost: 2-3 scans of each table

Merge the tables and compute the join
- Cost: 1 scan of each table

Overall Properties
- cost: 3(M+N) to 4(M+N)
- results are sorted
- highly sequential access
- (weapon of choice for very large datasets)

Quick Recap

Single operator optimizations
- Access paths
- Primary vs secondary index costs
- Projection/distinct
- Predicate/project push downs

2 operators aka Joins
- Nested loops, index nested loops, sort merge

Selectivity estimation
- Statistics and simple models

Where we are

We've discussed
- Optimizations for a single operator
- Different types of access paths, join operators
- Simple optimizations e.g., predicate push-down

What about for multiple operators?
- System R Optimizer

Selinger Optimizer

Granddaddy of all existing optimizers
- don't go for best plan, go for *least worst plan*

2 Big Ideas
1. Cost Estimator
   - "predict" cost of query from statistics
   - Includes CPU, disk, memory, etc (can get sophisticated!)
   - It's an art
2. Plan Space
   - avoid cross product
   - push selections & projections to leaves as much as possible
   - only join ordering remaining

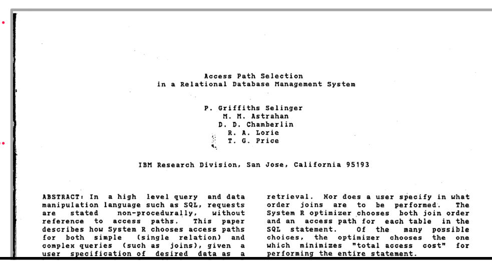
Selinger Optimizer

Granddaddy of all existing optimizers
- don't go for best plan, go for *least worst plan*

2 Big Ideas

1.

2.



Access Path Selection
in a Relational Database Management System

P. Griffiths Selinger
M. M. Astrahan
D. D. Chamberlin
R. A. Lorie
T. G. Price

IBM Research Division, San Jose, California 95193

ABSTRACT: In a high level query and data manipulation language such as SQL, requests are stated non-procedurally, without reference to access paths. This paper describes how System R chooses access paths for both simple (single relation) and complex queries (such as joins), given a user specification of desired data as a

retrieval. Nor does a user specify in what order joins are to be performed. The System R optimizer chooses both join order and an access path for each table in the SQL statement. Of the many possible choices, the optimizer chooses the one which minimizes "total access cost" for performing the entire statement.

ble

Cost Estimation

estimate(operator, inputs, stats) → cost

estimate cost for each operator
- depends on input *cardinalities* (# tuples)
- discussed earlier in lecture

estimate output size for each operator
- need to call estimate() on inputs!
- use selectivity. assume attributes are independent

Try it in PostgreSQL: `EXPLAIN <query>;`

Estimate Size of Output

```
SELECT   *
FROM     R1, …, Rn
WHERE    term1 AND … AND termm
```

Query input size
- $|R1| * \ldots * |Rn|$

Term selectivity

col = v	$1/ICARD_{col}$
col1 = col2	$1/\max(ICARD_{col1}, ICARD_{col2})$
col > v	$(\max_{col} - v) / (\max_{col} - \min_{col})$

Query output size
- $|R1| * \ldots * |Rn| * term_1 selectivity * \ldots * term_m selectivity$

Estimate Size of Output

```
Emp:    1000  Cardinality
Dept:   10  Cardinality

Cost(Emp  join  Dept)
```

Naïve

# total records	1000 * 10	= 10,000
Selectivity of Emp	1 / 1000	= 0.001
Selectivity of Dept	1 / 10	= 0.1
Join Selectivity	1 / max(1k, 10)	= 0.001
Output Card:	10,000 * 0.001	= 10

Key, Foreign Key join

Output Card: 1000

note: selectivity defined wrt cross product size

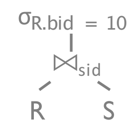
Try it out

R.sid = S.sid selectivity 0.01
R.bid selectivity 0.05
|R| = M
|S| = N

SELECT *
FROM R, S
WHERE R.sid = S.sid
AND R.bid = 10

Cost: M + MN
selection is pipelined

outputs: 0.0005MN



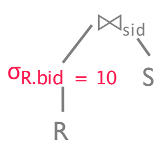
Try it out

R.sid = S.sid selectivity 0.01
R.bid selectivity 0.05
|R| = M
|S| = N

SELECT *
FROM R, S
WHERE R.sid = S.sid
AND R.bid = 10

Cost: ?????

outputs: 0.0005MN



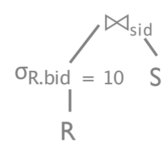
Try it out

R.sid = S.sid selectivity 0.01
R.bid selectivity 0.05
|R| = M
|S| = N

SELECT *
FROM R, S
WHERE R.sid = S.sid
AND R.bid = 10

Cost: M + (0.05MN)

outputs: 0.0005MN



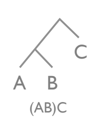
Selinger Optimizer

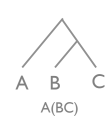
Granddaddy of all existing optimizers
don't go for best plan, go for *least worst plan*

- Cost Estimator
  - "predict" cost of query from statistics
  - Includes CPU, disk, memory, etc (can get sophisticated!)
  - It's an art
- Plan Space
  - avoid cross product
  - push selections & projections to leaves as much as possible
  - only join ordering remaining

Join Plan Space

A⋈B⋈C





How many plans?

(AB)C (AC)B (BC)A (BA)C (CA)B (CB)A
A(BC) A(CB) B(CA) B(AC) C(AB) C(BA)

$\#\text{ parenthetizations} * \underbrace{\#\text{strings}}_{N!}$

Join Plan Space

parenthetizations * #strings

A: (A)
AB: (AB)
ABC: ((AB)C), (A(BC))
ABCD: (((AB)C)D), ((A(BC))D), ((AB)(CD)), (A((BC)D)), (A(B(CD))
paren(n) choose(2(N-1), (N-1)) / N

(choose(2(N-1), (N-1)) / N) * N!

N=10 #plans = 17,643,225,600

The Art of Computer Programming, Volume 4A, page 440-450

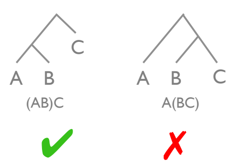
Selinger Optimizer

Simplify the set of plans so it's tractable and ~ok

1. Push down selections and projections
2. Ignore cross products (S&T don't share attrs)
3. Left deep plans only
4. Dynamic programming optimization problem
5. Consider interesting sort orders

Selinger Optimizer

parens(N) = 1
Only left-deep plans
ensures pipelining



Dynamic Programming
Idea: If considering ((ABC)DE)
compute best (ABC), cache, and reuse
figure out best way to combine with (DE)

```
Dynamic Programming Algorithm
   compute best join size 1, then size 2, …
```
~O(2^N)

Reducing the Plan Space

```
Dynamic Programming Algorithm
    compute best join size 1, then size 2, …

    R = relations  to join
    N = |R|
    for i in {1,… N}
       for S in {all size i subsets of R}
          bestjoin(S) = S-A join A
             where A is single relation that minimizes:
                cost(bestjoin(S-A)) +
                min cost to join A to (S-A) +
                min access cost for A
```

Selinger Algorithm i = 1

bestjoin(ABC), only nested loops join

i = 1
A = best way to access A
B = best way to access B
C = best way to access C

cost: N relations

Selinger Algorithm i = 2

bestjoin(ABC)

i = 2
A,B = (A)B or (B)A
A,C = (A)C or (C)A
B,C = (B)C or (C)B

cost: choose(N, 2) * 2

Selinger Algorithm i = 3

bestjoin(ABC)

i = 3
A,B,C = bestjoin(BC)A or
bestjoin(AC)B or
bestjoin(AB)C

cost: choose(N, 3) * 3

Selinger Algorithm Cost

cost = # subsets * # options per subset
set of relations R
N = |R|

#subsets = choose(N, 1) + choose(N,2) + choose(N,3)…
= 2^N

#options = k<N ways to remove a relation A +
1 way to join A with R-A (if only NLJ)
< N

Cost = $N*2^N$
N = 12 49152

Summary

Single operator optimizations
- Access paths
- Primary vs secondary index costs
- Projection/distinct
- Predicate/project push downs

2 operators aka Joins
- Nested loops, index nested loops, sort merge

Full plan optimizations
- Naïve vs Selinger join ordering

Selectivity estimation
- Statistics and simple models

Summary

Query optimization is a deep, complex topic
Pipelined plan execution
Different types of joins
Cost estimation of single and multiple operators
Join ordering is hard!