DATA 514

Lecture 4:
Query Execution and Indexes
Announcements

• WQ4 is out – due next Sunday
• HW3- due Feb 3

• Midterm, next Tuesday(Feb. 6)
• Check out past exams in our website
  – Note that the material covered in past exams is not necessarily the same as that covered in our exams
Query Evaluation Steps

1. Parse & Check Query
   - Translate query string into internal representation
   - Check syntax, access control, table names, etc.

2. Decide how best to answer query: query optimization
   - Logical plan → physical plan

3. Query Execution

4. Return Results

DATA514 - Winter 2018
Product(pid, name, price)
Purchase(pid, cid, store)
Customer(cid, name, city)

From SQL to RA

```
SELECT DISTINCT x.name, z.name
FROM Product x, Purchase y, Customer z
WHERE x.pid = y.pid and y.cid = z.cid and
    x.price > 100 and
    z.city = 'Seattle'
```
Product(pid, name, price)
Purchase(pid, cid, store)
Customer(cid, name, city)

**From SQL to RA**

```
SELECT DISTINCT x.name, z.name
FROM Product x, Purchase y, Customer z
WHERE x.pid = y.pid and y.cid = z.cid and
    x.price > 100 and
    z.city = 'Seattle'
```

Can you think of a “better” plan?
From SQL to RA

```
SELECT DISTINCT x.name, z.name
FROM Product x, Purchase y, Customer z
WHERE x.pid = y.pid and y.cid = y.cid and
  x.price > 100 and
  z.city = 'Seattle'
```

Can you think of a “better” plan?

Push selections down the query plan!

Product(pid, name, price)
Purchase(pid, cid, store)
Customer(cid, name, city)
From SQL to RA

SELICT DISTINCT x.name, z.name
FROM Product x, Purchase y, Customer z
WHERE x.pid = y.pid and y.cid = y.cid and
  x.price > 100 and
  z.city = 'Seattle'

Can you think of a “better” plan?

Push selections down the query plan!

Query optimization: find an equivalent optimal plan
From Logical Plans to Physical Plans
Physical Operators

Each of the logical operators may have one or more implementations = physical operators

Will discuss several basic physical operators, with a focus on join
Main Memory Algorithms

Logical operator:

\[
\text{Product}(\text{pid, name, price}) \bowtie_{\text{pid} = \text{pid}} \text{Purchase}(\text{pid, cid, store})
\]

Propose three physical operators for the join, assuming the tables are in main memory:

1.
2.
3.
Main Memory Algorithms

Logical operator:

\[
\text{Product}(\text{pid}, \text{name}, \text{price}) \bowtie_{\text{pid} = \text{pid}} \text{Purchase}(\text{pid}, \text{cid}, \text{store})
\]

Propose three physical operators for the join, assuming the tables are in main memory:

1. Nested Loop Join  \( O( ?? ) \)
2. Merge join  \( O( ?? ) \)
3. Hash join  \( O( ?? ) \)
Main Memory Algorithms

Logical operator:

\[
\text{Product}(\text{pid}, \text{name}, \text{price}) \bowtie_{\text{pid} = \text{pid}} \text{Purchase}(\text{pid}, \text{cid}, \text{store})
\]

Propose three physical operators for the join, assuming the tables are in main memory:

1. Nested Loop Join \( O(n^2) \)
2. Merge join \( O(n \log n) \)
3. Hash join \( O(n) \ldots O(n^2) \)
BRIEF Review of Hash Tables

Separate chaining:

A (naïve) hash function:

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

Operations:

- find(103) = ??
- insert(488) = ??

Duplicates OK

WHY??
BRIEF Review of Hash Tables

• insert(k, v) = inserts a key k with value v

• Many values for one key
  – Hence, duplicate k’s are OK

• find(k) = returns the list of all values v associated to the key k
Query Evaluation Steps Review

SQL query

Parse & Rewrite Query

Select Logical Plan

Select Physical Plan

Query Execution

Disk

Logical plan

Physical plan

Query optimization
Relational Algebra

SELECT sname
FROM Supplier x, Supply y
WHERE x.sid = y.sid
    and y.pno = 2
    and x.scity = 'Seattle'
    and x.sstate = 'WA'

Give a relational algebra expression for this query
Relational Algebra

\[ \Pi_{\text{sname}} (\sigma_{\text{scity} = \text{'Seattle'} \land \text{sstate} = \text{'WA'} \land \text{pno} = 2} (\text{Supplier} \bowtie_{\text{sid} = \text{sid}} \text{Supply})) \]

\( \text{Supplier}(\text{sid}, \text{sname}, \text{scity}, \text{sstate}) \)
\( \text{Supply}(\text{sid}, \text{pno}, \text{quantity}) \)
Relational Algebra

SELECT sname
FROM Supplier x, Supply y
WHERE x.sid = y.sid
  and y.pno = 2
  and x.scity = 'Seattle'
  and x.sstate = 'WA'

Relational algebra expression is also called the "logical query plan"
A physical query plan is a logical query plan annotated with physical implementation details.

SELECT sname
FROM Supplier x, Supply y
WHERE x.sid = y.sid
    and y.pno = 2
    and x.scity = 'Seattle'
    and x.sstate = 'WA'

Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

(On the fly)  \( \prod_{sname} \)
(On the fly)  \( \sigma_{scity='Seattle' \land sstate='WA' \land pno=2} \)
(Nested loop)  \( \bowtie \)  \( \text{sid} = \text{sid} \)
Physical Query Plan 2

(On the fly) \( \Pi_{\text{sname}} \)

(On the fly) \( \sigma_{\text{scity}=\text{Seattle} \land \text{sstate}=\text{WA} \land \text{pno}=2} \)

(Hash join) \( \text{sid} = \text{sid} \)

Same logical query plan
Different physical plan

SELECT \text{sname} 
FROM Supplier \text{x}, Supply \text{y} 
WHERE \text{x.sid} = \text{y.sid} 
\quad \text{and} \quad \text{y.pno} = 2 
\quad \text{and} \quad \text{x.scity} = \text{’Seattle’} 
\quad \text{and} \quad \text{x.sstate} = \text{’WA’}

Supplier(\text{sid}, \text{sname}, \text{scity}, \text{sstate})
Supply(\text{sid}, \text{pno}, \text{quantity})
Physical Query Plan 3

Different but equivalent logical query plan; different physical plan

\[
\text{SELECT} \text{ sname} \\
\text{FROM} \text{ Supplier} \ x, \text{ Supply} \ y \\
\text{WHERE} \ x.\text{sid} = y.\text{sid} \\
\qquad \text{and} \ y.\text{pno} = 2 \\
\qquad \text{and} \ x.\text{scity} = 'Seattle' \\
\qquad \text{and} \ x.\text{sstate} = 'WA' \\
\]
Different but equivalent logical query plan; different physical plan

```
SELECT sname
FROM Supplier x, Supply y
WHERE x.sid = y.sid
    and y.pno = 2
    and x.scity = 'Seattle'
    and x.sstate = 'WA'
```

Note difference from Plan 3
Physical Query Plan 5

Different but equivalent logical query plan; different physical plan

```
SELECT sname
FROM Supplier x, Supply y
WHERE x.sid = y.sid
  and y.pno = 2
  and x.scity = 'Seattle'
  and x.sstate = 'WA'
```

```
Π_{sname} (σ_{pno=2} (σ_{sstate='WA'} (σ_{scity='Seattle'} (Supplier)))
```

(On the fly)

(On the fly)

(Nested loop)
Query Optimization Problem

- For each SQL query... many logical plans
- For each logical plan... many physical plans
- Optimizer examines multiple equivalent plans, chooses one with minimum cost
Query Execution
Iterator Interface

- Iterators:
  - Do not materialize intermediate results
  - Children pipeline their results to parents
- Every physical operator maintains its own execution state and implements the following methods
  - `open()`: Initialize state and get ready for processing
  - `next()`: Operator invokes `get_next()` recursively on its inputs; Performs processing and produces an output tuple
  - `close()`: clean-up state
An iterator for file scan

- **state:** a block of memory for buffering input
  a pointer to a tuple within the block
- **open():** allocate a block of memory
- **next():**
  - If no block of has been read yet, read the first block from the disk and return the first tuple in the block
  - If there is no more tuple left in the current block, read the next block of from the disk and return the first tuple in the block
  - Otherwise, return the next tuple in the memory block
- **close():** deallocate the block of memory
Pipelined Query Execution

\( \sigma \text{sscity='Seattle'} \land \text{sstate='WA'} \land \text{pno=2} \)

\( \Pi_{\text{sname}} \)

\( \text{open()} \)

(On the fly)

(Nested loop)

Suppliers (File scan)

(On the fly)

Supplies (File scan)
Pipelined Query Execution

\(\text{(On the fly)}\)
\[\sigma_{\text{sscity} = 'Seattle' \land \text{sstate} = 'WA' \land pno = 2}\]
\[\Pi_{\text{sname}}\]
\[\text{next()}\]
\[\text{(On the fly)}\]
\[\text{next()}\]
\[\text{(Nested loop)}\]
\[\text{sno = sno}\]
\[\text{next()}\]
\[\text{next()}\]
\[\text{Suppliers (File scan)}\]
\[\text{Supplies (File scan)}\]

DATA514 - Winter 2018
Pipelined Execution

• Tuples generated by an operator are immediately sent to the parent

• Benefits:
  – No operator synchronization issues
  – No need to buffer tuples between operators
  – Saves cost of writing intermediate data to disk
  – Saves cost of reading intermediate data from disk

• This approach is used whenever possible
Intermediate Tuple Materialization

• Tuples generated by an operator are written to disk and in intermediate table

• No direct benefit
• Necessary:
  – For certain operator implementations
  – When we don’t have enough memory
Suppliers

\[ \sigma_{\text{sscity}='Seattle' \land \text{sstate}='WA'} \]

\[ \Pi_{\text{sname}} \]

\[ \sigma_{\text{pno}=2} \]

\[ \text{Suppliers (File scan)} \]

\[ \text{Supplies (File scan)} \]

(Scan: write to T1)

(Scan: write to T2)

(On the fly)

(Sort-merge join)

Intermediate Tuple Materialization

DATA514 - Winter 2018
Query Execution Bottom Line

• SQL query transformed into **physical plan**

  – **Access path selection** for each relation
    • Scan the relation or use an index (rest of this lecture)
  
  – **Implementation choice** for each operator
    • Nested loop join, hash join, etc.

  – **Scheduling decisions** for operators
    • Pipelined execution or intermediate materialization
Data Storage

- DBMSs store data in **files**
- Most common organization is row-wise storage
- On disk, a file is split into **blocks**
- Each block contains a set of tuples

In the example, we have **4 blocks** with 2 tuples each

<table>
<thead>
<tr>
<th>ID</th>
<th>fName</th>
<th>lName</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Tom</td>
<td>Hanks</td>
</tr>
<tr>
<td>20</td>
<td>Amy</td>
<td>Hanks</td>
</tr>
<tr>
<td>50</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>200</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>220</td>
<td></td>
<td></td>
</tr>
<tr>
<td>240</td>
<td></td>
<td></td>
</tr>
<tr>
<td>420</td>
<td></td>
<td></td>
</tr>
<tr>
<td>800</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Student

<table>
<thead>
<tr>
<th>ID</th>
<th>fName</th>
<th>lName</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Tom</td>
<td>Hanks</td>
</tr>
<tr>
<td>20</td>
<td>Amy</td>
<td>Hanks</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Data File Types

The data file can be one of:

• Heap file
  – Unsorted

• Sequential file
  – Sorted according to some attribute(s) called key

<table>
<thead>
<tr>
<th>ID</th>
<th>fName</th>
<th>lName</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Tom</td>
<td>Hanks</td>
</tr>
<tr>
<td>20</td>
<td>Amy</td>
<td>Hanks</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Data File Types

The data file can be one of:

- **Heap file**
  - Unsorted
- **Sequential file**
  - Sorted according to some attribute(s) called *key*

Note: *key* here means something different from primary key: it just means that we order the file according to that attribute. In our example we ordered by **ID**. Might as well order by **fName**, if that seems a better idea for the applications running on our database.
Index

• An **additional** file, that allows fast access to records in the data file given a search key
Index

• An **additional** file, that allows fast access to records in the data file given a search key

• The index contains (key, value) pairs:
  – The key = an attribute value (e.g., student ID or name)
  – The value = a pointer to the record
Index

• An **additional** file, that allows fast access to records in the data file given a search key

• The index contains (key, value) pairs:
  – The key = an attribute value (e.g., student ID or name)
  – The value = a pointer to the record

• Could have many indexes for one table

*Key = means here search key*
Index Classification

• **Clustered/unclustered**
  – Clustered = records close in index are close in data
    • Option 1: Data inside data file is sorted on disk
    • Option 2: Store data directly inside the index (no separate files)
  – Unclustered = records close in index may be far in data

• **Primary/secondary**
  – Meaning 1:
    • Primary = is over attributes that include the primary key
    • Secondary = otherwise
  – Meaning 2: means the same as clustered/unclustered
Example 1: Index on ID

Index **Student_ID** on **Student.ID**

Data File **Student**

<table>
<thead>
<tr>
<th>ID</th>
<th>fName</th>
<th>lName</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Tom</td>
<td>Hanks</td>
</tr>
<tr>
<td>20</td>
<td>Amy</td>
<td>Hanks</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Clustered Index
Example 2: Index on fName

Index **Student_fName**
on **Student.fName**

Data File **Student**

<table>
<thead>
<tr>
<th>ID</th>
<th>fName</th>
<th>lName</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Tom</td>
<td>Hanks</td>
</tr>
<tr>
<td>20</td>
<td>Tom</td>
<td>Cruise</td>
</tr>
<tr>
<td>...</td>
<td>Amy</td>
<td>Hanks</td>
</tr>
</tbody>
</table>

Unclustered Index
Index Organization

Several index organizations:

- Hash table
- B+ trees – most popular
  - They are search trees, but they are not binary instead have higher fanout
  - Will discuss them briefly next
- Specialized indexes: bit maps, R-trees, inverted index
B+ Trees Basics

- Parameter \( d \) = the degree
- Each node has \( d \leq m \leq 2d \) keys (except root)

Each node also has \( m+1 \) pointers

- Each leaf has \( d \leq m \leq 2d \) keys

DATA514 - Winter 2018
B+ Tree Index by Example

d = 2

Find the key 40
Searching a B+ Tree

• Exact key values:
  – Start at the root
  – Proceed down, to the leaf

• Range queries:
  – As above
  – Then sequential traversal

Select name
From people
Where age = 25

Select name
From people
Where 20 <= age
and age <= 30
Clustered vs Unclustered

Every table can have **only one** clustered and **many** unclustered indexes.
Scanning a Data File

- Disks are mechanical devices!
  - Technology from the 60s; density much higher now
- We read only at the rotation speed!
- Consequence:
  Sequential scan is MUCH FASTER than random reads
  - **Good**: read blocks 1,2,3,4,5,…
  - **Bad**: read blocks 2342, 11, 321,9, …
Scanning a Data File

• Disks are mechanical devices!
  – Technology from the 60s; density much higher now

• We read only at the rotation speed!

• Consequence:
  Sequential scan is MUCH FASTER than random reads
  – **Good**: read blocks 1,2,3,4,5,…
  – **Bad**: read blocks 2342, 11, 321,9, …

• Rule of thumb:
  – Random reading 1-2% of the file ≈ sequential scanning the entire file
  – Solid state (SSD): $$$ expensive; put indexes, other “hot” data there, not enough room for everything
Getting Practical: Creating Indexes in SQL

CREATE TABLE V(M int, N varchar(20), P int);

CREATE INDEX V1 ON V(N);

CREATE INDEX V2 ON V(P, M);

CREATE INDEX V3 ON V(M, N);

CREATE UNIQUE INDEX V4 ON V(N);

CREATE CLUSTERED INDEX V5 ON V(N);
Getting Practical: Creating Indexes in SQL

CREATE TABLE V(M int, N varchar(20), P int);

CREATE INDEX V1 ON V(N)

CREATE INDEX V2 ON V(P, M)

CREATE INDEX V3 ON V(M, N)

CREATE UNIQUE INDEX V4 ON V(N)

CREATE CLUSTERED INDEX V5 ON V(N)

What does this mean?
Creating Indexes in SQL

CREATE TABLE V(M int, N varchar(20), P int);

CREATE INDEX V1 ON V(N);

CREATE INDEX V2 ON V(P, M);

CREATE INDEX V3 ON V(M, N);

CREATE UNIQUE INDEX V4 ON V(N);

CREATE CLUSTERED INDEX V5 ON V(N);

What does this mean?

Not supported in SQLite
Which Indexes?

• How many indexes **could** we create?

• Which indexes **should** we create?
Which Indexes?

- How many indexes **could** we create?

  15, namely: (ID), (fName), (lName), (ID,fName),(fName,ID),…

- Which indexes **should** we create?
Which Indexes?

• How many indexes could we create?

  15, namely: (ID), (fName), (lName), (ID,fName),(fName,ID),…

• Which indexes should we create?

  Few! Each new index slows down updates to Student
Which Indexes?

• How many indexes could we create?

18, namely: (ID), (fName), (lName), (ID,fName),(fName,ID),…

• Which indexes should we create?

Few! Each new index slows down updates to Student

Index selection is a hard problem
Which Indexes?

- The *index selection problem*
  - Given a table, and a “workload” (big Java application with lots of SQL queries), decide which indexes to create (and which ones NOT to create!)

- Who does index selection:
  - The database administrator DBA
    - Semi-automatically, using a database administration tool
Which Indexes?

• The *index selection problem*
  – Given a table, and a “workload” (big Java application with lots of SQL queries), decide which indexes to create (and which ones NOT to create!)

• Who does index selection:
  – The database administrator DBA
  – Semi-automatically, using a database administration tool
Index Selection: Which Search Key

- Make some attribute $K$ a search key if the WHERE clause contains:
  - An exact match on $K$
  - A range predicate on $K$
  - A join on $K$
The Index Selection Problem 1

V(M, N, P);

Your workload is this

100000 queries:

SELECT *
FROM V
WHERE N=?

100 queries:

SELECT *
FROM V
WHERE P=?
The Index Selection Problem 1

Your workload is this

100000 queries:

100 queries:

What indexes?
The Index Selection Problem 1

\[ V(M, N, P); \]

Your workload is this

100000 queries:

\[
\text{SELECT * FROM } V \text{ WHERE } N=\
\]

100 queries:

\[
\text{SELECT * FROM } V \text{ WHERE } P=\
\]

A: \( V(N) \) and \( V(P) \) (hash tables or B-trees)
The Index Selection Problem 2

V(M, N, P);

Your workload is this
100000 queries:
SELECT * 
FROM V
WHERE N>? and N<?

100 queries:
SELECT * 
FROM V
WHERE P=?

100000 queries:
INSERT INTO V 
VALUES (?, ?, ?)

What indexes?
The Index Selection Problem 2

V(M, N, P);

Your workload is this

100000 queries:

SELECT *
FROM V
WHERE N>? and N<?

100 queries:

SELECT *
FROM V
WHERE P=?

100000 queries:

INSERT INTO V
VALUES (?, ?, ?)

A: definitely V(N) (must B-tree); unsure about V(P)
The Index Selection Problem 3

V(M, N, P);

Your workload is this

100000 queries:  
SELECT *  
FROM V  
WHERE N=?

1000000 queries:  
SELECT *  
FROM V  
WHERE N=? and P>?

100000 queries:  
INSERT INTO V  
VALUES (?, ?, ?)

What indexes?
The Index Selection Problem 3

\[ V(M, N, P); \]

Your workload is this

100000 queries: \[ SELECT * \]
FROM \[ V \]
WHERE \[ N=? \]

1000000 queries: \[ SELECT * \]
FROM \[ V \]
WHERE \[ N=? \] and \[ P>？ \]

100000 queries: \[ INSERT INTO V VALUES (？, ？, ？) \]

A: \[ V(N, P) \]

How does this index differ from:
1. Two indexes \[ V(N) \] and \[ V(P) \]?
2. An index \[ V(P, N) \]?
The Index Selection Problem 4

\[ V(M, N, P); \]

Your workload is this

1000 queries:

\[
\begin{align*}
\text{SELECT} & \quad \text{*} \\
\text{FROM} & \quad V \\
\text{WHERE} & \quad N>? \quad \text{and} \quad N<?
\end{align*}
\]

100000 queries:

\[
\begin{align*}
\text{SELECT} & \quad \text{*} \\
\text{FROM} & \quad V \\
\text{WHERE} & \quad P>? \quad \text{and} \quad P<?
\end{align*}
\]

What indexes?
The Index Selection Problem 4

V(M, N, P);

Your workload is this

1000 queries:

SELECT *
FROM V
WHERE N>? and N<?

100000 queries:

SELECT *
FROM V
WHERE P>? and P<?

A: V(N) secondary, V(P) primary index
Basic Index Selection Guidelines

• Consider queries in workload in order of importance

• Consider relations accessed by query
  – No point indexing other relations

• Look at WHERE clause for possible search key

• Try to choose indexes that speed-up multiple queries

• To Cluster or Not?
  – Range queries benefit mostly from clustering
SELECT * 
FROM R 
WHERE K>? and K<?

Cost

Percentage tuples retrieved
SELECT * 
FROM R 
WHERE K>? and K<?
72

Percentage tuples retrieved

Cost

Sequential scan

Clustered index

SELECT *
FROM R
WHERE K>? and K<?
SELECT *
FROM R
WHERE K>? and K<?
Using Subqueries to Solve Problems: More Examples
Example 0

Employees(id, name, salary)

Main query: Which employees have salaries greater than Richard’s salary?
- What is Richard’s salary?
Example 0

Employees(id, name, salary)

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>Richard</td>
<td>50</td>
</tr>
<tr>
<td>20</td>
<td>Roger</td>
<td>150</td>
</tr>
<tr>
<td>33</td>
<td>David</td>
<td>130</td>
</tr>
<tr>
<td>23</td>
<td>Nick</td>
<td>30</td>
</tr>
</tbody>
</table>

SELECT names
FROM Employees
WHERE Salary > (SELECT Salary
FROM Employees
WHERE name='Richard')
Example 0

Employees(id, name, salary)

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>Richard</td>
<td>50</td>
</tr>
<tr>
<td>20</td>
<td>Roger</td>
<td>150</td>
</tr>
<tr>
<td>33</td>
<td>David</td>
<td>130</td>
</tr>
<tr>
<td>23</td>
<td>Nick</td>
<td>30</td>
</tr>
</tbody>
</table>

Main query: get the second highest salary?
Example 0

Employees(id, name, salary)

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>Richard</td>
<td>50</td>
</tr>
<tr>
<td>20</td>
<td>Roger</td>
<td>150</td>
</tr>
<tr>
<td>33</td>
<td>David</td>
<td>130</td>
</tr>
<tr>
<td>23</td>
<td>Nick</td>
<td>30</td>
</tr>
</tbody>
</table>

Main query: get the second highest salary?
- Which employee receives the highest salary?
Example 0

Employees(id, name, salary)

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>Richard</td>
<td>50</td>
</tr>
<tr>
<td>20</td>
<td>Roger</td>
<td>150</td>
</tr>
<tr>
<td>33</td>
<td>David</td>
<td>130</td>
</tr>
<tr>
<td>23</td>
<td>Nick</td>
<td>30</td>
</tr>
</tbody>
</table>

Main query: get the second highest salary?

– Which employee receives the highest salary?
– Let’s find the other employees, i.e., those who do not receive the highest salary.
Example 0

Main query: get the second highest salary?
- Which employee receives the highest salary?
- Let’s find the other employees, i.e., those who do not receive the highest salary
- Let’s find the maximum salary among those who do not receive the highest salary
Example 0

Employees(id, name, salary)

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>Richard</td>
<td>50</td>
</tr>
<tr>
<td>20</td>
<td>Roger</td>
<td>150</td>
</tr>
<tr>
<td>33</td>
<td>David</td>
<td>130</td>
</tr>
<tr>
<td>23</td>
<td>Nick</td>
<td>30</td>
</tr>
</tbody>
</table>

Main query: get the second highest salary?

```
SELECT * FROM Employee WHERE Salary IN ( SELECT max(Salary) FROM Employee)
```

This will return the second record in our case
Example 0

Main query: get the second highest salary?

```
SELECT * FROM Employee WHERE Salary NOT IN (SELECT max(Salary) FROM Employee)
```

This will return all records except for the second one in our case.
Example 0

Main query: get the second highest salary?

```
Employees(id, name, salary)

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>Richard</td>
<td>50</td>
</tr>
<tr>
<td>20</td>
<td>Roger</td>
<td>150</td>
</tr>
<tr>
<td>33</td>
<td>David</td>
<td>130</td>
</tr>
<tr>
<td>23</td>
<td>Nick</td>
<td>30</td>
</tr>
</tbody>
</table>
```

```
SELECT max(Salary) FROM Employee WHERE Salary NOT IN ( SELECT max(Salary) FROM Employee)
```

This will return 130 in our case.
Example 0

Employees(id, name, salary)

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>Richard</td>
<td>50</td>
</tr>
<tr>
<td>20</td>
<td>Roger</td>
<td>150</td>
</tr>
<tr>
<td>33</td>
<td>David</td>
<td>130</td>
</tr>
<tr>
<td>23</td>
<td>Nick</td>
<td>30</td>
</tr>
</tbody>
</table>

Main query: get the second highest salary?

Exercise: use a correlated subquery!
Example 1

Find drinkers that frequent some bar that serves some beer they like.
Example 1

Find drinkers that frequent some bar that serves some beer they like.
Example 1

Find drinkers that frequent some bar that serves some beer they like.

\[
\text{SELECT DISTINCT } \text{X.drinker} \\
\text{FROM Frequents X} \\
\text{WHERE EXISTS (SELECT bar} \\
\text{FROM Serves Y, Likes Z} \\
\text{WHERE X.bar} = \text{Y.bar} \\
\text{AND X.drinker} = \text{Z.drinker} \\
\text{AND Y.beer} = \text{Z.beer})
\]
Negation of Quantifiers

The statement: "It is not true that all x have the property P" is equivalent to: "There is some x for which ~P is true".

Example:

– Not all people are honest = Some people are not honest.
Negation of Quantifiers

The statement: "It is not true that there is some $x$ with the property $P$"

is equivalent to: "No $x$ has the property $P$" or "All $x$ have the property $\sim P$.”

Example:

– This bar does not serve some beer that I like = This bar only serves beer that I don’t like.

– I only frequent bars that serve some beer I like = I do not frequent bars that only serves beer I don’t like
Example 2

Find drinkers that frequent some bar that serves only beers they don’t like.

<table>
<thead>
<tr>
<th>Likes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>drinker</strong></td>
</tr>
<tr>
<td>Roger</td>
</tr>
<tr>
<td>David</td>
</tr>
<tr>
<td>Nick</td>
</tr>
<tr>
<td>Richard</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Frequents</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>name</strong></td>
</tr>
<tr>
<td>Roger</td>
</tr>
<tr>
<td>Roger</td>
</tr>
<tr>
<td>Nick</td>
</tr>
<tr>
<td>Richard</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Serves</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>bar</strong></td>
</tr>
<tr>
<td>Joe’s</td>
</tr>
<tr>
<td>adam’s</td>
</tr>
<tr>
<td>adam’s</td>
</tr>
<tr>
<td>Sue’s</td>
</tr>
</tbody>
</table>
Likes(drinker, beer)  
Frequents(drinker, bar)  
Serves(bar, beer)

Example 2
Find drinkers that frequent some bar that serves only beers they don’t like.

<table>
<thead>
<tr>
<th>Likes</th>
<th>Frequents</th>
<th>Serves</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>drinker</strong></td>
<td><strong>beer</strong></td>
<td><strong>name</strong></td>
</tr>
<tr>
<td>Roger</td>
<td>Bud</td>
<td>Roger</td>
</tr>
<tr>
<td>David</td>
<td>Michelob</td>
<td>Roger</td>
</tr>
<tr>
<td>Nick</td>
<td>Bud Lite</td>
<td>Nick</td>
</tr>
<tr>
<td>Richard</td>
<td>Bud</td>
<td>Richard</td>
</tr>
</tbody>
</table>

Find drinkers that frequent some bar that DO NOT serves some beers they like
Example 2

Find drinkers that frequent some bar that DO NOT serves some beers they like

```
SELECT DISTINCT X.drinker
FROM Frequents X
WHERE NOT EXISTS (SELECT bar
                     FROM Serves Y, Likes Z
                     WHERE X.bar=Y.bar
                     AND X.drinker=Z.drinker
                     AND Y.beer = Z.beer)
```
Example 3

Find drinkers that frequent only bars that serves some beer they like.

<table>
<thead>
<tr>
<th>Likes</th>
<th>Frequents</th>
<th>Serves</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>drinker</strong></td>
<td><strong>name</strong></td>
<td><strong>bar</strong></td>
</tr>
<tr>
<td>Roger</td>
<td>Roger</td>
<td>Joe’s</td>
</tr>
<tr>
<td>David</td>
<td>Roger</td>
<td>adam’s</td>
</tr>
<tr>
<td>Nick</td>
<td>Nick</td>
<td>Sue’s</td>
</tr>
<tr>
<td>Richard</td>
<td>Richard</td>
<td>Sue’s</td>
</tr>
<tr>
<td></td>
<td>David</td>
<td>Sue’s</td>
</tr>
</tbody>
</table>

Find drinkers that **DO NOT** frequent some bar that only serves beers they don’t like!