Administrivia

Extensions

- Project 1 part 3 due 11/14 11:59PM EST
- Project 1 part 3 meetings: 11/14 – 11/21
- HW4, Project 2: 11/31 11:59PM EST

Administrivia

EC2: Data as Art. Due 12/2 11:59PM EST

Your submission should be ORIGINAL. Submitting existing content can result in 0 credit and may violate academic honesty policies

Extra Credit: Data as Art #405

Eugene Wu STAFF
1 minute ago in Social

UNPIN STAR WATCHING 2 VIEWS

Reply to this post with a meme, story, poem, or other artistic expression of a concept or essence of data management from this semester.

Extra credit: up to 1%

Grading criteria (in order of importance)

- Originality
- How much it captures the essence of the data management concept (should not need explanation)
- Quality of execution
- Number of likes

Query Execution & Optimization

Eugene Wu

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
- optimize for speed/storage

Optimization

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

What if

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

unique values of R.v: 1? 100? 1M? ← selectivity!

Overview of Query Optimization

SQL → query plan

How plans are executed

Some implementations of operators

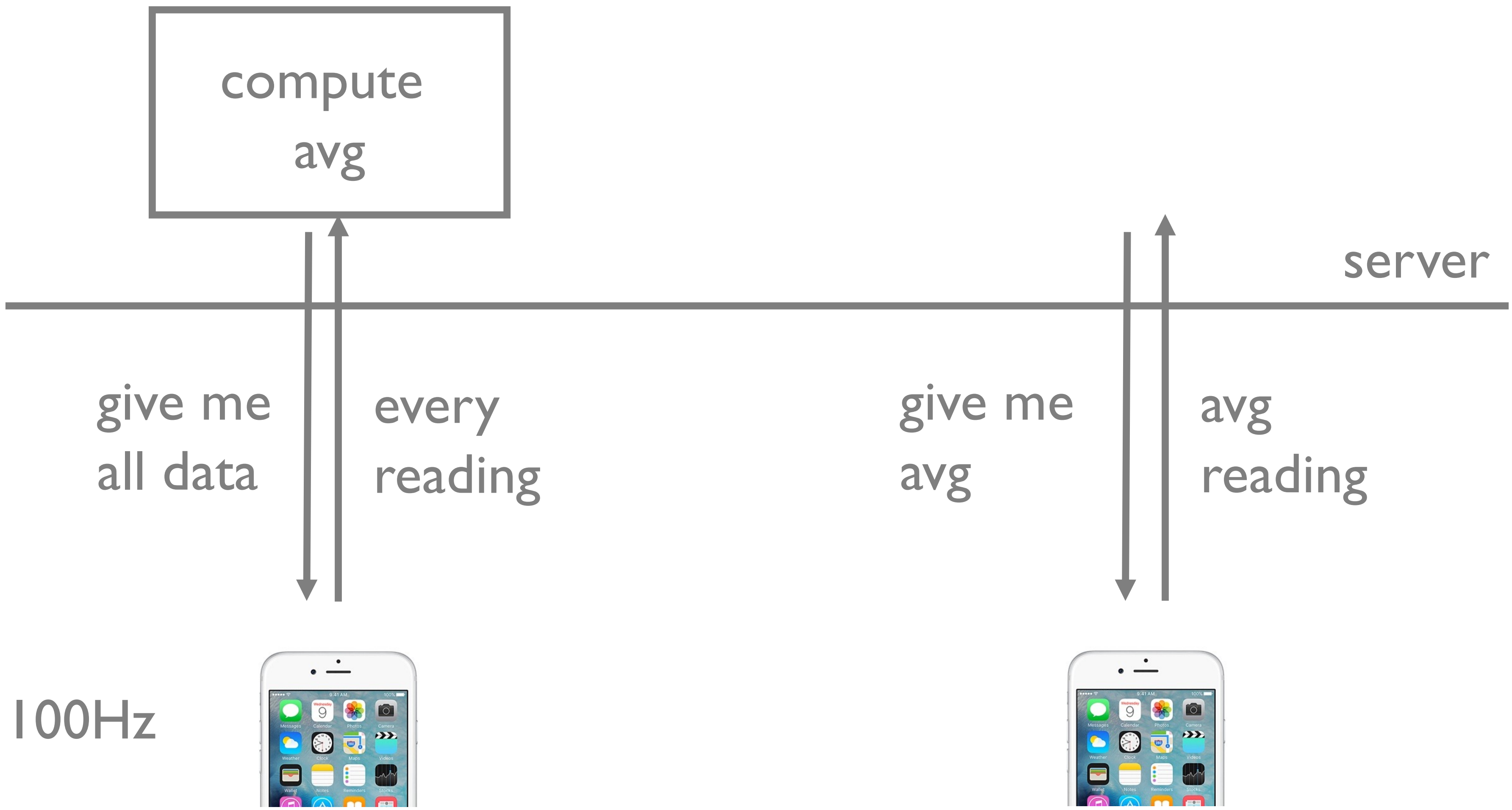
Cost + Selectivity estimation of a plan

System R dynamic programming

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

iPhones as a database

"avg acceleration over the past hour"



WARNING!

Confusingly, the logical operators in a query plan use the same symbols as relational algebra operators BUT

- Relational algebra uses set semantics
- Logical operators (such as in this lecture) use bag/multiset semantics

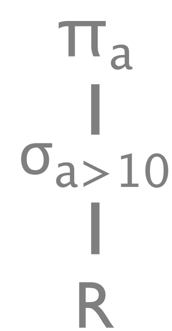
SQL → Query Plan

SELECT a FROM R

$\pi_a(R)$

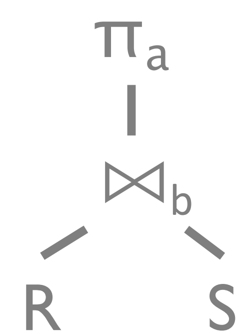
SELECT a FROM R
WHERE a > 10

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



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

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



Query Evaluation

Push vs Pull?

Push (e.g., a river)

- Operators are input-driven
- As operator (say reading input table) gets data, push it to parent operator.
- Often used in streaming systems

Pull (e.g., a straw)

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

Are cursors push or pull?

Query Evaluation

Op at a time

- read R
- filter a>10 and write out
- read and project a
- projected results send to user
- Cost: B + M + M

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

B # *data* pages

M # pages matched in WHERE clause

Query Evaluation

Pipelined exec (at page granularity)

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

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

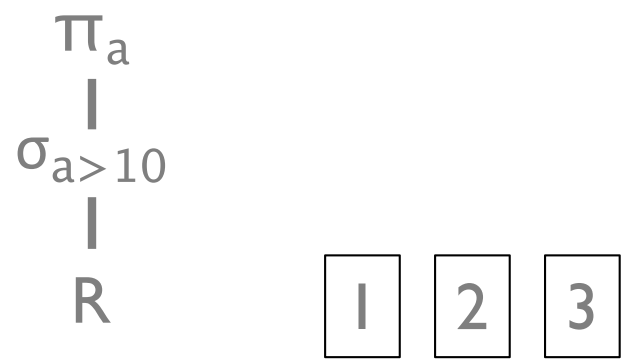
B # *data* pages

M # pages matched in WHERE clause

Query Evaluation

Pipelined exec (at page granularity)

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



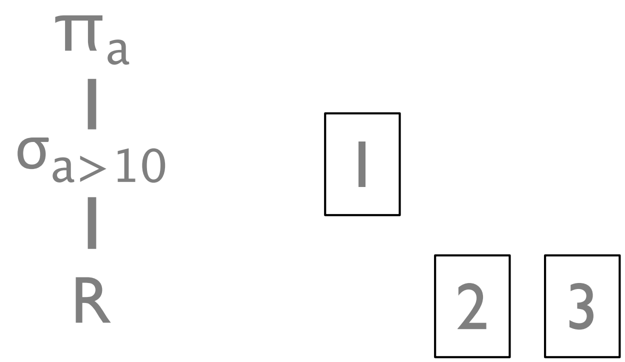
B # *data* pages

M # pages matched in WHERE clause

Query Evaluation

Pipelined exec (at page granularity)

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



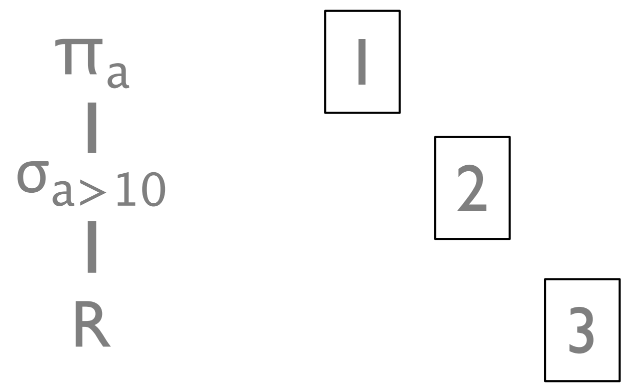
B # *data* pages

M # pages matched in WHERE clause

Query Evaluation

Pipelined exec (at page granularity)

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



B # *data* pages

M # pages matched in WHERE clause

Query Evaluation

Pipelined exec (at page granularity)

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

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

$\boxed{2}$

$\boxed{3}$

B # *data* pages

M # pages matched in WHERE clause

Query Evaluation

Pipelined exec (at page granularity)

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

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

3

B # *data* pages

M # pages matched in WHERE clause

Query Evaluation

Pipelined exec (at page granularity)

- 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

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: $\log_F B + M$

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

B # *data* pages

M # pages matched in WHERE clause

Push vs Pull?

What are the typical tradeoffs?

Pull

pro: easy to pipeline

con: more complexity, usually higher latency

Push

pro: vectorization, batching, simpler logic

con: hard to control rate of data

usually good for streaming

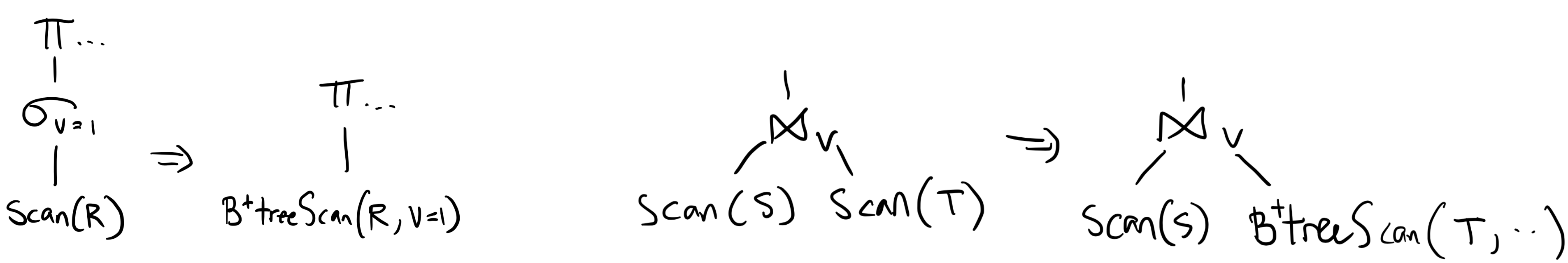
Access Paths

Access Path: how to access input data

file scan or

index + matching condition (e.g., a > 10)

Based on whether there is a "filter" operator **directly above** the Scan operator



Access Paths

Sequential Scan

doesn't accept any matching conditions

Hash index on <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?

Tree index on <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

Let:

a=1 has 0.1 selectivity

b>3 has 0.6 selectivity

What is selectivity of a=1 & b>3

0.1* 0.6 = 0.06

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)

Access Path selection looks at operator right above the Scan.
Thus, move filters close to Scan (change (b)→(a))

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

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

B # *data* pages
M # pages matched

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

The Join

Core database operation

- join of 100+ tables common in enterprise apps

Join algorithms is a large area of research

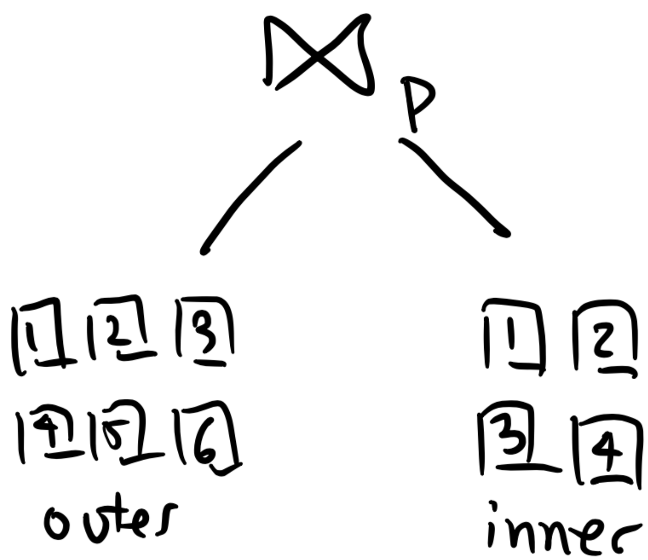
- e.g., distributed, temporal, geographic, multi-dim, range, sensors, graphs, etc
- Discuss three common join implementations
  - nested loops, indexed nested loops, hash join

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

Basic Join Algorithms

Costs for: `outer JOIN inner on p`

- Nested Loops Join
- Index Nested Loops Join
- Hash Join

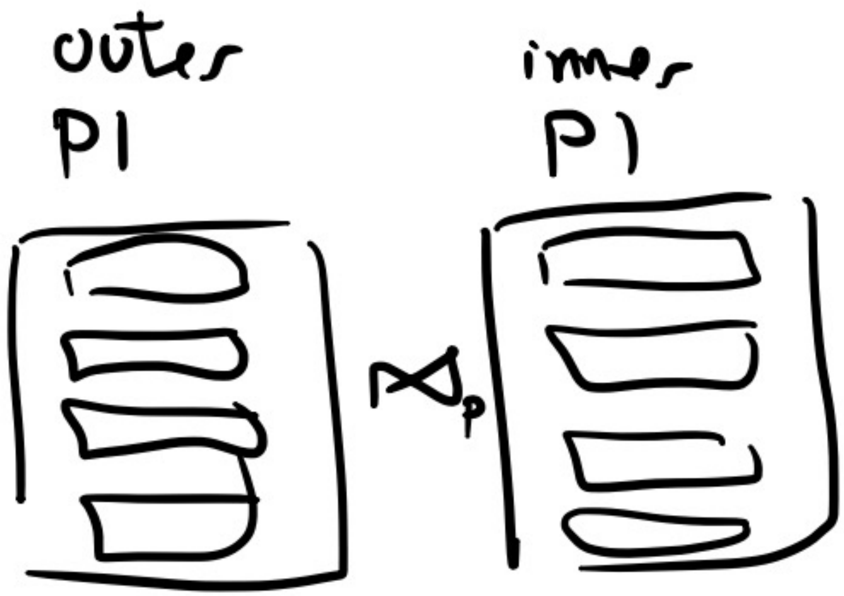


Prelim: Joins between two *pages*

Suppose we have one page of records from each join table

opage	outer relation
ipage	inner relation

If both pages in memory, the join itself is “free” in terms of disk costs



Prelim: Joins between two *pages*

Suppose we have one page of records from each join table

opage	outer relation
ipage	inner relation

If both pages in memory, the join itself is "free" in terms of disk costs

```
def join2pages(opage, ipage):
    for orow in opage:
        for resulttuple in joinrow(orow, ipage):
            yield resulttuple

def joinrow(orow, ipage):
    for irow in ipage:
        if orow.p == irow.p:
            yield (orow, irow)
```

Prelim: Joins between two *pages*

`join2pages()` will be our "atomic operation"

- considered "free" because runs in memory
- other join algorithms will call `join2pages()`

NLJ: Nested Loops Join

```
for opage in outer:                 # M pages from disk
    for ipage in inner:             # N pages from disk per opage
        join2pages(opage, ipage)
```

M pages in outer, N pages in inner, T tuples per page

Very flexible

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

Contrast with cross product?

INLJ: Indexed Nested Loops Join

```
for opage in outer:                        # M pages from disk
   for orow in opage:                      # in memory
      for ipage in index.get(orow.p):      # read from disk
         joinrow(orow, ipage)
```

inner is already indexed on join attribute *p*

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

$M + T * M * 0.05 * C_I$

HJ: Basic Hash Join

```
index = initialize hash index
for ipage in inner:     # N pages
    for irow in ipage:
        index.insert(irow.p, irow)

for opage in outer:     # M pages
    for orow in opage:
        for irow in index.get(orow.p):
            yield (row, irow)
```

Build **secondary** hash index in memory

INL Join

Less Flexible

- Equality joins
- M pages in outer, N pages in inner, T tuples/page
- Hash table in mem, assume no overflow pages→1 lookup to get tuple

Cost: N + M + (T * M) * 1

Join Cost Summary for S join T assuming B+ index

NCARD(S) = N_s

NCARD(T) = N_T

NPAGES(S) = P_S

NPAGES(T) = P_T

ICARD(S) = I_S

ICARD(T) = I_T

Height of index = H

total # data pages depends on primary vs secondary index

S NLJ T

$P_S + P_S * P_T$

S INLJ T

$P_S + N_S$ * (lookup cost)

S HJ T

$P_T + P_S + N_S$ * (lookup cost)

lookup cost:

H + # data pgs (+ # pointers)

data pgs:

selectivity * total # data pages

Quick Recap

Single relation operator optimizations

- Access paths
- Primary vs secondary index costs
- Predicate (Filter) push downs

2 relation operators aka Joins

- Nested loops, index nested loops, basic hash join

Selectivity estimation

- Statistics and simple models

Next:

- multi-operator plan optimization!

Adaptive Optimization of Very Large Join Queries

Thomas Neumann
Technische Universität München
neumann@in.tum.de

Bernhard Radke
Technische Universität München
radke@in.tum.de

Worst plan can be 100,000,000x slower!

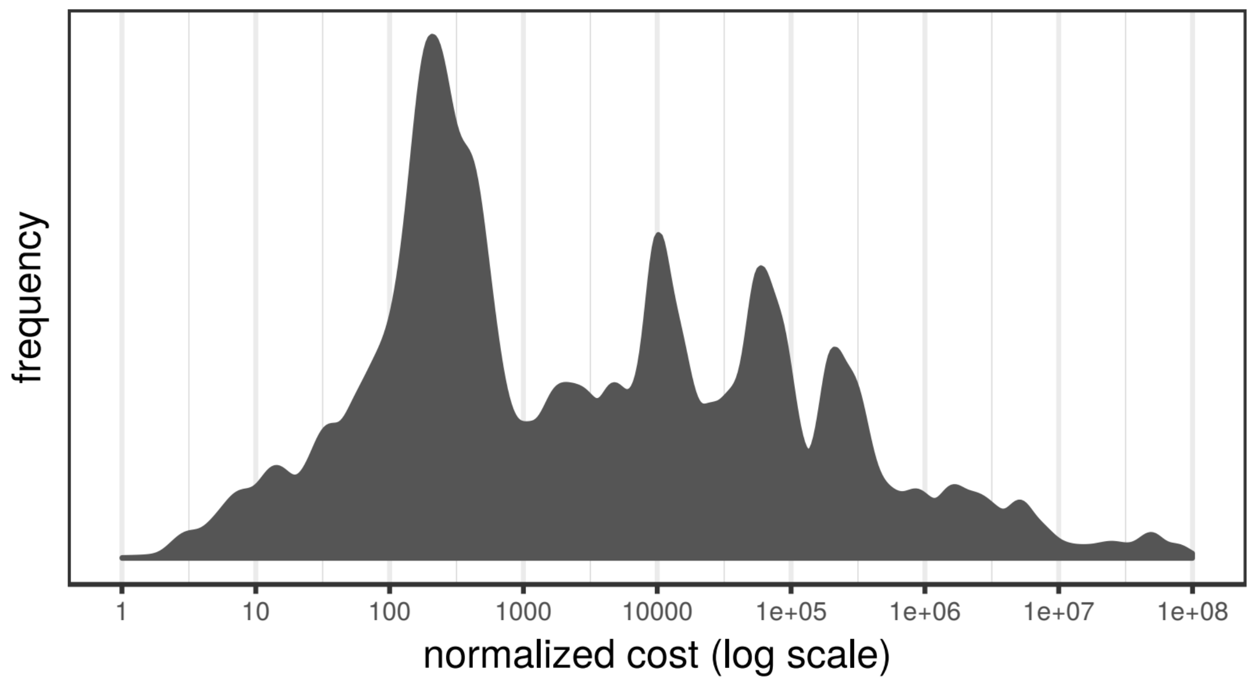


Figure 1: Normalized Cost Distribution of Random Plans for a Data-Warehouse-Style Query with 50 Relations

Selinger Optimizer

Origin 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

Origin 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

```
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)
```

In general

# 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

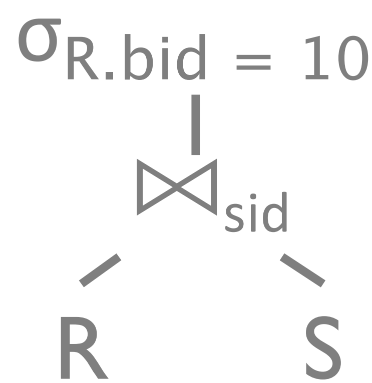
Try it out

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

Cost: M + MN
selection is pipelined

outputs: 0.0005MN

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



Try it out

R.sid = S.sid selectivity 0.01

R.bid selectivity 0.05

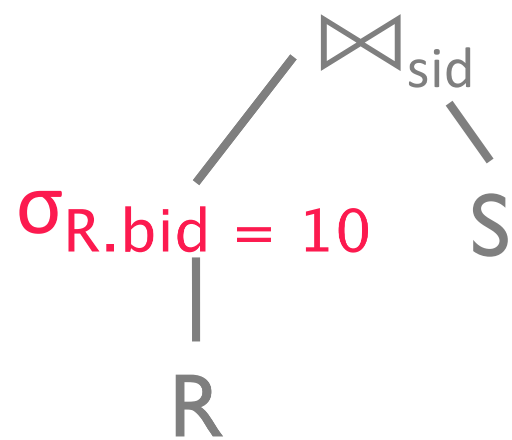
|R| = M

|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

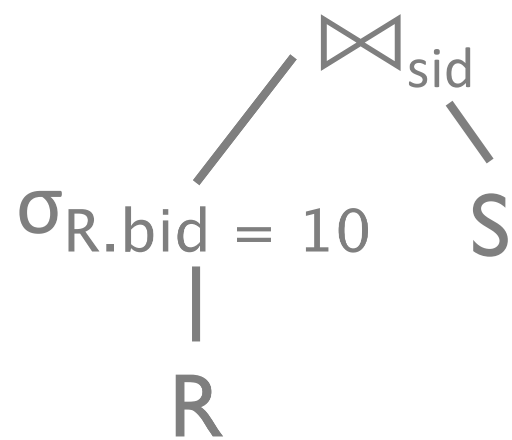
|R| = M

|S| = N

Cost: M + (0.05MN)

outputs: 0.0005MN

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



Selinger Optimizer

Granddaddy of all existing optimizers

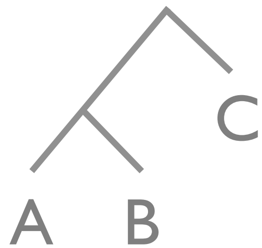
- don't go for best plan, go for *least worst plan*

2 Big Ideas

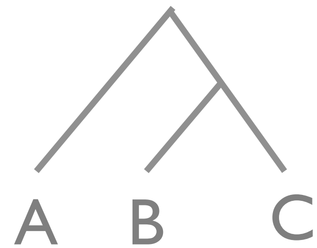
1. Cost Estimator
   - "predict" cost of query from statistics
   - Includes CPU, disk, memory, etc (can get sophisticated!)
   - It's an art
2. Plan Space
   - avoid cross product
   - push selections & projections to leaves as much as possible
   - only join ordering remaining

Join Plan Space

$A \bowtie B \bowtie C$



(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)

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

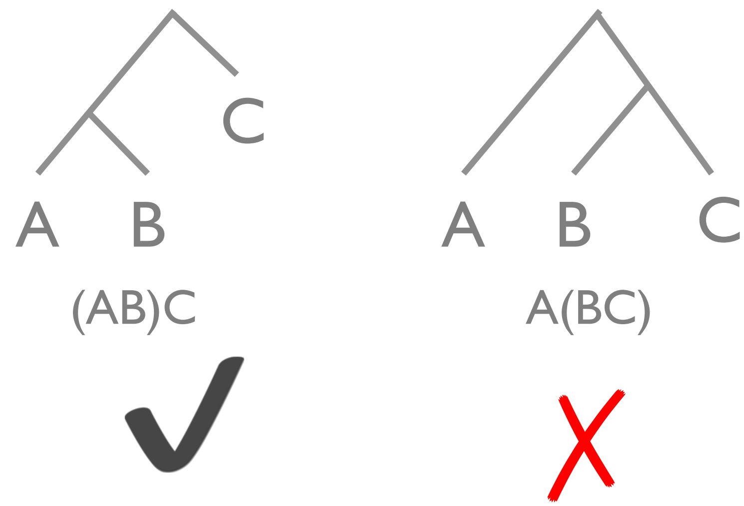
1. Push down selections and projections
2. Ignore cross products (S&T don't share attrs)
3. Left deep plans only
4. Dynamic programming optimization problem
5. Consider interesting sort orders (ignored in this class)

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, …
```

~$O(N*2^N)$

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}   # from join size 1 to join size N
        for S in {all size i subsets of R}
            bestjoin(S) = S-A join A
            # A is relation that minimizes the join cost:
            #  use bestjoin(S-A) as the outer relation
            #  min cost join algo of (S-A) with A using
            #  minimum access cost for A
            #  calculate for every possible A, pick the best
```

Selinger Algorithm i = 1

bestjoin(ABC), only nested loops join

i = 1

A = ways to access A

B = ways to access B

C = ways to access C

cost: N relations

Selinger Algorithm i = 2

bestjoin(ABC), only nested loops join

i = 2

A,B = bestjoin(A)B or bestjoin(B)A

A,C = bestjoin(A)C or bestjoin(C)A

B,C = bestjoin(B)C or bestjoin(C)B

cost: choose(N, 2) * 2

Selinger Algorithm i = 3

bestjoin(ABC), only nested loops join

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 subsets to be inner relation (right side) *
J join algorithms (NL, INL, ...)
< J*N

Cost = $J*N*2^N$
N = 12   49152   # if only using INL

Summary

Single operator optimizations

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

2 operators aka Joins

- Nested loops, index nested loops

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