L21
Query Execution & Optimization
Continued

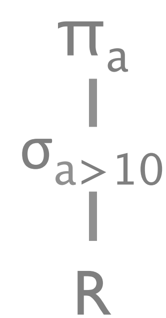
SQL → Query Plan

SELECT a FROM R

$\pi_a(R)$

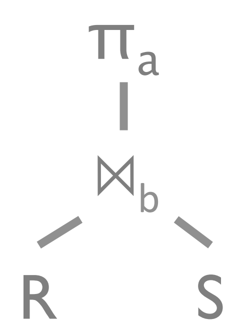
SELECT a FROM R
WHERE a > 10

$\pi_a(\sigma_{a>10}(R))$



SELECT a
FROM R JOIN S
ON R.b = S.b

$\pi_a(\bowtie_b(R,S))$



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

$$\begin{array}{c}\pi_a \\ | \\ \sigma_{a>10} \\ | \\ R\end{array}$$

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

```
SELECT a
FROM R
WHERE a > 10
```

π_a
|
$\sigma_{a>10}$
|
R

B # *data* pages

M # pages matched in WHERE clause

Note: can't pipeline some operators!

e.g., sort, some joins, aggregates

why?

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

$$\pi_a$$
$$|$$
$$\sigma_{a>10}$$
$$|$$
$$R$$

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

typically best with equality on all but last column

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

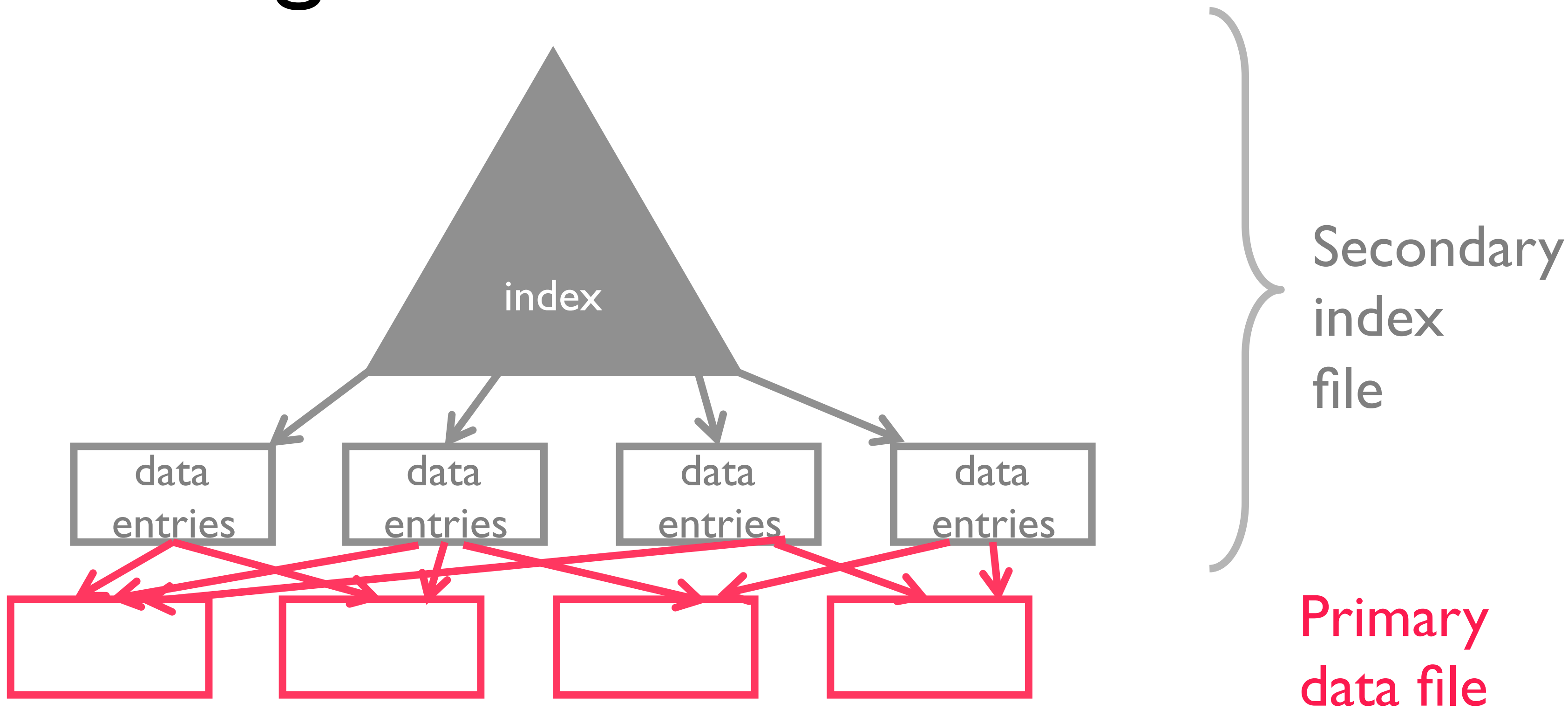
Assume

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!

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

$$\pi_a \;|\; \sigma_{a>10} \;|\; R$$

(a)

$$\sigma_{a>10} \;|\; \pi_a \;|\; R$$

(b)

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
  - remove duplicates on insert

Index on projected fields

- scan the index pages, avoid reading data

The Join

Core database operation

- join of 10s of 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

Nested Loops Join:

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

Nested Loops Join

What this means in terms of disk IO

tableA join tableB; tableA is “outer”; tableB is “inner”
M pages in tableA, N pages in tableB, T tuples per page

$M + T \times M \times N$

for each tuple *t* in tableA, (M pages, TM tuples)
    scan through each page *pi* in the inner (N pages)
        compare all the tuples in *pi* with *t*

Nested Loops Join: Order?

Does order matter?

$M + T \times M \times N$

$N + T \times N \times M$

Scan “outer” once; Scan “inner” multiple times:

If inner is small IO cost is M + N!

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

Indexed Nested Loops Join

What this means in terms of disk IO

outer join inner on sid
M pages in outer, N pages in inner, T tuples/page
inner has primary key index on sid
Cost of looking up in index is C_I
predicate on outer has 5% selectivity

$M + T \times M \times 0.05 \times C_I$

```
for each tuple t in the outer:  (M pages, TM tuples)
     if predicate(t):  (5% of tuples satisfy pred)
          lookup_in_index(t.sid) (C_I disk IO)
```

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)

Sort Merge Join

What does this mean in terms of disk IO?

R join T on sid

R has M pages, T has N pages, 50 tuples/page

Assume sort takes 3 scans, merge takes 1 scan

$3 * M + 1 * M + 3 * N + 1 * N$

(note, tuples/page didn't matter)

Join Cost Summary

\# tuples (S) = N_s

\# tuples (T) = N_T

\# pages (S) = P_S

\# pages (T) = P_T

\# index values (S) = I_S

\# index values (T) = I_T

Secondary index on T.id

Height of index = H

S NLJ T

$P_S + N_S \times P_T$

T NLJ S

$P_T + N_T \times P_S$

S INLJ T

$P_S + N_S \times$ (index cost)

index cost:

H + # leaf pages + # tuples

S SM T

$3 \times (P_S + P_T)$

Quick Recap

Single relation operator optimizations

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

2 relation 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.

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

```
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

note: selectivity defined wrt cross product size

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

(AB)C A(BC)

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

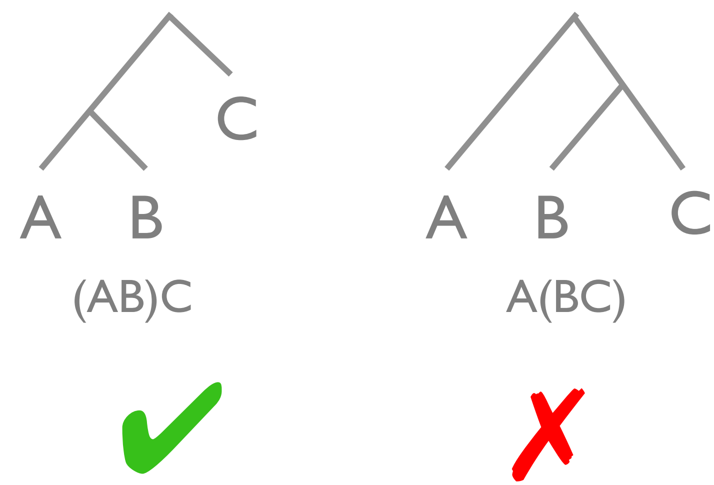
1. Push down selections and projections
2. Ignore cross products (S&T don't share attrs)
3. Left deep plans only
4. Dynamic programming optimization problem
5. Consider interesting sort orders

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, …
```

~$O(N*2^N)$

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!

You should understand

Estimate query cardinality, selectivity

Apply predicate push down

Given primary/secondary indexes and statistics,

- pick best index for access method + est cost
- pick best index for join + est cost
- pick best join order for 3 tables
- pick cheaper of two execution plans