Administrivia

Project 2 not ready :(

How did the evaluations go?

How did the projects go?

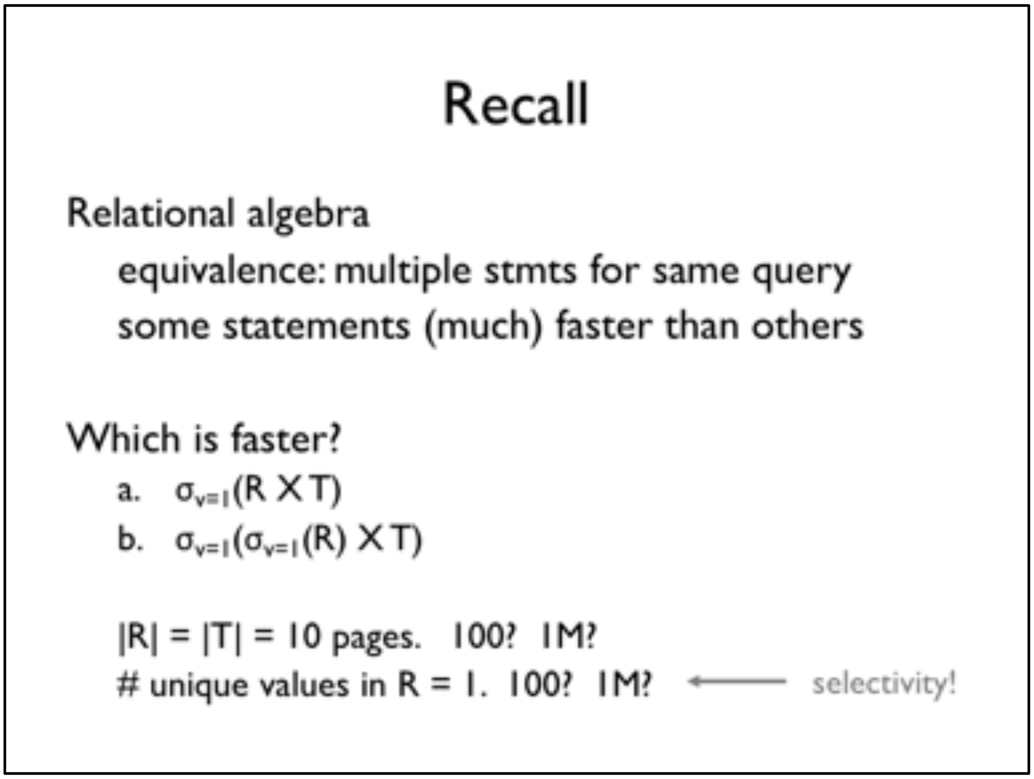
L9
Query Execution & Optimization

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



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

Overview of Query Optimization

SQL → query plan

How plans are executed

Some implementations of operators

Cost estimation of a plan

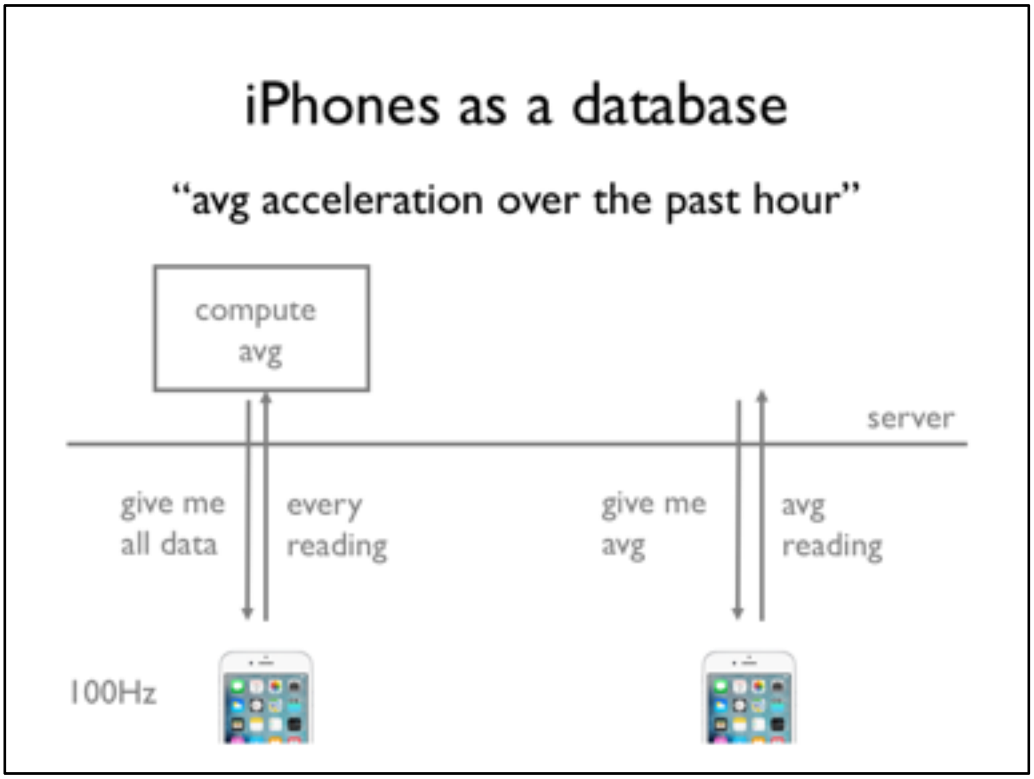
- Selectivity

System R dynamic programming

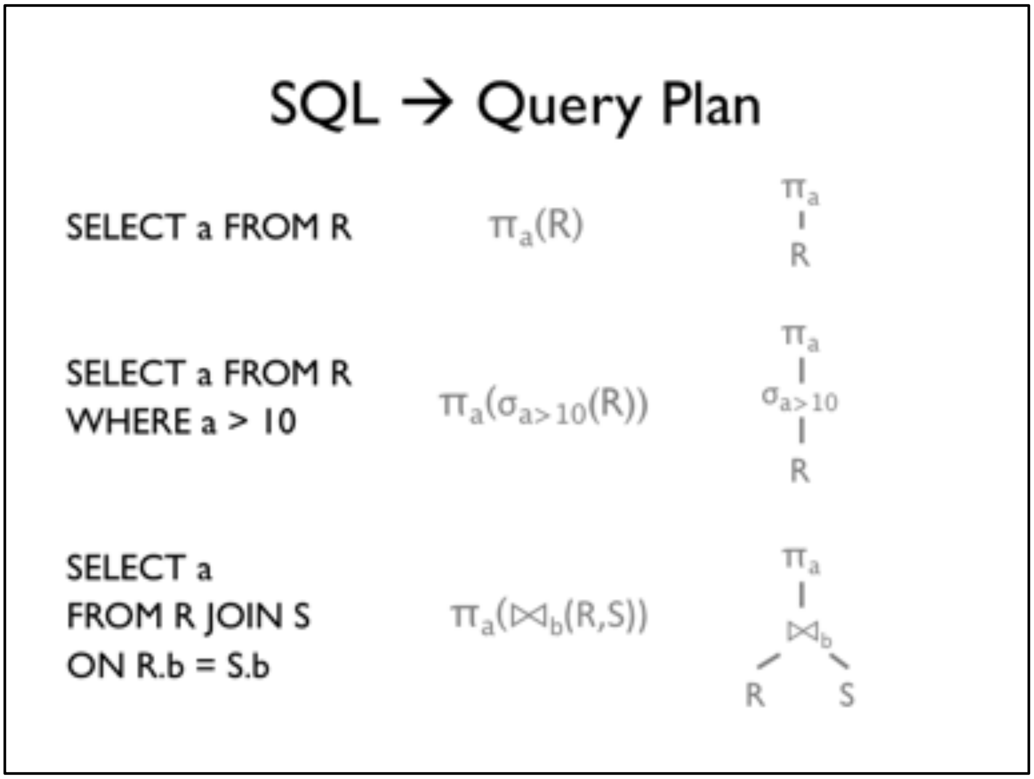
All ideas from System R's "Selinger Optimizer" 1979

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?

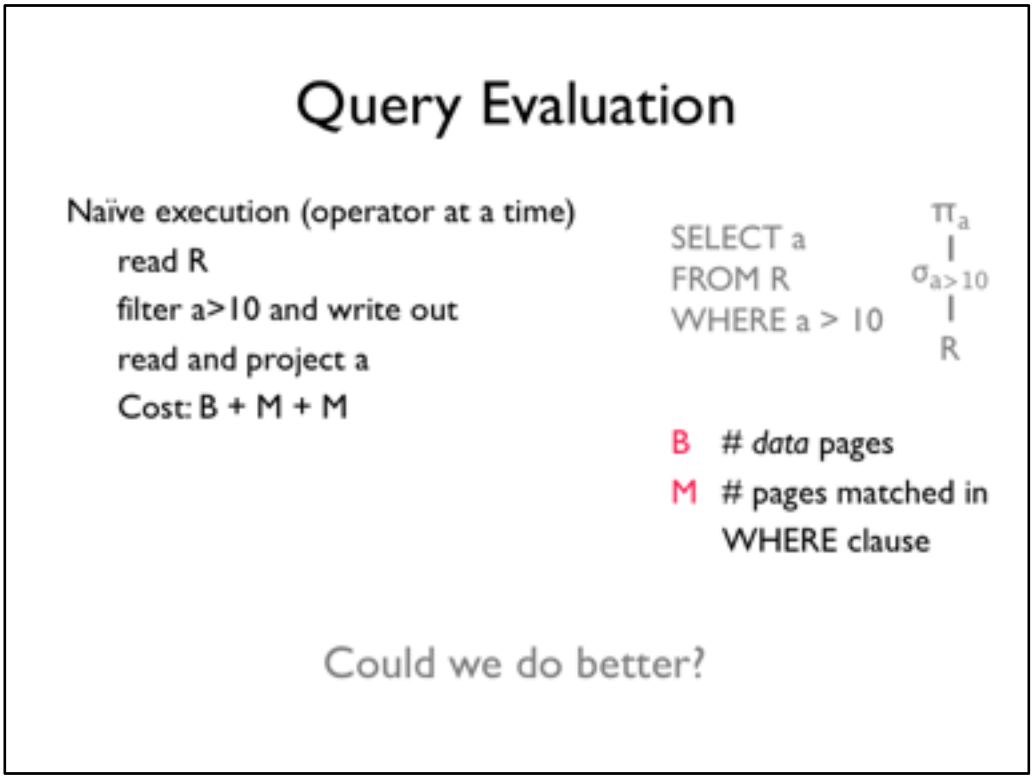
Push

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

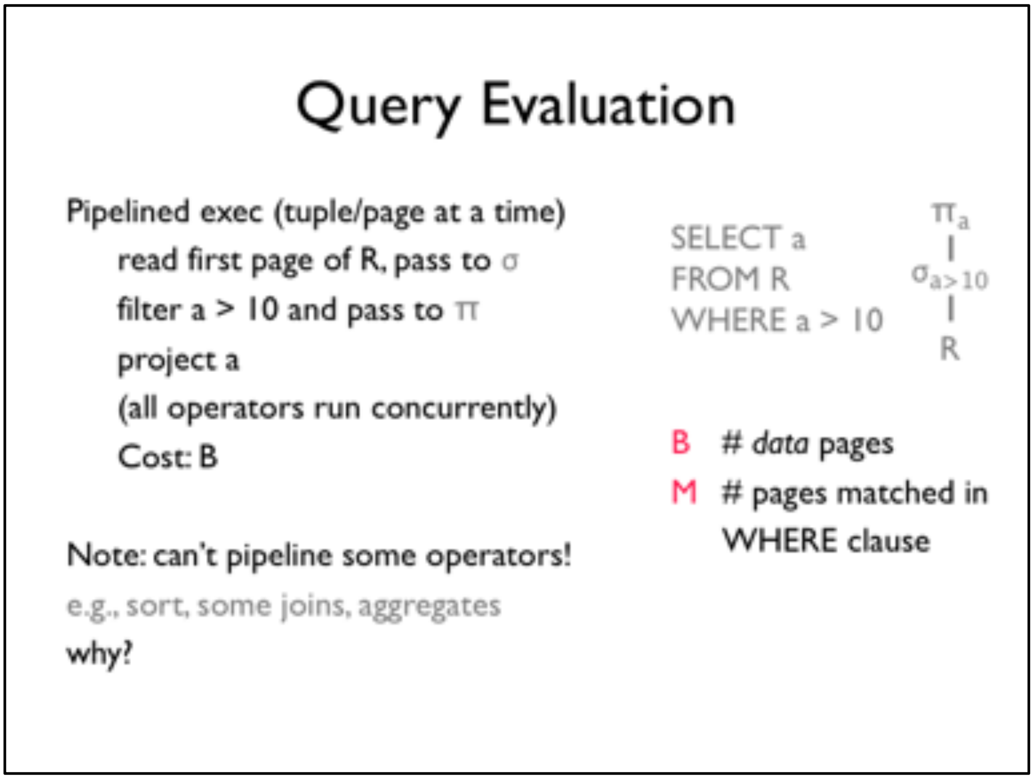
Pull

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

Are cursors push or pull?



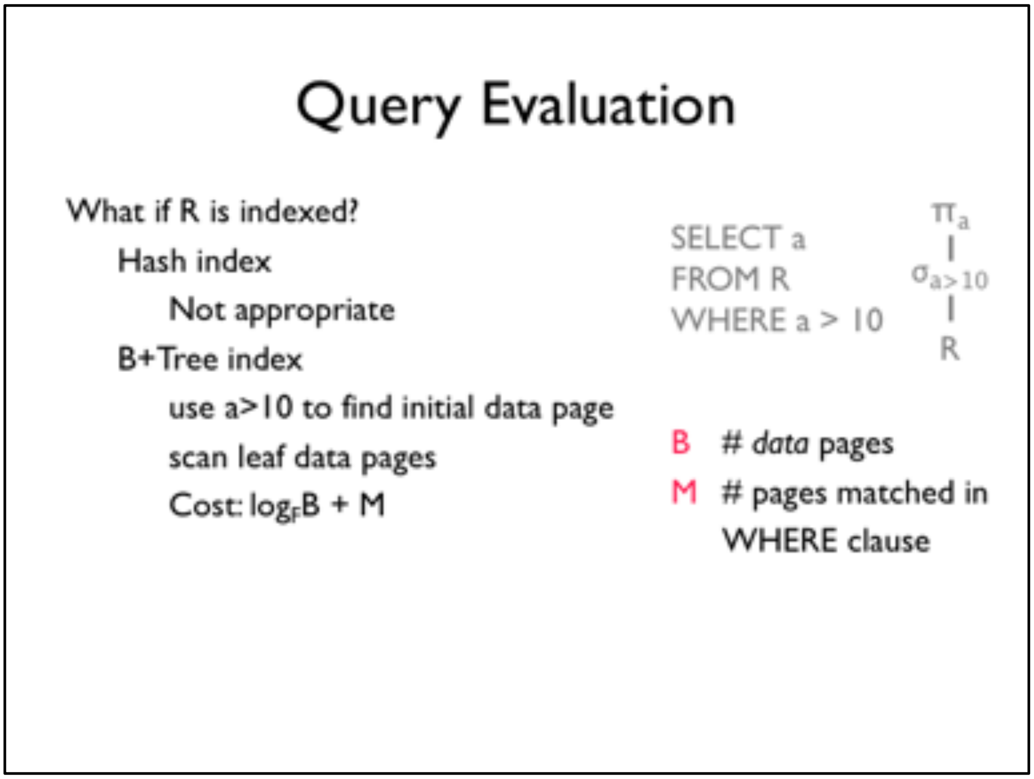
I read R, maybe after the filter the result doesn't fit into memory, or just barely, so I need to write it out. then I go to next operator, and read its data to execute the projection



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?

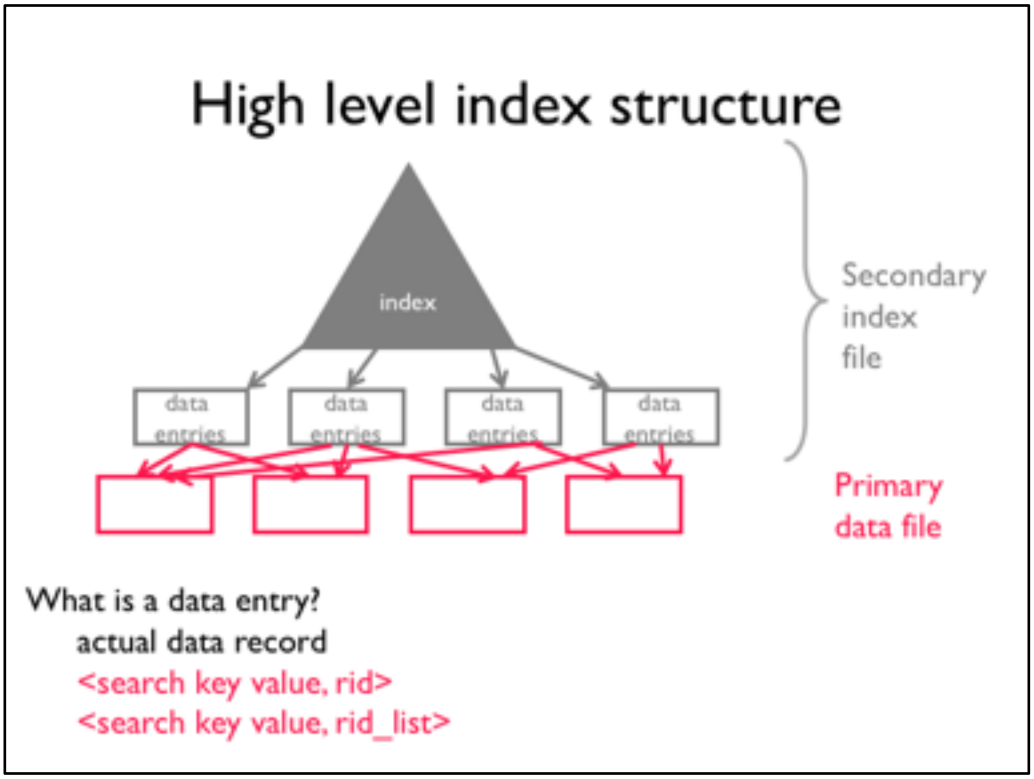
Selectivity

- 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

How to pick Access Paths?

Hash index on <a, b, c>

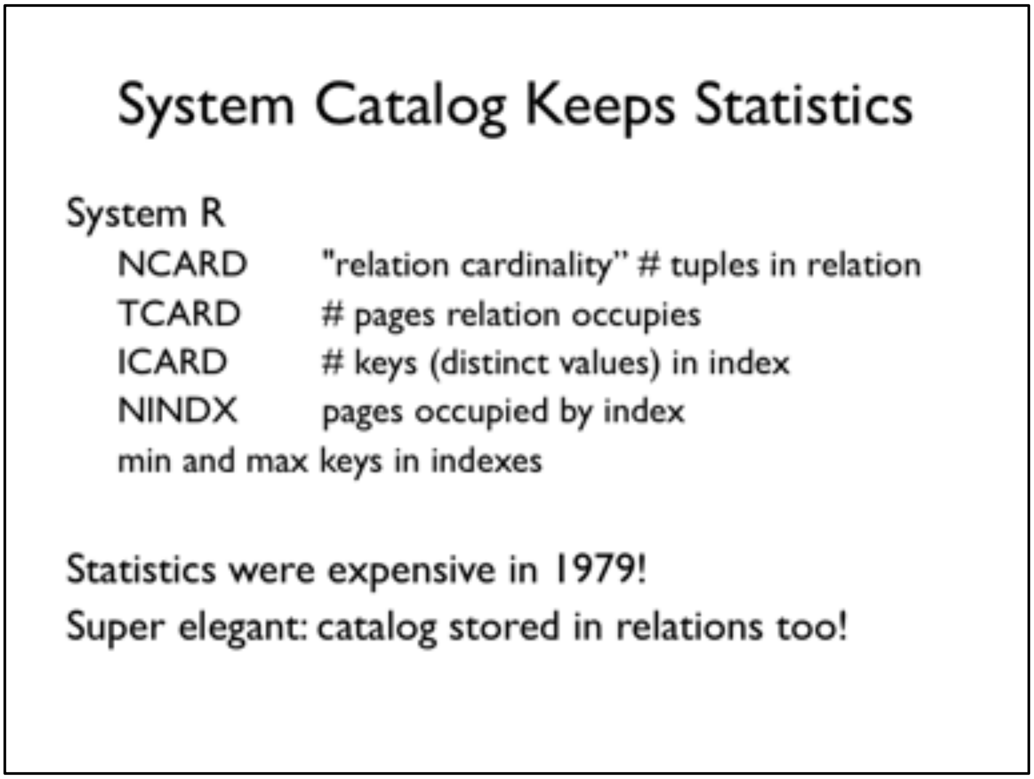
a = 1, b = 1, c = 1 how to estimate selectivity?

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

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?

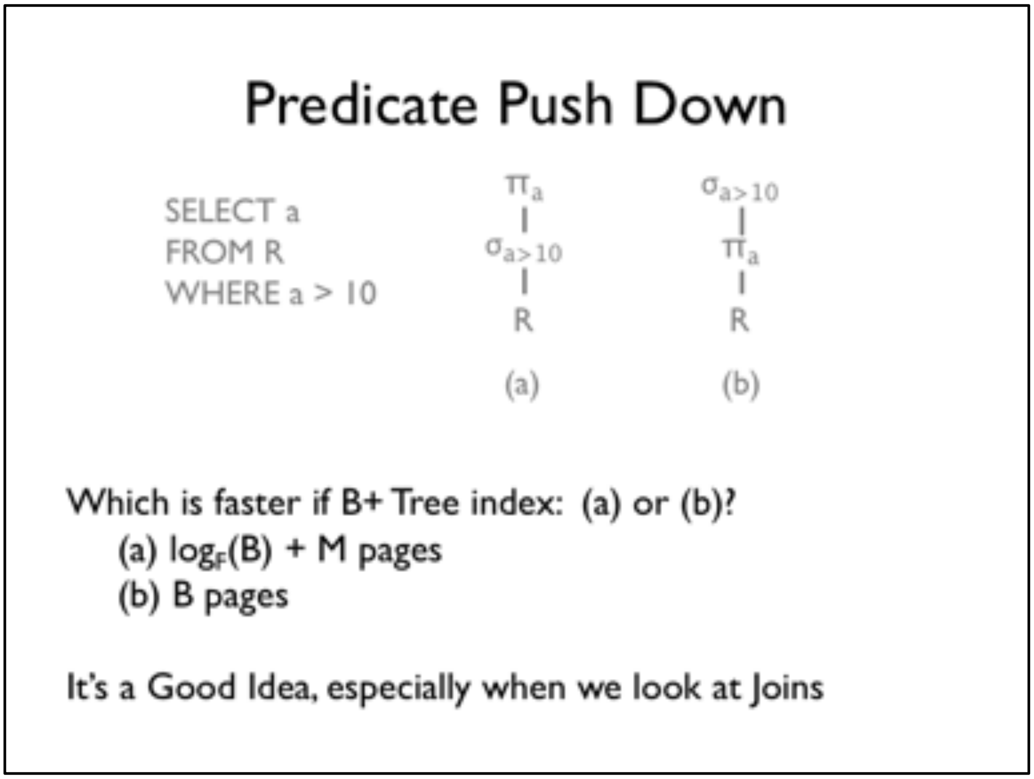
Access Path ✔

Predicate push-down

Join implementation

Join ordering

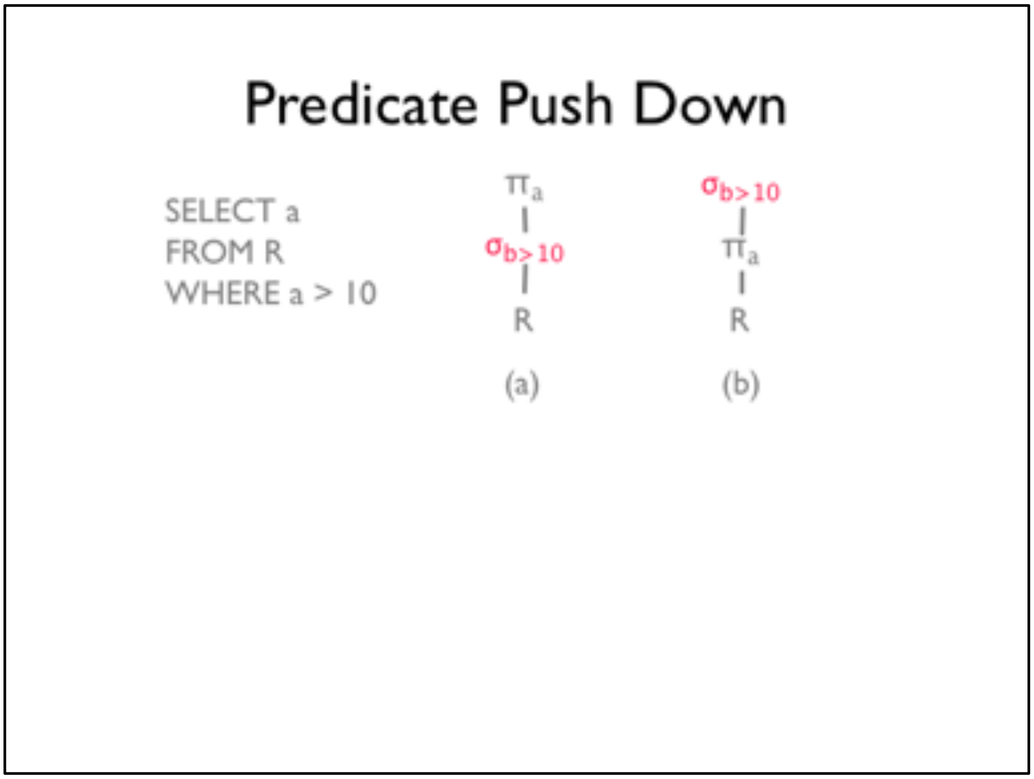
In general, depends on operator implementations. So let's take a look



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

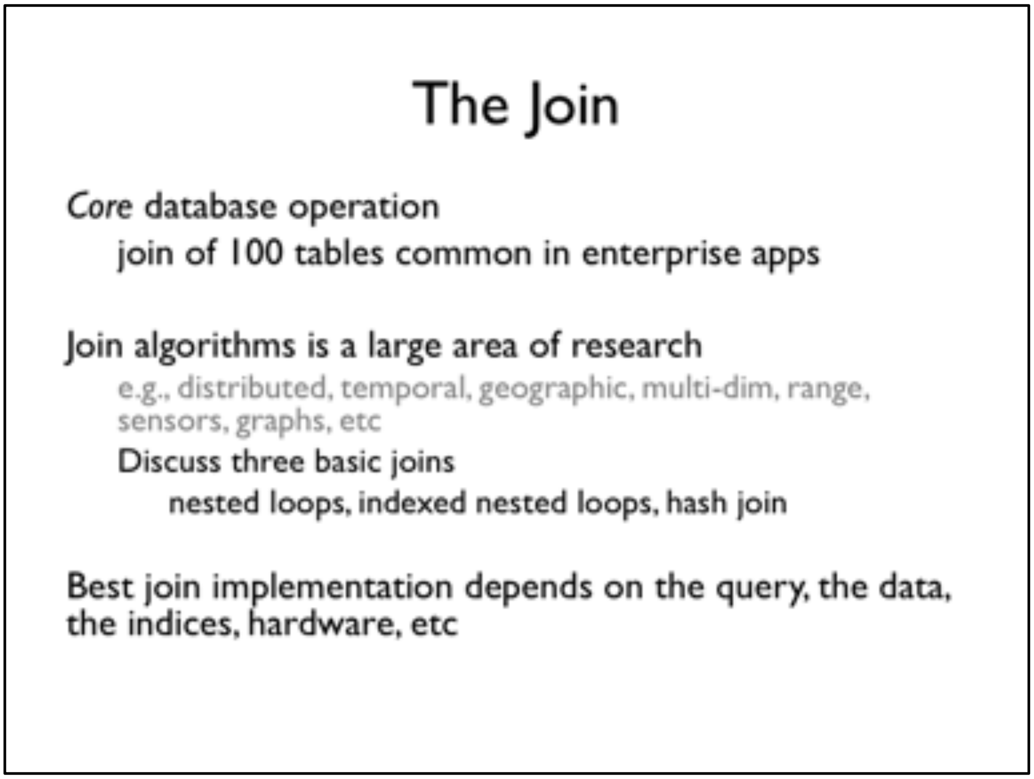
What would be a rule we could create that tells us to pick the faster one?



You have to be a bit careful about how you do the predicate pushdown and when youa er allowed to

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)

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

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

System R 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!)

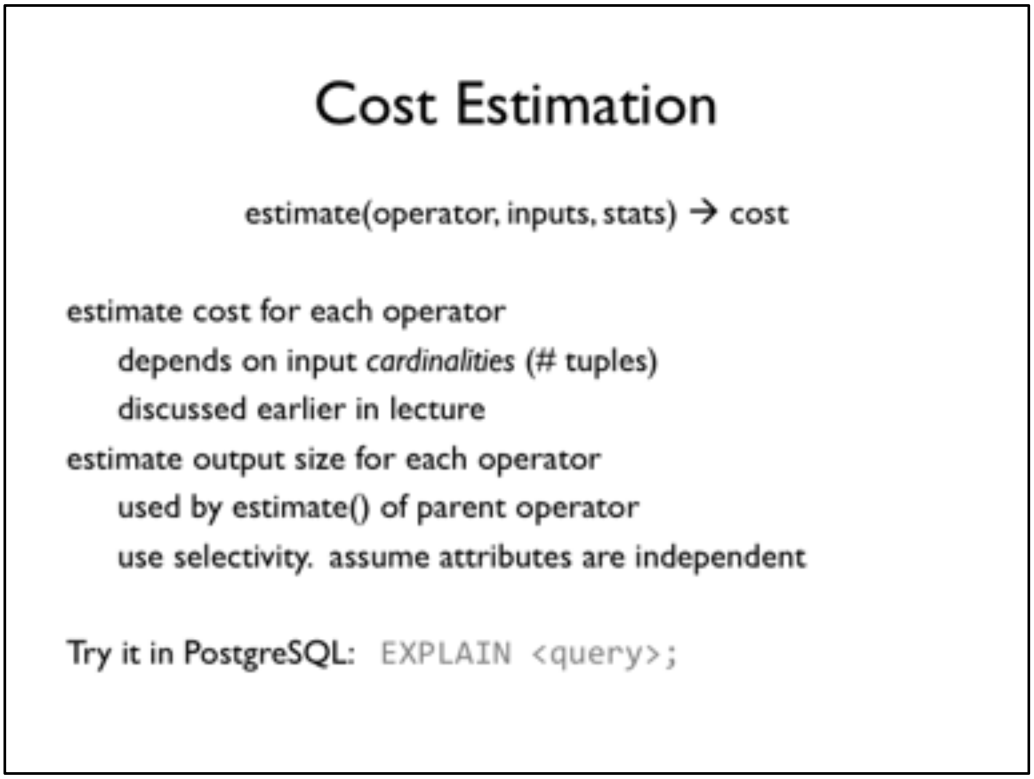
  It's an art

- Plan Space

  avoid cross product

  push selections & projections to leaves as much as possible

  only join ordering remaining



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

Estimate Size of Output

```
SELECT    *
FROM      R1, …, Rn
WHERE     term1 AND … AND termm
```

Query input size

|R1| * … * |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

$|R1|*…*|Rn| * term_1selectivity * … * term_mselectivity$

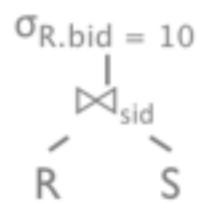
Try it out

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
projection is pipelined

outputs: 0.0005MN



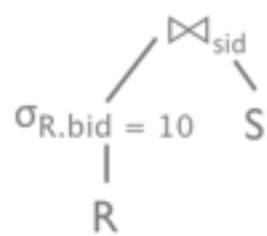
Try it out

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 + (0.05MN)
projection is pipelined

outputs: 0.0005MN



Plan Space

A⋈B⋈C

How many plans?

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

parentheizations * #strings

Plan Space

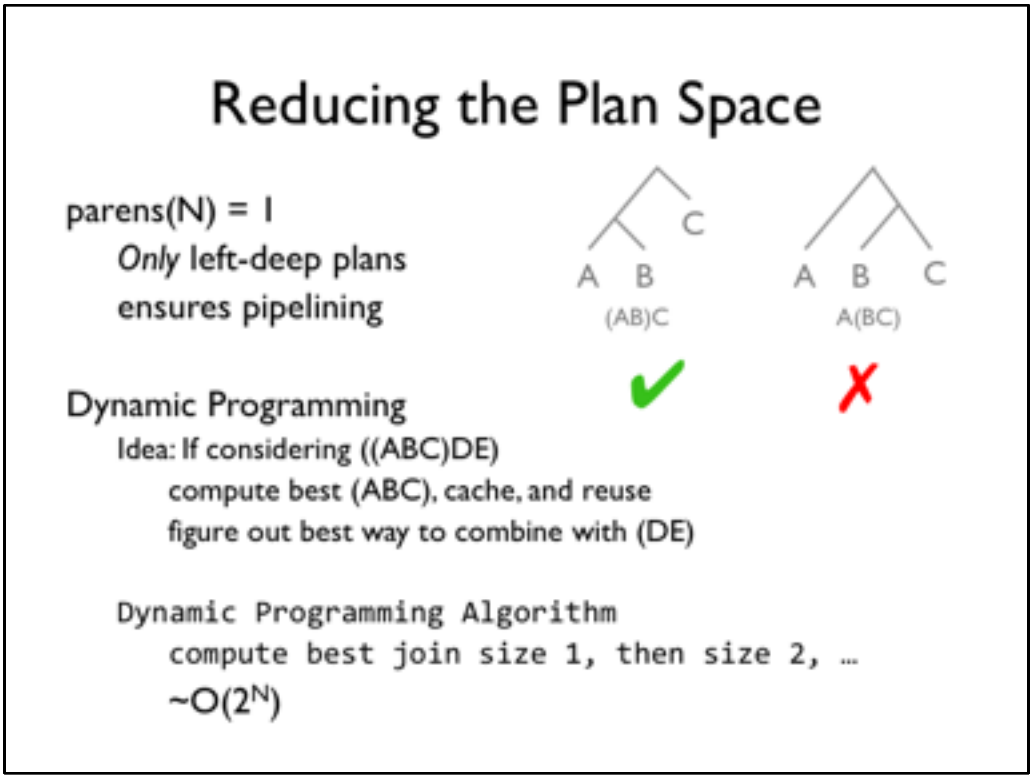
parenthetizations * #strings

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=1, #plans = 17,643,225,600



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

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

Reducing the Plan Space

```
Dynamic Programming Algorithm
    compute best join size 1, then size 2, …

    R = relations to join
    for n in {1,… |R|}
        for S in {all size n subsets of R}
            bestjoin(S) = A join S-A
                where A is single relation that minimizes:
                cost(bestjoin(S-A)) +
                min cost to join A to (S-A) +
                min access cost for A
```

Selinger Algorithm N = 1

bestjoin(ABC), only nested loops join

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

cost: N relations

Selinger Algorithm N = 2

bestjoin(ABC)

N = 2
A,B = AB or BA
A,C = AC or CA
B,C = BC or CB

cost: choose(N, 2) * 2

Selinger Algorithm N = 3

bestjoin(ABC)

N = 3

A,B,C = (AB)C or (AC)B or (BC)A

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 +

2 ways to join A with R-A

< 2N

Cost = $2N*2^N$

2 ways to join A with R-A only because considering NL join only. otherwies it should be 2* M, where M is the numbero fpossible join algorithms