Review lecture - 2

CS348 Spring 2023
Instructor: Sujaya Maiyya
Sections: 002 & 004 only
Announcements

• **Milestone 2**
  • Due Tuesday, June 11th
  • Late policy: 25% penalty per 24 hrs

• **Assignment 3 - released**
  • Due July 20th
  • Late policy: 15% penalty per 24 hrs

• Expect delays in grading due to a change in TA
  • We will announce on Piazza when grades are ready
Topics covered so far

- Relational model (lecture 2)
- SQL (lectures 3-6)
- Database design (lectures 7-10)
- Storage management & indexing (lectures 11-12)
- Query processing & optimizations (lectures 13-14)

Conceptual/Logical level

Review these topics
Storage hierarchy

- Registers
- Cache
- Memory
- Disk

Secondary storage
- Tapes

Tertiary storage

Non-volatile
A typical hard drive

“Moving parts” are slow
Top view

“Zoning”: more sectors/data on outer tracks

A block is a logical unit of transfer consisting of one sector
Disk access time

Disk access time: time from when a read or write request is issued to when data transfer begins

Sum of:

- **Seek time**: time for disk heads to move to the correct cylinder
- **Rotational delay**: time for the desired block to rotate under the disk head

- **Transfer time**: time to read/write data in the block (= time for disk to rotate over the block)

- Total data access time = seek time + rotational delay + transfer time
Random disk access

→ Successive requests are for blocks that are randomly located on disk

Delay = Seek time + rotational delay + transfer time

• Average seek time
  • Seek the right cylinder for each access
  • “Typical” value: 5 ms

• Average rotational delay
  • Rotate for the right block for each access
  • “Typical” value: 4.2 ms (7200 RPM)
Sequential disk access

→ Successive requests are for successive block numbers, which are on the same track, or on adjacent tracks

Delay = Seek time + rotational delay + transfer time

• Seek time
  • 1 time delay: seek the right cylinder once

• Rotational delay
  • 1 time delay: rotate to the right block once

• Easily an order of magnitude faster than random disk access!
Record layout

Record = row in a table

• Variable-format records
  • Rare in DBMS—table schema dictates the format
  • Relevant for semi-structured data such as XML

• Focus on fixed-format records
  • With fixed-length fields only, or
  • With possible variable-length fields
Fixed-length fields

• All field lengths and offsets are constant
  • Computed from schema, stored in the system catalog

• Example: CREATE TABLE User(uid INT, name CHAR(20), age INT, pop FLOAT);

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>24</th>
<th>28</th>
<th>36</th>
</tr>
</thead>
<tbody>
<tr>
<td>142</td>
<td>Bart (padded with space)</td>
<td>10</td>
<td>0.9</td>
<td></td>
</tr>
</tbody>
</table>

• If block size != 36, one row maybe split across multiple blocks or move to next block & leave the remaining space empty

• What about NULL?
  • Add a bitmap at the beginning of the record
Variable-length records

- Example: `CREATE TABLE User(uid INT, name VARCHAR(20), age INT, pop FLOAT, comment VARCHAR(100));`
- Put all variable-length fields at the end
- Approach 1: use field delimiters (‘\0’ okay?)

```
 0   4   8  16
142 10 0.9 Bart\0 Weird kid!\0
```

- Approach 2: use an offset array

```
 0   4   8  16  18  22  32
142 10 0.9 Bart         Weird kid!
```

- Scheme update is messy if it changes the length of a field
Block layout

How do you organize records in a block?

• **NSM** (N-ary Storage Model)
  • Most commercial DBMS

• **PAX** (Partition Attributes Across)
  • Ailamaki et al., VLDB 2001
NSM

• Store records from the beginning of each block
• Use a directory at the end of each block
  • To locate records and manage free space
  • Necessary for variable-length records

Why store data and directory at two different ends?
So both can grow easily!
Cache behavior of NSM

- **Query:** `SELECT uid FROM User WHERE pop > 0.8;`
- **Assumptions:** no index, and cache line size < record size
- **Lots of cache misses & wasted prefetching**

```
142          Bart           10       0.9
123       Milhouse        10    0.2
456.       Ralph            8.   0.3
857            Lisa             8.    0.7
```

```
Cache
```

```plaintext
142  Bart  10
0.9  123  Milhouse
10  0.2  857  Lisa
8  0.7
456  Ralph  8
0.3
```
PAX

• Most queries only access a few columns
• Cluster values of the same columns in each block
• Better sequential reads for queries that read a single column

Reorganize after every update (for variable-length records only) and delete to keep fields together

(number of records)

(IS NOT NULL bitmap)
# Column vs. row oriented db

**User:**

<table>
<thead>
<tr>
<th></th>
<th>uid</th>
<th>name</th>
<th>pop</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Bart</td>
<td>.6</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>Lisa</td>
<td>.9</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>Abe</td>
<td>.3</td>
<td>65</td>
</tr>
</tbody>
</table>

**Row oriented**

1. Bart, .6, 12
2. Lisa, .9, 10
3. Abe, .3, 65

**Column oriented**

1. Bart, Lisa, Abe
2. .6, .9, .3
3. 12, 10, 65
Indexes
Dense v.s. sparse indexes

- **Dense**: one index entry for each search key value
  - One entry may “point” to multiple records (e.g., two users named Jessica)
- **Sparse**: one index entry for each block
  - Records must be clustered according to the search key on disk
Dense v.s. sparse indexes

- **Dense**: one index entry for each search key value
  - One entry may “point” to multiple records (e.g., two users named Jessica)
- **Sparse**: one index entry for each block
  - Records must be **clustered** according to the search key

**Dense index on name**

<table>
<thead>
<tr>
<th>Name</th>
<th>UID</th>
<th>Size</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bart</td>
<td>123</td>
<td>10</td>
<td>0.2</td>
</tr>
<tr>
<td>Jessica</td>
<td>456</td>
<td>10</td>
<td>0.9</td>
</tr>
<tr>
<td>Lisa</td>
<td>279</td>
<td>10</td>
<td>0.9</td>
</tr>
<tr>
<td>Martin</td>
<td>345</td>
<td>8</td>
<td>2.3</td>
</tr>
<tr>
<td>Ralph</td>
<td>456</td>
<td>8</td>
<td>0.3</td>
</tr>
<tr>
<td>Nelson</td>
<td>512</td>
<td>10</td>
<td>0.4</td>
</tr>
<tr>
<td>Sherri</td>
<td>679</td>
<td>10</td>
<td>0.6</td>
</tr>
<tr>
<td>Terri</td>
<td>697</td>
<td>10</td>
<td>0.6</td>
</tr>
<tr>
<td>Lisa</td>
<td>857</td>
<td>8</td>
<td>0.7</td>
</tr>
<tr>
<td>Windel</td>
<td>912</td>
<td>8</td>
<td>0.5</td>
</tr>
<tr>
<td>Jessica</td>
<td>997</td>
<td>8</td>
<td>0.5</td>
</tr>
</tbody>
</table>

**Sparse index on uid**

- Smaller size
- Easier to update
- Must be clustered

**Can tell directly if a record exists**

**May not fit into memory**
Clustering v.s. non-clustering indexes

- An index on attribute A is a clustering index if tuples in the relation with similar values for A are stored together in the same block.
- Other indices are non-clustering (or secondary) indices.
- Note: A relation may have at most one clustering index, and any number of non-clustering indices.
B$^+$-tree

- A hierarchy of nodes with intervals
- Balanced: good performance guarantee
- Disk-based: one node per block; large fan-out

Max fan-out: 4
Sample B\(^+\)-tree nodes

Max fan-out: 4

Non-leaf

- to keys: 100 ≤ k
- to keys: 100 ≤ k < 120
- to keys: 120 ≤ k < 150
- to keys: 150 ≤ k < 180
- to keys: 180 ≤ k

Leaf

- to keys: 120
- to keys: 130

- to next leaf node in sequence
- to records with these k values;
or, store records directly in leaves
Lookups

- SELECT * FROM R WHERE $k = 179$;
- SELECT * FROM R WHERE $k = 32$;
Range query

- SELECT * FROM R WHERE $k > 32$ AND $k < 179$;

And follow next-leaf pointers until you hit upper bound

Max fan-out: 4
Insertion

- Insert a record with search key value 32

Max fan-out: 4

Look up where the inserted key should go...

And insert it right there
Another insertion example

• Insert a record with search key value 152

Oops, node is already full!
Node splitting

Max fan-out: 4

Oops, that node becomes full!

Need to add to parent node a pointer to the newly created node
More node splitting

• In the worst case, node splitting can “propagate” all the way up to the root of the tree (not illustrated here)
  • Splitting the root introduces a new root of fan-out 2 and causes the tree to grow “up” by one level

Max fan-out: 4

Need to add to parent node a pointer to the newly created node
Index-only plan

• For example:
  • SELECT firstname, pop FROM User WHERE pop > ‘0.8’ AND firstname = ‘Bob’;
  • non-clustering index on (firstname, pop)

• A (non-clustered) index contains all the columns needed to answer the query without having to access the tuples in the base relation.
  • Avoid one disk I/O per tuple
  • The index is much smaller than the base relation
Query processing
Notation

• Relations: $R, S$
• Tuples: $r, s$
• Number of tuples: $|R|, |S|$
• Number of disk blocks: $B(R), B(S)$
• Number of memory blocks available: $M$
• Cost metric
  • Number of I/O’s
  • Memory requirement
Table scan

- Scan table $R$ and process the query
  - Selection over $R$
  - Projection of $R$ without duplicate elimination

- I/O's: $B(R)$
  - Trick for selection:
    - stop early if it is a lookup by key

- Memory requirement: 2 (blocks)
  - 1 for input, 1 for buffer output
  - Increase memory does not improve I/O

- Not counting the cost of writing the result out
  - Same for any algorithm!
Basic nested-loop join

\[ R \bowtie_p S \]

- For each \( r \) in a block \( B_R \) of \( R \):
  - For each \( s \) in a block \( B_S \) of \( S \):
    - Output \( rs \) if \( p \) is true over \( r \) and \( s \)

- \( R \) is called the outer table; \( S \) is called the inner table
- I/O’s: \( B(R) + |R| \cdot B(S) \)

- Memory requirement: \( 3 \)
Improvement: block nested-loop join

\[ R \bowtie_p S \]

- For each block \( B_R \) of \( R \):
  - For each block \( B_S \) of \( S \):
    - For each \( r \) in \( B_R \):
      - For each \( s \) in \( B_S \):
        - Output \( rs \) if \( p \) is true over \( r \) and \( s \)

- I/O's: \( B(R) + B(R) \cdot B(S) \)

  - Blocks of \( R \) are moved into memory only once
  - Blocks of \( S \) are moved into memory \( B(R) \) number of times

- Memory requirement: 3
More improvements

• Stop early if the key of the inner table is being matched

• Make use of available memory
  • Stuff memory with as much of $R$ as possible, stream $S$ by, and join every $S$ tuple with all $R$ tuples in memory
  
  • I/O’s: $B(R) + \left\lfloor \frac{B(R)}{M-2} \right\rfloor \cdot B(S)$
    • Or, roughly: $B(R) \cdot B(S)/M$
  • Memory requirement: $M$ (as much as possible)

• Which table would you pick as the outer? (exercise)
Indexes: Selection using index

• Equality predicate: $\sigma_{A=v}(R)$
  • Use an ISAM, B⁺-tree, or hash index on R(A)

• Range predicate: $\sigma_{A>v}(R)$
  • Use an ordered index (e.g., ISAM or B⁺-tree) on R(A)
  • Hash index is not applicable

• Indexes other than those on R(A) may be useful
  • Example: B⁺-tree index on R(A, B)
  • How about B⁺-tree index on R(B, A)?
Index nested-loop join

\[ R \bowtie_{R.A=S.B} S \]

- Idea: use a value of \( R.A \) to probe the index on \( S(B) \)
- For each block of \( R \), and for each \( r \) in the block:
  - Use the index on \( S(B) \) to retrieve \( s \) with \( s.B = r.A \)
  - Output \( rs \)

- I/O’s: \( B(R) + |R| \cdot \text{(index lookup)} + \text{I/O for record fetch} \)
  - Typically, the cost of an index lookup is 2-4 I/O’s (depending on the index tree height if B+ tree)
  - Beats other join methods if \( |R| \) is not too big
  - Better pick \( R \) to be the smaller relation

- Memory requirement: \( 3 \) (extra memory can be used to cache index, e.g. root of B+ tree)
External merge sort

Recall in-memory merge sort: Sort progressively larger runs, 2, 4, 8, ..., |R|, by merging consecutive “runs”

Problem: sort \( R \), but \( R \) does not fit in memory

1. **Phase 0**: read \( M \) blocks of \( R \) at a time, sort them, and write out a level-0 run
2. **Phase 1**: merge \( (M - 1) \) level-0 runs at a time, and write out a level-1 run
3. **Phase 2**: merge \( (M - 1) \) level-1 runs at a time, and write out a level-2 run

... 

• Final phase produces one sorted run
Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0

Arrows indicate the blocks in memory
Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0

Arrows indicate the blocks in memory
Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0

Arrows indicate the blocks in memory
Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0

Arrows indicate the blocks in memory
Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0

Arrows indicate the blocks in memory
Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0

Arrows indicate the blocks in memory
Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0
- Phase 1

Disk

Arrows indicate the blocks in memory

R: 1 7 4 5 2 8 9 6 3

1 4 7 2 5 8 3 6 9
Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0
- Phase 1

Arrows indicate the blocks in memory
Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0
- Phase 1

Arrows indicate the blocks in memory
Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0
- Phase 1
Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0
- Phase 1
Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0
- Phase 1

Arrows indicate the blocks in memory
Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0
- Phase 1

Arrows indicate the blocks in memory
Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0
- Phase 1
Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0
- Phase 1

Arrows indicate the blocks in memory
Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0
- Phase 1

Arrows indicate the blocks in memory
Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0
- Phase 1
- Phase 2 (final)
Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0
- Phase 1
- Phase 2 (final)
Sort-merge join

$R \bowtie_{R.A=S.B} S$

• Sort $R$ and $S$ by their join attributes; then merge
  • $r, s =$ the first tuples in sorted $R$ and $S$
  • Repeat until one of $R$ and $S$ is exhausted:
    If $r.A > s.B$
    then $s =$ next tuple in $S$
    else if $r.A < s.B$
    then $r =$ next tuple in $R$
    else output all matching tuples, and
      $r, s =$ next in $R$ and $S$

• I/O’s: sorting + $O(B(R) + B(S))$
  • In most cases (e.g., join of key and foreign key)
  • Worst case is $B(R) \cdot B(S)$: everything joins
Query optimization
A query’s trip through the DBMS

SQL query

Parser

Parse tree

Validator

Logical plan

Optimizer

Physical plan

Executor

Result

SELECT name, uid
FROM Member, Group
WHERE Member.gid = Group.gid;

\[
\pi_{\text{name, uid}} \\
\sigma_{\text{Member.gid} = \text{Group.gid}}
\]
Logical plan

- Nodes are **logical** operators (often relational algebra operators)
- There are many equivalent logical plans

\[
\pi_{\text{Group.name}} \\
\sigma_{\text{User.name}="Bart" \land \text{User.uid} = \text{Member.uid} \land \text{Member.gid} = \text{Group.gid}}
\times
\pi_{\text{Group.name}} \\
\Join_{\text{Member.gid} = \text{Group.gid}}
\times
\sigma_{\text{User.name}="Bart"}
\times
\pi_{\text{Group.name}} \\
\sigma_{\text{User.uid} = \text{Member.uid}}
\times
\pi_{\text{Group.name}} \\
\Join_{\text{Member.gid} = \text{Group.gid}}
\times
\sigma_{\text{name} = "Bart"}
\times
\pi_{\text{Group.name}} \\
\sigma_{\text{User.uid} = \text{Member.uid}}
\times
\pi_{\text{Group.name}} \\
\sigma_{\text{name} = "Bart"}
\times
\pi_{\text{Group.name}} \\
\sigma_{\text{User.uid} = \text{Member.uid}}
\times
\pi_{\text{Group.name}} \\
\sigma_{\text{name} = "Bart"}
\times
\pi_{\text{Group.name}} \\
\sigma_{\text{User.uid} = \text{Member.uid}}
\times
\pi_{\text{Group.name}} \\
\sigma_{\text{name} = "Bart"}
\times
\pi_{\text{Group.name}} \\
\sigma_{\text{User.uid} = \text{Member.uid}}
\times
\pi_{\text{Group.name}} \\
\sigma_{\text{name} = "Bart"}
\times
\pi_{\text{Group.name}} \\
\sigma_{\text{User.uid} = \text{Member.uid}}
\times
\pi_{\text{Group.name}} \\
\sigma_{\text{name} = "Bart"}
\times
\pi_{\text{Group.name}} \\
\sigma_{\text{User.uid} = \text{Member.uid}}
\times
\pi_{\text{Group.name}} \\
\sigma_{\text{name} = "Bart"}
\times
\pi_{\text{Group.name}} \\
\sigma_{\text{User.uid} = \text{Member.uid}}
\times
\pi_{\text{Group.name}} \\
\sigma_{\text{name} = "Bart"}
\times
\pi_{\text{Group.name}} \\
\sigma_{\text{User.uid} = \text{Member.uid}}
\times
\pi_{\text{Group.name}} \\
\sigma_{\text{name} = "Bart"}
\times
\pi_{\text{Group.name}} \\
\sigma_{\text{User.uid} = \text{Member.uid}}
\times
\pi_{\text{Group.name}} \\
\sigma_{\text{name} = "Bart"}
\times
\pi_{\text{Group.name}} \\
\sigma_{\text{User.uid} = \text{Member.uid}}
\times
\pi_{\text{Group.name}} \\
\sigma_{\text{name} = "Bart"}
\times
\pi_{\text{Group.name}} \\
\sigma_{\text{User.uid} = \text{Member.uid}}
\times
\pi_{\text{Group.name}} \\
\sigma_{\text{name} = "Bart"}
\times
\pi_{\text{Group.name}} \\
\sigma_{\text{User.uid} = \text{Member.uid}}
\times
\pi_{\text{Group.name}} \\
\sigma_{\text{name} = "Bart"}
\times
\pi_{\text{Group.name}} \\
\sigma_{\text{User.uid} = \text{Member.uid}}
\times
\pi_{\text{Group.name}} \\
\sigma_{\text{name} = "Bart"}
\times
\pi_{\text{Group.name}} \\
\sigma_{\text{User.uid} = \text{Member.uid}}
\times
\pi_{\text{Group.name}} \\
\sigma_{\text{name} = "Bart"}
\times
\pi_{\text{Group.name}} \\
\sigma_{\text{User.uid} = \text{Member.uid}}
\times
\pi_{\text{Group.name}} \\
\sigma_{\text{name} = "Bart"}
\times
\pi_{\text{Group.name}} \\
\sigma_{\text{User.uid} = \text{Member.uid}}
\times
\pi_{\text{Group.name}} \\
\sigma_{\text{name} = "Bart"}
\times
\pi_{\text{Group.name}} \\
\sigma_{\text{User.uid} = \text{Member.uid}}
\times
\pi_{\text{Group.name}} \\
\sigma_{\text{name} = "Bart"}
\times
\pi_{\text{Group.name}} \\
\sigma_{\text{User.uid} = \text{Member.uid}}
\times
\pi_{\text{Group.name}} \\
\sigma_{\text{name} = "Bart"}
\times
\pi_{\text{Group.name}} \\
\sigma_{\text{User.uid} = \text{Member.uid}}
\times
\pi_{\text{Group.name}} \\
\sigma_{\text{name} = "Bart"}
\times
\pi_{\text{Group.name}} \\
\sigma_{\text{User.uid} = \text{Member.uid}}
Physical (execution) plan

• A complex query may involve multiple tables and various query processing algorithms
  • E.g., table scan, basic & block nested-loop join, index nested-loop join, sort-merge join, ... (Lecture 13)

• A **physical plan** for a query tells the DBMS query processor how to execute the query
  • A tree of **physical plan operators**
  • Each operator implements a query processing algorithm
  • Each operator accepts a number of input tables/streams and produces a single output table/stream
Examples of physical plans

SELECT Group.name
FROM User, Member, Group
WHERE User.name = 'Bart'
AND User.uid = Member.uid AND Member.gid = Group.gid;

- Many physical plans for a single query
  - Equivalent results, but different costs and assumptions!
  ✨ DBMS query optimizer picks the “best” possible physical plan
Cost estimation

Physical plan example:

- We have: cost estimation for each operator
  - Example: INDEX-NESTED-LOOP-JOIN(uid) takes \( O(B(R) + |R| \cdot (\text{index lookup} + \text{record fetch})) \)

- We need: size of intermediate results
Cardinality estimation

Cardinality estimation for:

- Equality predicates
- Range predicates
- Joins
- Textbook has more operators
Selections with equality predicates

- $Q: \sigma_{A=v} R$
- DBMSs typically store the following in the catalog
  - Size of $R$: $|R|$
  - Number of distinct $A$ values in $R$: $|\pi_A R|$
- Assumptions
  - Values of $A$ are uniformly distributed in $R$
- $|Q| \approx \frac{|R|}{|\pi_A R|}$
  - Selectivity factor of $(A = v)$ is $\frac{1}{|\pi_A R|}$
Example

Physical plan example:

- \(|User|=1000, \pi_{name}(User)| = 50 \Rightarrow \sigma_{name="Bart"}(User)| = ?\)
- Assumptions:
  - Values of name are uniformly distributed in User
  - \(|\sigma_{name="Bart"}(User)| = \frac{1000}{50} = 20\)
Range predicates

• $Q: \sigma_{A > v} R$
• Not enough information!
  • Just pick, say, $|Q| \approx |R| \cdot \frac{1}{3}$

• With more information
  • Largest R.A value: high($R.A$)
  • Smallest R.A value: low($R.A$)
  • $|Q| \approx |R| \cdot \frac{\text{high}(R.A) - v}{\text{high}(R.A) - \text{low}(R.A)}$
Two-way equi-join

- $Q$: $R(A, B) \bowtie S(A, C)$

- Assumption: containment of value sets
  - Every tuple in the “smaller” relation (one with fewer distinct values for the join attribute) joins with some tuple in the other relation
    - That is, if $|\pi_A R| \leq |\pi_A S|$ then $\pi_A R \subseteq \pi_A S$
  - Certainly not true in general
  - But holds in the common case of foreign key joins

- $|Q| \approx \frac{|R| \cdot |S|}{\max(|\pi_A R|, |\pi_A S|)}$
  - Selectivity factor of $R.A = S.A$ is $\frac{1}{\max(|\pi_A R|, |\pi_A S|)}$
Example

• Database:
  • User(uid, name, age, pop), Member(gid, uid, date), Group(gid, gname)
  • |User|=1000 rows, |Group|=100 rows, |Member|=50000 rows
  • $|\pi_{name}(User)| = 50$
  • $|\pi_{uid}(Member)| = 500$

• Estimate size $|User \bowtie Member| = ?$
  • $|\pi_{uid}(User)| = 1000$
  • $|\pi_{uid}(Member)| = 500$
  • $1000*50000/\max(500,1000)=50000$
Search space is huge

• Characterized by “equivalent” logical query plans

```
SELECT Group.name FROM User, Member, Group WHERE
User.name = 'Bart'
AND User.uid = Member.uid AND Member.gid = Group.gid;
```

An equivalent plan:

Do we need to exam all the logical plans?
No. We can apply heuristic transformation rules to find a cheaper logical plan
Transformation rules (a sample)

• Convert \( \sigma_p \times \) to/from \( \bowtie_p \): \( \sigma_p (R \times S) = R \bowtie_p S \)
  - Example: \( \sigma_{User.uid=Member.uid} (User \times Member) = User \bowtie Member \)

• Merge/split \( \sigma \)'s: \( \sigma_{p_1} (\sigma_{p_2} R) = \sigma_{p_1 \land p_2} R \)
  - Example: \( \sigma_{age>20} (\sigma_{pop=0.8 User}) = \sigma_{age>20 \land pop=0.8 User} \)

• Merge/split \( \pi \)'s: \( \pi_{L_1} (\pi_{L_2} R) = \pi_{L_1} R \), if \( L_1 \subseteq L_2 \)
  - Example: \( \pi_{age} (\pi_{age, pop} User) = \pi_{age} User \)
Transformation rules (a sample)

• Push down/pull up $\sigma$:

$$\sigma_{p \land p_r \land p_s}(R \bowtie_{p'} S) = (\sigma_{p_r} R) \bowtie_{p \land p'} (\sigma_{p_s} S),$$

where

- $p_r$ is a predicate involving only $R$ columns
- $p_s$ is a predicate involving only $S$ columns
- $p$ and $p'$ are predicates involving both $R$ and $S$ columns
- Example:

$$\sigma_{U1.name=U2.name \land U1.pop>0.8 \land U2.pop>0.8}(\rho_{User} \bowtie_{U1.uid \neq U2.uid} \rho_{User})$$

$$= \sigma_{pop>0.8}(\rho_{User}) \bowtie_{U1.uid \neq U2.uid \land U1.name=U2.name}(\sigma_{pop>0.8}(\rho_{User}))$$
Transformation rules (a sample)

• Push down $\pi$: $\pi_L(\sigma_p R) = \pi_L \left( \sigma_p \left( \pi_{L,L'} R \right) \right)$, where
  • $L'$ is the set of columns referenced by $p$ that are not in $L$
  • Example:
    $\pi_{\text{age}}(\sigma_{\text{pop}>0.8 User}) = \pi_{\text{age}}(\sigma_{\text{pop}>0.8}(\pi_{\text{age, pop User}}))$

• Many more (seemingly trivial) equivalences...
  • Can be systematically used to transform a plan to new ones
Relational query rewrite example

\[ \pi_{\text{Group.name}} \sigma_{\text{User.name}=“Bart” \land \text{User.uid} = \text{Member.uid} \land \text{Member.gid} = \text{Group.gid}} \times \text{Group} \times \text{User} \]

Push down \( \sigma \)

Convert \( \sigma_p \times \) to \( \bowtie_p \)
Heuristics-based query optimization

• Start with a logical plan

• Push selections/projections down as much as possible
  • Why? Reduce the size of intermediate results

• Join smaller relations first, and avoid cross product
  • Why? Joins are more optimized and have alternate implementations

• Convert the transformed logical plan to a physical plan (by choosing appropriate physical operators)