Introduction to Data Management

Query Cost Estimation

Alyssa Pittman
Based on slides by Jonathan Leang, Shana Hutchinson, Dan Suciu, et al

Paul G. Allen School of Computer Science and Engineering
University of Washington, Seattle
Announcements

▪ HW5 out
  • You know how to write the SQL queries, but it’s a lot of code!
  • Transactions can be hard to debug.
Midterm results

- Scores released today via Gradescope
- Solutions on the website
- Regrade requests open until February 26
  - Please be specific/descriptive when asking for a question to be regraded
Goals for Today

- Move to a short unit on RDBMS optimization
- Learn how an RDMS translates a logical query plan to a physical query plan and executes it
Outline

- Query execution
- Cost estimation ideas and assumptions
- Join algorithm analyses
- Basic cardinality estimation
So you wrote a SQL query...
• SQL only tells the computer *what* you want
• RDBMS needs to find a good way to actually do it
Logical vs Physical Plans

- SQL is translated into RA
- RA (logical plan) does not fully describe execution
- RA with algorithms (physical plan) is needed
Logical vs Physical Plans

- SQL is translated into RA
- RA (logical plan) does not fully describe execution
- RA with algorithms (physical plan) is needed
Disclaimer

- Cost estimation is an active research topic
- Equations and methods discussed in this class form a foundation of concepts, but usually cannot compare to a commercialized solution
RDBMs optimize by selecting the **least cost plan**

- SQL $\rightarrow$ RA
- RA $\rightarrow$ Set of eq. RA
- Set of eq. RA $\rightarrow$ Set of physical plans
- Set of physical plans $\rightarrow$ The least cost plan

...Execute!
RDBMS

SQL
SELECT *
FROM T, R, S
WHERE ...
Plan Enumeration

```
SELECT * 
FROM T, R, S 
WHERE ...
```

RDBMS

Logical Plan

```
T R S
```
Plan Enumeration

SQL

SELECT *
FROM T, R, S
WHERE ...

RDBMS

Logical Plan

Equivalent Logical Plans
Plan Enumeration

SELECT * FROM T, R, S WHERE ...

Physical Plans

...
Plan Enumeration

SQL
SELECT *
FROM T, R, S
WHERE ...

RDBMS

Logical Plan

Equivalent Logical Plans

Least Cost Plan

Physical Plans
Plan Enumeration

SQL

```
SELECT *
FROM T, R, S
WHERE ...
```

RDBMS

Logical Plan

Equivalent Logical Plans

Physical Plans

Least Cost Plan

Execution

```
100101010110
000101111010
100010101000
001010010100
```

```
100101010110
000101111010
100010101000
001010010100
```
Assumptions

For this class we make a lot of assumptions

- **Disk-based storage**
  - HDD not SDD

- **Row-based storage**
  - Tuples are stored contiguously

- **IO cost** (reading from disk) only considered
  - Comprehensive cost estimation involves many factors
    - Network, disk, and CPU cost
    - Cache (main mem., L1 cache, L2 cache, disk cache, …)
  - Reading from disk is usually the biggest component
    - One IO access is ~100000x more expensive than one main memory access

- **Cold cache** (no data preloaded)
Disk Storage

- Mechanical hard drive
- Smallest unit of memory that can be read at once is a **block**
  - Usually 512B to 4kB
- DBMS will attempt to store table files in **contiguous chunks of memory** on disk
- Sequential disk reads are faster than random ones
Disk Storage

### Numbers Everyone Should Know

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache reference</td>
<td>0.5</td>
</tr>
<tr>
<td>Branch mispredict</td>
<td>5</td>
</tr>
<tr>
<td>L2 cache reference</td>
<td>7</td>
</tr>
<tr>
<td>Mutex lock/unlock</td>
<td>100</td>
</tr>
<tr>
<td>Main memory reference</td>
<td>100</td>
</tr>
<tr>
<td>Compress 1K bytes with Zippy</td>
<td>10,000</td>
</tr>
<tr>
<td>Send 2K bytes over 1 Gbps network</td>
<td>20,000</td>
</tr>
<tr>
<td><strong>Read 1 MB sequentially from memory</strong></td>
<td><strong>250,000</strong></td>
</tr>
<tr>
<td>Round trip within same datacenter</td>
<td>500,000</td>
</tr>
<tr>
<td><strong>Disk seek</strong></td>
<td><strong>10,000,000</strong></td>
</tr>
<tr>
<td>Read 1 MB sequentially from network</td>
<td>10,000,000</td>
</tr>
<tr>
<td>Read 1 MB sequentially from disk</td>
<td>30,000,000</td>
</tr>
<tr>
<td>Send packet CA-&gt;Netherlands-&gt;CA</td>
<td>150,000,000</td>
</tr>
</tbody>
</table>

Jeff Dean’s “Numbers Everyone Should Know”
Disk Storage

- Tables are stored as files
  - **Heap file** ▪ Unsorted tuples (this lecture)
  - **Sequential file** ▪ Sorted tuples (next lecture)
    - Attribute(s) sorted on is called a **key** (because that term isn’t overloaded…)

February 12, 2020
RDBMS keeps statistics about our tables

- $B(R) = \# \text{ of blocks}$ in relation $R$
- $T(R) = \# \text{ of tuples}$ in relation $R$
- $V(\text{attr}, R) = \# \text{ of distinct values}$ of attr in $R$

- We only discuss **join algorithms** because they are usually the most expensive part of a query
- We only discuss **nested-loop** and **single-pass** join algorithms because cost equations get complex
Join Algorithm Summary

- Nested-Loop Join
  - Versatile

- Hash Join (single pass)
  - Fast
  - Needs at least one input to be small

- Sort-Merge Join (single pass)
  - Fast
  - Sorts data at the same time!
  - Needs both inputs to be small
Join Algorithm Summary

- **Nested-Loop Join**
  - Versatile

- **Hash Join (single pass)**
  - Fast
  - Needs at least one input to be small

- **Sort-Merge Join (single pass)**
  - Fast
  - Sorts data at the same time!
  - Needs both inputs to be small
Nested Loop Join Algorithm

- Similar execution logic as nested-loop semantics

for each tuple t1 in R:
  for each tuple t2 in S:
    if t1 and t2 can join:
      output (t1,t2)
Nested Loop Join Algorithm

- Similar execution logic as nested-loop semantics

\[
\text{for each tuple } t_1 \text{ in } R: \\
\quad \text{for each tuple } t_2 \text{ in } S: \\
\quad \quad \text{if } t_1 \text{ and } t_2 \text{ can join: } \\
\quad \quad \quad \text{output } (t_1, t_2)
\]

To save time, we’ll read tuples from disk to memory in blocks. For fixed-size tuples, each block will have the same number of tuples.
Nested Loop Join Algorithm

Example equijoin

```
SELECT * 
FROM R, S 
WHERE R.attr = S.attr
```

Block-at-a-time nested loop join:

for each block \( b_R \) in \( R \):
  for each block \( b_S \) in \( S \):
    for each tuple \( t_R \) in \( b_R \):
      for each tuple \( t_S \) in \( b_S \):
        if \( t_R \) and \( t_S \) can join:
          output \( (t_R, t_S) \)
Nested Loop Join Algorithm

Example equijoin

```
SELECT *  
FROM R, S  
WHERE R.attr = S.attr
```

Block-at-a-time nested loop join:

for each block \( b_R \) in \( R \):
  for each block \( b_S \) in \( S \):
    for each tuple \( t_R \) in \( b_R \):
      for each tuple \( t_S \) in \( b_S \):
        if \( t_R \) and \( t_S \) can join:
          output \((t_R, t_S)\)
Nested Loop Join Algorithm

Example equijoin

```
SELECT *  
FROM R, S  
WHERE R.attr = S.attr
```

Block-at-a-time nested loop join:

for each block \(b_R\) in \(R\):
for each block \(b_S\) in \(S\):
  for each tuple \(t_R\) in \(b_R\):
    for each tuple \(t_S\) in \(b_S\):
      if \(t_R\) and \(t_S\) can join:
        output \((t_R, t_S)\)

(blocks are joined in memory)
Nested Loop Join Algorithm

Example equijoin

```
SELECT * 
FROM R, S 
WHERE R.attr = S.attr
```

Assume block size = 2 tuples

A tuple where x is the join attribute value
Example equijoin

```
SELECT * 
FROM R, S 
WHERE R.attr = S.attr
```

A tuple where x is the join attribute value

Disk

```
R: 1 7 3 5
S: 3 3 1
```

Main Memory

```
1 7
```

Assume block size = 2 tuples
Nested Loop Join Algorithm

Example equijoin

```
SELECT * 
FROM R, S
WHERE R.attr = S.attr
```

A tuple where x is the join attribute value

Assume block size = 2 tuples
Nested Loop Join Algorithm

Example equijoin

```
SELECT *  
FROM R, S  
WHERE R.attr = S.attr
```

![Diagram of nested loop join algorithm](image)

- A tuple where x is the join attribute value
- Main memory join (0 cost)
- Assume block size = 2 tuples
Nested Loop Join Algorithm

Example equijoin

```
SELECT * FROM R, S WHERE R.attr = S.attr
```

A tuple where x is the join attribute value

Disk

```
R  1  7  3  5
S  3  3  1
```

Main Memory

```
Main memory join (0 cost)
```

Assume block size = 2 tuples

Example equijoin (block-at-a-time nested loop join)
Nested Loop Join Algorithm

Example equijoin

```
SELECT *  
FROM R, S  
WHERE R.attr = S.attr
```

A tuple where \( x \) is the join attribute value

Assume block size = 2 tuples

Disk

```
R 1 7 | S 3 3 1
    |     |
    | 3 5 |
```

Main Memory

```
3 5
```

Main memory join (0 cost)

```
1 1
```
Nested Loop Join Algorithm

Example equijoin

```sql
SELECT * 
FROM R, S 
WHERE R.attr = S.attr
```

A tuple where x is the join attribute value

![Diagram showing nested loop join algorithm](image)

Main memory join (0 cost)

Assume block size = 2 tuples
Nested Loop Join Algorithm

Example equijoin

```
SELECT * 
FROM R, S 
WHERE R.attr = S.attr
```

A tuple where x is the join attribute value

Assume block size = 2 tuples
Nested Loop Join Algorithm

Example equijoin

```
SELECT * 
FROM R, S
WHERE R.attr = S.attr
```

A tuple where x is the join attribute value

<table>
<thead>
<tr>
<th>R</th>
<th>1</th>
<th>7</th>
<th>3</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Main Memory

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

Assume block size = 2 tuples
Nested Loop Join Algorithm

Block-at-a-time nested loop join

Cost = B(R) + B(R)*B(S)

Reading all of R...

...for each block of R read all of S
Example equijoin

```sql
SELECT * 
FROM R, S 
WHERE R.attr = S.attr
```

Can I do it faster?
Nested Loop Join Algorithm

Example equijoin

```
SELECT * 
FROM R, S 
WHERE R.attr = S.attr
```

Can I do it faster?
Yeah… if you’re willing to use more memory

Algorithms 101:
Time complexity vs space complexity tradeoff
Nested Loop Join Algorithm

Example equijoin

```
SELECT * 
FROM R, S 
WHERE R.attr = S.attr
```

Optimized block-nested-loop join:

```
for each group of N blocks bR in R:
    for each block bS in S:
        for each tuple tR in bR:
            for each tuple tS in bS:
                if tR and tS can join:
                    output (tR,tS)
```
Nested Loop Join Algorithm

Example equijoin

```sql
SELECT * 
FROM R, S 
WHERE R.attr = S.attr
```

N = 2 blocks

```
R 1 7 3 5 2
S 3 3 1
```

Main Memory

Assume block size = 2 tuples
Nested Loop Join Algorithm

Example equijoin

```
SELECT * 
FROM R, S 
WHERE R.attr = S.attr
```

N = 2 blocks

Disk

```
R 1 7 3 5 2
S 3 3 1
```

Main memory

```
1 7 3 5
3 3
```

Assume block size = 2 tuples
Nested Loop Join Algorithm

Example equijoin

```
SELECT * 
FROM R, S 
WHERE R.attr = S.attr
```

N = 2 blocks

Disk

```
R: 1 7 3 5 2
S: 3 3 1
```

Main Memory

```
1 7 3 5
```

Assume block size = 2 tuples

Main memory join (0 cost)

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
```
Nested Loop Join Algorithm

Example equijoin

\[
\text{SELECT} \quad * \\
\text{FROM} \quad R, S \\
\text{WHERE} \quad R.\text{attr} = S.\text{attr}
\]

\[\text{Disk}\]

\[\begin{array}{c}
R & 1 & 7 & 3 & 5 & 2 \\
S & 3 & 3 & 1
\end{array}\]

\[\text{Main Memory}\]

\[\begin{array}{cc}
2 & \text{Main memory join (0 cost)} \\
3 & 3 \\
3 & 3 \\
1 & 1
\end{array}\]

\(N = 2\) blocks

Assume block size = 2 tuples

\[
\text{Example equijoin (block-nested-loop join)}
\]

\[
R.\text{attr}=S.\text{attr}
\]
Nested Loop Join Algorithm

Example equijoin

SELECT * 
FROM R, S 
WHERE R.attr = S.attr

N = 2 blocks

Disk

R | 1 | 7 | 3 | 5 | 2
S | 3 | 3 | 1 |

Main Memory

Join (0 cost)

R | 1 |
S | 3 | 3 |

Assume block size = 2 tuples
Block-nested-loop join
Cost = $B(R) + \frac{B(R)}{N} \cdot B(S)$

Reading all of R...

... for each group of N blocks of R read all of S
Join Algorithm Summary

- Nested-Loop Join
  - Versatile

- **Hash Join** (single pass)
  - Fast
  - Needs at least one input to be small

- Sort-Merge Join (single pass)
  - Fast
  - Sorts data at the same time!
  - Needs both inputs to be small
Hash Tables 101

A naive hash function:

\[ h(x) = x \mod 10 \]

Operations:

\[ \text{find}(103) = \text{??} \]
\[ \text{insert}(488) = \text{??} \]

Separate chaining:
Hash Tables 101

- **insert**\((k, v)\) inserts key \(k\) with value \(v\)
- Many values for one key
  - Duplicates are ok for our bag semantics
- **find**\((k)\) returns a *list* of all values associated with the key

Separate Chaining:

- Insertion at index 3
- Slots 0, 1, 2, 4, 5, 6, 7, 8, 9
- Values: 503, 103, 503, 75, 555, 48
Make a lookup/hash table from the smaller table

- Smaller table has to be smaller than total main memory available ($B(R) < M$ or $B(S) < M$)

- For each block of the larger table, join using the lookup/hash table
Hash Join

Example equijoin

\[
\text{SELECT } * \text{ FROM } R, S \text{ WHERE } R.\text{attr} = S.\text{attr}
\]

\[M = 10 \text{ blocks, hash}(x) = x \mod 5\]
Hash Join

Example equijoin

SELECT *
FROM R, S
WHERE R.attr = S.attr

M = 10 blocks, hash(x) = x mod 5

Disk

<table>
<thead>
<tr>
<th>R</th>
<th>1</th>
<th>7</th>
<th>3</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>3</td>
<td>8</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Main Memory

Assume block size = 2 tuples

<table>
<thead>
<tr>
<th>hash table</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>3, 8</td>
</tr>
</tbody>
</table>
Hash Join

Example equijoin

\[
\text{SELECT } * \\
\text{FROM } R, S \\
\text{WHERE } R.\text{attr} = S.\text{attr}
\]

\[M = 10 \text{ blocks, } \text{hash}(x) = x \mod 5\]

Assume block size = 2 tuples
Hash Join

Example equijoin equijoin

M = 10 blocks, hash(x) = x mod 5

Disk

Main Memory

Assume block size = 2 tuples
Hash Join

Hash join
Cost = $B(R) + B(S)$

Assuming $B(R) < M$
Read all of $R$ into a hash table…

…and join with all of $S$
Hash Join

Cost = \( B(R) + B(S) \)

Isn't this the same as block-nested-loop join where \( B(R) = N \)?

Cost = \( B(R) + B(R)/N \times B(S) \)
Hash Join

The cost of a Hash Join is:

$$\text{Cost} = B(R) + B(S)$$

Isn’t this the same as block-nested-loop join where $$B(R) = N$$?

$$\text{Cost} = B(R) + \frac{B(R)}{N}B(S)$$

Yes! It’s the optimal “one-pass” join!
Join Algorithm Summary

- Nested-Loop Join
  - Versatile

- Hash Join (single pass)
  - Fast
  - Needs at least one input to be small

- **Sort-Merge Join** (single pass)
  - Fast
  - Sorts data at the same time!
  - Needs both inputs to be small
Sort-Merge Join

- Sort both tables into lists in memory
  - Since the sorted lists must contain all tuples, both tables together must fit in memory ($B(R) + B(S) < M$)

- Merge the lists in memory to join
  - Preserves order!
Sort-Merge Join

Example equijoin

\[
\begin{align*}
\text{SELECT } & \ast \\
\text{FROM } & \ R, S \\
\text{WHERE } & \ R.\text{attr} = S.\text{attr}
\end{align*}
\]

\(M = 10\) blocks

Disk

\[
\begin{array}{cccc}
R & 1 & 7 & 3 & 5 \\
S & 3 & 8 & 1 \\
\end{array}
\]

Main Memory

\[
\begin{array}{ccc}
1 & 7 & 3 & 5 \\
3 & 8 & 1 \\
\end{array}
\]
Sort-Merge Join

Example equijoin

```
SELECT *  
FROM R, S  
WHERE R.attr = S.attr
```

M = 10 blocks

Disk

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>7</th>
<th>3</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>3</td>
<td>8</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Main Memory

```
1 3 5 7
```

```
1 3 8
```

(sort merge join)

R.attr=S.attr

Main Memory

```
1 3 5 7
```

```
1 3 8
```

Example equijoin (sort merge join)
Sort-Merge Join

Example equijoin

```sql
SELECT * 
FROM R, S 
WHERE R.attr = S.attr
```

\( M = 10 \) blocks

![Diagram of disk and main memory with data]

We don’t care about exact implementation after disk read since it’s small compared to IO

\( \text{merge join} \)
Cardinality Estimation

- Another building block when estimating the overall cost of a plan
- If we have an RA tree, we need to estimate the output cardinality of the “lower” operations since it’s the input to “upper” operations

How many tuples here??
Cardinality Estimation

- Estimate the number of tuples in the output of each RA operator
  - err, without actually computing the output

- Let's go grocery shop!
  - Safeway(id, name, category, price)
  - QFC(id, name, category, price)

- Let's use store stats to estimate the cardinality of some queries
Cardinality Estimation

- **Safeway** (id, name, category, price)
  - $T = 1000$ \# of tuples
  - $V(name) = 900$ \# of distinct values
  - $V(category) = 10$
  - $V(price) = 200$
  - Range(price) = [1,50) \ range of values

- **QFC** (id, name, category, price)
  - $T = 2000$
  - $V(name) = 1900$
  - $V(category) = 12$
  - $V(price) = 500$

Underline = primary key
Cardinality Estimation: SELECT

Safeway(id, name, category, price)  \( T = 1000 \)

\[
\begin{align*}
\text{SELECT} & \quad \text{name} \\
\text{FROM} & \quad \text{Safeway} \\
\pi_{\text{name}} & \quad \text{Safeway}
\end{align*}
\]

How many tuples do we expect this query to output?
Cardinality Estimation: SELECT

Safeway\((id, \text{name}, \text{category}, \text{price})\) \(T = 1000\)

SELECT name
FROM Safeway

\(\pi_{name}\) 
| Safeway

How many tuples do we expect this query to output?
ANSWER: 1000 (no change)
Cardinality Estimation: DISTINCT

\[ \text{Safeway}(id, \text{name}, \text{category}, \text{price}) \quad T = 1000 \]
\[ \nu(\text{name}) = 900 \]

\[
\text{SELECT DISTINCT name}
\text{FROM Safeway}
\]

How many tuples do we expect this query to output?
Cardinality Estimation: DISTINCT

\[ \text{Safeway}(id, \text{name}, \text{category}, \text{price}) \quad T = 1000 \]

\[ V(\text{name}) = 900 \]

\[ \text{SELECT DISTINCT } \text{name} \]
\[ \text{FROM } \text{Safeway} \]

How many tuples do we expect this query to output?

**ANSWER:** 900 (set to distinct values)
How many tuples do we expect this query to output? 
ASSUME: that ‘45’ exists in the distinct values of id

Answer is 0 otherwise...

ANSWER: 1
Cardinality Estimation: WHERE Value

Safeway\((id, name, category, price)\) \(T = 1000\)

\(V(name) = 900\)

\[
\begin{align*}
\text{SELECT} & \quad * \\
\text{FROM} & \quad \text{Safeway} \\
\text{WHERE} & \quad \text{name} = 'Milk' \\
\end{align*}
\]

\(\sigma_{name="Milk"} \quad \text{Safeway}\)

ASSUME: distinct values uniformly distributed

Without assumptions, estimation is impossible…

ANSWER: \(1000 / 900 \approx 1.11\) tuples
Cardinality Estimation: WHERE Value

\[ \text{Safeway}(\text{id, name, category, price}) \quad T = 1000 \]
\[ V(\text{name}) = 900 \]

\[ \text{SELECT} \quad * \]
\[ \text{FROM} \quad \text{Safeway} \]
\[ \text{WHERE} \quad \text{name} = 'Milk' \]

Assume: distinct values uniformly distributed

Answer: \( \frac{1000}{900} \approx 1.11 \) tuples

The selectivity factor
Cardinality Estimation: WHERE Range

Safeway(id, name, category, price) \( T = 1000 \)
\( V(\text{price}) = 200 \) \( \text{Range}(\text{price}) = [1,50) \)

\[
\text{SELECT } * \\
\text{FROM } \text{Safeway} \\
\text{WHERE } \text{price} < 20
\]

\( \sigma_{\text{price} < 20} \ |
\text{Safeway} \)

ASSUME: distinct values uniformly distributed & continuous

Without assumptions, estimation is impossible...

\[
\text{ANSWER: } 1000 \times \frac{(20 - 1)}{(50 - 1)} \approx 387.8 \text{ tuples}
\]
Cardinality Estimation: WHERE Range

Safeway(id, name, category, price)  \( T = 1000 \)

\( V(\text{price}) = 200 \) \( \text{Range} (\text{price}) = [1, 50) \)

\[ \begin{align*}
\text{SELECT} & \quad * \\
\text{FROM} & \quad \text{Safeway} \\
\text{WHERE} & \quad \text{price} < 20
\end{align*} \]

\[ \text{Select Range: } T(op) * \frac{(Val-Min)}{(Max-Min)} \]

ASSUME: distinct values uniformly distributed & continuous

\[ \text{ANSWER: } 1000 \times \frac{(20 - 1)}{(50 - 1)} \approx 387.8 \] tuples

The selectivity factor

\( \sigma_{\text{price}<20} \)

\( \text{Safeway} \)
Cardinality Estimation: WHERE and

\[ \text{SELECT} \quad * \quad \text{FROM} \quad \text{Safeway} \quad \text{WHERE} \quad \text{price} < 20 \quad \text{AND} \quad \text{name} = 'Milk' \]
Cardinality Estimation: WHERE and

Safeway\((id, name, category, price)\) \(T = 1000\)

\(V(name) = 900\quad V(price) = 200\quad \text{Range}(price) = [1,50)\)

\[
\begin{align*}
\text{SELECT } & \quad \ast \\
\text{FROM } & \quad \text{Safeway} \\
\text{WHERE } & \quad \text{price} < 20 \\
& \quad \text{AND name} = 'Milk' \\
\end{align*}
\]

Hard to say
If conditions disjoint, \(0\) tuples result
If conditions independent, \textbf{multiply} estimates
If conditions fully overlap, take \textbf{minimum} of estimates

\(\sigma_{\text{name} = 'Milk'}\)

\(\sigma_{\text{price} < 20}\)

\(\text{Safeway}\)

\(\text{e.g. no milk costs} < 20\)

\(\text{e.g. milk & price independent}\)

\(\text{e.g. all milk costs} < 20\)

\(\text{ASSUME independent unless you know for sure}\)
Cardinality Estimation: WHERE and

\[
\text{Safeway}(id, \text{name}, \text{category, price}) \quad T = 1000 \\
V(\text{name}) = 900 \quad V(\text{price}) = 200 \quad \text{Range}(\text{price}) = [1,50)
\]

\[
\begin{align*}
&\text{SELECT } * \\
&\text{FROM Safeway} \\
&\text{WHERE } \text{price} < 20 \\
&\quad \text{AND name} = '\text{Milk}'
\end{align*}
\]

\[
\text{ANSWER: assuming independence} \\
\approx 1000 \times \frac{(20 - 1)}{(50 - 1)} \times \frac{1}{900} \approx 0.431 \text{ tuples}
\]

\[
\sigma_{\text{name}='\text{Milk'}}, \sigma_{\text{price}<20} \quad \text{Safeway}
\]
Cardinality Estimation: WHERE and

\[ \text{Safeway}(id, \text{name}, \text{category}, \text{price}) \quad T = 1000 \]

\[ V(\text{name}) = 900 \quad V(\text{price}) = 200 \quad \text{Range}(\text{price}) = [1,50) \]

**AND / INTERSECT**

Assume independence: \( T(\text{op}) \times \text{cond1} \times \text{cond2} \)

unless full overlap: \( T(\text{op}) \times \min\{\text{cond1}, \text{cond2}\} \)

unless disjoint: 0

**WHERE** \( \text{price} < 20 \)

**AND** \( \text{name} = \text{'Milk'} \)

**ANSWER:**

assuming independence

\[ \approx 1000 \times \left( \frac{20 - 1}{50 - 1} \right) \times \frac{1}{900} \approx 0.431 \text{ tuples} \]
Cardinality Estimation: JOIN

- Read 16.4.4 in the book for cardinality estimation of JOINs
- We’ll use this later!
Takeaways

- Nested-Loop Joins
  - Block-at-a-time: $B(R) + B(R) \times B(S)$
  - Nested-block-loop: $B(R) + B(R)/N \times B(S)$

- Hash Join and Sort-Merge Join: $B(R) + B(S)$

- Cardinality estimation helps give us inputs for more complex RA trees.