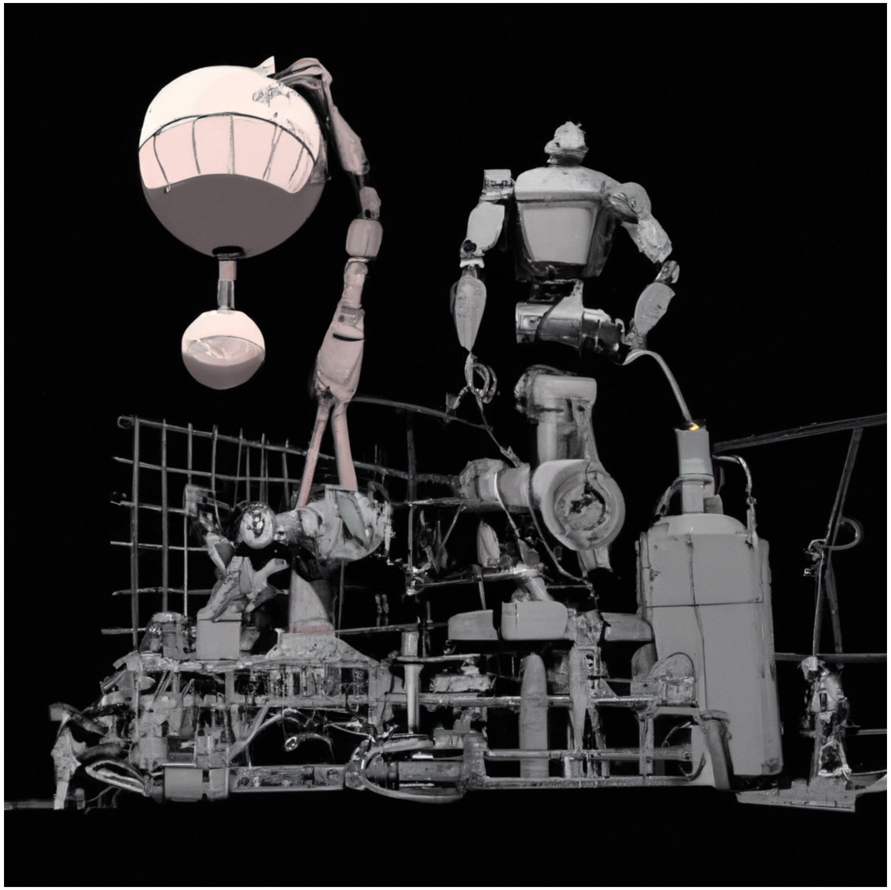
6.5830 Lecture 8



Query Optimization

September 30, 2024

Recap: Basic Join Summary

	CPU Complexity	I/O Complexity	Notes
Nested loops	{R} x {S}	|S| + {S}|R| *R doesn't fit in memory* |S| + |R| *R fits in memory*	Choice of inner / outer matters when R fits in memory and S doesn't
Blocked nested loops	{R} x {S}	+ |R|	Better to partition R (fewer passes)
Index nested loops	{R} x D *D is tree depth, < ~5*	{R} x D *I/O random unless R sorted & index clustered on join attr*	Assuming index on S.
Hash join	{R} + {S}	|R| + |S|	R must fit in memory (if index on R)
Blocked hash join		+ |R|	
Sort merge join	{R}log{R} + {S}log{S} + {S} + {R}	|R| + |S|	Assumes both tables fit in memory; If already sorted, can avoid logn step

Clicker (http://clicker.mit.edu/6.5830)

When would you prefer sort-merge over hash join? (Select all valid statements)

A) When the inner table is only a few records large
B) For an equi-join and when both tables are already sorted
C) For non-equi joins
D) When the hash table does not fit into main memory
E) Sort-merge joins are always better as the IO complexity is only O(*n log n)*

Clicker (http://clicker.mit.edu/6.5830)

When would you prefer a nested loop join over a hash join? (Select all valid statements)

A) When the inner table is only a few records large
B) For an equi-join and when both tables are already sorted
C) For a non-equi join
D) When the hash table does not fit into main memory
E) Nested loop joins are always better

Recap: Questions

- When would you prefer sort-merge over hash join?
- When would you prefer index-nested-loops join over hash join?

Join Processing in Database Systems with Large Main Memories

LEONARD D. SHAPIRO
North Dakota State University

“External” Sort Merge Join

Equi-join of two tables S & R

|S| = Pages in S; {S} = Tuples in S

|S| ≥ |R|

M pages of memory; M > sqrt(|S|)

Algorithm:

- Partition S and R into memory sized sorted runs, write out to disk
- Merge all runs simultaneously

Total I/O cost: Read |R| and |S| twice, write once

3(|R| + |S|) I/Os

Example

R=1,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R1

S1

If each run is M pages and M > sqrt(|S|), then there are at most

|S|/sqrt(|S|) = sqrt(|S|)

runs of S

So if |R| = |S|, we actually need M to be 2 x sqrt(|S|)

[handwavy argument in paper for why it's only sqrt(|S|)]

						OUTPUT
1						
3						
4	14	11	7	12	15	

Need enough memory to keep 1 page of each run in memory at a time

Example

R=1,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R1 = 1,3,4 R2 = 6,9,14 R3 = 1,7,11

S1 = 2,3,7 S2 = 8,9,12 S3 = 4,6,15

R1	R2	R3	S1	S2	S3
1	6 ←	1	2 ←	8 ←	4 ←
3 ←	9	7 ←	3	9	6
4	14	11	7	12	15

OUTPUT

Example

R=1,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R1 = 1,3,4　　R2 = 6,9,14　　R3 = 1,7,11

S1 = 2,3,7　　S2 = 8,9,12　　S3 = 4,6,15

R1	R2	R3	S1	S2	S3
1	6 ←	1	2	8 ←	4 ←
3 ←	9	7 ←	3 ←	9	6
4	14	11	7	12	15

OUTPUT

Example

R=1,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R1 = 1,3,4 R2 = 6,9,14 R3 = 1,7,11

S1 = 2,3,7 S2 = 8,9,12 S3 = 4,6,15

R1	R2	R3	S1	S2	S3
1	6 ←	1	2	8 ←	4 ←
3 ←	9	7 ←	3 ←	9	6
4	14	11	7	12	15

OUTPUT
(3,3)

Example

R=1,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R1 = 1,3,4

R2 = 6,9,14

R3 = 1,7,11

S1 = 2,3,7

S2 = 8,9,12

S3 = 4,6,15

R1	R2	R3	S1	S2	S3
1	6 ←	1	2	8 ←	4 ←
3	9	7 ←	3 ←	9	6
4 ←	14	11	7	12	15

OUTPUT
(3,3)
(4,4)

Example

R=1,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R1 = 1,3,4 R2 = 6,9,14 R3 = 1,7,11

S1 = 2,3,7 S2 = 8,9,12 S3 = 4,6,15

R1	R2	R3	S1	S2	S3
1	6 ←	1	2	8 ←	4 ←
3	9	7 ←	3	9	6
4 ←	14	11	7 ←	12	15

OUTPUT
(3,3)
(4,4)

Example

R=1,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R1 = 1,3,4 R2 = 6,9,14 R3 = 1,7,11

S1 = 2,3,7 S2 = 8,9,12 S3 = 4,6,15

R1	R2	R3	S1	S2	S3
1	6 ←	1	2	8 ←	4
3	9	7 ←	3	9	6 ←
4 ←	14	11	7 ←	12	15

OUTPUT
(3,3)
(4,4)
(6,6)

Example

R=1,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R1 = 1,3,4
R2 = 6,9,14
R3 = 1,7,11

S1 = 2,3,7
S2 = 8,9,12
S3 = 4,6,15

R1	R2	R3	S1	S2	S3
1	6 ←	1	2	8 ←	4
3	9	7 ←	3	9	6 ←
4 ←	14	11	7 ←	12	15

…

OUTPUT
(3,3)
(4,4)
(6,6)
(7,7)

Output in sorted order!

Simple “External” Hash

Idea: Avoid repeated passes over S in blocked hash

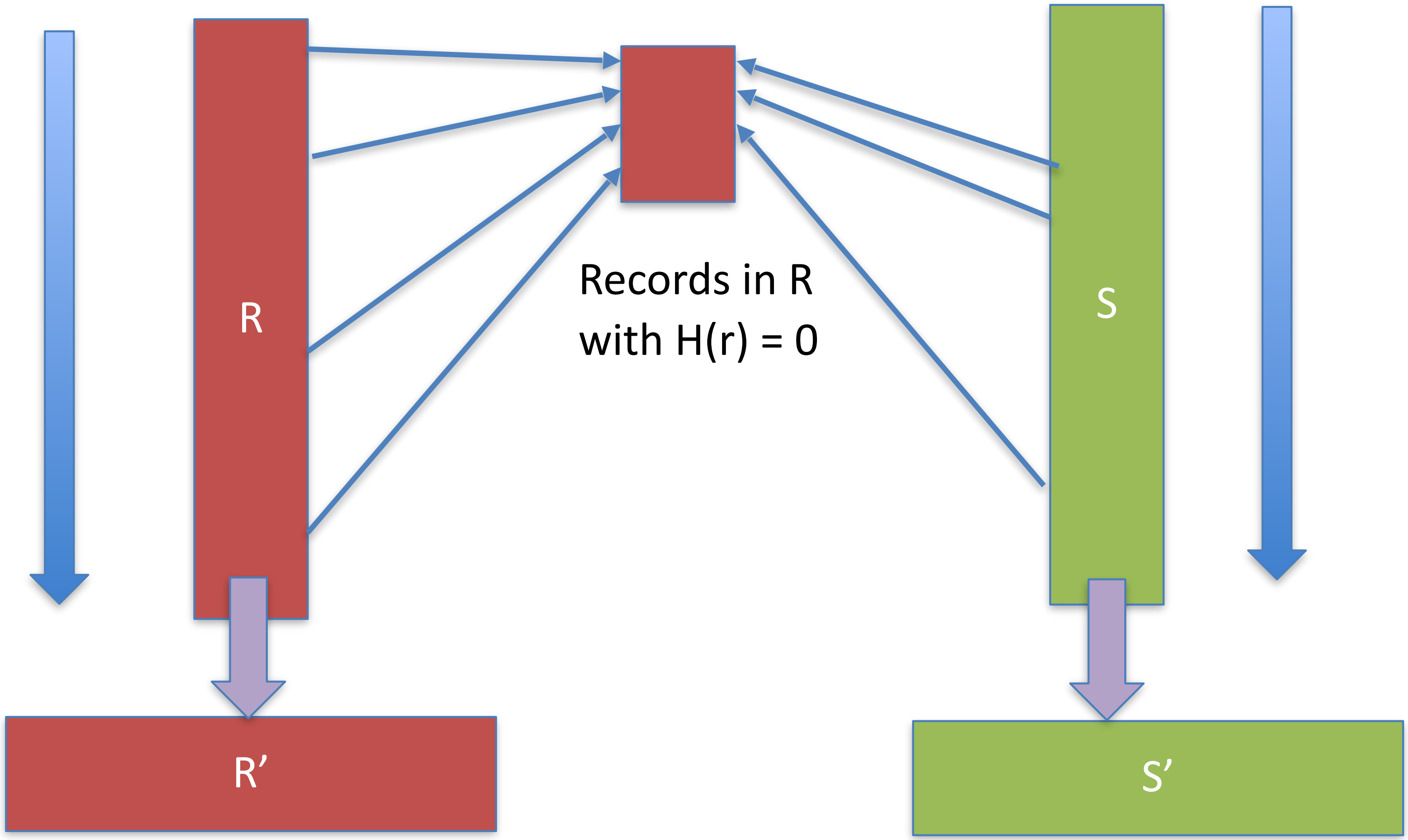
Algorithm:

```
    Given hash function H(x) → [0,…,P-1] (e.g., x mod P)
        where P is number of partitions
    for i in [0,…,P-1]:
        for each r in R:
            if H(r)=i, add r to in memory hash
            otherwise, write r back to disk in R’
        for each s in S:
            if H(s)=i, lookup s in hash, output matches
            otherwise, write s back to disk in S’
        replace R with R’, S with S’
```

Pass 0

Illustration

Hash function in 0…P



Scan of R

In memory hash table

Scan of S

R

S

Records in R with H(r) = 0

R’

S’

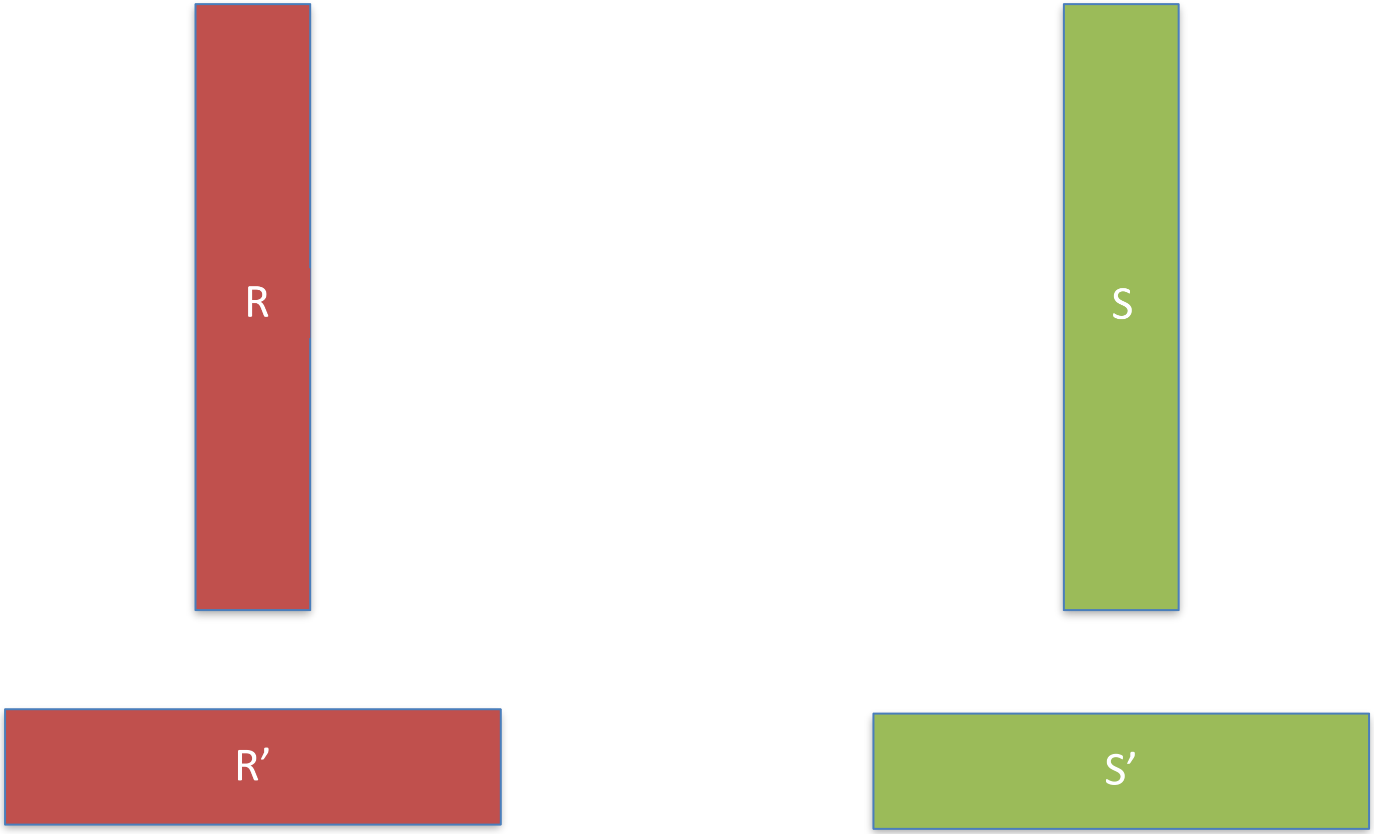
Remainder of R (Records with H(r) > 0

Remainder of S (Records with H(s) > 0

Pass 0

Illustration

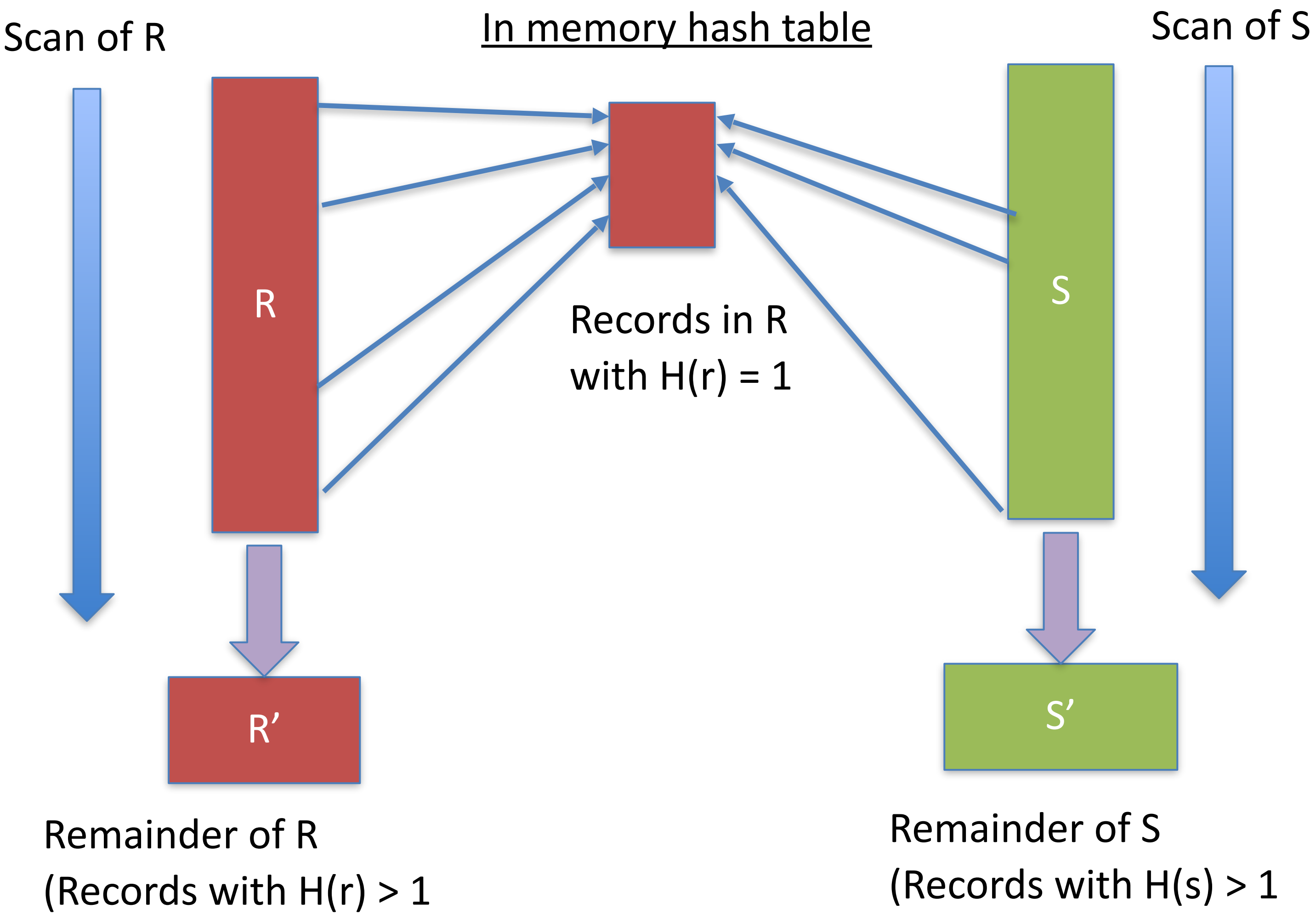
Hash function in 0…P



Pass 1

Illustration

Hash function in 0…P



Scan of R

In memory hash table

Scan of S

R

Records in R with H(r) = 1

S

R’

S’

Remainder of R (Records with H(r) > 1

Remainder of S (Records with H(s) > 1

Repeat for P passes

https://clicker.mit.edu/6.5830/

	I/O Complexity
(A)	
(B)	
(C)	
(D)	

N = number of partitions

Simple Hash I/O Analysis

Suppose P=2, and hash uniformly maps tuples to partitions

Read |R| + |S|
Write 1/2 (|R| + |S|)
Read 1/2 (|R| + |S|)

2 (|R| + |S|)

P=3

Read |R| + |S|
Write 2/3 (|R| + |S|)
Read 2/3 (|R| + |S|)
Write 1/3 (|R| + |S|)
Read 1/3 (|R| + |S|)

3 (|R| + |S|)

P=4

|R| + |S| + 2 * (3/4 (|R| + |S|)) + 2 * (2/4 (|R| + |S|)) + 2 * (1/4 (|R| + |S|))
= 4 (|R| + |S|)

➔ P = n ; n * (|R| + |S|) I/Os

Grace Hash

Can we avoid rewriting some records many times?

```
Algorithm:
Partition:
    Suppose we have P partitions, and H(x) → [0…P-1]
    Choose P = |S| / M ➔   P ≤ sqrt(|S|)  //may need to leave a little slop for imperfect hashing
    Allocate P 1-page output buffers, and P output files for R
    For each r in R:
        Write r into buffer H(r)
        If buffer full, append to file H(r)
    Allocate P output files for S
    For each s in S:
        Write s into buffer H(s)
        if buffer full, append to file H(s)
Join:
    For i in [0,…,P-1]
        Read file i of R, build hash table (memory should hold this)
        Scan file i of S, probing into hash table and outputting matches
```

Need one page of RAM for each of P partitions

Since
$M > \sqrt{|S|}$ and
$P \le \sqrt{|S|}$, all is well

Total I/O cost: Read |R| and |S| once, write once, read back once more

3(|R| + |S|) I/Os

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2

P output buffers

F0	F1	F2

P output files

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2
		5

F0	F1	F2

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2
	4	5

F0	F1	F2

Example

P = 3; H(x) = x mod P

↓

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2
3	4	5

F0	F1	F2

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2
3	4	5
6		

F0	F1	F2

Example

P = 3; H(x) = x mod P

↓

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2
3	4	5
6		

Need to flush R0 to F0!

F0	F1	F2

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2
	4	5

F0	F1	F2
3		
6		

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2
9	4	5

F0	F1	F2
3		
6		

Example

P = 3; H(x) = x mod P

⬇

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2
9	4	5
		14

F0	F1	F2
3		
6		

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2
9	4	5
	1	14

F0	F1	F2
3		
6		

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2
9	4	5
	1	14

F0	F1	F2
3		
6		

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2
9		5
		14

F0	F1	F2
3	4	
6	1	

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	**R1**	**R2**
9	7	5
		14

F0	**F1**	**F2**
3	4	
6	1	

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2
9	7	5
		14

F0	F1	F2
3	4	
6	1	

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2
9	7	

F0	F1	F2
3	4	5
6	1	14

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2
9	7	11

F0	F1	F2
3	4	5
6	1	14

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2

F0	F1	F2
3	4	5
6	1	14
9	7	11

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R Files

F0	F1	F2
3	4	5
6	1	14
9	7	11

S Files

F0	F1	F2
3	7	2
12	4	8
9		
15		
6		

Example

P = 3; H(x) = x mod P

Matches:

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R Files

F0	F1	F2
3	4	5
6	1	14
9	7	11

Load F0 from R into memory

S Files

F0	F1	F2
3	7	2
12	4	8
9		
15		
6		

Example

P = 3; H(x) = x mod P

Matches:

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R Files

F0	F1	F2
3	4	5
6	1	14
9	7	11

Load F0 from R into memory

S Files

F0	F1	F2
3	7	2
12	4	8
9		
15		
6		

Scan F0 from S

Example

P = 3; H(x) = x mod P

Matches:
3,3

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R Files

F0	F1	F2
3	4	5
6	1	14
9	7	11

Load F0 from R into memory

S Files

F0	F1	F2
3	7	2
12	4	8
9		
15		
6		

Scan F0 from S

Example

P = 3; H(x) = x mod P

Matches:
3,3

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R Files

F0	F1	F2
3	4	5
6	1	14
9	7	11

Load F0 from R into memory

S Files

F0	F1	F2
3	7	2
12	4	8
9		
15		
6		

Scan F0 from S

Example

P = 3; H(x) = x mod P

Matches:
3,3
9,9

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R Files

F0	F1	F2
3	4	5
6	1	14
9	7	11

Load F0 from R into memory

S Files

F0	F1	F2
3	7	2
12	4	8
9		
15		
6		

Scan F0 from S

Example

P = 3; H(x) = x mod P

Matches:
3,3
9,9

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R Files

F0	F1	F2
3	4	5
6	1	14
9	7	11

Load F0 from R into memory

S Files

F0	F1	F2
3	7	2
12	4	8
9		
15		
6		

Scan F0 from S

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

Matches:
3,3
9,9
6,6

R Files

F0	F1	F2
3	4	5
6	1	14
9	7	11

Load F0 from R into memory

S Files

F0	F1	F2
3	7	2
12	4	8
9		
15		
6		

Scan F0 from S

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

Matches:
3,3
9,9
6,6

R Files

F0	F1	F2
3	4	5
6	1	14
9	7	11

S Files

F0	F1	F2
3	7	2
12	4	8
9		
15		
6		

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

Matches:

3,3

9,9

6,6

7,7

4,4

R Files

F0	**F1**	**F2**
3	4	5
6	1	14
9	7	11

S Files

F0	**F1**	**F2**
3	7	2
12	4	8
9		
15		
6		

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

Matches:
3,3
9,9
6,6
7,7
4,4

R Files

F0	F1	F2
3	4	5
6	1	14
9	7	11

S Files

F0	F1	F2
3	7	2
12	4	8
9		
15		
6		

Hybrid

- Acts like simple for small tables, grace for large tables
- Suppose we have M = $\sqrt{|R|}$ + E
  - E is additional memory beyond the minimum
- Make the first partition size E, and join as in simple
- For remaining partitions write out as in grace
- Repeat with S, joining first partition on the fly, and writing out remaining partitions as in grace
- Join remaining partitions as in grace

External Join Summary

Notation: P partitions / passes over data; assuming hash is O(1)

Sort-Merge	Simple Hash	Grace Hash
I/O: 3 (|R| + |S|) CPU: O(P x {S}/P log {S}/P)	I/O: P (|R| + |S|) CPU: O({R} + {S})	I/O: 3 (|R| + |S|) CPU: O({R} + {S})

Grace hash is generally a safe bet, unless memory is close to size of tables, in which case simple can be preferable

Extra cost of sorting makes sort merge unattractive unless there is a way to access tables in sorted order (e.g., a clustered index), or a need to output data in sorted order (e.g., for a subsequent ORDER BY)

Many fancier versions exist, e.g., using modern sorting techniques (radix or counting sort), parallel cores, etc

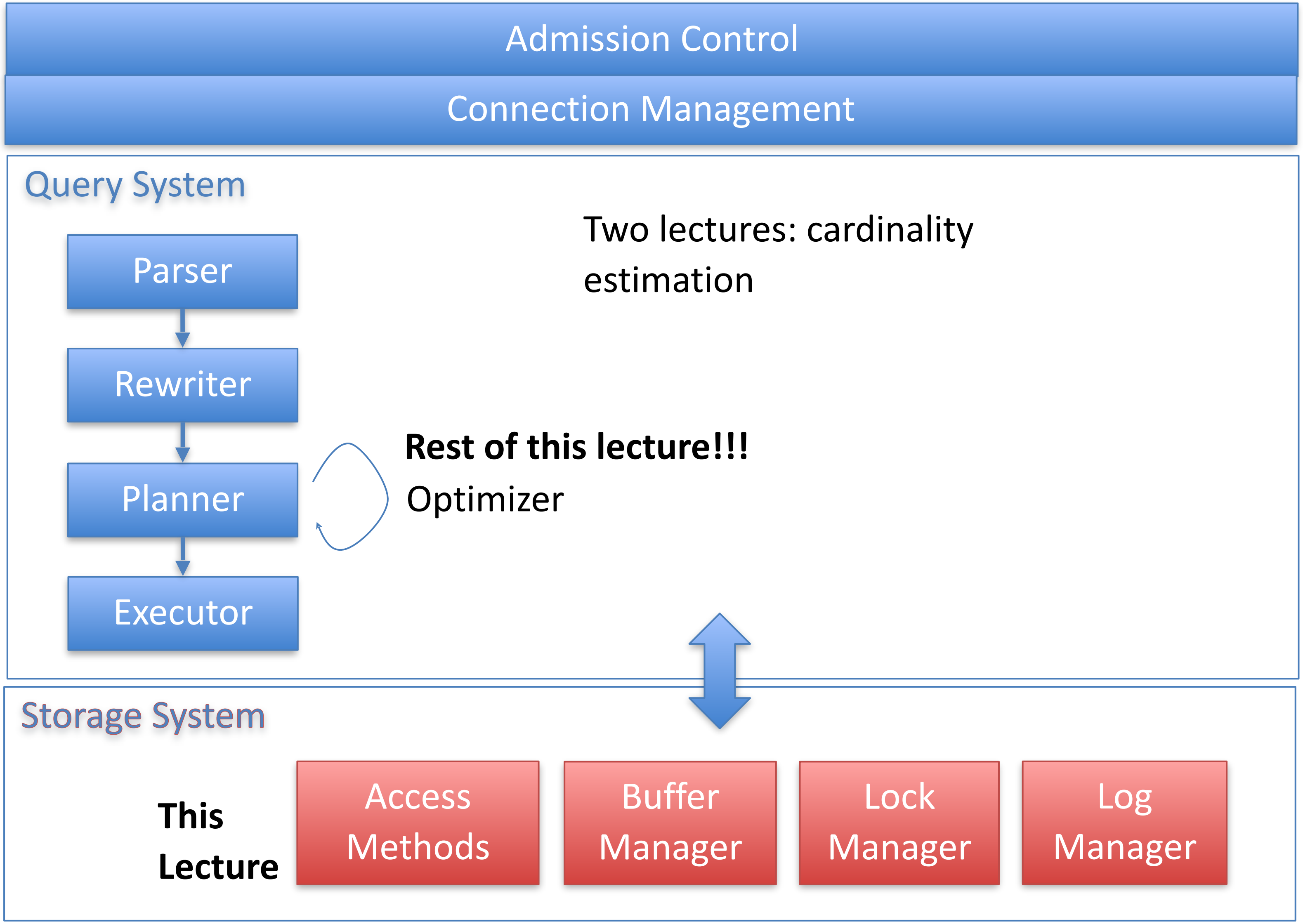
Join Algo Summary

Algo	I/O cost	CPU cost	In Mem?
Nested loops	|R|+|S|	O({R}x{S})	R in mem
Nested loops	{S}|R| + |S|	O({R}x{S})	No
Index nested loops (R index)	|S| + {S}c (c <5)	O({S}log{R})	No
Block nested loops	|S| + B|R| (B=|S|/M)	O({R}x{S})	No
Sort-merge	|R|+|S|	O({S}log{S})	Both
Hash (Hash R)	|R|+|S|	O({S} + {R})	R in mem
Blocked hash (Hash S)	|S| + B|R| (B=|S|/M)	O({S} + B{R}) (*)	No
External Sort-merge	3(|R| + |S|)	O(P x {S}/P log {S}/P)	No
Simple hash (not covered)	P(|R|+|S|) (P=|S|/M)	O({R} + {S})	No
Grace hash	3(|R| + |S|)	O({R} + {S})	No

Grace hash is generally a safe bet, unless memory is close to size of tables, in which case simple can be preferable

Extra cost of sorting makes sort merge unattractive unless there is a way to access tables in sorted order (e.g., a clustered index), or a need to output data in sorted order (e.g., for a subsequent ORDER BY)

Database Internals Outline



Admission Control

Connection Management

Query System

Parser

Rewriter

Planner

Executor

Two lectures: cardinality estimation

Rest of this lecture!!!

Optimizer

Storage System

This Lecture

Access Methods

Buffer Manager

Lock Manager

Log Manager

Query Optimization Objective

- Find the query plan of minimum cost
  - Many possible cost functions, as we've discussed
- Requires a way to:
  - Evaluate cost of a plan
  - Enumerate (iterate through) plan options

Cost Estimation

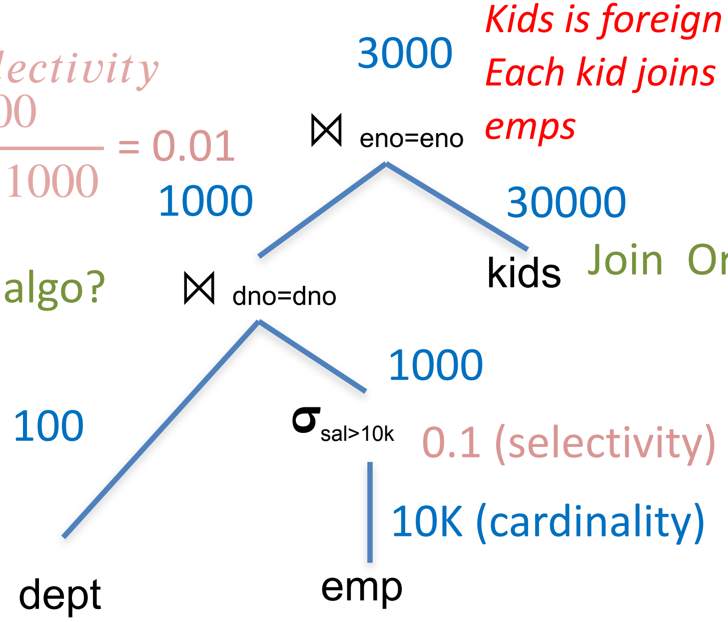
- Cost Plan = ∑(Cost Plan Operators)
- Cost Plan Operator ∝ Size of Operator Input

- Determining Size of Operator Input
  - For base tables, equal to size on disk
    - Tables with indexes may support predicate push down
  - For other operators, equal to “selectivity” x size of children
    - **Selectivity** is fraction of input size that the operator emits
    - Join selectivity defined relative to the size of the cross product

Example (Lec 5)

```
SELECT * FROM emp, dept, kids
WHERE sal > 10k
AND emp.dno = dept.dno
AND emp.eid = kids.eid
```

100 tuples/page
10 pages RAM
10 KB/page



Kids is foreign key; Each kid joins w/ 3 emps

Selectivity

$$\frac{1000}{100 \times 1000} = 0.01$$

Join algo?

Join Ordering? Why not kids / emp first?

Index vs scan?

|dept| = 100 records = 1 page = 10 KB
|emp| = 10K = 100 pages = 1 MB
|kids| = 30K = 300 pages = 3 MB

Steps:

For each plan alternative:

1. Estimate sizes of relations
2. Estimate selectivities
3. Compute intermediate sizes
4. Evaluate cost of plan operations

5. Select best plan

Selinger Statistics

Steps:
1. ➡ Estimate sizes of relations
2. Estimate selectivities
3. Compute intermediate sizes
4. Evaluate cost of plan operations
5. Find best overall plan

NCARD(R) - "relation cardinality" - number of records in R

TCARD(R) - # pages R occupies

ICARD(I) - # keys (distinct values) in index I

NINDX(I) - pages occupied by index I

Min and max keys in indexes

Modern databases use much more sophisticated stats – will look at Postgres and learn about some research techniques in 2 lectures

Steps:

1. Estimate sizes of relations
2. Estimate selectivities
3. Compute intermediate sizes
4. Evaluate cost of plan operations
5. Find best overall plan

Selinger Selectivities

F(pred) = Selectivity of predicate = Fraction of records that a predicate does not filter

NCARD(R) - "relation cardinality" - number of records in R

TCARD(R) - # pages R occupies

ICARD(I) - # keys (distinct values) in index I

NINDX(I) - pages occupied by index I

Min and max keys in indexes

Predicate types

1. col = val

Clicker (http://clicker.mit.edu/6.5830)

Which is the best estimate for the selectivity of *col = val*?

A. 1/TCARD(R)

B. ICARD(I)/NCARD(I)

C. 1/ICARD(I)

D. (max key – val) / (ICARD(I))

Selinger Selectivities

Steps:

1. Estimate sizes of relations
2. Estimate selectivities
3. Compute intermediate sizes
4. Evaluate cost of plan operations
5. Find best overall plan

F(pred) = Selectivity of predicate = Fraction of records that a predicate does not filter

NCARD(R) - "relation cardinality" - number of records in R

TCARD(R) - # pages R occupies

ICARD(I) - # keys (distinct values) in index I

NINDX(I) - pages occupied by index I

Min and max keys in indexes

Predicate types

1. col = val

F = 1/ICARD() *(if index available)*

F = 1/10 otherwise → Modern DBs use fancier stats!

2. col > val

(max key - value) / (max key - min key) *(if index available)*

1/3 otherwise

3. col1 = col2

1/MAX(ICARD(col1), ICARD(col2)) *(if index available)*

1/10 otherwise

Assumes key-foreign key join

Note a better estimate is 1/ICARD(PK table)

We use 1/ICARD(PK table) going forward

Complex Predicates

Steps:

1. Estimate sizes of relations
2. Estimate selectivities
3. Compute intermediate sizes
4. Evaluate cost of plan operations
5. Find best overall plan

F(pred) = Selectivity of predicate = Fraction of records that a predicate does not filter

- P1 and P2

  F(P1) x F(P2)

- P1 or P2

  1 – P(neither predicate is satisfied) =

  1 – (1-F(P1)) x (1-F(P2))

Note uniformity assumption

Steps:

1. Estimate sizes of relations
2. Estimate selectivities
3. Compute intermediate sizes
4. Evaluate cost of plan operations
5. Find best overall plan

Intermediate Sizes

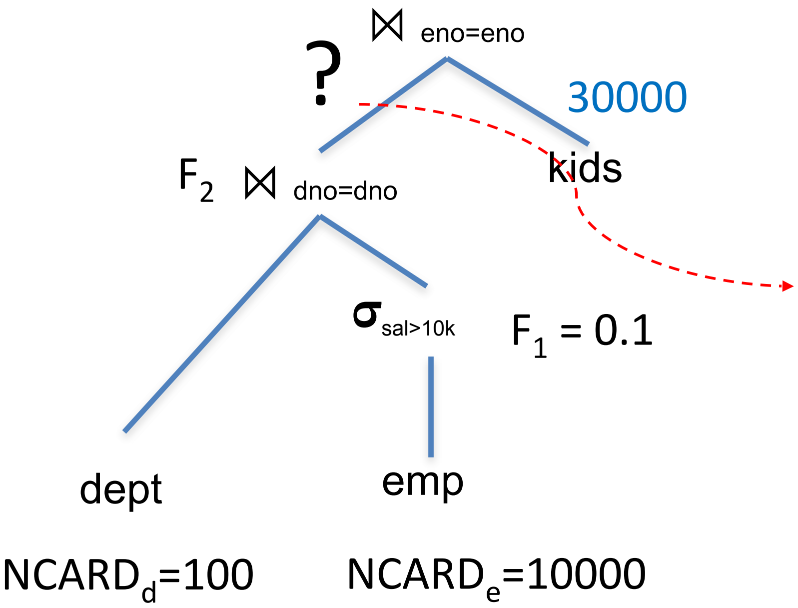
NCARD(R) - "relation cardinality" - number of records in R

TCARD(R) - # pages R occupies

ICARD(I) - # keys (distinct values) in index I

NINDX(I) - pages occupied by index I

Min and max keys in indexes



Clicker (http://clicker.mit.edu/6.5830)

A) $100 \times (10000 \times 0.1) \times 0.01 = 1000$

B) $10000 \times 0.1 = 1000$

C) $10000 \times 0.1 \times 0.01 = 10$

D) $10000 \times 0.1 \times 100 = 100000$

Steps:

1. Estimate sizes of relations
2. Estimate selectivities
3. Compute intermediate sizes
4. Evaluate cost of plan operations
5. Find best overall plan

Intermediate Sizes

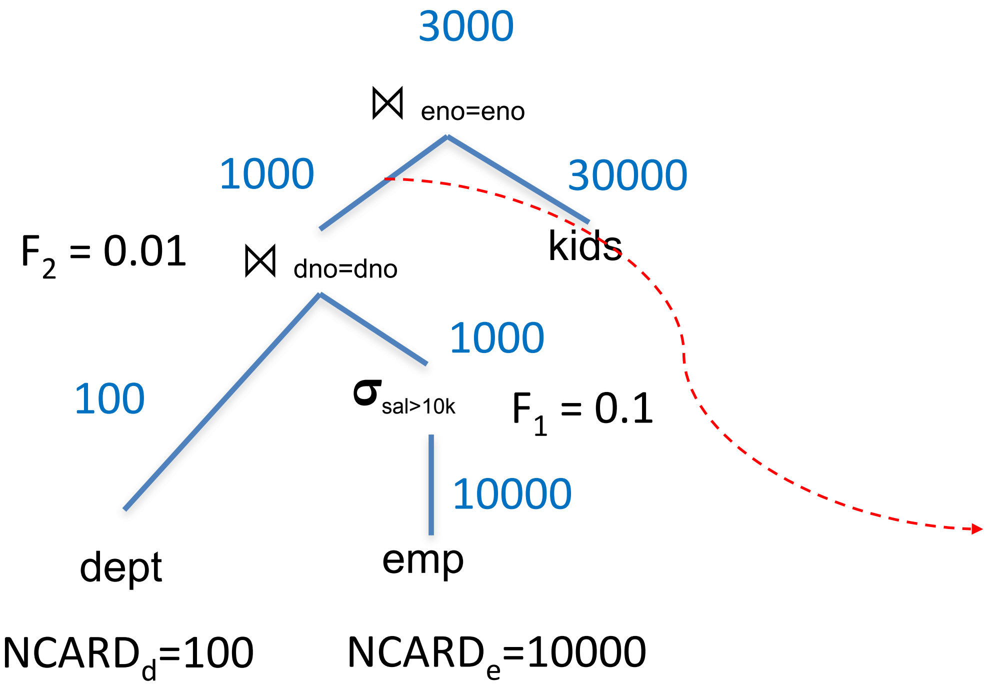
NCARD(R) - "relation cardinality" - number of records in R

TCARD(R) - # pages R occupies

ICARD(I) - # keys (distinct values) in index I

NINDX(I) - pages occupied by index I

Min and max keys in indexes



$$NCARD_d \times NCARD_e \times F_1 \times F_2 = \\ 100 \times 10000 \times 0.1 \times 0.01 = 1000$$

Steps:

1. Estimate sizes of relations
2. Estimate selectivities
3. Compute intermediate sizes
4. Evaluate cost of plan operations
5. Find best overall plan

Cost of Base Table Operations

Cost = pages read + weight x (records evaluated)

NCARD(R) - "relation cardinality" - number of records in R

TCARD(R) - # pages R occupies

ICARD(I) - # keys (distinct values) in index I

NINDX(I) - pages occupied by index I

Min and max keys in indexes

W: weight of CPU operations

Equality predicate with unique index: $\underset{\text{B+Tree lookup}}{1} + \overset{\text{Heap File lookup}}{1} + \underset{\text{Predicate evaluation}}{W}$

Steps:

1. Estimate sizes of relations
2. Estimate selectivities
3. Compute intermediate sizes
4. Evaluate cost of plan operations
5. Find best overall plan

Cost of Base Table Operations

NCARD(R) - "relation cardinality" - number of records in R

TCARD(R) - # pages R occupies

ICARD(I) - # keys (distinct values) in index I

NINDX(I) - pages occupied by index I

Min and max keys in indexes

W: weight of CPU operations

Cost = pages read + weight x (records evaluated)

Equality predicate with unique index: $1 + 1 + W$

B+Tree lookup

Heap File lookup

Predicate evaluation

Clustered index, range w/ selectivity F

A: F x TCARD + W x (tuples read)
B: F x (NINDX + NCARD) + W x (tuples read)
C: F x NINDX + W x (tuples read)
D: F x (NINDX + TCARD) + W x (tuples read)

Clicker (http://clicker.mit.edu/6.5830)

Cost of Base Table Operations

Steps:

1. Estimate sizes of relations
2. Estimate selectivities
3. Compute intermediate sizes
4. ➡ Evaluate cost of plan operations
5. Find best overall plan

Cost = pages read + weight x (records evaluated)

NCARD(R) - "relation cardinality" - number of records in R

TCARD(R) - # pages R occupies

ICARD(I) - # keys (distinct values) in index I

NINDX(I) - pages occupied by index I

Min and max keys in indexes

W: weight of CPU operations

Equality predicate with unique index: $\underbrace{1}_{\text{B+Tree lookup}} + \overbrace{1}^{\text{Heap File lookup}} + \underbrace{W}_{\text{Predicate evaluation}}$

Clustered index, range w/ selectivity F: F x (NINDX + TCARD) + W x (tuples read)

One I/O per page

Unclustered index, range w/ selectivity F : F x (NINDX + NCARD) + W x (tuples read)

One I/O per record

Seq (segment) scan: TCARD + W x (NCARD)

Steps:
1. Estimate sizes of relations
2. Estimate selectivities
3. Compute intermediate sizes
4. Evaluate cost of plan operations
5. Find best overall plan

Cost of Joins

NCARD(R) - "relation cardinality" - number of records in R

TCARD(R) - # pages R occupies

ICARD(I) - # keys (distinct values) in index I

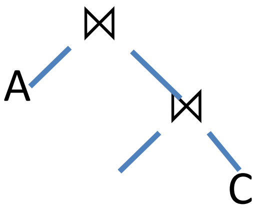
W: weight of CPU operations

NestedLoops(A,B,pred)

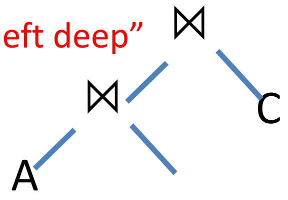
$$\underbrace{\mathrm{Cost}(A)}_{\text{Outer Plan}} + \mathrm{NCARD}(A) \times \underbrace{\mathrm{Cost}(B)}_{\text{Inner Plan}}$$

- Selinger only considers “left deep” plans, i.e., B is always a base table T_{right}
- In an index on T_{right}, $\mathrm{Cost}(B) = 1 + 1 + W$
- If no index, $\mathrm{Cost}(B) = \mathrm{TCARD}(T_{right}) + W \times \mathrm{NCARD}(T_{right})$
- Cost(A) is just cost of outer subtree

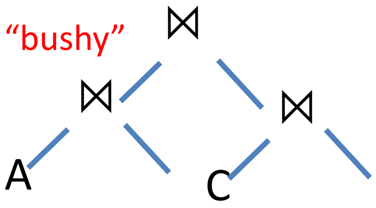
“right deep”



“left deep”



“bushy”



Steps:

1. Estimate sizes of relations
2. Estimate selectivities
3. Compute intermediate sizes
4. → Evaluate cost of plan operations
5. Find best overall plan

Cost of Joins

Merge(A,B,pred)

Cost(A) + Cost(B) + sort cost

Varies depending on whether sort is in memory or on disk, and whether one or both tables are already sorted

If either table is a base table, cost is just the sequential scan cost

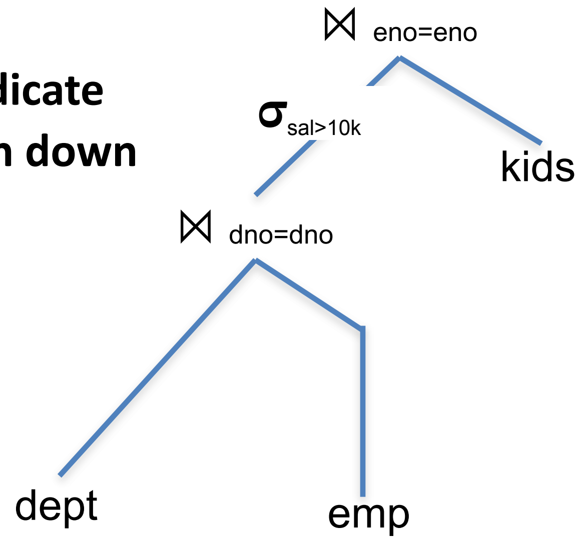
Steps:

1. Estimate sizes of relations
2. Estimate selectivities
3. Compute intermediate sizes
4. Evaluate cost of plan operations
5. Find best overall plan

Enumerating Plans

- Selinger combines several heuristics with a search over join orders
- Heuristics
  - Push down selections
  - Don’t consider cross products
  - Only “left deep” plans
    - Right side of all joins is base relation
- Still have to order joins!

Predicate push down



Join ordering

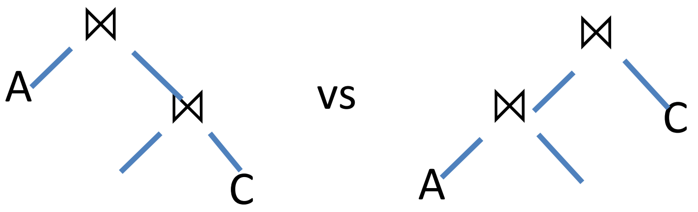
- Suppose I have 3 tables, A ⋈ B ⋈ C
    - Predicates between all 3 (no cross products)
- How many orderings?

ABC
ACB
BAC
BCA
CAB
CBA

n!

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

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



This plan is not
left deep!

Left deep plans are all of
the form (…(((AB)C)D)E)…)

n! left deep plans
10! = 3.6 M
15! = 1.3 T

Can we do
better?

Dynamic Programming Algorithm

- **Idea**: compute the best way to join each sub-plan, from smallest to largest
  - Don't need to reconsider subplans in larger plans
- For example, if the best way to join ABC is (AC)B, that will always be the best way to join ABC, whenever* these three relations occur as a part of a subplan.

* *Except when considering interesting orders*

Postgres example

*explain select * from emp join kids using (eno);*

```
Hash Join  (cost=34730.02..132722.07 rows=3000001 width=35)
 Hash Cond: (kids.eno = emp.eno)
 ->  Seq Scan on kids  (cost=0.00..49099.01 rows=3000001 width=18)
 ->  Hash  (cost=16370.01..16370.01 rows=1000001 width=21)
       ->  Seq Scan on emp  (cost=0.00..16370.01 rows=1000001 width=21)
```

*explain select * from dept join emp using(dno) join kids using (eno);*

```
Hash Join  (cost=35000.04..140870.43 rows=3000001 width=39)
 Hash Cond: (emp.dno = dept.dno)
 ->  Hash Join  (cost=34730.02..132722.07 rows=3000001 width=35)
       Hash Cond: (kids.eno = emp.eno)
       ->  Seq Scan on kids  (cost=0.00..49099.01 rows=3000001 width=18)
       ->  Hash  (cost=16370.01..16370.01 rows=1000001 width=21)
             ->  Seq Scan on emp  (cost=0.00..16370.01 rows=1000001 width=21)
 ->  Hash  (cost=145.01..145.01 rows=10001 width=8)
       ->  Seq Scan on dept  (cost=0.00..145.01 rows=10001 width=8)
```

Identical subplans

Selinger Algorithm

```
R ← set of relations to join
For i in {1...|R|}:
    for S in {all length i subsets of R}:
        $optcost_s = \infty$
        $optjoin_S = \emptyset$
        for a in S:  //a is a relation
            $c_{sa} = optcost_{s-a} +$
                min. cost to join (S-a) to a +
                min. access cost for a
            if $c_{sa} < optcost_s$ :
                $optcost_s = c_{sa}$
                $optjoin_s$ = optjoin(S-a) joined optimally w/ a
```

Cached in previous step!

Example

4 Relations: ABCD

Optjoin:

A = best way to access A

(e.g., sequential scan,

or predicate pushdown into index...)

B = " " " " B

C = " " " " C

D = " " " " D

{A,B} = AB or BA

{A,C} = AC or CA

{B,C} = BC or CB

{A,D}

{B,D}

{C,D}

Relations	Best Plan	Cost

Dynamic Programming Table

Example (con't)

Relations	Best Plan	Cost
A	Index Scan	5
B	Seq Scan	15
…		
{A,B}	BA	75
{A,C}	AC	12
{B,C}	CB	22
..		
[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]
[illegible]		
[illegible]	[illegible]	[illegible]

Already computed!

Optjoin

{A,B,C} = remove A: compare A({B,C}) to ({B,C})A
remove B: compare ({A,C})B to B({A,C})
remove C: compare C({A,B}) to ({A,B})C

{A,C,D} = …

{A,B,D} = …

{B,C,D} = …

…

{A,B,C,D} = remove A: compare A({B,C,D}) to ({B,C,D})A
remove B: compare B({A,C,D}) to ({A,C,D})B
remove C: compare C({A,B,D}) to ({A,B,D})C
remove D: compare D({A,C,C}) to ({A,B,C})D

Complexity

- Have to enumerate all sets of size 1…n

$$\binom{n}{1} + \binom{n}{2} \dots + \binom{n}{n}$$

- Number of subsets of set of size n =

  |power set of n| =

  2^n (here, n is number of relations)

Equivalent to all binary strings of length N, where a 1 in the ith position indicates that relation i is included:

001, 010, 100, … , 011, 111

Complexity (cont.)

2^n Subsets

How much work per subset?

Have to iterate through each element of each subset, so this at most n

$n2^n$ complexity (vs n!)

n=12 ➔ 49K vs 479M

Interesting Orders

- Some query plans produce data in sorted order – E.g scan over a primary index, merge-join – Called an *interesting order*
- Next operator may use this order – E.g. can be another merge-join
- For each subset of relations, compute multiple optimal plans, one for each interesting order
- Increases complexity by factor k+1, where k=number of interesting orders

Summary

- Selinger Optimizer is the foundation of modern cost-based optimizers
  - Simple statistics
  - Several heuristics, e.g., left-deep
  - Dynamic programming algo for join ordering
- Easy to extend, e.g., with:
  - More sophisticated statistics
  - Fewer heuristics