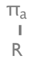
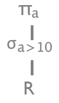
L20
Query Execution & Optimization Continued

SQL → Query Plan

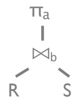
SELECT a FROM R $\pi_a(R)$



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

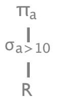
Are cursors push or pull?

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



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

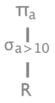
Could we do better?

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



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

Note: can't pipeline some operators!

e.g., sort, some joins, aggregates

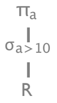
why?

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



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?

Depends on # data pages we would need to read

Selectivity

ratio of # tuples that satisfy predicates vs # inputs

0.01 means 1 output tuple for every 100 input tuples

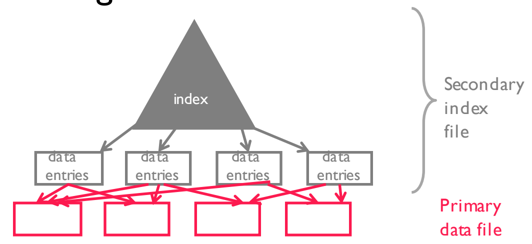
Assume

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 as 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

π_a
|
$\sigma_{a>10}$
|
R

(a)

$\sigma_{a>10}$
|
π_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 × M × 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)
```