**Today:**

- Estimating costs of operations.
- Join algorithms: nested-loop join, sort-join, indexed join, hash join, parallel hash join.

**Soon:**

- Algorithms for other operations: $\cup$, $\delta$, $\gamma$, etc.
- Summarize architecture of query optimizer.
- Begin study of transaction processing, starting with resilience: logging, recovery.

## Estimating the Cost of a Query Plan

- Goal is to count disk I/O's.
- But we first have to estimate sizes of intermediate results.
- $V(A, R)$ = number of distinct values of attribute $A$ in relation $R$.
  - Guides estimate of size of $\sigma_{A=1}$, e.g.
- $N(R)$ = number of tuples in relation $R$.

## Estimating Size of a Selection

- Simple assumption:
  1. All $V(A, R)$ values are equally likely for attribute $A$.
  2. Selection asks for $A = c$ for some one of these values $c$. Thus, $N\big(\sigma_{A=c}(R)\big) = N(R)/V(A, R)$.

  ✦ What about a selection for a value that doesn't appear?

- Selection involving inequality: $\sigma_{A<c}(R)$?

  ✦ Common assumption: 1/3 will meet condition. (**Discussion**: Why not 1/2 as SKS suggests)?

- Complex conditions:

  ✦ `AND` of conditions: use decomposition.

  ✦ Example: $\sigma_{A=a\ AND\ B<b}(R)$ has size estimate $N(R)/3V(A, R)$.

  ✦ **Problem**: What about OR? For that matter, how do you estimate the size of a union?

## Estimating Size of a Projection

- Since duplicates are not eliminated, there is no change in size, strictly speaking.
- But if one follows the $\pi_{A_1 \cdots A_k}(R)$ with a $\delta$, there will be a size reduction.
  - Suggestion: minimum of $N(R)$ and $V(A_1, R) \times \cdots \times V(A_k, R)$.

## Estimating Sizes of Joins

Consider $T = R \bowtie S$. Let $X$, $Y$ be sets of attributes of $R$, $S$, respectively.

1. $X \cap Y = \emptyset$. Join is a product, and $N(T) = N(R)N(S)$.
   - Remember that if there are duplicates in joins or products, duplicate tuples are treated as distinct.
2. $X \cap Y$ is a key for $R$ (for $S$: similar). Each tuple of $S$ can join with at most one tuple of $R$, so $N(T) \leq N(S)$.
   - Generally can't tell how much less.

3. $X \cap Y$ not empty, not a key. One view: consider each tuple of $R$; assume $X \cap Y = A$.

- A tuple of $R$ then joins with $N(S)/V(A,S)$ tuples of $S$; $N(T) = N(R)N(S)/V(A,S)$.
- But symmetrically, we could start with tuples of $S$ and derive the estimate $N(T) = N(R)N(S)/V(A,R)$.
- Take minimum? Why? Consider a case where $A$ has values 1 and 2 in $R$ and 1, 2, 3, 4, and 5 in $S$.
- What about the case where $X \cap Y$ has more than one attribute?

## Example

An important application of size estimates is to evaluate plans that order joins in different ways.

$$\begin{array}{lll} R(A,B) & S(B,C) & T(C,D) \\ N(R)=1000 & N(S)=2000 & N(T)=5000 \\ V(B,R)=20 & V(B,S)=50 & V(C,T)=500 \\ & V(C,S)=100 & \end{array}$$

1. Join $R$ and $S$ first:

$$N(R \bowtie S) = 1000 \times 2000/50 = 40{,}000$$
$$V(C, R \bowtie S) = 100$$
$$N\big((R \bowtie S) \bowtie T\big) = 40{,}000 \times 5000/500 = 400{,}000$$

2. Join $S$ and $T$ first:

$$N(S \bowtie T) = 2000 \times 5000/500 = 20{,}000$$
$$V(B, S \bowtie T) = 50$$
$$N\big(R \bowtie (S \bowtie T)\big) = 20{,}000 \times 1000/50 = 400{,}000$$

3. Product of $R$ and $T$ first:

$$N(R \bowtie T) = 1000 \times 5000 = 5{,}000{,}000$$
$$V(B, R \bowtie T) = 20;\ V(C, R \bowtie T) = 500$$
$$N\big((R \bowtie T) \bowtie S\big) = 5{,}000{,}000 \times 2000/(50 \times 500) = 400{,}000$$

## Issues in Join Ordering

- Note the size estimate for the result is 400,000 tuples, regardless of how we order the join.
  - Coincidence? I don't think so.
  - **Problem**: Do you?
- In this case, the size of the result swamps the size of the intermediate, unless we do the dumb thing of starting with a Cartesian product (case 3).
- Size of the intermediate(s) is one important criteria, since it takes time to create the intermediate.
  - But there are other important issues, such as existence of indexes.

**Example**

Suppose there were an index on $T.C$. Even though $R \bowtie S$ is bigger than $S \bowtie T$, we could pipeline the tuples of $R \bowtie S$ to the second join, and use the $C$-value from each tuple to look up matching tuples from $T$.

- Saves the disk I/O's involved in creating and retrieving $R \bowtie S$.

## Nested-Loop Join

To compute $R \bowtie S$:

```
for each tuple r of R do
    for each tuple s of S do
        if r and s join then
            output the resulting tuple
```

## Improvement to Take Advantage of Disk I/O Model

- Instead of retrieving tuples of $S$ $N(R)$ times, load memory with as many tuples of $R$ as can fit, and match tuples of $S$ against all $R$-tuples in memory.

## Example

- Let $N(R) = 10{,}000$ and $N(S) = 5000$.
- Assume 10 tuples of either $R$ or $S$ fit in one block; i.e., $R$, $S$ occupy 1000 and 500 blocks, respectively.
- Assume there are 101 input buffers in memory available for the join.
  - ❖ Ignores the need for at least one output buffer.
  - ❖ *Important Aside*: 101 buffer blocks is not as unrealistic as it sounds. There may be many queries at the same time, competing for main-memory buffers.
- Assume that both $R$ and $S$ are *clustered*, i.e., their tuples are packed in blocks consisting of only tuples of the same relation.

**Strategy**

1. Load 100 buffers with 1000 tuples of $R$.

2. Read all tuples of $S$, one block at a time, into memory; compare these tuples with tuples of $R$ in memory, and output any matches.

- Repeat steps (1) and (2) 10 times, until all tuples of $R$ have had their turn in memory.

## Analysis of Nested-Loop Join

- Each block of $R$ is read once = 1000 disk I/O's.
- Each block of $S$ is read 10 times = 5000 disk I/O's.
- Ignores writing of result, which could take a widely varying number of blocks, depending on the size of the result.
  - But do we really write them? Perhaps they are the source for another join in which they play the role of $R$, being read into a separate set of buffers until those buffers are filled.

## Problem

We could interchange the roles of $R$ and $S$. Should we?

**Sort Join (Merge-Join in SKS)**

To join $R(A, B) \bowtie S(B, C)$:

1. Sort them by $B$ if they are not already sorted.
2. "Merge" the two sorted lists, thus matching all tuples with common values of $B$.

**Example**

$R = (a, 10),\ (b, 10),\ (c, 20),\ (d, 40)$

$S = (10, x),\ (10, y),\ (30, x),\ (40, z),\ (40, y)$

- General idea: $R = r_1, r_2, \ldots, r_n$, $S = s_1, s_2, \ldots, s_m$.

```
i := 1; j := 1;
while i ≤ n and j ≤ m do
    if r_i and s_j join then
        OUTPUT(i,j)
    else if r_i.B < s_j.B then i := i+1
    else j := j+1
end;
```

- Function OUTPUT pairs $r_i$ with $s_j$ and as many following $S$-tuples as join with $r_i$:

```
OUTPUT(i,j):
    k := j;
    while r_i and s_k join do
        output the join of r_i and s_k;
        k := k+1;
    end;
    i := i+1;
```

## Analysis of Sort Join

- Sorting by 2PMMS takes 4 disk I/O's per block of data = 4000 for $R$, 2000 for $S$.
  - Better check there are enough blocks to do 2PMMS.
- Assuming that merge-joining the two sorted relations does not require more than a few blocks of each in buffers (i.e., not too many tuples share a value of the join attribute) then merging requires another disk I/O per data block = 1500 in our example.
  - Remember: we don't count the last write in any of these join methods.

## Better Implementation of Sort Join

Do only phase 1 of 2PMMS for each relation. In phase 2, generate a few output blocks at a time and pass them to the joining process.

- Saves one write and one read per block of data.
  - In our example, reduces disk I/O's to 4500 (plus the final write).

## Limitations

- For our example $R$ and $S$, we sort $R$ (using 101 buffers) into 10 sorted sublists and $S$ into 5 sublists, using the same buffers.
- In the merge phase, we need 15 buffers, one per sublist, for input.
- That leaves 43 buffers each for the sorted $R$ and $S$.
  - Thus, we can join unless there are more than 430 records from one relation that share a $B$-value.

## Hash Join

- Pick a hash function $h$ that maps $B$-values to buckets.
  - ✧ If "$B$" is really a combination of attributes, then the hash function involves them all.
- Send $R$ and $S$ tuples to separate hash tables, each based on $h$.
- Examine the $i$th buckets of both hash tables to find joining tuples.

## Example

Two hash tables of 50 buckets, used for our example $R$ and $S$.

- As we read $R$, we hash to buckets, which may fill up. If so, we move the current block for that bucket to disk and regard its buffer as a new block, to which the old block is chained.
  - ✧ Total disk I/O's = 1000 reads and a little more than 1000 writes (because some blocks may not be completely full).

- Similarly, hashing $S$ requires about 1000 disk I/O's.
- Average bucket for $R$ has 20–21 blocks; those for $S$ average 10–11.
  - ❖ Thus, unless there is a lot of data skew, we should be able to bring an entire $R$-bucket and an entire $S$ bucket into memory at the same time.
  - ❖ Additional disk I/O's for reading buckets: about 1500.

**Comparison**

For our example:

1. Nested-loop: 5500 (not 6000: interchange $R$ and $S$).
2. Sort-Join, naive: 7500.
3. Sort-Join, combine phase 2 of 2PMMS with join step: 4500
4. Hash-Join: 4500+.

But note:

- Nested-loop is essentially quadratic, the other approaches are linear.
- We might take advantage of one argument being sorted already.
    - Example: $AB \bowtie AC \bowtie AD$.
- In hash-join, we can hash only recordID-joinAttribute pairs, using fewer blocks for the buckets.
    - Downside: we may need to read a lot of blocks for the tuples themselves if a lot of pairs join.

## Parallel Hash Join

If there are many processors, hash join allows all processors to be useful at the same time (even in a "shared nothing" architecture).

- Assume $R$ and $S$ are distributed across the processors.

## Distribution Phase

- If there are $p$ processors, all use a hash function $h$ from the values of the join attribute(s) to $[0..p-1]$.
- Each processor hashes its $R$ and $S$ tuples, sending tuple $t$ to the processor numbered $h(t[B])$, where $B$ represents the join attribute(s).
  - In a message-passing architecture, we should bundle a number of tuples in one message.

## Local Join Phase

- Once distribution is complete, each processor takes a look at what it has received, and hashes it to new, local hash tables.
  - ❖ One table for each of $R$ and $S$, just like uniprocessor hash-join.
  - ❖ But make sure you use a hash function other than $h$, or you will get a nasty surprise.

## Analysis

Once the data is distributed, the elapsed time is similar to hash-join on relations each $1/p$th as large.

- But we have an additional disk read for every block of data, and whatever communication costs there are.

## Using Indexes

What if we want $R(A, B) \bowtie S(B, C)$ and we have an index on $R.B$?

- Keep index buffered in memory (will it fit?).

```
for each S-tuple s do
    consult index for matching
            R-tuples;
    output all matches;
end
```

## Analysis

For our running example:

- 500 disk I/O's to read $S$.
- If each $S$-tuple matches $k$ $R$-tuples, we make $5000k$ disk reads for $R$.
  - Could be best or worst method for this example.

## Other Things to Do With an Index

- If the index is based on a sorted $R$, use it to read $R$ sorted without paying the sorting price in sort-join.
- If $R$ is not clustered, then all the other analyses are bogus (count of disk I/O's to read $R$ is wrong).
    - Gives an advantage to index-join.

## Problem

Suppose there are indexes on both $R.B$ and $S.B$. How could we take advantage?