L9
Query Execution & Optimization

Eugene Wu
Fall 2015

Remember in the history of SQL, the CODASYL folks were making the performance argument, that there's no way that SQL could run fast? Well history has proved them wrong, and we'll talk enough to give you a feel for how we went about making this fast

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

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

\# unique values in R = 1. 100? 1M? ⟵ selectivity!

In the first relational algebra query, we first perform a cross product, which will return R*T pages, then we filter by v=1

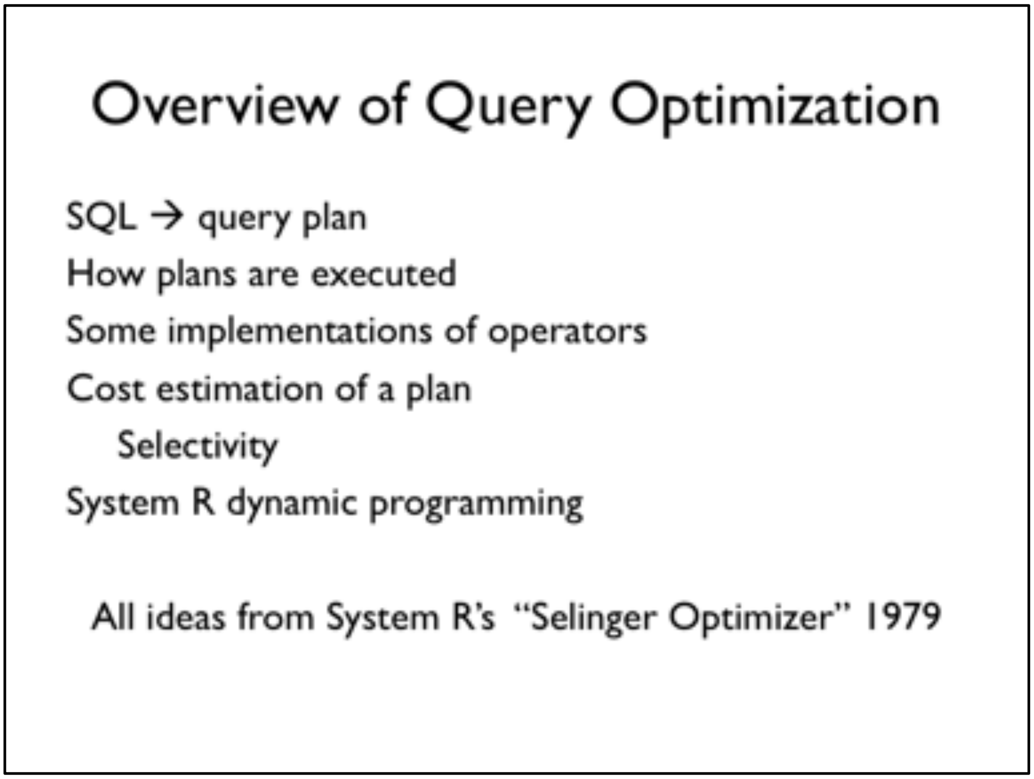
In the second, we first filter R, and then perform the cross product

in the extreme case, where there are no R records with v = 1, then this is an obviously better option.

2*(R * T) + R*T: for each R tuple, read each T and generate result (R*T). write it out. then read it back in for filter

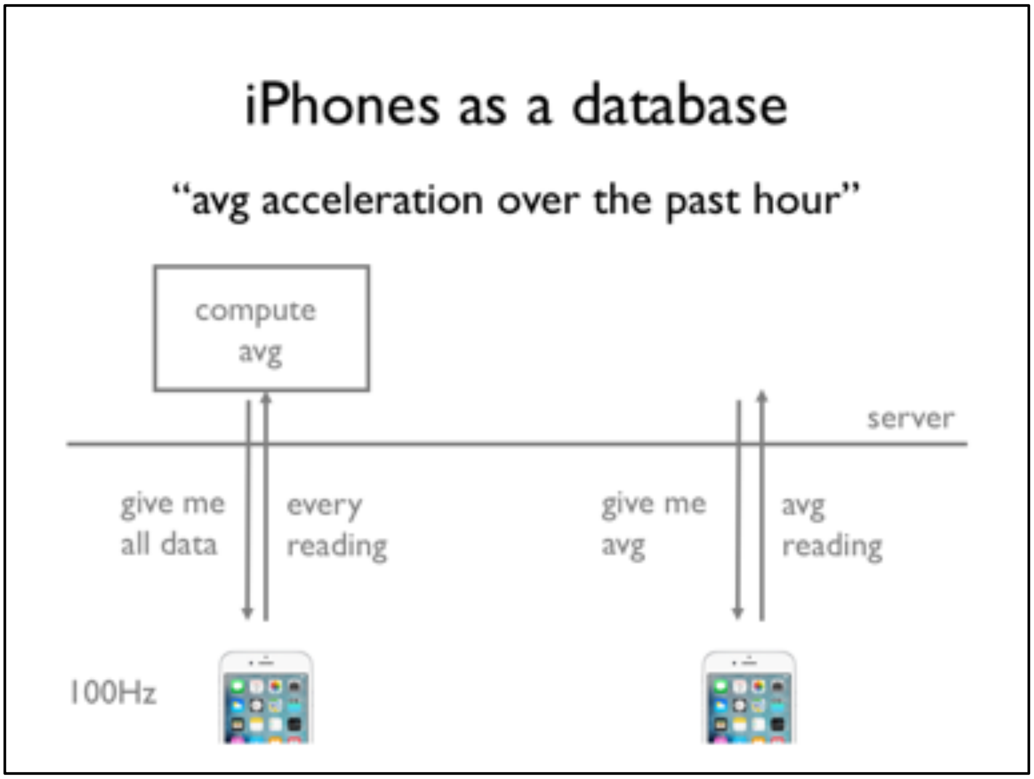
vs

(1+selectivity) * R + (selectivity*R)*T

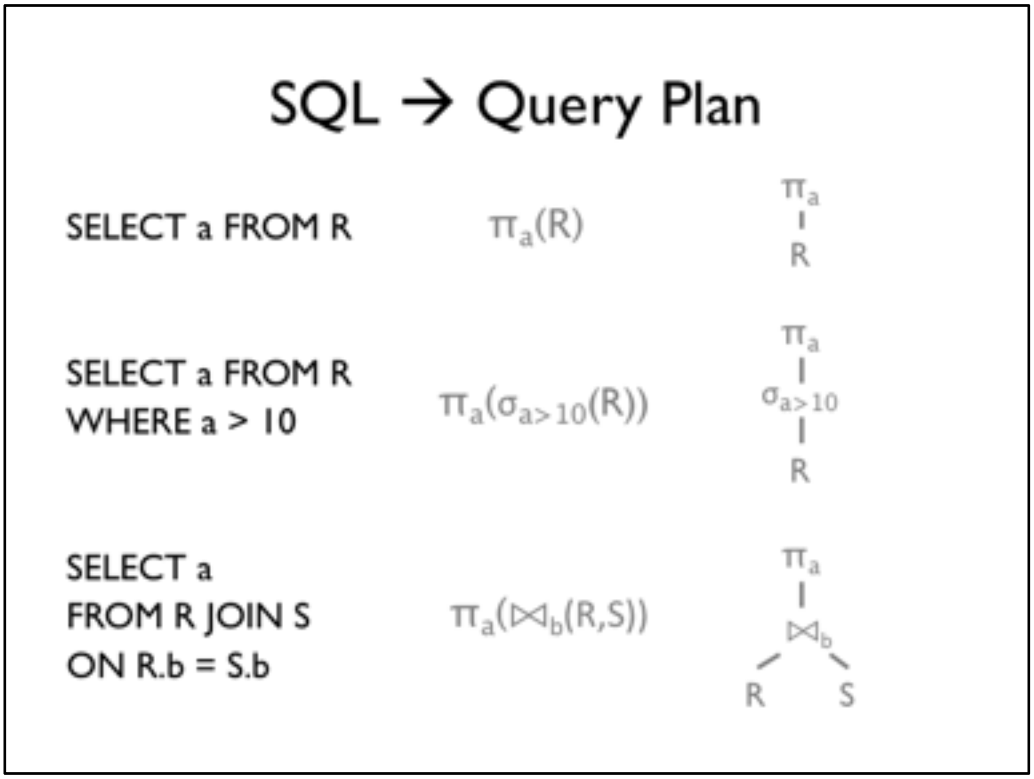


First, we need to talk abou how exactly to translate SQL into a standardized data structure that can be manipulated, compared, optimized. That form is the query plan
Go through the bottom up execution of these query plans
Look at some alternative operator implementation – different join algorithms, etc
Because of this there is as distinction between a logical operator such as the greek symbols we have seen
(describe the result semantics)
and the physical implementation of the operator or physical operator, that is exactly how it should be executed

pipelining
access paths
Optimizers are hugely important – often the trade secret of a database. not surprising to have hundreds of programmer years in an optimizer like Oracle or IBM's
A huge chunk of database research, at the core, are optimization algorithms that could be added to an optimizer



100hz = 100 samples per second, 6000 per minute, 360,000 per hour
exmaple where doing the avg computation on the phone (pushing it down) makes sense.



arguments are children

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

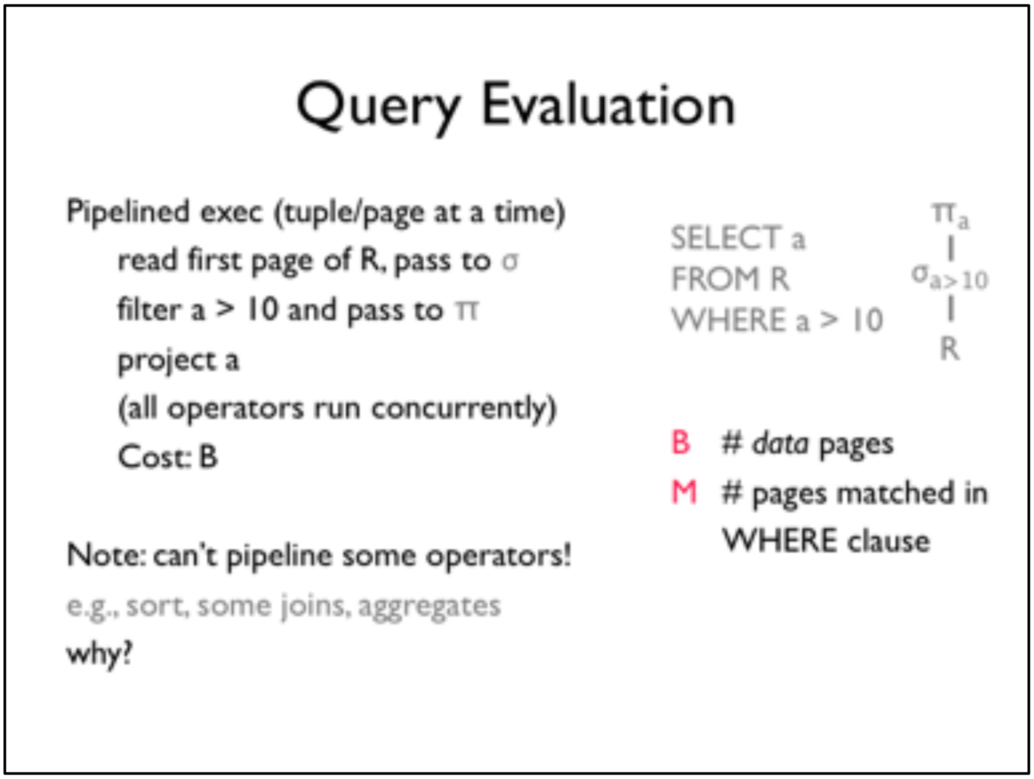
π_a
|
$\sigma_{a>10}$
|
R

B # *data* pages

M # pages matched in WHERE clause

Could we do better?

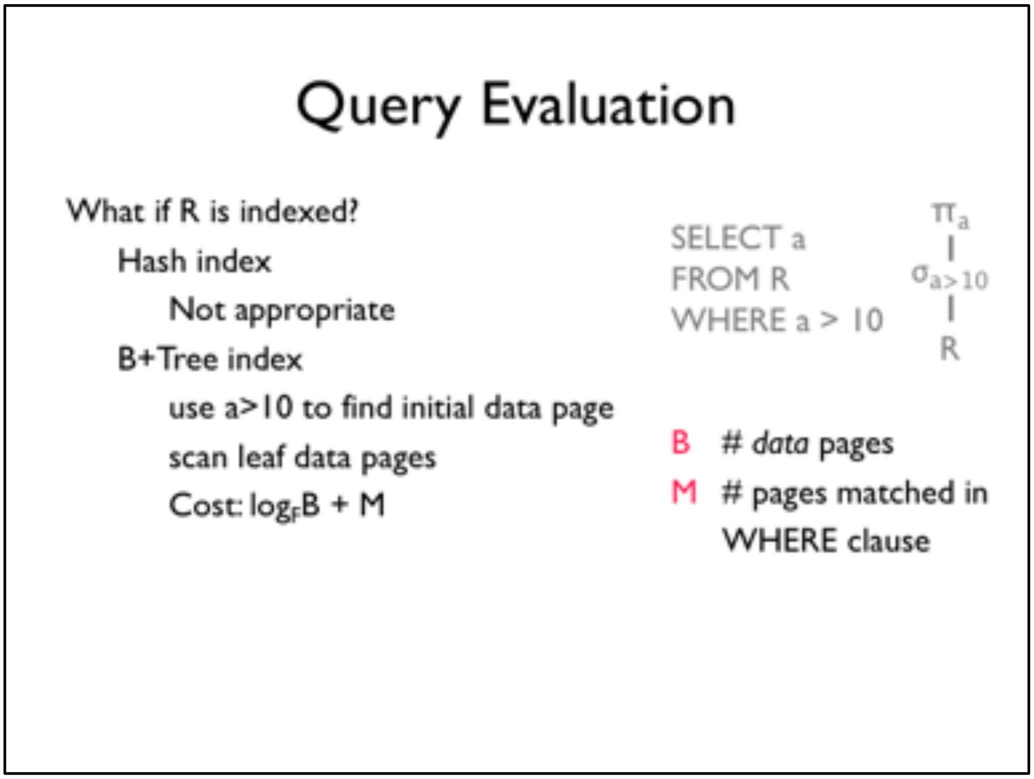
I read R, maybe after the filter the result doesn't fit into memory, or just barely, so I need to write it out. then I go to next operator, and read its data to execute the projection



Not exactly correct, but provides the intuition of why pipelining is a good idea.
In reality, each operator is often implemented using an iterator interface with get_next() calls.
The user would call next() on the root node, pi in this case, and the query execution will do just the work to compute the next result tuple.
In this case, if I only called get_next() once, we would only need to read a single page!

Keep in mind that this is an example of pipelined execution – all of the operators are running at the same time on the same or different pieces of data. If each operator were a separate CPU, then they are running at the same time and don't need to wait on each other.
Remember the monotonicity property from talking about relational operators?
there are operators such as some join operators that are blocking, meaning

Could we do better?



Not exactly correct, but provides the intuition of why pipelining is a good idea.
In reality, each operator is often implemented using an iterator interface with get_next() calls.
The user would call next() on the root node, pi in this case, and the query execution will do just the work to compute the next result tuple.
In this case, if I only called get_next() once, we would only need to read a single page!

Could we do better?

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

e.g., a > 1 and b = 5 and c < 5

will (a > 1 and b = 5) work?

will (a > 1 and c > 9) work?

(a > 1 and c > 9) will work because a > 1 uses a, which is a prefix of the search key. When we get to the left-most leaf data node that matches a > 1, we will then scan towards the seft

if the data has been sorted at the leaves, then it's just a sequential scan for all pages that contain tuples that match a > 1

How to pick Access Paths?

Selectivity

ratio of # outputs satisfying predicates vs # inputs

0.01 means 1 output tuple for every 100 input tuples

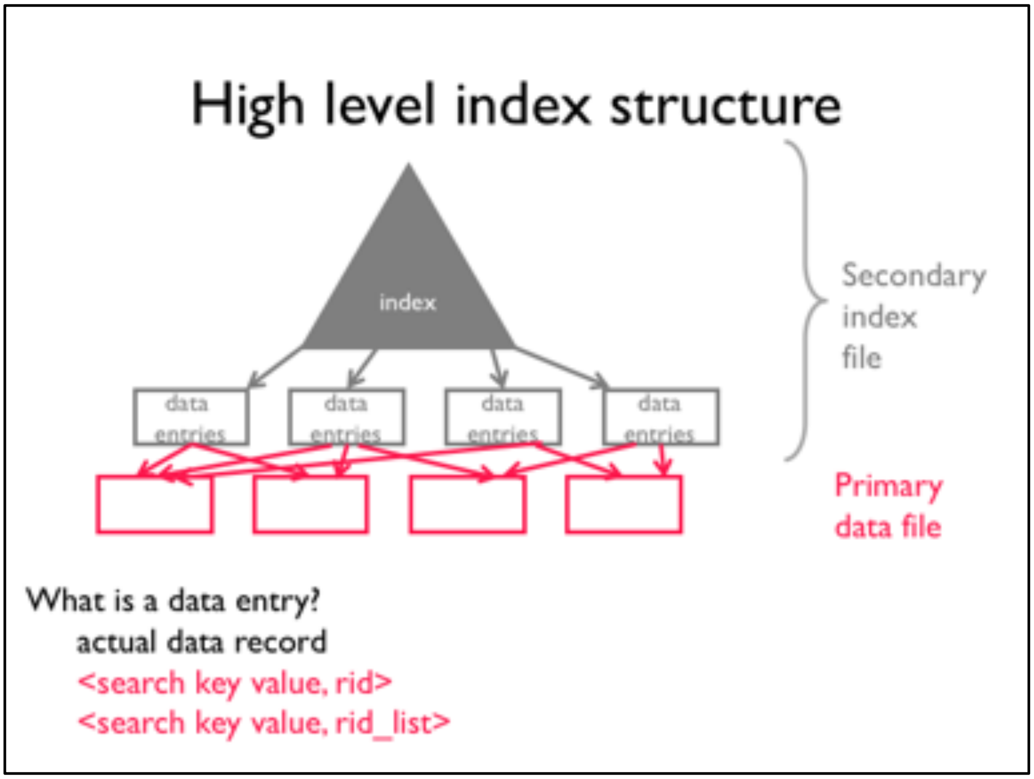
Assume

attribute selectivity is independent

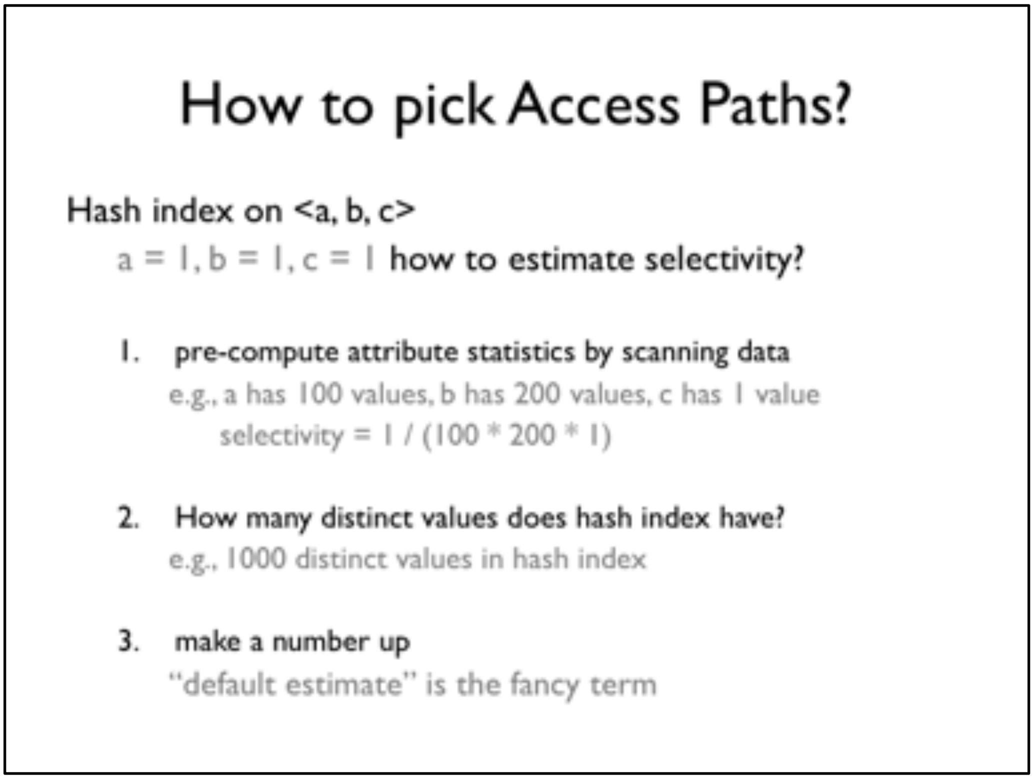
if selectivity(a=1) = 0.1, selectivity(b>3) = 0.6

selectivity(a=1 and b>3) = 0.1*0.6 = 0.06

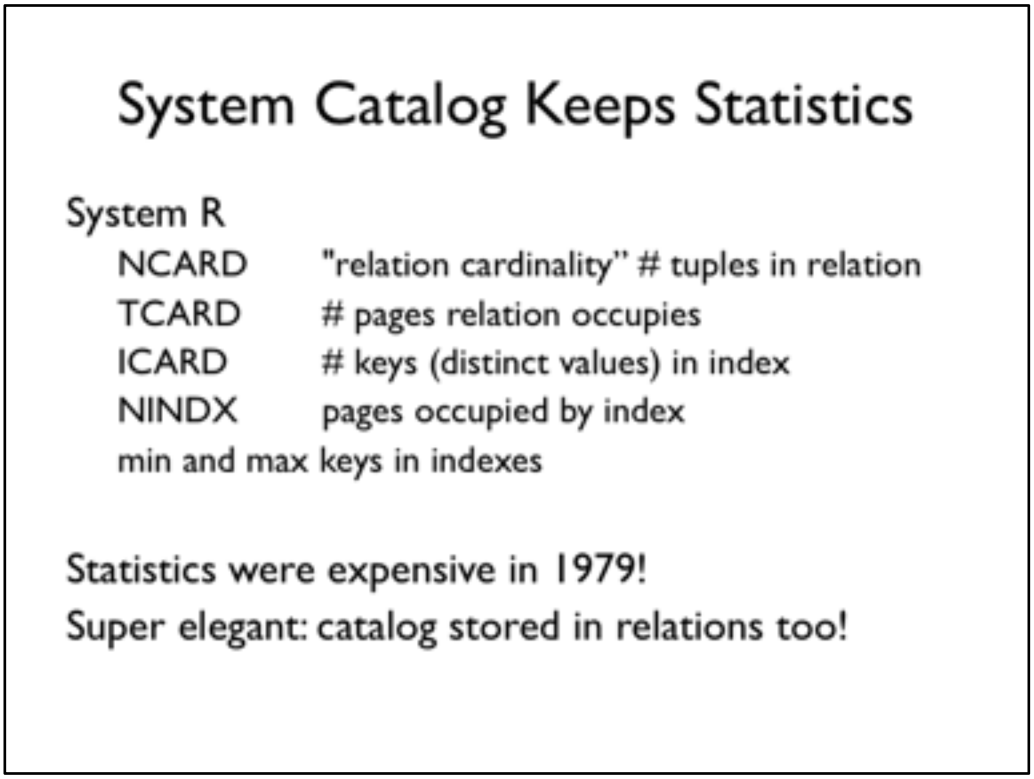
Why does selectivity matter?



baseline is sequential scan
if storing data in index as RID, then one page access per record
same with hash tables



1) look at each attribute individually
2) look at combination of all attributes and their distinctness
3) make something up



What is a statistic? It's a data structure (sometimes justa single number) that describes enough about the data to estimate the selectivities – and thus the costs
for example, you could get perfect informatino by just running the query.
so it's a trade off between accuracy and time

for example, the index is a compressed, summarized version of the table for ONLY the search key attributes,
but presumably those are the attribute you CARE about estimating selectivities correctly

if stats stored in database, then estimates are just queries.

What Optimization Options Do We Have?

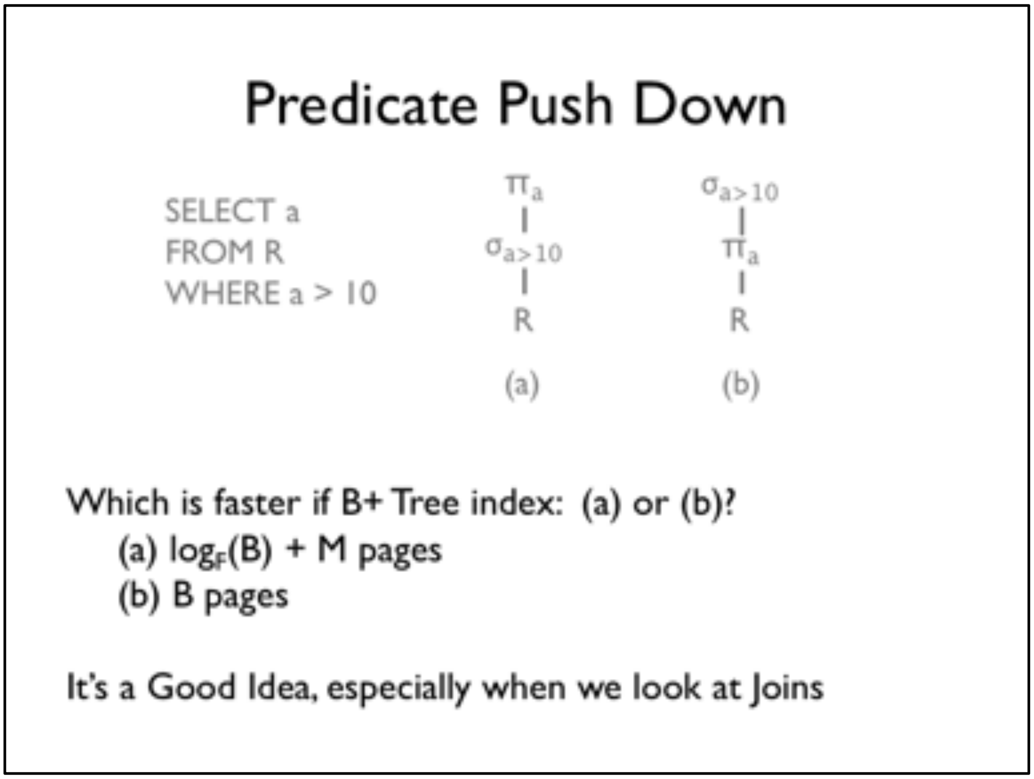
Access Path ✔

Predicate push-down

Join implementation

Join ordering

In general, depends on operator implementations. So let's take a look



do (a) and (b) do the same things?
if you're thinking, well it's obvious that a and b are equivalent plans, then you're right!

in (b), the selection predicate a > 10 is at the top, and in (a), it is "pushed down" lower into the query plan

What would be a rule we could create that tells us to pick the faster one?

Projection with DISTINCT clause

need to deduplicate e.g., π_{rating}Sailors

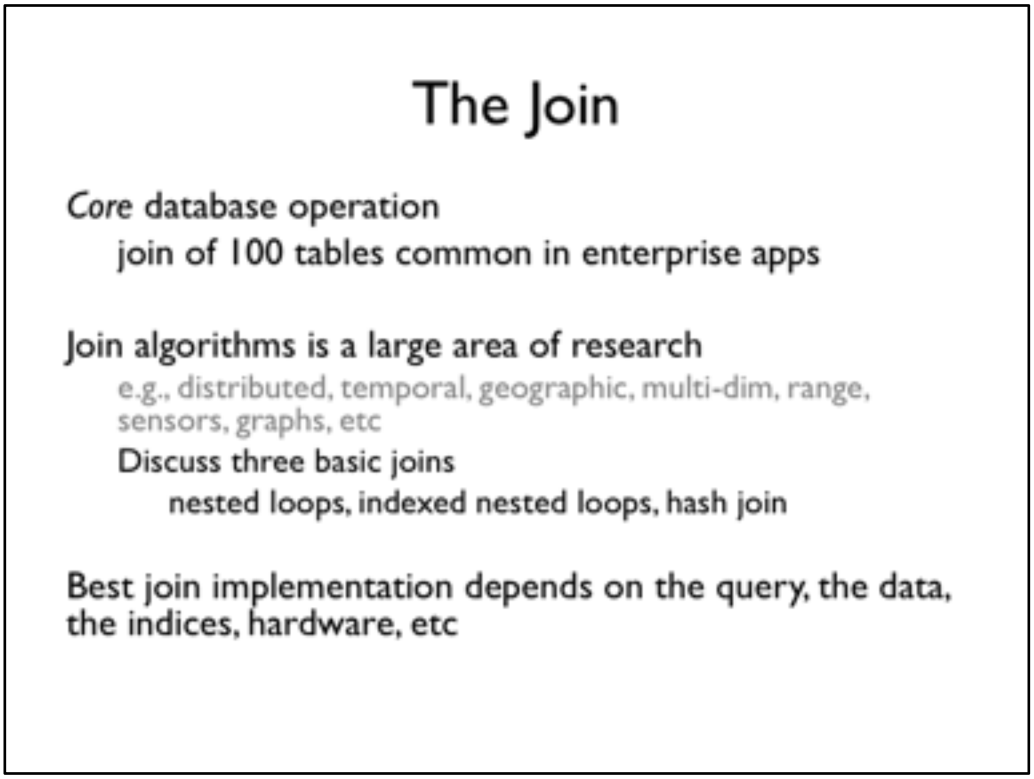
Two basic approaches

- Sort: fundamental database operation
  - sort on rating, remove dups on scan of sorted data
- Hash:
  - partition into N buckets
  - sort each bucket and remove dups

Index on projected fields

- scan the index pages, avoid reading data

Sorting is



if you squint, almost everything can be viewed as a join, and its ideas are constantly being rediscovered

For example, graph analysis – we saw that it is logically a join
When you have a dataset of actors and download a dataset of movies to analyze them, that's a join in space – performing type of join
When you high five a friend, that's a join in space.. .and time
When a florescent marker binds to a target protein, that's a chemical join

By thinking of this way, we can make use of everything we know about joins
One could say it's a fundamental operation in life

Datasets

```
from collections import defaultdict
from random import randint

M = 10000
N = 1000
outer = [ [randint(0, 1000), randint(0, 1000)]
          for i in xrange(M)]
inner = [ [randint(0, 1000), randint(0, 1000)]
          for i in xrange(N)]

# outer ⋈_1 inner
# outer JOIN inner ON outer.1 = inner.1
```

OK, we'll illustrate a few join algorithms using python code. You should be able to copy and past this code to run it

As opposed to cost estimation before, we'll be pretty general about join costs. Partly because you'll go into substantially more detail in 4112,
I want you to understand nested loops and indexed nested loops join,
for the other joins, just the properties

Nested Loops Join

```
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?

Different than cross product because output size is not M*N
but otherwise, the algorithm is the same!

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
- Assume hash index, 50 tuples per page, equality join on <u>sid</u>, 5% of outer tuples match
- Cost: M + (50M * 0.05 * 1page)

M pages * 50 tuples per page = 50M tuples in outer. Each is a probe of the index, and 5% of the probes are a hit that require accessing a data page (1 page)
+ M pages to read for the outer table.

You could imagine, if the inner table is small enough, building the hash table on the fly and running indexed nested loops join
or, building hash tables on both outer and inner tables
There are a bunch of variations of this idea, and they primarily think about how to best use RAM vs going to disk, as well as using sequential access of the disk

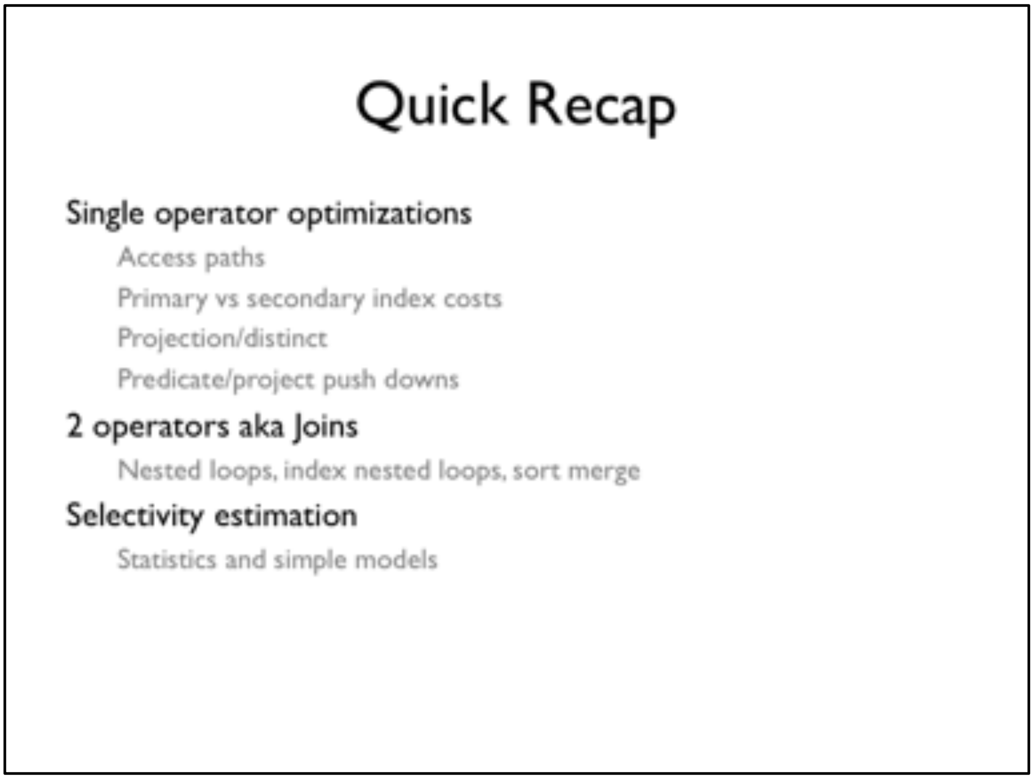
Sort Merge Join

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)

mention mcsherry's blog post – often for huge datasets, sort merge can out perform hash lookups due to random access vs sequential



Using an Index for Selections
Cost depends on #qualifying tuples.
●Cost of finding qualifying data entries (typically small) plus cost of retrieving records (could be large).
●In example, assuming uniform distribution of names, about 10% of tuples qualify (100 pages, 10000 tuples). With a “clustered” index, cost is little more than 100 I/Os; if “unclustered,” up to 10000 I/Os!

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.

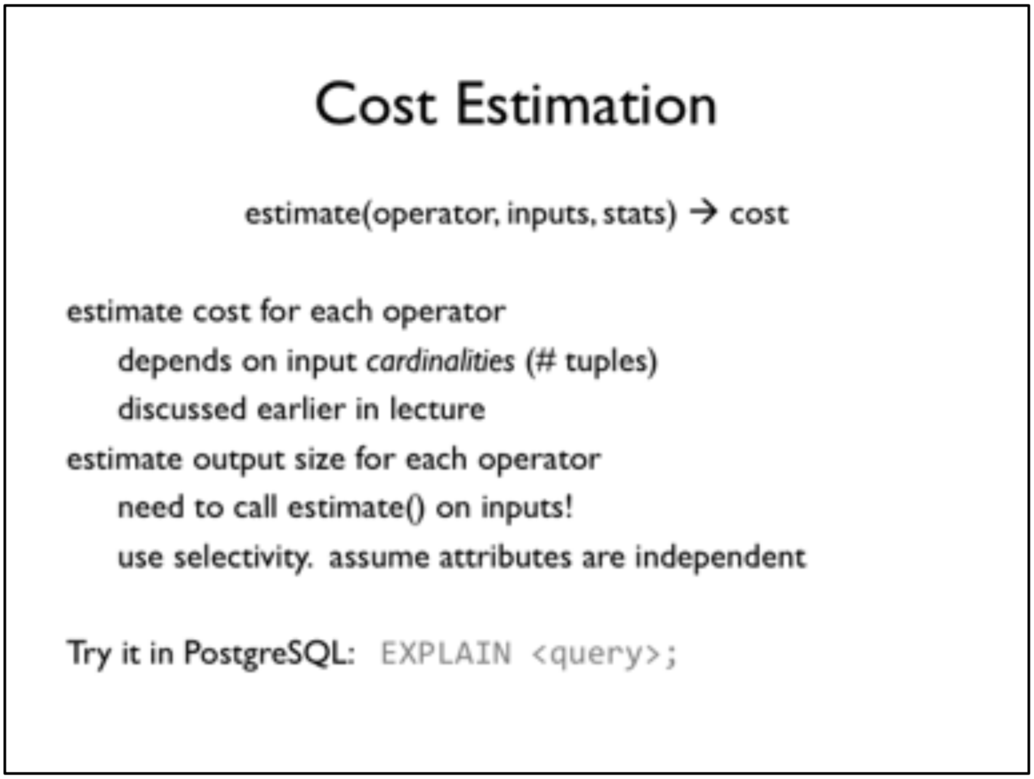
2.

Access Path Selection
in a Relational Database Management System

P. Griffiths Selinger
M. M. Astrahan
D. D. Chamberlin
R. A. Lorie
T. G. Price

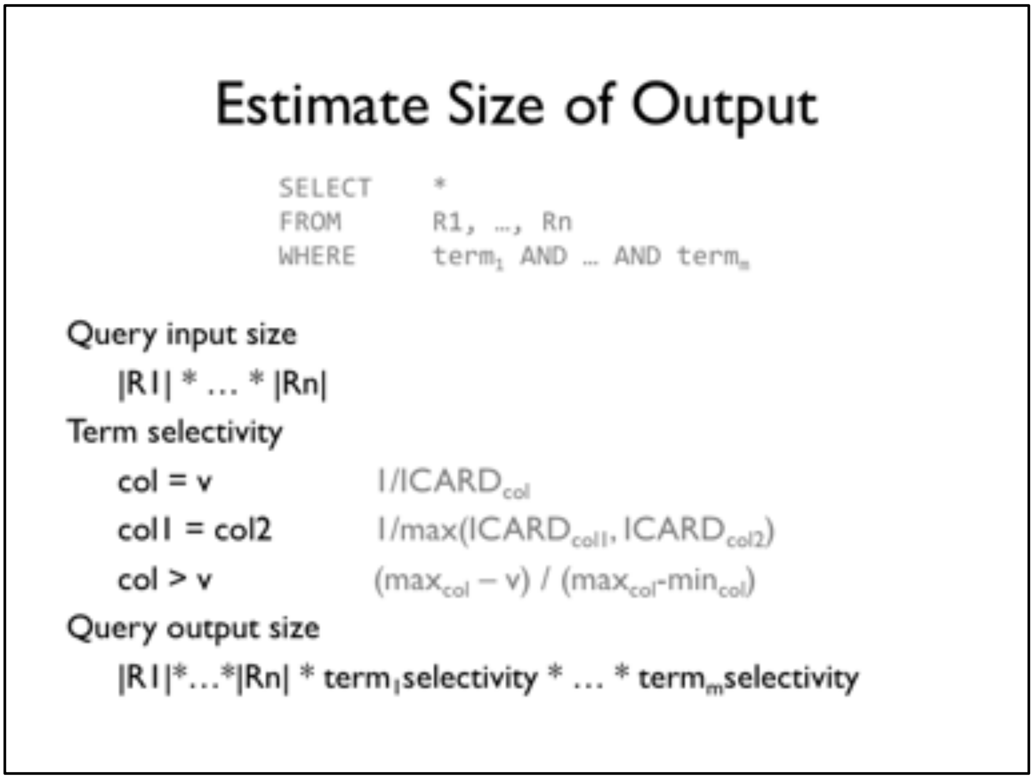
IBM Research Division, San Jose, California 95193

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.

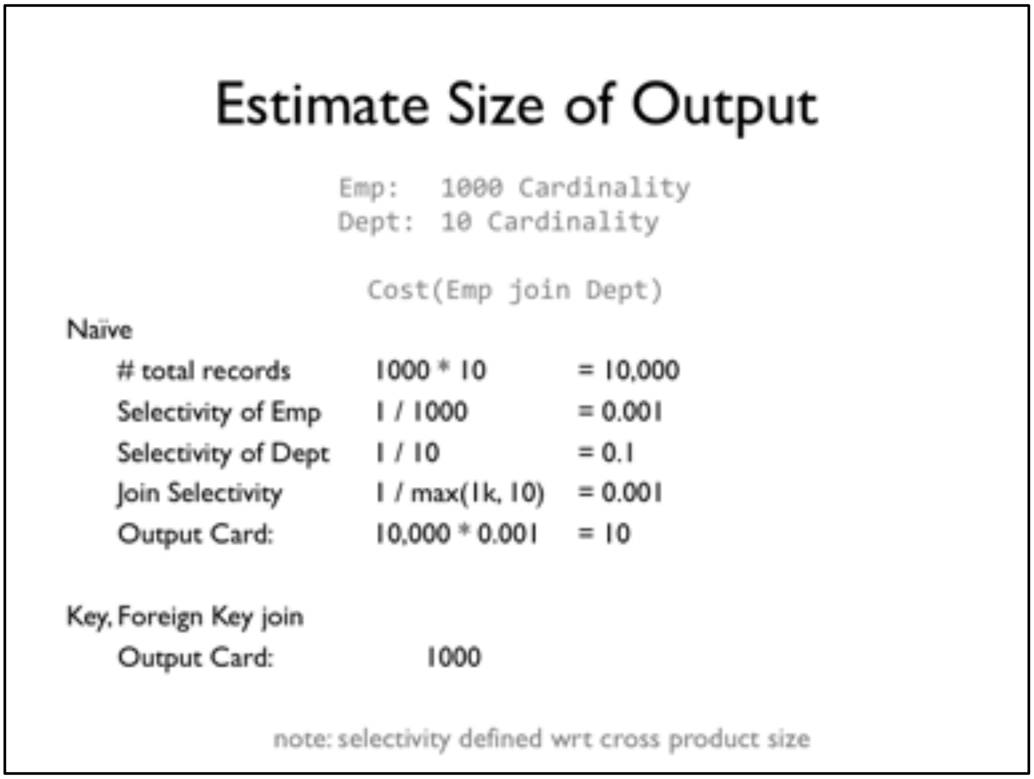


In order to estimate the cost of an operator, we need to be able to estimate the SIZE of its inputs.
Well the inputs may be other operators, so we need the ability to estimate output sizes

you'll note that I don't have any unitl for the cost. That's because estimators tend to be so wildly off that they bear almost no relation to wall clock time.
If you



ex: suppose emp has 1000 records, dept has 10 records
total records is 1000 * 10, selectivity is 1/1000, so 10 tuples expected to pass join
(note that this is wrong if doing key/fk join on emp.did = dept.did, which will produce 1000 results!)
Note that selectivity is defined relative to size of cross product for joins! p1 and p2



ex: suppose emp has 1000 records, dept has 10 records
total records is 1000 * 10, selectivity is 1/1000, so 10 tuples expected to pass join
(note that this is wrong if doing key/fk join on emp.did = dept.did, which will produce 1000 results!)
Note that selectivity is defined relative to size of cross product for joins! p1 and p2

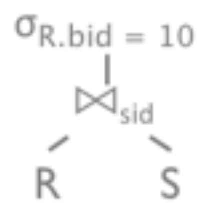
Try it out

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: M + MN
selection is pipelined

outputs: 0.0005MN



Try it out

R.sid = S.sid selectivity 0.01

R.bid selectivity 0.05

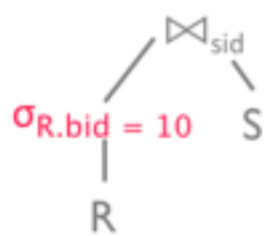
|R| = M

|S| = N

Cost: ?????

\# 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

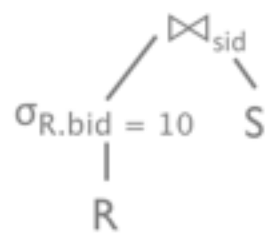
|R| = M

|S| = N

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*

- Cost Estimator

  "predict" cost of query from statistics

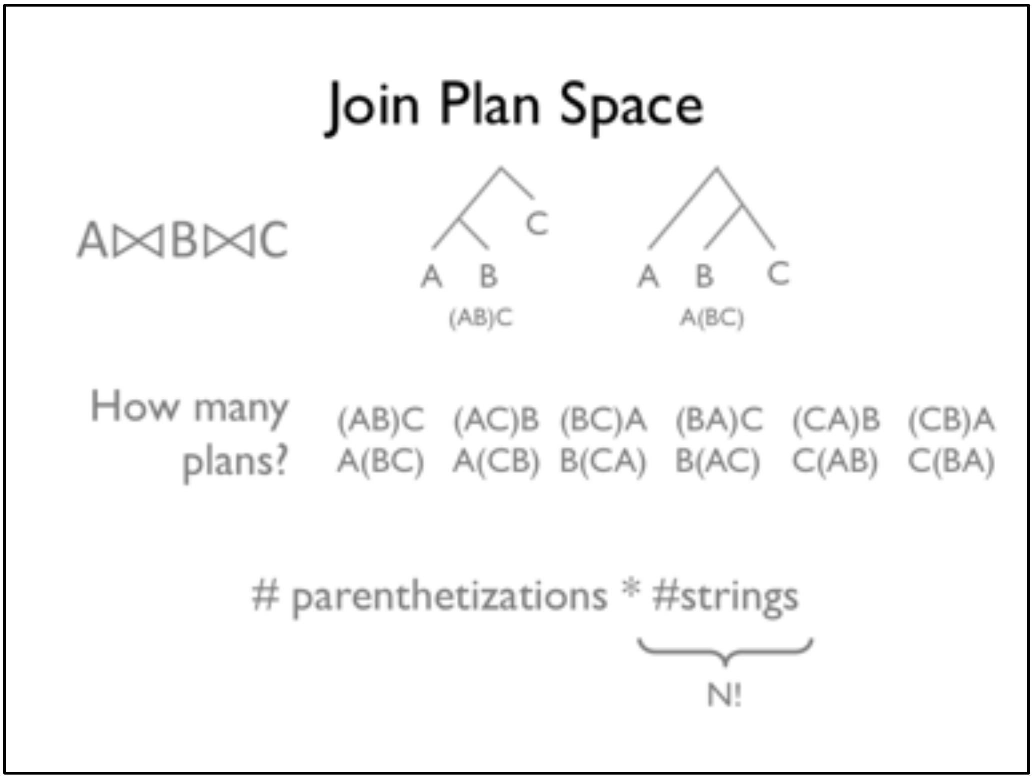
  Includes CPU, disk, memory, etc (can get sophisticated!)

  It's an art
- Plan Space

  avoid cross product

  push selections & projections to leaves as much as possible

  only join ordering remaining

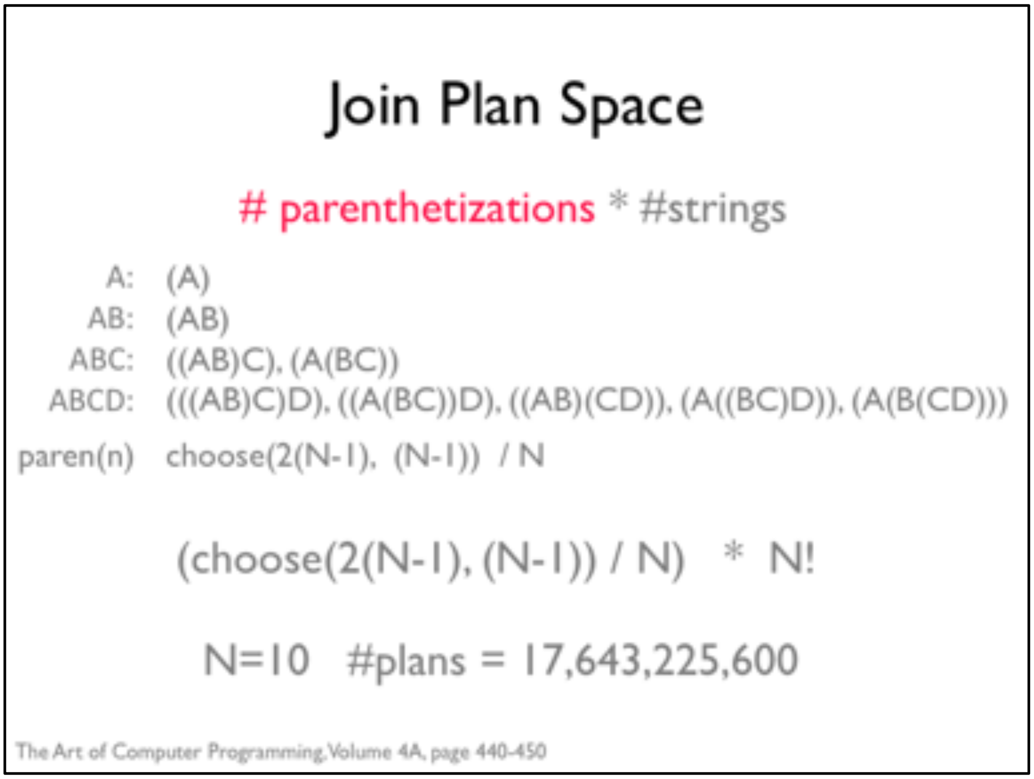


when talking about join orderings, we are ignoring the other operators (e.g., selection, projection), so a query plan is a single binary tree where each node is a join

#strings: For abc, how many distinct strings can these characters create?
#parens: For a given string, how many ways to construct a binary tree on top of that string?

Each column is a possible string ordering
2 possible parenthetizations for three tables
1*2*3 = 3! String orderings
6*2 possible join orders for three tables.

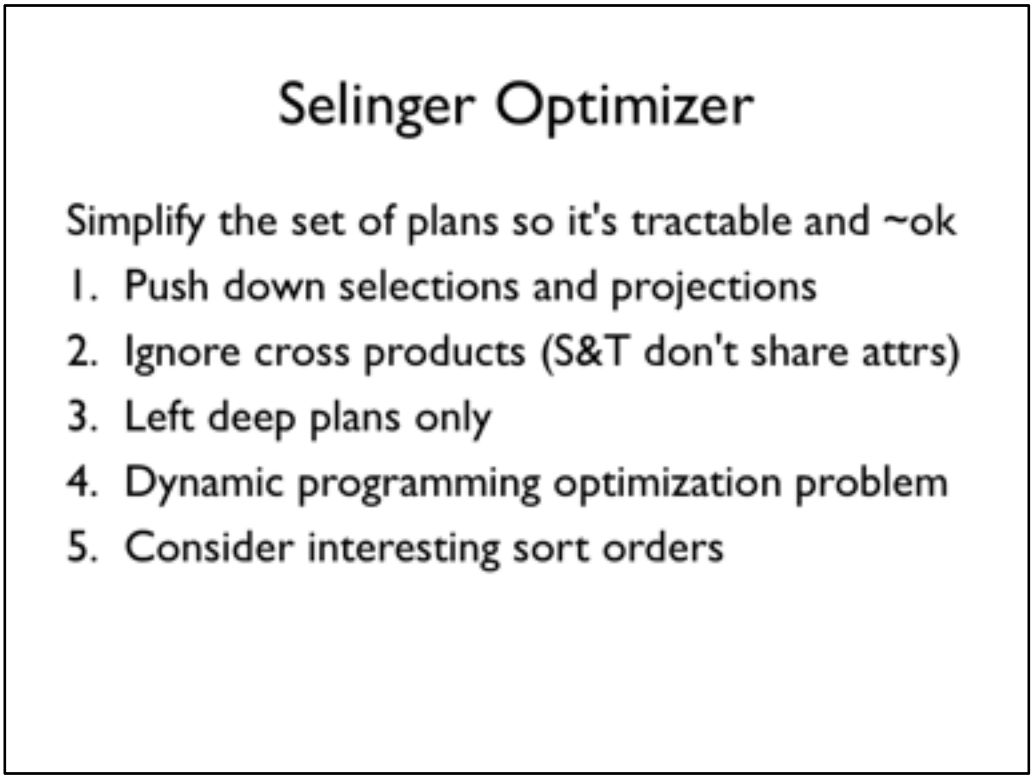
What about four?



How many possible parenthetizations?

==> n! * choose(2(N-1),(N-1))/(N)
==> 4 choose 2 / 3 == 6 / 3 = 2
6 * 2 == 12 for 3 relations

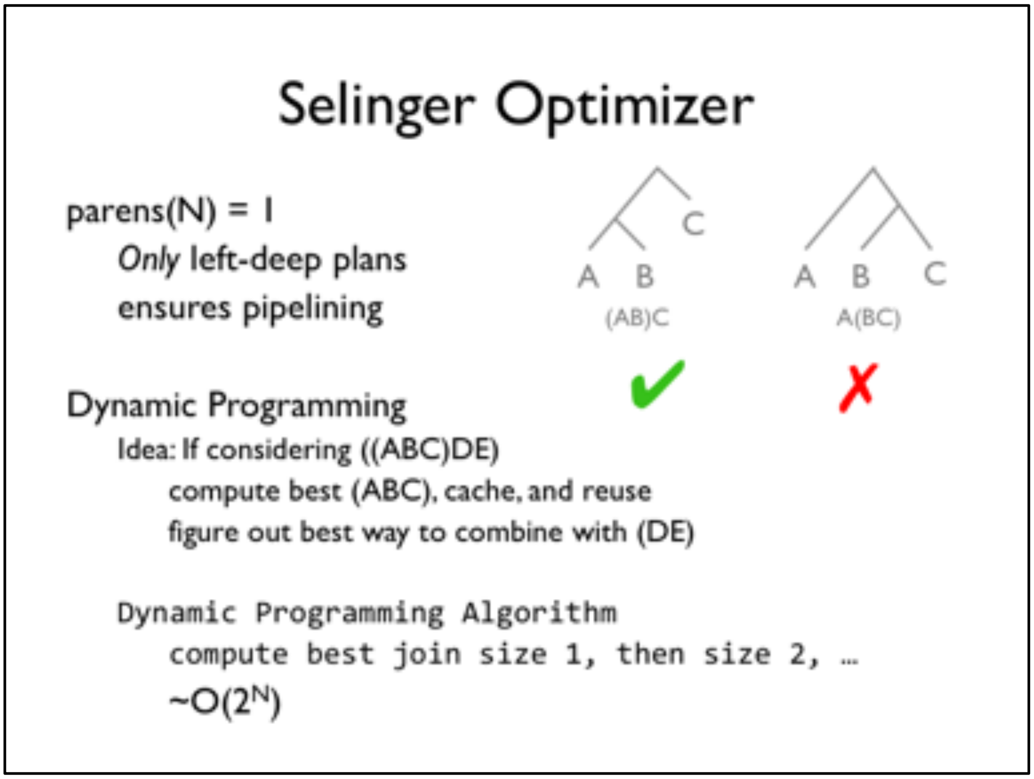
choose(2*(5-1), (5-1)) / 6



additional simplifying assumptions to make this tractable
At left deep plans only, still N! Possible tBle orders

This is a classic interview question, one way to solve is to simplify into a dynamic programming problem.

Finally run this for different sort orders that may make future joins or operators cheaper. Eg if there is an order by clause, or downstream join table is already sorted.

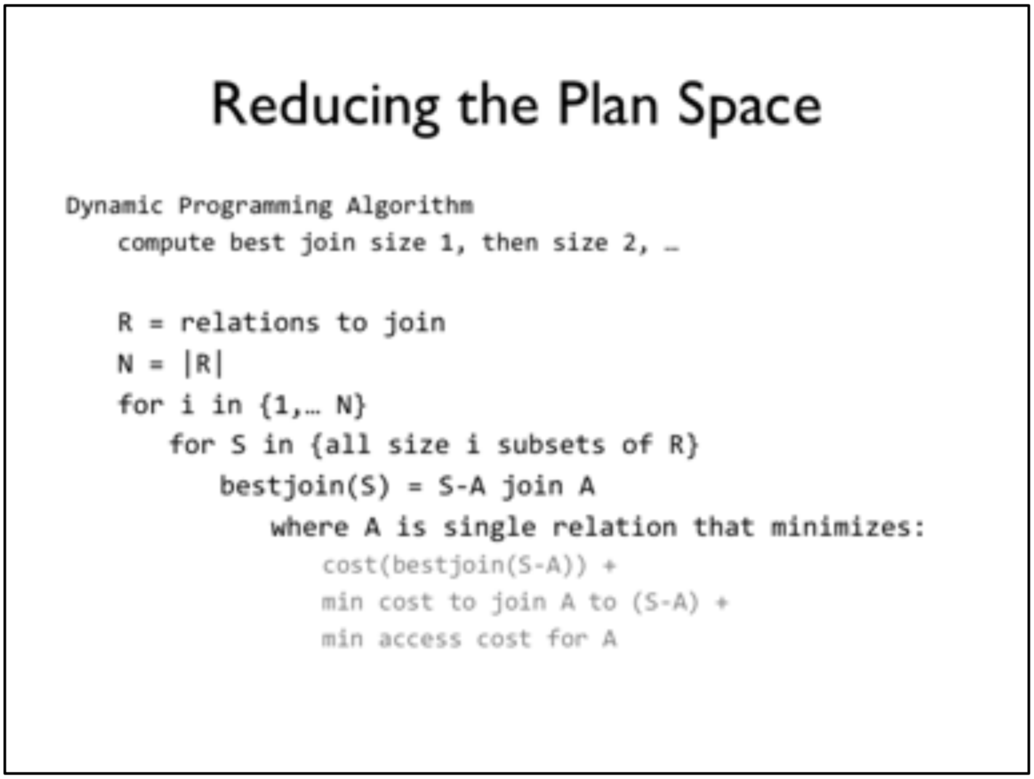


left deep: first join the two left most tables, then join the result with the next left-most table, and so on.
for a give node, only its left child is allowed to be a join operator

pipelining: right deep tree on the right: before A can join with it's inner relation (result of B join C), need to wait until B join C is completed

With left deep plan, the righ side of join is always a base table that is immediately available.
Note this applies to JOIN ORDERING. Selections and projects can still happen on the right side

Still pretty expensive, since the number of plans is parens=1 * N!



Recursive algorithm
In first iteration, look at all size one subsets of R.
For each table A in the subset, check how it can join with the rest of the subset. For n=1, basically what is the best access path for A

For next iteration, look at all size two subsets. Say PQ, then check:
A=P: cost(bestjoin(Q)) + costofbestjoinoperator(P, Q) + minaccesscost(P).
Bedtjoin(Q) already computed

A=Q. Same procedure

Selinger Algorithm i = 1

bestjoin(ABC), only nested loops join

i = 1
A = best way to access A
B = best way to access B
C = best way to access C

cost: N relations

Selinger Algorithm i = 2

bestjoin(ABC)

i = 2
A,B = (A)B or (B)A
A,C = (A)C or (C)A
B,C = (B)C or (C)B

cost: choose(N, 2) * 2

There are choose n, 2 ways to pick a set of size 2. Then there are two possible orders of the joins.

Selinger Algorithm i = 3

bestjoin(ABC)

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 ways to remove a relation A +
1 way to join A with R-A (if only NLJ)
< N

Cost = $N*2^N$
N = 12 49152

1 way to join A with R-A only because considering left deep NL join only. otherwies it should be M, where M is the numbero fpossible join algorithms

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!