Lecture 03.04.

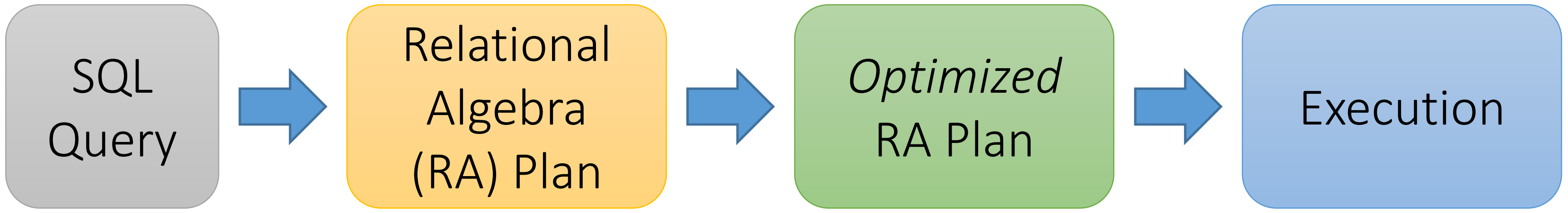
Query evaluation

Cost-based transformations

By Marina Barsky
Winter 2017, University of Toronto

RDBMS query evaluation

How does a RDBMS answer your query?



Declarative query (user declares what results are needed)

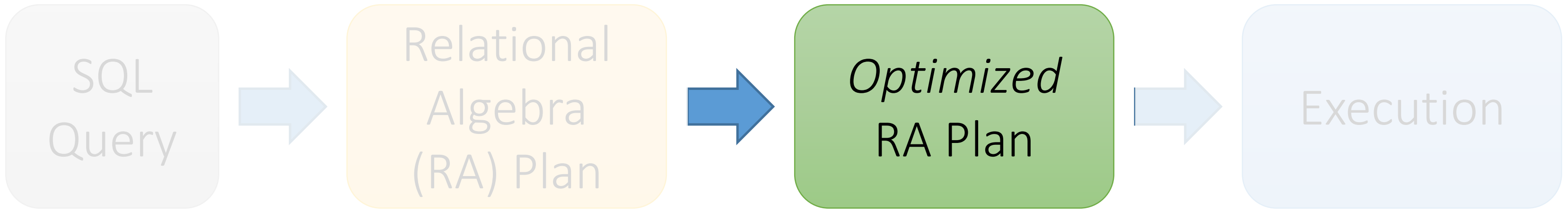
Translate to relational algebra expression

Find logically equivalent- but more efficient- RA expression

Execute each operator of the optimized plan!

RDBMS query evaluation

How does a RDBMS answer your query?

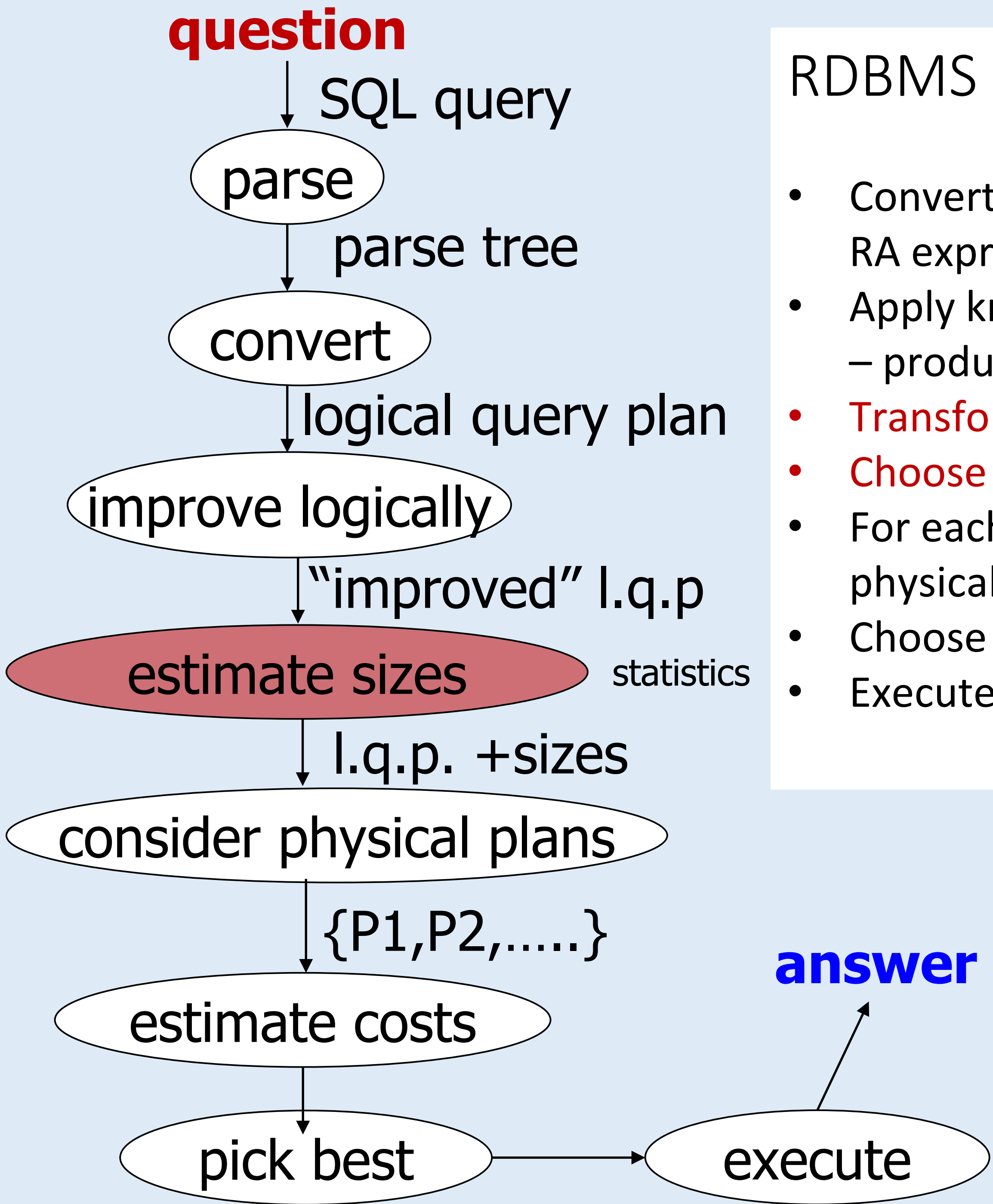


Declarative query (user declares what results are needed)

Translate to relational algebra expression

Find logically equivalent- but more efficient- RA expression

Execute each operator of the optimized plan!



RDBMS query optimizer: steps

- Convert parsed SQL into corresponding RA expression
- Apply known algebraic transformations – produce improved logical query plan
- Transform based on estimated cost
- Choose one min-cost logical expression
- For each step, consider alternative physical implementations
- Choose physical plan with min I/Os
- Execute

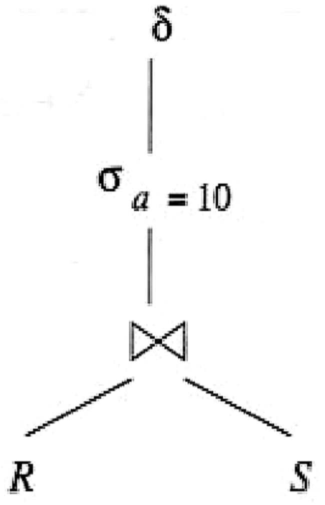
Cost-based query optimization

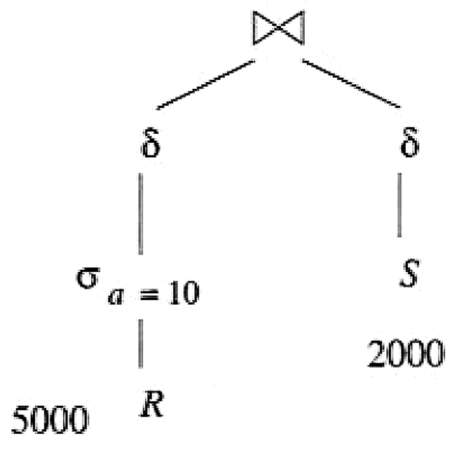
- Sometimes we don't need to compute the cost to decide applying some heuristic transformation.
  - E.g. Pushing selections down the tree, can be expected almost certainly to improve the cost of a logical query plan.

- However, there are points where estimating the cost both before and after a transformation will allow us to apply a transformation where it appears to reduce cost and avoid the transformation otherwise.

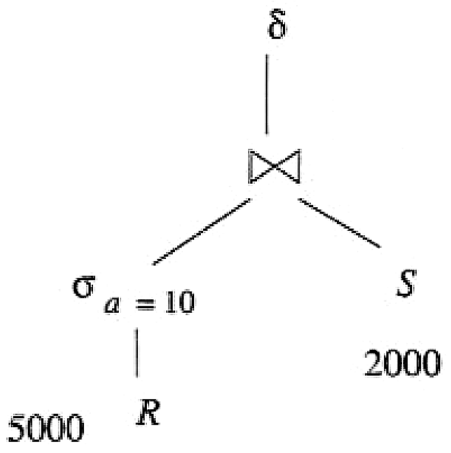
Example: which plan to choose

$R(a,b)$	$S(b,c)$
$T(R) = 5000$	$T(S) = 2000$
$V(R,a) = 50$	
$V(R,b) = 100$	$V(S,b) = 200$
	$V(S,c) = 100$





(a)



(b)

Initial logical query plan

Two candidates for the best logical query plan.
Which one to choose?

Parameters to estimate the cost

- **R**: the name of the relation on disk
- **B(R)**: number of blocks of R
- **T(R)**: number of tuples of R
- **V(R, a)**: number of distinct values in column ***a*** of R

Plus:

- Indexes
- Clustered-unclustered

Estimating the output of each operation

- How can we estimate B(R) and T(R) in an intermediate relation?

- We don’t want to execute the query in order to learn the sizes. So, we need to estimate them.

Rules about estimation formulas:

1. Give (somehow) accurate estimates
2. Easy to compute

Estimating the output: Projection

The size of a (bag) projection is the only one we can compute **exactly**.

1. Bag projection retains duplicates, so the number of tuples T(R) in the result is the same as in the input.
2. Result tuples are usually shorter than the input tuples.

So what changes in the parameters of the output relation?

Estimating the output: Selection

Equality: $S = \sigma_{A=c}(R)$

We can estimate the size of the result as

$$\mathbf{T(S) = T(R) / V(R,A)}$$

Range: $S = \sigma_{A<c}(R)$

On average, T(S) would be T(R)/2, but from practice:

$$\mathbf{T(S) = T(R)/3}$$

Inequality: $S = \sigma_{A \neq c}(R)$

Then, an estimate is: $T(S) = T(R) * [(V(R,A)-1)/V(R,A)]$,

or close to

$$\mathbf{T(S) = T(R)}$$

Estimating the output: Selection with conjunction

$\mathbf{S = \sigma_{C\ AND\ D}(R) = \sigma_C(\sigma_D(R))}$

Remember? Splitting rule

First estimate $T(\sigma_D(R))$ and then use this to estimate T(S).

Example

$S = \sigma_{a=10\ AND\ b<20}(R)$

$T(R) = 10{,}000,\ V(R,a) = 50$

$T(S) = (1/50) * (1/3) * T(R) = 67$

Note: Watch for selections like:

$\sigma_{a=10\ AND\ a>20}(R)$

Why?

Cost-based optimization

1. Estimating cost of intermediate results
2. Maintaining vital statistics
3. Selecting order of joins

Estimating the output: Selection

Equality: $S = \sigma_{A=c}(R)$

$T(S) = T(R) / V(R,A)$

Range: $S = \sigma_{A<c}(R)$

$T(S) = T(R)/3$

Inequality: $S = \sigma_{A\neq c}(R)$

$T(S) = T(R)$

Estimating the output: Selection with disjunction

$S = \sigma_{C\ OR\ D}(R)$

Add up: $T(S) = T(\sigma_C(R)) + T(\sigma_D(R))$.

Problem: It's possible that $T(S)>T(R)$!

A more accurate estimate

Let:

$T(R)=n$,

m_1 = size of selection on C, and

m_2 = size of selection on D.

Then **$T(S) = n(1-(1-m_1/n)(1-m_2/n))$**

Why?

Selection with disjunction: example

Simple: $T(S) = T(\sigma_C(R)) + T(\sigma_D(R))$

Accurate: $T(S) = n(1-(1-m_1/n)(1-m_2/n))$

- **Example**: $S = \sigma_{a=10 \text{ OR } b<20}(R)$.

  $T(R) = 10{,}000$, $V(R,a) = 50$

- Simple estimation: *$T(S) = 3533$*
- More accurate:

$n = 10000$, $m1 = T(R)/V(R,a) = 200$, $m2 = T(R)/3 = 3333$

$T(S) = 10000 * [1 - (1 – 200/10000) * (1 – 3333/10000)] = 3466$

Natural Join $R(X,Y) \bowtie S(Y,Z)$

- Anything could happen!

Extremes

- No tuples join

  $T(R \bowtie S) = 0$

- All tuples join: *i.e.* R.Y=S.Y = *a*

  $T(R \bowtie S) = T(R)*T(S)$

Natural Join size estimation: heuristics

Estimate size of: $P = R(X,Y) \bowtie S(Y,Z)$

If $V(R,Y) \leq V(S,Y)$, then (in the worst case) each tuple **r** of **R** is going match with some tuples of **S**. How many tuples on average that match to a given value of Y? $T(S)/V(S,Y)$ tuples of S.

Hence,

$$T(P) = T(R)*T(S)/V(S,Y)$$

By a similar reasoning, for the case when $V(S,Y) \leq V(R,Y)$, we get

$$T(P) = T(S)*T(R)/V(R,Y)$$

In summary: $T(P) = T(R)*T(S)/\max\{V(R,Y),V(S,Y)\}$

$T(R \bowtie S) = T(R)*T(S)/\max\{V(R,Y),V(S,Y)\}$

Join output estimation: example

R(a,b):	T(R)=1000,	V(R,b)=20	
S(b,c):	T(S)=2000,	V(S,b)=50,	V(S,c)=100
U(c,d):	T(U)=5000,	V(U,c)=500	

a. Estimate the size of intermediate outputs for **($R \bowtie S$) $\bowtie$ U**

$T(R \bowtie S) = 1000*2000 / 50 =$ **40,000**

T (($R \bowtie S$**)** $\bowtie U) = 40000 * 5000 / 500 =$ **400,000**

b. Estimate the size of intermediate outputs for **R $\bowtie$ (S $\bowtie$ U)**

$T(S \bowtie U) =$ **20,000**

$T(R \bowtie$ **(**$S \bowtie U$**))** $= 1000*20000 / 50 =$ **400,000**

intermediate results could be of different sizes

Estimate of **final result** should not depend on the evaluation order!

T(R ⊳⊲ S) = T(R)*T(S)/max{V(R,Y),V(S,Y)}

Join output estimation: more

c. Yet another order for **R ⊳⊲ S ⊳⊲ U:** (R × U) ⊳⊲ S

R(a,b):	T(R)=1000,	V(R,b)=20	
S(b,c):	T(S)=2000,	V(S,b)=50,	V(S,c)=100
U(c,d):	T(U)=5000,	V(U,c)=500	

T(R × U) = 1000*5000 = **5,000,000**

Note that the cardinality of *b*'s in the product is 20 (=V(R,b)), and the cardinality of *c*'s is 500 (=V(U,c)).

T(**(**R × U**)** $\rhd\lhd_{R.b=S.b \text{ AND } U.c=S.c}$ S) = 5,000,000 * 2000 / (50 * 500) = **400,000**

Join output estimation: the question is…

A $T(R \bowtie S) = 1000*2000 / 50 = \mathbf{40{,}000}$

$T((R \bowtie S) \bowtie U) = 40000 * 5000 / 500 = \mathbf{400{,}000}$

T(U)=5000

B $T(S \bowtie U) = \mathbf{20{,}000}$

$T(R \bowtie (S \bowtie U)) = 1000*20000 / 50 = \mathbf{400{,}000}$

T(R)=1000

C $T(R \times U) = 1000*5000 = \mathbf{5{,}000{,}000}$

$T((R \times U) \bowtie S) = 5{,}000{,}000 * 2000 / (50 * 500) = \mathbf{400{,}000}$

T(S)=2000

Which order is better?

Size estimates for other operators

Cartesian product: $T(R \times S) = T(R) * T(S)$

Bag Union: $T(R) + T(S)$

Set union:

larger + half the smaller. Why?

Because a set union can be as large as the sum of sizes or as small as the larger of the two arguments. Something in the middle is suggested.

Intersection: half the smaller. Why?

Because intersection can be as small as 0 or as large as the sizes of the smaller. Something in the middle is suggested.

Difference: $T(R-S) = T(R) - 1/2*T(S)$

Because the result can be between T(R) and T(R)-T(S). Something in the middle is suggested.

Size estimates for other operators (contd.)

Duplicate elimination δ in $(R(a_1,\ldots,a_n))$:

The size ranges from 1 to T(R).

$T(\delta(R)) = V(R,[a_1 \ldots a_n])$, if available (but usually not available).

Otherwise: $T(\delta(R)) = \min[V(R,a_1) * \ldots * V(R,a_n), 1/2 * T(R)]$ is suggested.

Why?

$V(R,a_1) * \ldots * V(R,a_n)$ is the upper limit on the number of distinct tuples that could exist

$1/2 * T(R)$ is because the size can be as small as 1 or as big as T(R)

Grouping and Aggregation:

similar to δ, but only with respect to grouping attributes.

Exercises for size estimation

W (a,b)	X (b,c)	Y (c,d)	Z (d,e)
T (W) = 100	T (X) = 200	T (Y) = 300	T (Z) = 400
V (W, a) = 20	V (X, b) = 50	V (Y, c) = 50	V (Z, d) = 40
V (W, b) = 60	V (X, c) = 100	V (Y, d) = 50	V (Z, e) = 100

Estimate the sizes of relations that are the results of the following expressions:

a) $W \bowtie X \bowtie Y \bowtie Z$
b) $\sigma_{a=10} (W)$
c) $\sigma_{c=20} (W)$
d) $\sigma_{c=20} (Y) \bowtie Z$
e) $W \times Y$
f) $\sigma_{d>10} (Z)$
g) $\sigma_{a=1 \text{ AND } b=2} (W)$
h) $\sigma_{a=1 \text{ AND } b>2} (W)$

Computing the statistics

- In order to compute sizes of intermediate relations we need to know T(R) and V(R) for each relation. In addition we would want to know more precise estimates for each attribute value if the values are skewed

- Computation of statistics by DBMS is triggered automatically or manually.

- T(R)'s, and V(R,A)'s are just aggregation queries (COUNT queries).
- However, they are expensive to compute after each update operation.

Incremental computation of statistics

Maintaining T(R):

Add 1 for every insertion and subtract 1 for every deletion.

Incremental computation of statistics

Maintaining V(R,A):

If there is an index on attribute A of a relation R, then:

> On insert into R, we must find the A-value for the new tuple in the index anyway, and so we can determine whether there is already such a value for A. If not increment V(R,A).
>
> On deletion…

If there is no index on A, the system could in effect create a rudimentary index by keeping a data structure (e.g. B-Tree) that holds every distinct value of A.

Final option: Sample the relation.

For non-uniform value distributions we want to estimate frequency of each value

- For **index selection**:
    - What is the cost of an index lookup?

- Also for **deciding which algorithm to use**:
    - Ex: To execute $\mathrm{R} \bowtie S$, which join algorithm should DBMS use?

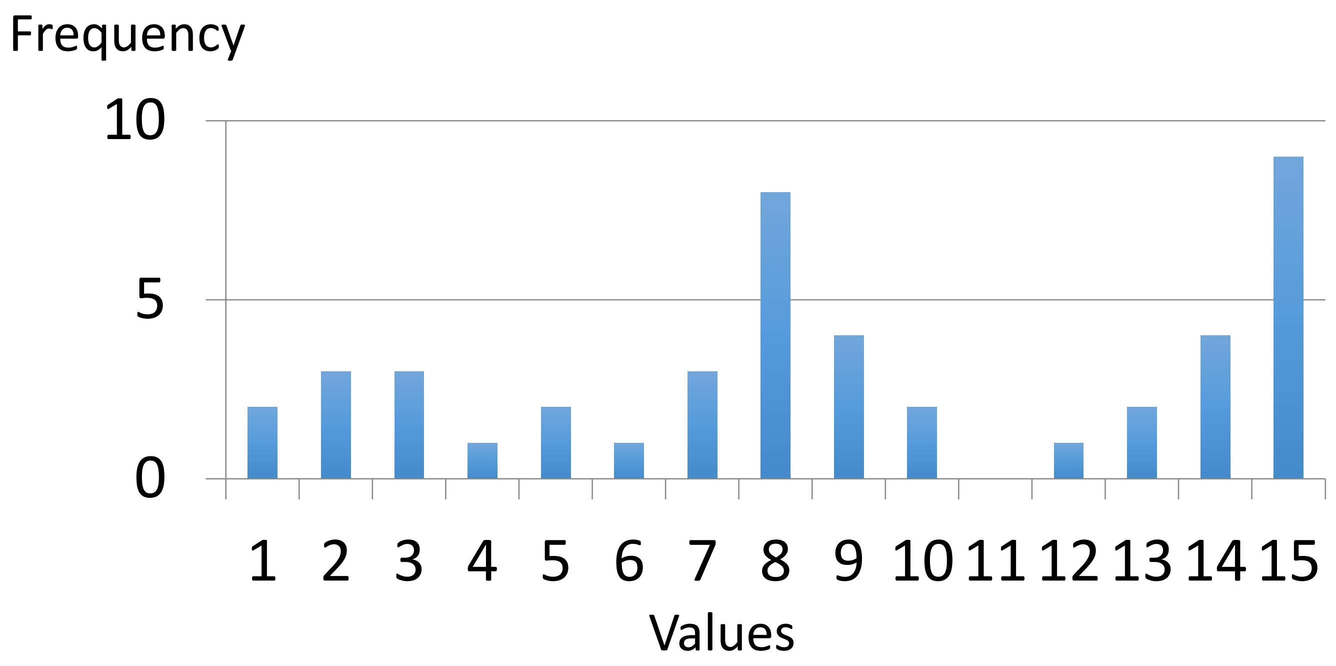
    - **What if we want to compute $\boldsymbol{\sigma_{A>10}(\mathbf{R}) \bowtie \sigma_{B=1}(S)}$?**

Histograms provide a way to efficiently store estimates of these quantities

Histograms

- A histogram is a set of value ranges (“buckets”) and the frequencies of values occurring in those buckets
- How to choose the buckets?
  - Equiwidth & Equidepth
- Turns out high-frequency values are **very** important

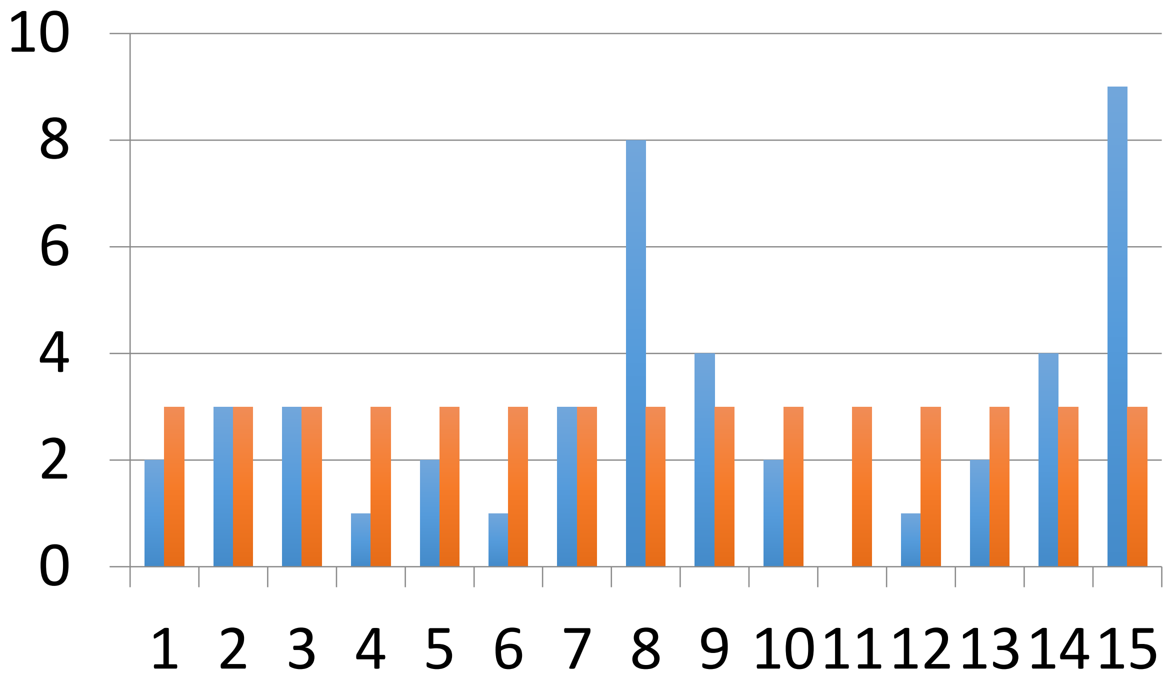
Store counts for each distinct value



How do we compute how many values between 8 and 10?
(Yes, it's obvious)

Problem: counts take up too much space!

Full vs. Uniform Counts



How much space do the full counts (bucket_size=1) take?

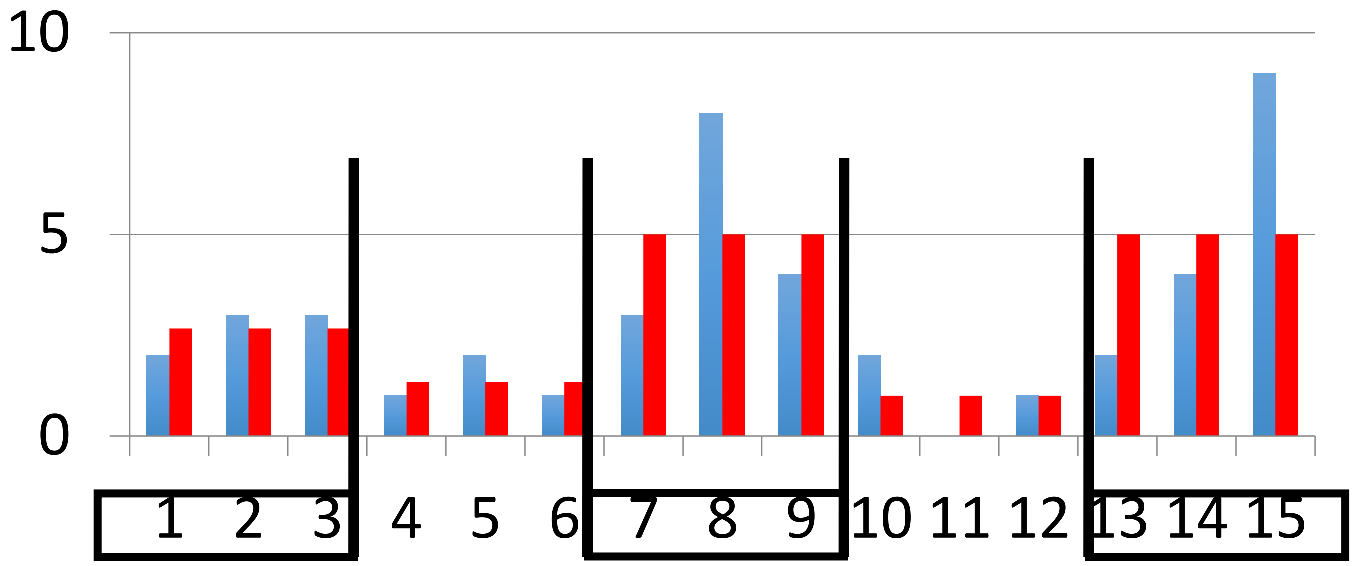
How much space do the uniform counts (bucket_size=ALL) take?

Fundamental Tradeoffs

- Want high resolution (like the full counts)
- Want low space (like uniform)
- Histograms are a compromise!

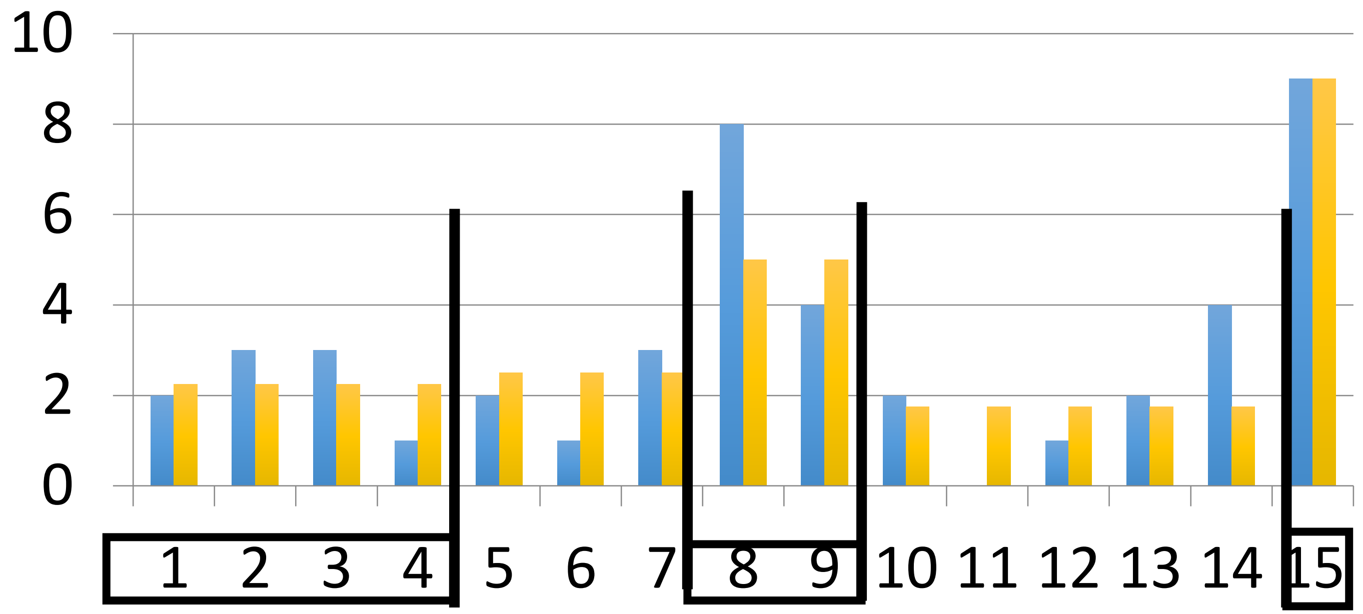
So how do we select the “bucket” sizes?

Equi-width



All buckets roughly the same width

Equi-depth



All buckets contain roughly the same number of items (total frequency)

Histograms

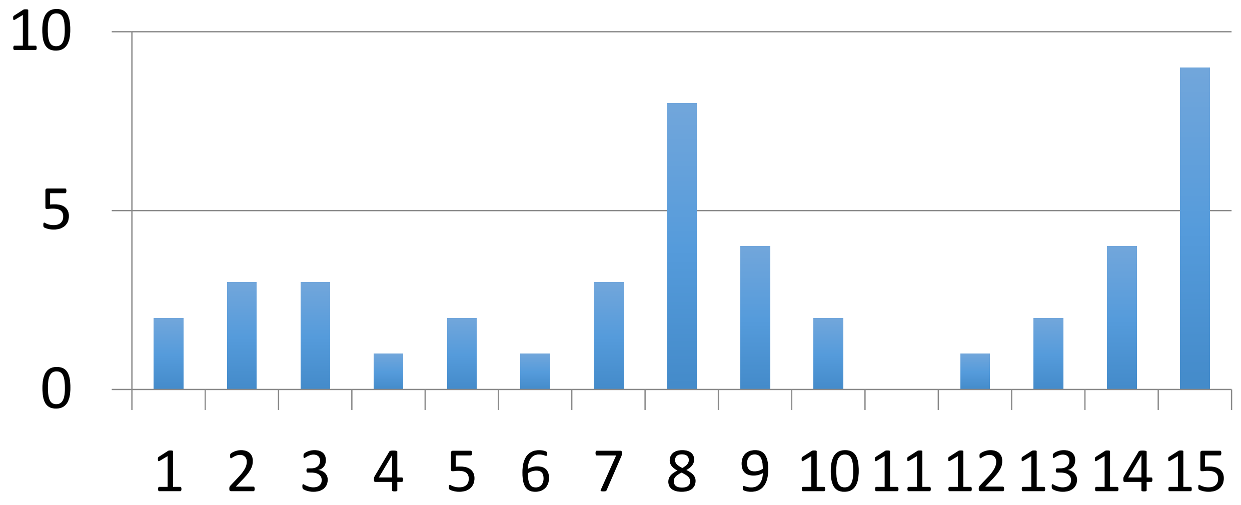
- Simple, intuitive and popular
- Parameters: # of buckets and type
- Can extend to many attributes (multidimensional)

Maintaining Histograms

- Histograms require that we update them!
  - Typically, you must run/schedule a command to update statistics on the database
  - Out of date histograms can be terrible!

- There is research work on self-tuning histograms and the use of query feedback
  - Oracle 11g

Nasty example



1. we insert many tuples with value > 16
2. we do **not** update the histogram
3. we ask for values > 20?

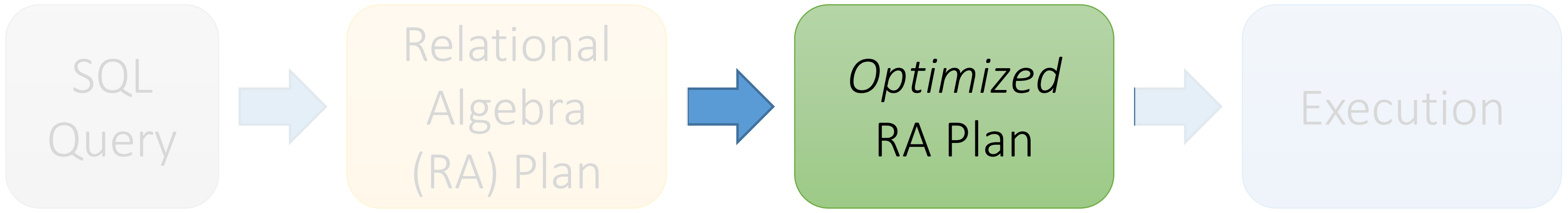
Compressed Histograms

- One popular approach:
  1. Store the most frequent values and their counts explicitly
  2. Keep an equi-width or equi-depth one for the rest of the values

People continue to try all manner of fanciness here
wavelets, graphical models, entropy models,…

RDBMS query evaluation

How does a RDBMS answer your query?



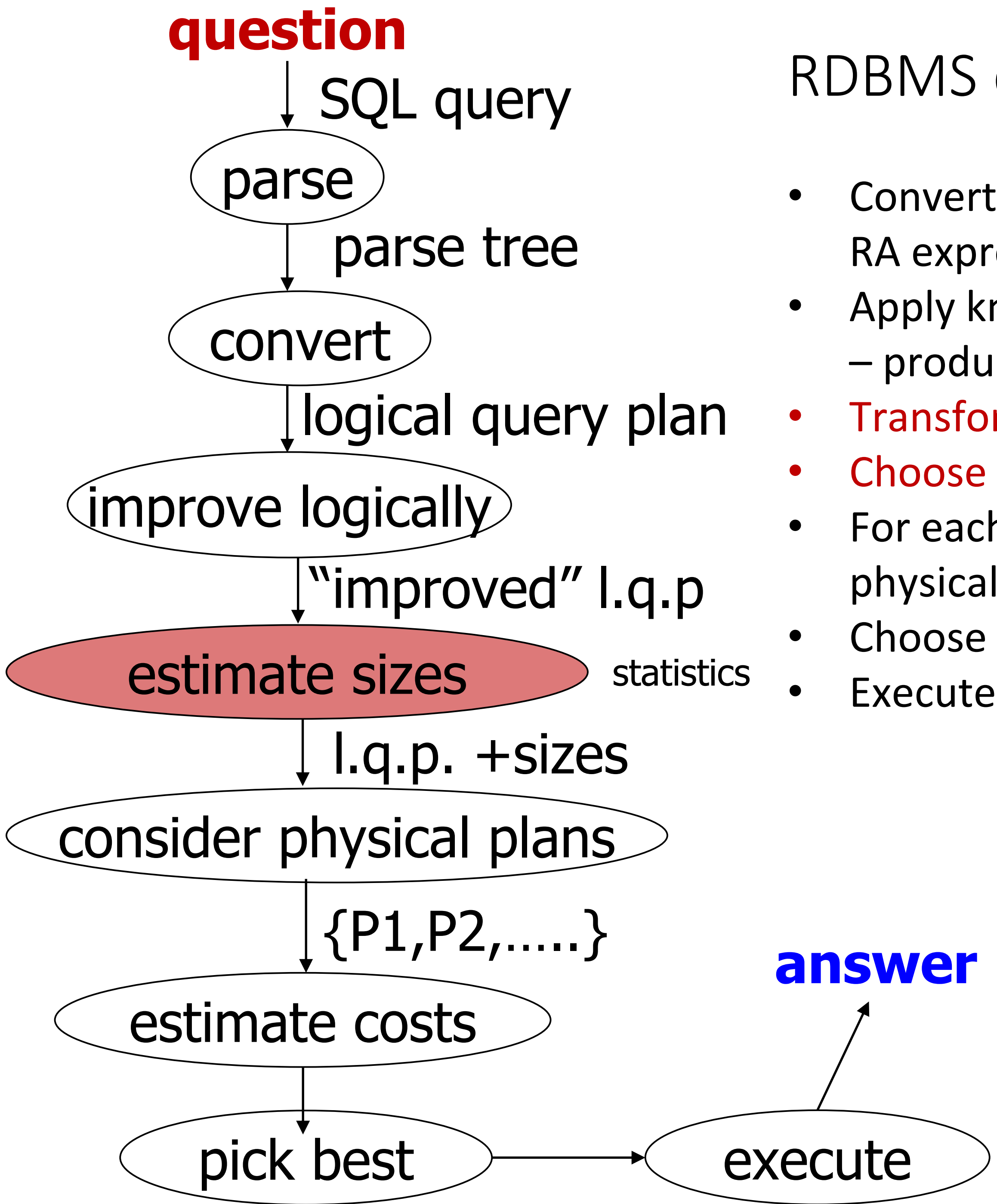
Declarative query (user declares what results are needed)

Translate to relational algebra expression

Find logically equivalent- but more efficient- RA expression

Execute each operator of the optimized plan!

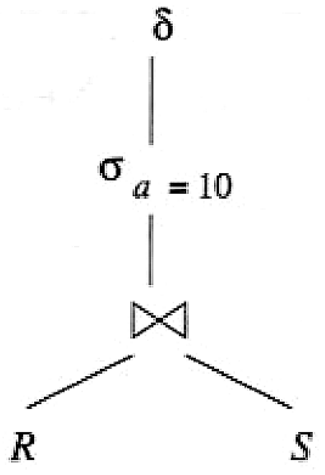
RDBMS query optimizer: steps

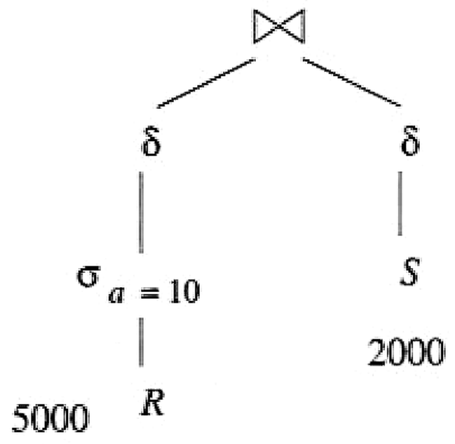


- Convert parsed SQL into corresponding RA expression
- Apply known algebraic transformations – produce improved logical query plan
- Transform based on estimated cost
- Choose one min-cost logical expression
- For each step, consider alternative physical implementations
- Choose physical plan with min I/Os
- Execute

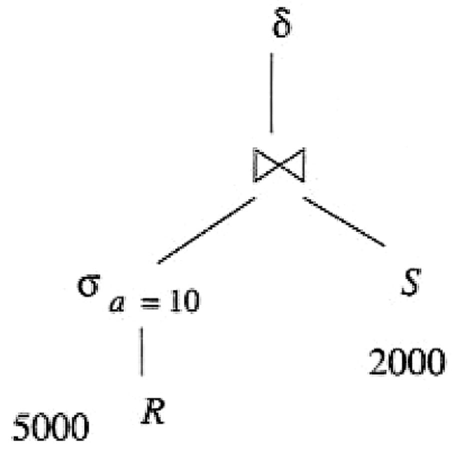
Cost-based transformations of RA trees

$R(a,b)$	$S(b,c)$
$T(R) = 5000$	$T(S) = 2000$
$V(R,a) = 50$	
$V(R,b) = 100$	$V(S,b) = 200$
	$V(S,c) = 100$





(a)



(b)

Initial logical query plan

Two candidates for the best logical query plan

Estimating sizes of relations to be joined

(a)

The (estimated) size of $\sigma_{a=10}(R)$ is

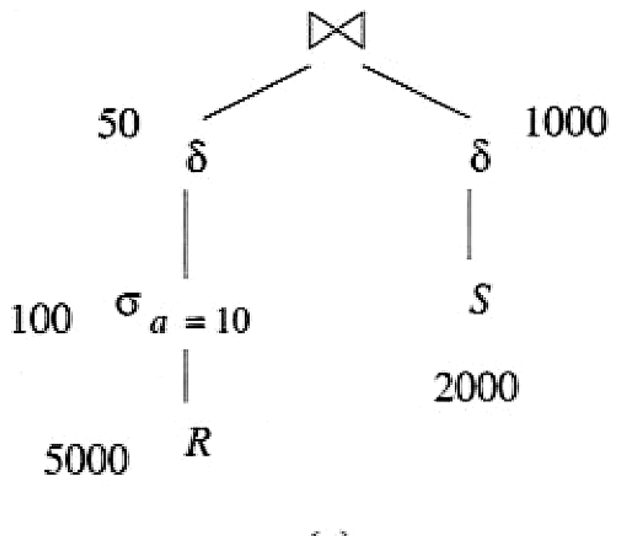
$5000/50 = \mathbf{100}$

The (estimated) size of $\delta(\sigma_{a=10}(R))$ is

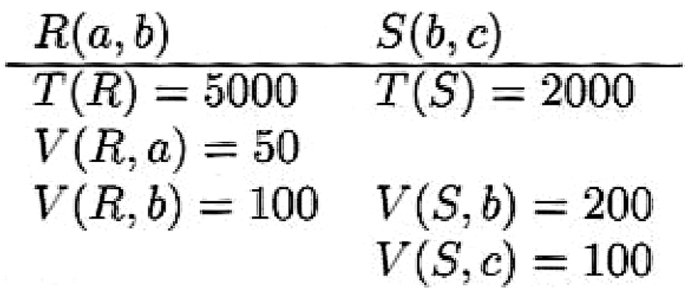
$\min\{1*100, 100/2\} = \mathbf{50}$

The (estimated) size of $\delta(S)$ is

$\min\{200*100, 2000/2\} = \mathbf{1000}$



(a)



$R(a,b)$	$S(b,c)$
$T(R) = 5000$	$T(S) = 2000$
$V(R,a) = 50$	
$V(R,b) = 100$	$V(S,b) = 200$
	$V(S,c) = 100$

$T(\delta(R)) = \min[V(R,a_1)*\ldots*V(R,a_n), 1/2*T(R)]$

Estimating sizes of relations to be joined

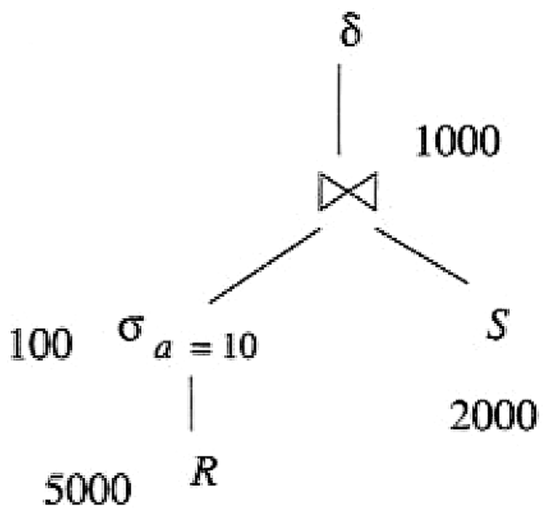
(b)

The (estimated) size of $\sigma_{a=10}(R)$:

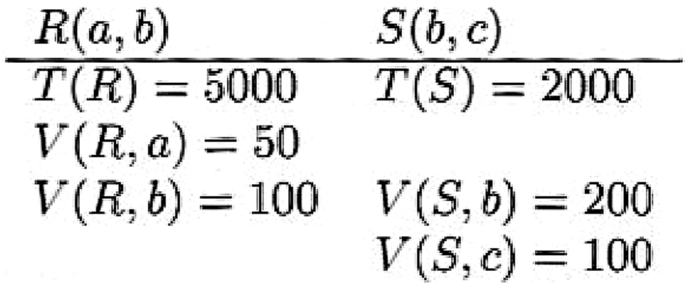
$5000/50 = \mathbf{100}$

The (estimated) size of $\sigma_{a=10}(R) \bowtie S$:

$100*2000/200 = \mathbf{1000}$

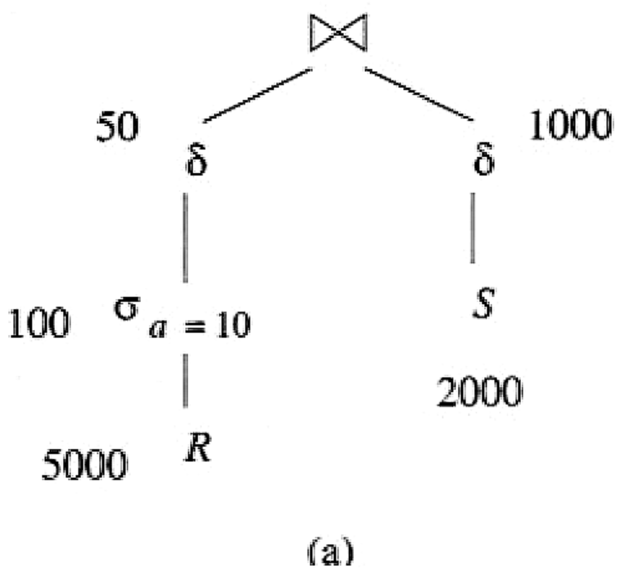


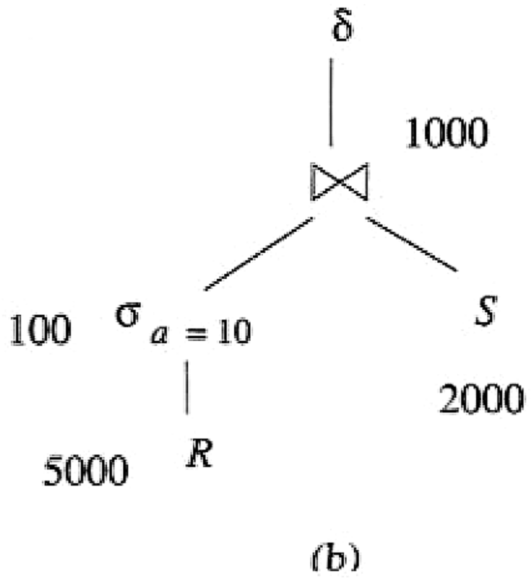
(b)



$R(a,b)$	$S(b,c)$
$T(R) = 5000$	$T(S) = 2000$
$V(R,a) = 50$	
$V(R,b) = 100$	$V(S,b) = 200$
	$V(S,c) = 100$

Comparing two plans





Adding up the costs of plan (a) and (b), (sizes of intermediate relations) we get:

(a) 1150

(b) 1100

So, the conclusion is that plan **(b)** is better,

i.e. deferring the duplicate elimination to the end is a better plan for this query.

Choosing
an order of joins

Choosing a join order

- Critical problem in cost-based optimization: Selecting an order for the (natural) join of three or more relations.
  - Cost is **the total size of all intermediate relations**.

- We have, of course, many possible join orders
- Does it matter which one we pick?
- If so, how do we do this?

Basic intuition

R(a,b), T(R) = 1000, V(R,b) = 20

S(b,c), T(S) = 2000, V(S,b) = 50, V(S,c) = 100

U(c,d) T(U) = 5000, V(U,c) = 500

(R ▷◁ S) ▷◁ U versus **R ▷◁ (S ▷◁ U)**?

- T(R ▷◁ S) = 1000*2000 / 50 = **40,000**
- T(**(**R ▷◁ S**)** ▷◁ U) =

  40000 * 5000 / 500 = **400,000**

- T(S ▷◁ U) = 2000*5000 / 500 = **20,000**
- T(R ▷◁ **(**S ▷◁ U**)**) = 1000*20000 / 50 = **400,000**

Both plans are estimated to produce the same number of tuples (no coincidence here).

However, the first plan is more costly than the second plan because the **size of its intermediate relation** is bigger than the size of the intermediate relation in the second plan (40K versus 20K)

Asymmetricity of Joins

- In some join algorithms, the roles played by the two argument relations are different, and the cost of join depends on which relation plays which role.
  - E.g., the one-pass BNLJ join reads one relation - the smaller - into main memory.
    - Left relation (**the smaller**) is called the *build relation*.
    - Right relation, called the *probe relation*, is read a block at a time and its tuples are matched in main memory with those of the build relation.
  - Other join algorithms that distinguish between their arguments:
    - Nested-Loop join, where we assume the left argument is the relation of the outer loop.
    - Index-join, where we assume the right argument has the index.
    - Hash-join, where we need the *smaller* be less than M^2

Join of two relations

- When we have the join of two relations, we need to optimize the order of the arguments.

- The preferred order is to have the smaller relation on the left

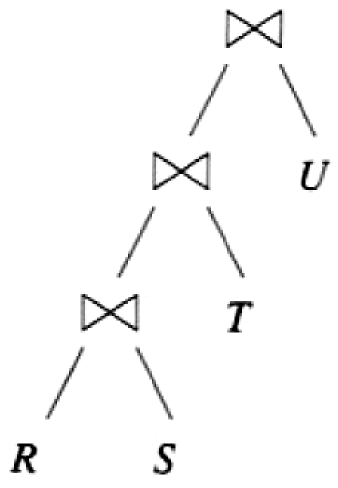
Join Trees for more than two relations

- When the join involves more than two relations, the number of possible join trees grows rapidly.

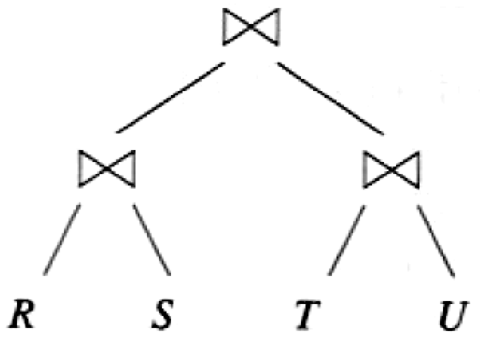
E.g. suppose *R, S, T,* and *U,* being joined. **How many possible join trees?**

- There are 5 possible *shapes* for the tree.
- Each of these trees can have the four relations in any order. So, the total number of trees is 5*4! =5*24 = 120 different trees!!

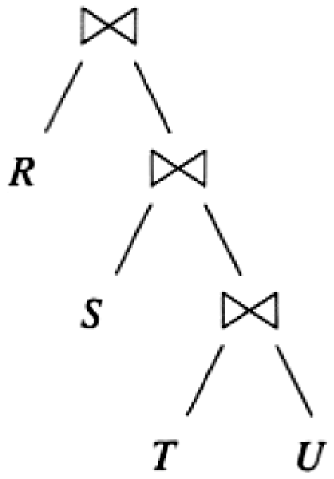
Types of join trees



(a)



(b)



(c)

left-deep tree
All right children are leaves

bushy tree

righ-deep tree
All left children are leaves.

Considering only Left-Deep Join Trees

Good choice because:

1. The number of possible left-deep trees with a given number of leaves is large, but not nearly as large as the number of all trees.

2. Left-deep trees for joins interact well with common join algorithms - nested-loop joins and one-pass join in particular.

Number of plans with Left-Deep Join Trees

- For *n* relations, there is only one left-deep tree shape, to which we may assign the relations in *n*! ways.
- However, the total number of tree shapes $T(n)$ for *n* relations is given by the recurrence:
  - $T(1) = 1$
  - $T(n) = \Sigma_{i=1 \ldots n-1} T(i)T(n - i)$

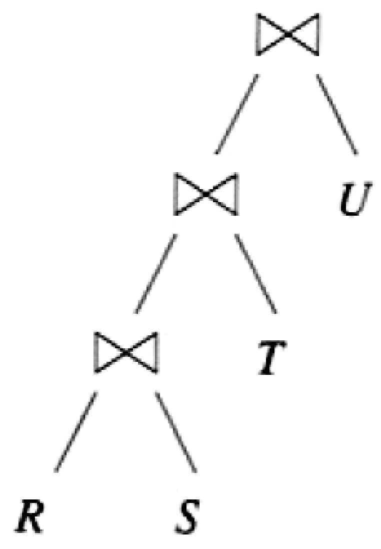
We may pick any number *i* between 1 and *n* - 1 to be the number of leaves in the left subtree of the root, and those leaves may be arranged in any of the $T(i)$ ways that trees with *i* leaves can be arranged. Similarly, the remaining *n-i* leaves in the right subtree can be arranged in any of $T(n-i)$ ways.

And for each shape – n! possible assignments

For example, for 6 relations there are **30,240** different trees, of which only 6!, or **720** are left-deep trees

Left-Deep Trees and One Pass Join Algorithm with pipelining

- A left-deep join tree that is computed by a one-pass algorithm requires main-memory space for **two** of the temporary relations any time.
  - Left argument is the build relation; i.e., held in main memory.
  - To compute R▷◁S, we need to keep R in main memory, and as we compute R▷◁S we need to keep the result in main memory as well – to use as a build relation for the next step.
  - Thus, we need B(R) + B(R▷◁S) buffers.
  - And so on…



(a)

Searching for the best order of joins

- Exhaustively enumerating all possible join orders (even in left-deep trees) is not feasible (n!)

- Two main algorithms:

1. Dynamic programming

   Use the best plan for (*k-1*)-way join to compute the best *k*-way join

2. Greedy heuristic algorithms

   Start with the smallest-cost join, and add one best at a time

Dynamic Programming algorithm

- 70s, seminal work on join order optimization in System R
- The best way to join k relations is drawn from k plans in which the left argument is the least cost plan for joining k-1 relations

```
BestPlan (A,B,C,D,E) = min of (
      BestPlan (A,B,C,D) ⋈ E,
      BestPlan (A,B,C,E) ⋈ D,
      BestPlan (A,B,D,E) ⋈ C,
      BestPlan (A,C,D,E) ⋈ B,
      BestPlan (B,C,D,E) ⋈  A )
```

Dynamic Programming algorithm: Complexity

- Finds optimal join order but must evaluate all 2-way, 3-way, …, n-way joins (n choose k)
- Time $O(n*2^n)$, Space $O(2^n)$
- Exponential complexity, but joins on > 10 relations are rare

Dynamic Programming: additional considerations

- Choosing the best join order or algorithm for each subset of relations may not be the best decision
  - Sort-join produces sorted output that may be useful (i.e., ORDER BY/GROUP BY, sorted on join attribute)
  - Pipelining with nested-loop join
  - Availability of indexes

Dynamic programming with *interesting order*

- Identify interesting sort order. For each order, find the best plan for each subset of relations (one plan per interesting property)
- Low overhead (few interesting orders)

```
BestPlan(A,B,C,D; sort order) = min of (
    BestPlan(A,B,C; sort order) ⋈ D,
    BestPlan(A,B,D; sort order) ⋈ C,
    BestPlan(A,C,D; sort order) ⋈ B,
    BestPlan(B,C,D; sort order) ⋈ A )
```

Partial Order Dynamic Programming

- Optimization of multiple parameters (i.e., response time, network cost, throughput)
  - Each plan assigned a *p*-dimensional cost vector (generalization of interesting orders)

- Complexity: 2^p explosion in search space ($O(2^p*n*2^n)$ time, $O(2^p*2^n)$ space)

Heuristic Approaches

- Dynamic programming may still be too expensive
- Sample heuristics:
    - Join from smallest to largest relation
    - Perform the most selective join operations first
    - Index-joins if available
    - Precede Cartesian product with selection
- Most systems use hybrid of heuristic and cost-based optimization

DP algorithm Example

R(a,b)
V(R,a) = 100
V(R,b) = 200

S(b,c)
V(S,b) = 100
V(S,c) = 500

T(c,d)
V(T,c) = 20
V(T,d) = 50

U(d,a)
V(U,a) = 50
V(U,d) = 1000

Singleton Sets				
	{R}	{S}	{T}	{U}
Size	1000	1000	1000	1000
Cost	0	0	0	0
Best Plan	R	S	T	U

Pairs of relations						
	{R,S}	{R,T}	{R,U}	{S,T}	{S,U}	{T,U}
Size	5000	1M	10,000	2000	1M	1000
Cost	0	0	0	0	0	0
Best Plan	R ▷◁ S	R ▷◁ T	R ▷◁ U	S ▷◁ T	S ▷◁ U	T ▷◁ U

Note that the size is simply the size of the relation, while cost = the size of the intermediate result (0 at this stage).

DP algorithm Example

R(a,b)
V(R,a) = 100
V(R,b) = 200

S(b,c)
V(S,b) = 100
V(S,c) = 500

T(c,d)
V(T,c) = 20
V(T,d) = 50

U(d,a)
V(U,a) = 50
V(U,d) = 1000

Singleton Sets

	{R}	{S}	{T}	{U}
Size	1000	1000	1000	1000
Cost	0	0	0	0
Best Plan	R	S	T	U

Pairs of relations

	{R,S}	{R,T}	{R,U}	{S,T}	{S,U}	{T,U}
Size	5000	1M	10,000	2000	1M	1000
Cost	0	0	0	0	0	0
Best Plan	R ▷◁ S	R ▷◁ T	R ▷◁ U	S ▷◁ T	S ▷◁ U	T ▷◁ U

Also, because the initial sizes of all relations happened to be the same, we can select any order in best plans for pairs – they give the same cost, and the same output size

DP algorithm Example

R(a,b)	S(b,c)	T(c,d)	U(d,a)
V(R,a) = 100	V(S,b) = 100	V(T,c) = 20	V(U,a) = 50
V(R,b) = 200	V(S,c) = 500	V(T,d) = 50	V(U,d) = 1000

Pairs of relations						
	{R,S}	{R,T}	{R,U}	{S,T}	{S,U}	{T,U}
Size	5000	1M	10,000	2000	1M	1000
Cost	0	0	0	0	0	0
Best Plan	R ▹◃ S	R ▹◃ T	R ▹◃ U	S ▹◃ T	S ▹◃ U	T ▹◃ U

Triples of Relations				
	{R,S,T}	{R,S,U}	{R,T,U}	{S,T,U}
Size	10,000	50,000	10,000	2000
Cost	2000	5000	1000	1000
Best Plan	(S ▹◃ T) ▹◃ R	(R ▹◃ S) ▹◃ U	(T ▹◃ U) ▹◃ R	(T ▹◃ U) ▹◃ S

Here, for each triple we select the best among all options:
For example, min for {R, S, T} = min ({R,S} + T, {R,T}+S, {S,T}+R)

Example…cont'd

BestPlan(R,S,T,U) = min of (

  BestPlan (R,S,T) ⋈ U,

  BestPlan (R,S,U) ⋈ T,

  BestPlan (S,T,U) ⋈ R,

  BestPlan (R,T,U) ⋈ S)

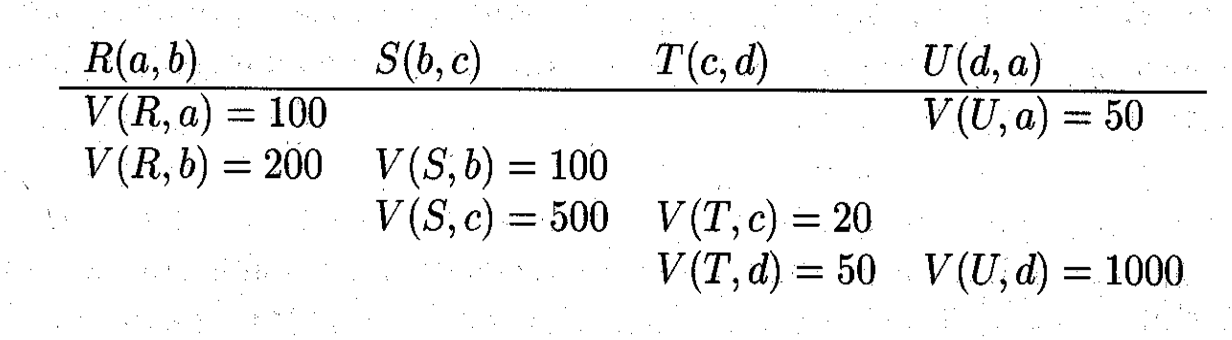
4 relations	Cost
((S ⋈ T) ⋈ R) ⋈ U	12,000
((R ⋈ S) ⋈ U) ⋈ T	55,000
((T ⋈ U) ⋈ R) ⋈ S	11,000
((T ⋈ U) ⋈ S) ⋈ R	**3,000**

Simple Greedy algorithm

- **BASIS**: Start with the pair of relations whose estimated join size is smallest. The join of these relations becomes the current tree.
- **INDUCTION**: Find, among all those relations not yet included in the current tree, the relation that, when joined with the current tree, yields the relation of smallest estimated size.

Note, however, that it is not guaranteed to get the best order.

Example of greedy algorithm



$R(a,b)$	$S(b,c)$	$T(c,d)$	$U(d,a)$
$V(R,a) = 100$			$V(U,a) = 50$
$V(R,b) = 200$	$V(S,b) = 100$		
	$V(S,c) = 500$	$V(T,c) = 20$	
		$V(T,d) = 50$	$V(U,d) = 1000$

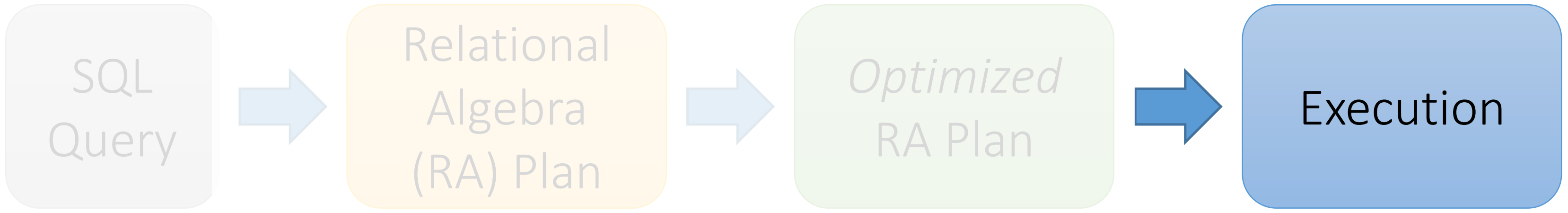
- The basis step is to find the pair of relations that have the smallest join.
- This honor goes to the join *T* ▷◁ *U,* with a cost of 1000. Thus, *T* ▷◁ *U* is the "current tree."

- We now consider whether to join *R* or S into the tree next.
- We compare the sizes of (*T* ▷◁ *U*) ▷◁ *R* and (*T* ▷◁ *U*)▷◁S.
- The latter, with a size of 2000 is better than the former, with a size of 10,000. Thus, we pick as the new current tree (*T* ▷◁ *U*)▷◁S.

- Now there is no choice; we must join *R* at the last step, leaving us with a total cost of **3000**, the sum of the sizes of the two intermediate relations.

RDBMS query evaluation

How does a RDBMS answer your query?

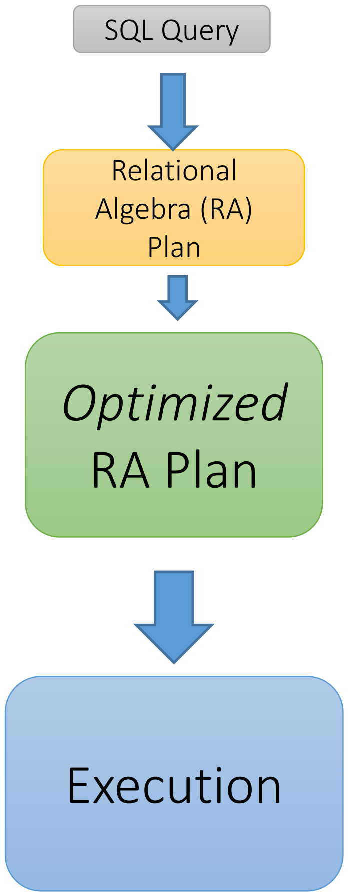


When optimal-cost plan is selected, we still have variations on selecting physical implementation for each operator

Logical vs. Physical Optimization

- **Logical optimization:**
  - Find equivalent plans that are more efficient
  - *Intuition: Minimize # of tuples at each step by changing the order of RA operators*

- **Physical optimization:**
  - Find algorithm with lowest IO cost to execute our plan
  - *Intuition: Calculate based on physical parameters (buffer size, etc.) and estimates of data size (histograms)*



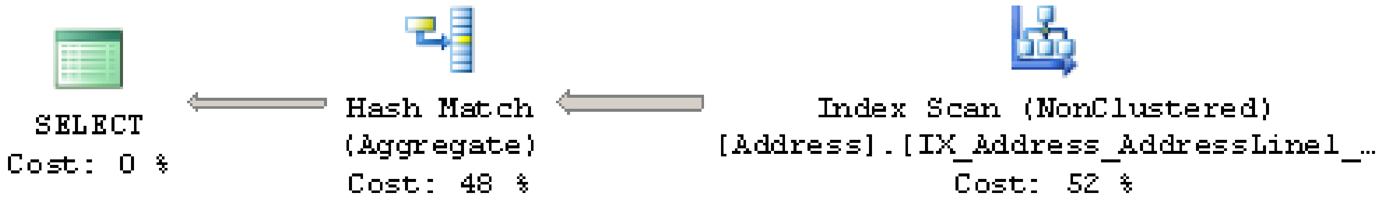
RA Plan Execution

- Natural Join / Join:
  - We saw how to use **memory & IO cost considerations to pick the correct algorithm to execute a join with (BNLJ, SMJ, HJ…)!**
- Selection:
  - We saw how to use **indexes to aid selection**
  - Can always fall back on scan / binary search as well
- Projection:
  - The main operation here is finding *distinct* values of the project tuples; we briefly discussed how to do this with e.g. **hashing** or **sorting**

We already know how to execute all the basic operators!

Exposing the optimizer: MS SQL Server

SELECT DISTINCT(City) FROM Person.Address

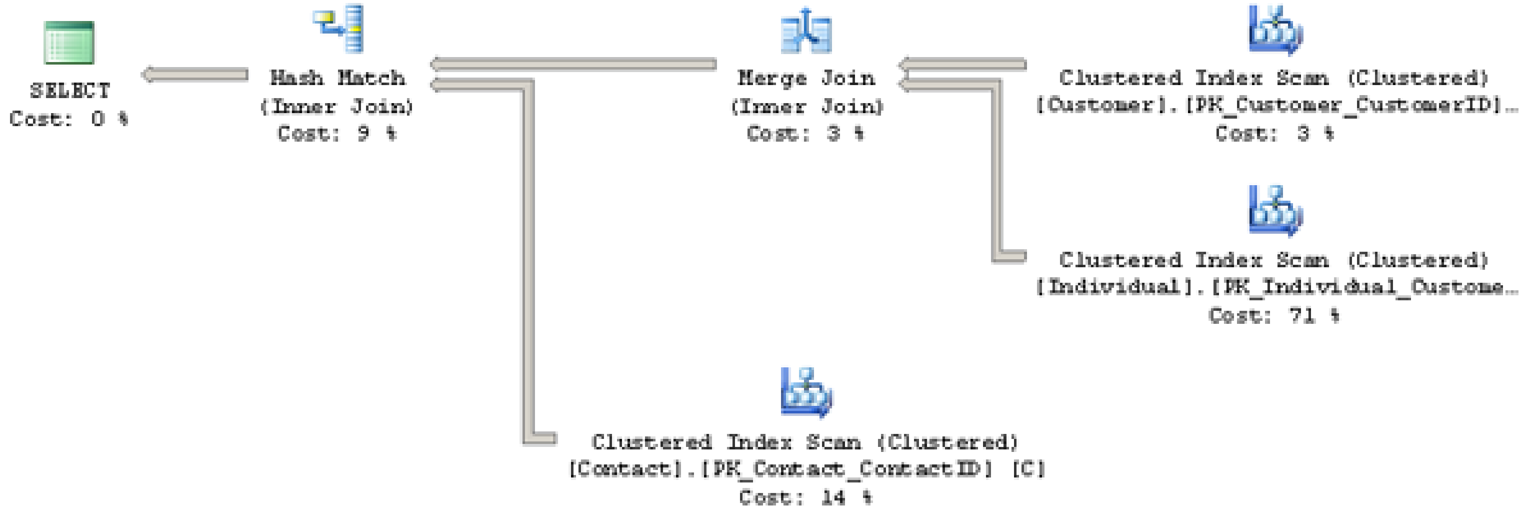


```
SELECT              Hash Match              Index Scan (NonClustered)
Cost: 0 %           (Aggregate)       [Address].[IX_Address_AddressLine1_…
                    Cost: 48 %                   Cost: 52 %
```

Exposing the optimizer: MS SQL Server

```sql
SELECT FirstName, LastName
FROM Person.Contact AS C
   JOIN Sales.Individual AS I
      ON C.ContactID = I.ContactID
   JOIN Sales.Customer AS Cu
      ON I.CustomerID = Cu.CustomerID
WHERE Cu.CustomerType = 'I'
```

explain


SELECT
Cost: 0 %
Hash Match
(Inner Join)
Cost: 9 %
Merge Join
(Inner Join)
Cost: 3 %
Clustered Index Scan (Clustered)
[Customer].[PK_Customer_CustomerID]…
Cost: 3 %
Clustered Index Scan (Clustered)
[Individual].[PK_Individual_Custome…
Cost: 71 %
Clustered Index Scan (Clustered)
[Contact].[PK_Contact_ContactID] [C]
Cost: 14 %

Summary

- Main optimization rules:
    1. Push σ as far down as possible
    2. Do splitting of complex conditions in σ in order to push σ even further
    3. Push π as far down as possible, introduce new early π (but take care for exceptions)
    4. Combine σ with $\times$ to produce Θ-joins or equijoins
    5. Select order of joins

- To choose best plans – need to estimate sizes of intermediate relations
- To choose best physical operators – use estimated outputs of intermediate operations

This simple set of tools allows us to greatly improve the execution time of queries by optimizing RA plans!