Administrivia

HW 3 was due today

Project 2 out
HW 4 out (tonight or tomorrow)

L21
Query Execution & Optimization Continued

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)	(b)
π_a	$\sigma_{a>10}$
$\sigma_{a>10}$	π_a
R	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
  - 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.attr == irow.attr:    # 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

A join B
- A is outer. M pages
- B is inner. N pages
- 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

A join B on sid
- M pages in A
- N pages in B
- T tuples/page

Primary B+index on B(sid)
Cost of looking up in index is C_I

$M + T \times M \times C_I$

for each tuple t in the outer: (M pages,TM tuples)

lookup_in_index(t.sid) (C_I disk IO)

Indexed Nested Loops Join

What this means in terms of disk IO

A join B on sid
- M pages in A
- N pages in B
- T tuples/page

Primary B+index on B(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 A: (M pages,TM tuples)
- if predicate(t): (5% of tuples satisfy pred)
  - lookup_in_index(t.sid) (C_I disk IO)

(Simple) Hash Join

Type of index Nested Loops Join; When no index on inner table
A join B on sid
  M pages in A
  N pages in B
  T tuples/page
  Cost of looking up in index is C_I
  predicate on outer has 5% selectivity

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

```
index = build_hash_table(B)          (N pages)
for each tuple t in the A:           (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
Index on T.id
Height of index = H
(note: # leaf pages & height may differ for primary and secondary trees!)

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)

Primary B+ index cost:
H + # leaf pages

Secondary B+ index cost
H + # leaf pages + # tuples

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

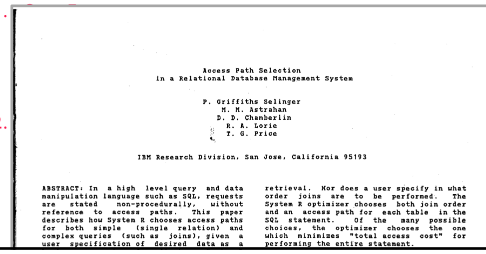
Granddaddy of all existing optimizers

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.

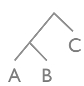
Cost Estimation

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

Note: estimate wrong if this is a key/fk join on emp.did = dept.did: 1000 results

Selinger Optimizer

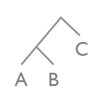
Granddaddy of all existing optimizers

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

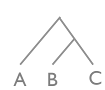
- Cost Estimator
  - "predict" cost of query from statistics
  - Includes CPU, disk, memory, etc (can get sophisticated!)
  - It's an art
- Plan Space
  - avoid cross product
  - push selections & projections to leaves as much as possible
  - only join ordering remaining

Join Plan Space

A⋈B⋈C



(AB)C



A(BC)

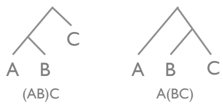
How many plans?

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

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

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)

parenthetizations * #strings

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

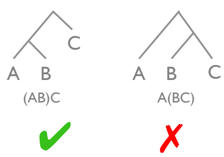
Selinger Optimizer

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 cheaper of two execution plans

Transactions, Concurrency, Recovery

Transfer $1000 from Evan to Neha

Check if Evan has $1000

Evan's Account -= $1000

Neha's Account += $1000

Transfer $1000 from Evan to Neha

Check if Evan has $1000

Evan's Account -= $1000

~~Neha's Account += $1000~~

Program crash or user presses cancel:

Money disappeared

Transfer $1000 from Evan to Neha

Check if Evan has $1000

~~Evan's Account -= $1000~~

~~Neha's Account += $1000~~

OOPS! Not enough money

Two transfers: Starting with $1500

Check if Evan has $1000

    Check if Evan has $1000

Evan's Account -= $1000

    Evan's Account -= $1000

Neha's Account += $1000

    Eugene's Account += $1000

Negative balance!

Transactions

Sequence of actions treated as a single unit

Atomicity: Apply all changes or none

- ("atomic" because it is indivisible)
- Solves the crash problem

Isolation: Illusion that each transaction executes sequentially, without concurrency

Transaction Guarantees

Atomicity
"all or nothing": All changes applied, or none are
users never see in-between transaction state

Consistency
database always satisfies Integrity Constraints
Transactions move from valid database to valid database

Isolation:
from transaction's point of view, it's the only one running

Durability:
if transaction commits, its effects *must persist*

Transactions

Transaction: a sequence of actions
action = read object, write object, commit, abort
API between app semantics and DBMS's view

User's view

T1: begin	A=A+100	B=B-100	END
T2: begin	A=A-50	B=B+50	END

DBMS's logical view

T1: begin	r(A) w(A)	r(B) w(B)	END
T2: begin	r(A) w(A)	r(B) w(B)	END

Concepts

Concurrency Control
techniques to ensure correct results when running transactions concurrently

what does this mean?

Recovery
On crash or abort, how to get back to a consistent (correct) state?

The two are intertwined!

What is Correct?

Serializability
Regardless of the interleaving of operations, result same as a serial ordering

Schedule
One specific interleaving of the operations

Serial Schedules

Logical xacts
T1: r(A) w(A) r(B) w(B) (e.g. A=A+100; B=B-100)
T2: r(A) w(A) r(B) w(B) (e.g. A=A*1.5; B=B*1.5)

No concurrency (serial 1)
T1: r(A) w(A) r(B) w(B)
T2: r(A) w(A) r(B) w(B)

No concurrency (serial 2)
T1: r(A) w(A) r(B) w(B)
T2: r(A) w(A) r(B) w(B)

Are serial 1 and serial 2 equivalent?

More Example Schedules

Logical xacts e.g. A=0 B=0
T1: r(A) w(A) r(A) w(B) e.g. A=A+1; B=A+1
T2: r(A) w(A) r(B) w(B) e.g. A=A+10; B=B+1

Concurrency (bad)
T1: r(A) w(A) r(A) w(B)
T2: r(A) w(A) r(B) w(B)

Concurrency (same as serial T1, T2!)
T1: r(A) w(A) r(A) w(B)
T2: r(A) w(A) r(B) w(B)

Concepts

Serial schedule

One transaction at a time. no concurrency.

Equivalent schedule

the database state is the same at end of both schedules

Serializable schedule (gold standard)

equivalent to a serial schedule