Introduction to Database Systems
CSE 344

Lecture 7:
Basic Query Evaluation and Indexes
Announcements

• Webquiz 3 due next Tuesday night, 11 pm
  • Short, covers nested queries
• Homework 2 due this Thursday night, 11 pm

• Today: nested queries, indexes
• Reading: (6.3, review), 14.1
Question for Database Fans and their Friends

• When can we *unnest* subqueries?
Monotone Queries

• Definition A query Q is \textbf{monotone} if:
  – Whenever we add tuples to one or more input tables, the answer to the query will not lose any of the tuples

<table>
<thead>
<tr>
<th>Product</th>
<th>Company</th>
</tr>
</thead>
<tbody>
<tr>
<td>pname</td>
<td>cid</td>
</tr>
<tr>
<td>price</td>
<td>cname</td>
</tr>
<tr>
<td>Gizmo</td>
<td>c001</td>
</tr>
<tr>
<td>19.99</td>
<td>c002</td>
</tr>
<tr>
<td>999.99</td>
<td>c001</td>
</tr>
<tr>
<td>Camera</td>
<td>c003</td>
</tr>
<tr>
<td>149.99</td>
<td>c003</td>
</tr>
</tbody>
</table>
Monotone Queries

- Definition A query Q is **monotone** if:
  - Whenever we add tuples to one or more input tables, the answer to the query will not lose any of the tuples.

---

<table>
<thead>
<tr>
<th>Product</th>
<th>Company</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>pname</strong></td>
<td><strong>cid</strong></td>
</tr>
<tr>
<td>Gizmo</td>
<td>c001</td>
</tr>
<tr>
<td>Gadget</td>
<td>c004</td>
</tr>
<tr>
<td>Camera</td>
<td>c003</td>
</tr>
<tr>
<td>iPad</td>
<td>c001</td>
</tr>
</tbody>
</table>

---

CSE 344 - Winter 2015
Monotone Queries

• **Theorem**: If Q is a SELECT-FROM-WHERE query that does not have subqueries, and no aggregates, then it is monotone.
Monotone Queries

• **Theorem:** If Q is a SELECT-FROM-WHERE query that does not have subqueries, and no aggregates, then it is monotone.

• Proof. We use the nested loop semantics: if we insert a tuple in a relation \( R_i \), this will not remove any tuples from the answer.

\[
\text{SELECT } a_1, a_2, ..., a_k \\
\text{FROM } R_1 \text{ AS } x_1, R_2 \text{ AS } x_2, ..., R_n \text{ AS } x_n \\
\text{WHERE } \text{Conditions}
\]
Monotone Queries

- The query:

Find all companies s.t. all their products have price < 200 is not monotone

<table>
<thead>
<tr>
<th>pname</th>
<th>price</th>
<th>cid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>19.99</td>
<td>c001</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cid</th>
<th>cname</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>c001</td>
<td>Sunworks</td>
<td>Bonn</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sunworks</td>
</tr>
</tbody>
</table>

Product \((\text{pname, price, cid})\)
Company\((\text{cid, cname, city})\)
Monotone Queries

• The query:

Find all companies s.t. all their products have price < 200

is not monotone

<table>
<thead>
<tr>
<th>pname</th>
<th>price</th>
<th>cid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>19.99</td>
<td>c001</td>
</tr>
<tr>
<td>Gadget</td>
<td>999.99</td>
<td>c001</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cid</th>
<th>cname</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>c001</td>
<td>Sunworks</td>
<td>Bonn</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cid</th>
<th>cname</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>c001</td>
<td>Sunworks</td>
<td>Bonn</td>
</tr>
</tbody>
</table>

• Consequence: we cannot write it as a SELECT-FROM-WHERE query without nested subqueries
Queries that must be nested

- Queries with universal quantifiers or with negation
Queries that must be nested

• Queries with universal quantifiers or with negation
• Queries that use aggregates in funny ways
  – Note: \( \text{sum}(..) \) and \( \text{count}(*) \) are NOT monotone, because they do not satisfy set containment
  – \textbf{select count}(*) from } R \textbf{ is not monotone!}
Where We Are

• We learned importance and benefits of DBMSs
• We learned how to use a DBMS
  – How to specify what our data will look like: schema
  – How to load data into the DBMS
  – How to ask SQL queries
Where We Are

• We learned importance and benefits of DBMSs
• We learned how to use a DBMS
  – How to specify what our data will look like: schema
  – How to load data into the DBMS
  – How to ask SQL queries
• Next:
  – How the DBMS executes a query
  – How we can help it run faster
Query Evaluation Steps

1. **Parse & Check Query**
2. **Decide how best to answer query: query optimization**
3. **Query Execution**
4. **Return Results**
### Example

#### Student

<table>
<thead>
<tr>
<th>ID</th>
<th>fName</th>
<th>lName</th>
</tr>
</thead>
<tbody>
<tr>
<td>195428</td>
<td>Tom</td>
<td>Hanks</td>
</tr>
<tr>
<td>645947</td>
<td>Amy</td>
<td>Hanks</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

#### Takes

<table>
<thead>
<tr>
<th>studentID</th>
<th>courseID</th>
</tr>
</thead>
<tbody>
<tr>
<td>195428</td>
<td>344</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

#### Course

<table>
<thead>
<tr>
<th>courseID</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>344</td>
<td>Databases</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**SQL Query**

```
SELECT *
FROM  Student x, Takes y
WHERE x.ID=y.studentID AND y.courseID > 300
```

How can the DBMS answer this query?
Possible Query Plan 1

for y in Takes
    if courseID > 300 then
        for x in Student
            if x.ID=y.studentID
                output *
Possible Query Plan 1

```
for y in Takes
  if courseID > 300 then
    for x in Student
      if x.ID=y.studentID
        output *
```
Possible Query Plan 2

sort Student on ID
sort Takes on studentID (and filter on coursesID > 300)
merge join Student, Takes on ID = studentID
for (x,y) in merged_result output *

SELECT *
FROM  Student x, Takes y
WHERE x.ID=y.studentID AND y.courseID > 300
Possible Query Plan 2

**sort** Student on ID
**sort** Takes on studentID (and filter on coursesID > 300)
**merge join** Student, Takes on ID = studentID
**for** (x,y) in merged_result **output** *

```
SELECT *
FROM  Student x, Takes y
WHERE x.ID=y.studentID AND y.courseID > 300
```
Possible Query Plan 3

create a hash-table
for x in Student
  insert x in the hash-table on x.ID

for y in Takes
  if courseID > 300
    then probe y.studentID in hash-table
      if match found
      then output *

SELECT *
FROM Student x, Takes y
WHERE x.ID=y.studentID AND y.courseID > 300
Possible Query Plan 3

create a hash-table
for x in Student
    insert x in the hash-table on x.ID

for y in Takes
    if courseID > 300
        then probe y.studentID in hash-table
            if match found
                then output *

SELECT *
FROM Student x, Takes y
WHERE x.ID=y.studentID AND y.courseID > 300
Discussion

Which plan is best? Choose one:

• Nested loop join

• Merge join

• Hash join
Discussion

Which plan is best? Choose one:

• **Nested loop join**: \(O(N^2)\)
  – Could be \(O(N)\) when few courses \(> 300\)

• **Merge join**:

• **Hash join**: 

```python
for y in Takes
    if courseID > 300 then
        for x in Student
            if x.ID = y.studentID
                output *

create a hash-table
for x in Student
    insert x in the hash-table on x.ID

for y in Takes
    if courseID > 300
        then probe y.studentID in hash-table
            if match found
                then return match
```
Discussion

Which plan is best? Choose one:

- **Nested loop join**: $O(N^2)$
  - Could be $O(N)$ when few courses $>300$
- **Merge join**: $O(N \log N)$
  - Could be $O(N)$ if tables already sorted
- **Hash join**:

```sql
for y in Takes
  if courseID > 300 then
    for x in Student
      if x.ID=y.studentID
        output *
  sort Student on ID
  sort Takes on studentID (and filter on coursesID > 300)
  merge join Student, Takes on ID = studentID
  return results

create a hash-table
for x in Student
  insert x in the hash-table on x.ID

for y in Takes
  if courseID > 300
    then probe y.studentID in hash-table
      if match found
        then return match
```
Discussion

Which plan is best? Choose one:

- **Nested loop join**: \(O(N^2)\)
  - Could be \(O(N)\) when few courses > 300

- **Merge join**: \(O(N \log N)\)
  - Could be \(O(N)\) if tables already sorted

- **Hash join**: \(O(N)\) expectation

```
for y in Takes
    if courseId > 300 then
        for x in Student
            if x.ID = y.studentID
                output *

sort Student on ID
sort Takes on studentID (and filter on coursesID > 300)
merge join Student, Takes on ID = studentID
return results
```

```
create a hash-table
for x in Student
    insert x in the hash-table on x.ID

for y in Takes
    if courseId > 300
        then probe y.studentID in hash-table
        if match found
            then return match
```
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>...</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
This is Not A Key

Different keys:

- **Primary key** – uniquely identifies a tuple
- **Key of the sequential file** – how the datafile is sorted, if at all
- **Index key** – how the index is organized
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>
</tbody>
</table>

...
Example 2: Index on fName

Index **Student** \_fName

on **Student.fName**

<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>
</tbody>
</table>

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>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>
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+ Tree Index by Example

$d = 2$

Find the key 40
Clustered vs Unclustered

Every table can have only one clustered and many unclustered indexes

CLUSTERED

UNCLUSTERED

B+ Tree

Data entries

(Data file)

(Index File)

Data Records

Data Records
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
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
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

- **Organization** B+ tree or Hash table
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; this is decreasing over time (because of increased density of disks)

• Solid state (SSD): $$$ expensive; put indexes, other “hot” data there, not enough room for everything
Query Plan 1 Revisited

\[
\text{for } y \text{ in Takes}
\]
\[
\text{if courseID > 300 then}
\]
\[
\text{for } x \text{ in Student}
\]
\[
\text{if } x.\text{ID}=y.\text{studentID}
\]
\[
\text{output } *
\]

Assume the database has indexes on these attributes:
- \text{index\_takes\_courseID} = index on Takes.courseID
- \text{index\_student\_ID} = index on Student.ID

\[
\text{for } y \text{ in index\_Takes\_courseID where } y.\text{courseID} > 300
\]
\[
\text{for } x \text{ in Takes where } x.\text{ID} = y.\text{studentID}
\]
\[
\text{output } *
\]
SELECT *
FROM Student x, Takes y
WHERE x.ID=y.studentID AND y.courseID > 300

for y in Takes
  if courseID > 300 then
    for x in Student
      if x.ID=y.studentID
        output *
  end

Assume the database has indexes on these attributes:
• index_takes_courseID = index on Takes.courseID
• index_student_ID = index on Student.ID

for y in index_Takes_courseID where y.courseID > 300
  for x in Takes where x.ID = y.studentID
    output *
SELECT *
FROM Student x, Takes y
WHERE x.ID=y.studentID AND y.courseID > 300

Assume the database has indexes on these attributes:
- `index_takes_courseID` = index on Takes.courseID
- `index_student_ID` = index on Student.ID

```plaintext
for y in Takes
  if courseID > 300 then
    for x in Student
      if x.ID=y.studentID
        output *

for y in index_Takes_courseID where y.courseID > 300
  for x in Takes where x.ID = y.studentID
  output *
```
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);

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

In general this is a very 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

<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>
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

<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>
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

\[ \text{V}(M, N, P); \]

Your workload is this

100000 queries:

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

100 queries:

\[
\text{SELECT * FROM V WHERE P=}? \]
The Index Selection Problem 1

V(M, N, P);

Your workload is this

100000 queries: 100 queries:

SELECT * FROM V WHERE N=?
SELECT * FROM V WHERE P=?

What indexes?
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=?

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>?

1000000 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)?