CS 133: Databases

Spring 2017
Lec 02 – 1/19
Relational Model & Memory and Buffer Manager

Prof. Beth Trushkowsky

Goals for Today

• Reason about the conceptual evaluation of an SQL query

• Understand the storage hierarchy and why disk input/output (I/O) is an important metric for query cost

• See how different policies for managing which data stays in RAM can impact cost of queries

• Users write declarative queries using logical schema
  – May actually interact with application that queries the database
  – Database administrator (DBA) typically creates database

• Given declarative query, DBMS figures out efficient execution strategy

- We’ll start discussion of “choices” today!

<table>
<thead>
<tr>
<th>College database</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Students</strong></td>
</tr>
<tr>
<td><strong>Courses</strong></td>
</tr>
<tr>
<td><strong>Courses</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Students</th>
<th>Courses</th>
</tr>
</thead>
<tbody>
<tr>
<td>SID</td>
<td>name</td>
</tr>
<tr>
<td>CS 121</td>
<td>Software Dev</td>
</tr>
<tr>
<td>CS 70</td>
<td>Data Structures</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SID</th>
<th>name</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>Alice</td>
<td>3.4</td>
</tr>
<tr>
<td>67</td>
<td>Bobtas</td>
<td>3.9</td>
</tr>
</tbody>
</table>

Administrivia

• Problem set 1 out tonight, due Thursday 11:59pm
  – Honor code: can use lectures notes and textbook, can discuss general ideas with classmates
  – On Sakai

• Lab 1: “Getting started” due Wednesday
  – On course website
  (labs will also be linked from assignment on Sakai)
  – Nothing to submit yet

• Grutoring hours almost set, will post on Piazza

Relational Model

Courses (cid: string, name: string, credits: integer)
Multi-Relation Queries

<table>
<thead>
<tr>
<th>Students</th>
<th>Enrolled</th>
</tr>
</thead>
<tbody>
<tr>
<td>SID</td>
<td>name</td>
</tr>
<tr>
<td>45</td>
<td>Alice</td>
</tr>
<tr>
<td>67</td>
<td>Bob</td>
</tr>
<tr>
<td>78</td>
<td>Carl</td>
</tr>
<tr>
<td>SID</td>
<td>CID</td>
</tr>
<tr>
<td>45</td>
<td>CS133</td>
</tr>
<tr>
<td>45</td>
<td>CS121</td>
</tr>
<tr>
<td>78</td>
<td>CS5</td>
</tr>
</tbody>
</table>

SELECT S.name, E.CID
FROM Students S, Enrolled E
WHERE S.sid=E.sid AND E.grade = “B”;

Basic Query: Select-From-Where

SELECT [DISTINCT] \[A_1, A_2, ..., A_n\]
FROM \[R_1, R_2, ..., R_n\]
WHERE \[condition(s)\];

Relation List.
Relations used in query, implicitly JOINed.

Target List
Attributes from relation list.

Comparisons. Conjunctive (“AND”), and Disjunctive (“OR”)

Also called an SPJ (select-project-join)

Query Semantics

Conceptual query evaluation steps:
1. do FROM clause: cross-product of tables
2. do WHERE clause: check conditions, discard tuples that fail
3. do SELECT clause: delete unwanted fields
4. do DISTINCT: eliminate duplicate tuples
(SQL SELECT defaults to keeping duplicates)

Actually very inefficient in practice!
An optimizer will find more efficient strategies to get the same answer.
(2) WHERE: Discard tuples that fail conditions

WHERE S.sid=E.sid AND E.grade='B'

Students X Enrolled

<table>
<thead>
<tr>
<th>S.SID</th>
<th>S.name</th>
<th>S.login</th>
<th>S.gpa</th>
<th>E.SID</th>
<th>E.CID</th>
<th>E.grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>Alice</td>
<td>alicious</td>
<td>3.4</td>
<td>45</td>
<td>CS133</td>
<td>A</td>
</tr>
<tr>
<td>67</td>
<td>Bob</td>
<td>bobtastic</td>
<td>3.9</td>
<td>45</td>
<td>CS133</td>
<td>A</td>
</tr>
<tr>
<td>78</td>
<td>Carl</td>
<td>carl</td>
<td>2.5</td>
<td>45</td>
<td>CS133</td>
<td>A</td>
</tr>
<tr>
<td>45</td>
<td>Alice</td>
<td>alicious</td>
<td>3.4</td>
<td>45</td>
<td>CS121</td>
<td>B</td>
</tr>
<tr>
<td>67</td>
<td>Bob</td>
<td>bobtastic</td>
<td>3.9</td>
<td>45</td>
<td>CS121</td>
<td>B</td>
</tr>
<tr>
<td>78</td>
<td>Carl</td>
<td>carl</td>
<td>2.5</td>
<td>45</td>
<td>CS121</td>
<td>B</td>
</tr>
<tr>
<td>45</td>
<td>Alice</td>
<td>alicious</td>
<td>3.4</td>
<td>78</td>
<td>CS5</td>
<td>A</td>
</tr>
<tr>
<td>67</td>
<td>Bob</td>
<td>bobtastic</td>
<td>3.9</td>
<td>78</td>
<td>CS5</td>
<td>A</td>
</tr>
<tr>
<td>78</td>
<td>Carl</td>
<td>carl</td>
<td>2.5</td>
<td>78</td>
<td>CS5</td>
<td>A</td>
</tr>
</tbody>
</table>

(3) SELECT: Delete Unwanted Fields

SELECT S.name, E.CID

<table>
<thead>
<tr>
<th>S.SID</th>
<th>S.name</th>
<th>S.login</th>
<th>S.gpa</th>
<th>E.SID</th>
<th>E.CID</th>
<th>E.grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>Alice</td>
<td>alicious</td>
<td>3.4</td>
<td>45</td>
<td>CS121</td>
<td>B</td>
</tr>
</tbody>
</table>

Exercise 2: Writing SQL

Students (sid, name, login, gpa)
Courses (cid, name)
Enrolled (sid, cid, grade)

Write an SQL query that finds the course cid for only the courses that gave at least one A grade

SELECT COUNT(*)
FROM Enrolled E;

• What about the count of enrollments per course?
SELECT cid, COUNT(*) courseCount
FROM Enrolled E
GROUP BY cid;

• Enrollments for only “large” classes
SELECT COUNT(sid)
FROM Enrolled E
GROUP BY cid
HAVING COUNT(sid) > 50;
[Less] Basic Query Anatomy

**SELECT** [DISTINCT] \( A_1, A_2, \ldots, A_n \)

**FROM** \( R_1, R_2, \ldots, R_n \)

**WHERE** condition(s)

**GROUP BY** \( A_1, A_2, \ldots, A_n \)

Groups list. Attributes from relation list.

**HAVING** conditions(s);

Group qualifications. Conditions on each group.

---

Query Semantics (cntd)

Conceptual query evaluation steps:

1. do **FROM** clause: *cross-product* of tables

2. do **WHERE** clause: check conditions, discard tuples that fail

3. Remove fields not in **SELECT**, **GROUP BY**, or **HAVING** clauses

4. do **GROUP BY**: partition into groups

5. do **HAVING**: delete groups that do not meet conditions

**Result:** one answer tuple per qualifying group

---

Simplified RDBMS Architecture

- **Application**
- **Query**
- **Query optimizer**
- **Query executor**
- **Access methods**
- **Buffer management**
- **Disk management**
- **Data records**

Let’s look at the system bottom-up!

Concerned with concurrency control and recovery

---

Computer Storage

- **Primary storage**
  - E.g., “main memory” a.k.a random-access memory (RAM)
  - Typically volatile

- **Secondary storage**
  - Hard disk drive
  - Non-volatile

Where variable values go when your program is running!

CPU registers

Cache

RAM

Disk

Tape

SMALL and FAST

BIG and SLOW
Why Not Keep All Data in Memory?

- Costs too much!
  - $100 for 100 GB of RAM or around 2 TB of disk
  - Databases can be in the petabyte (1000 TB) range

- Main memory volatile
  - Want persistence

A Typical Disk

- Moving parts!
  - Platters spin
  - Arms move in/out to position heads with track
  - Tracks under heads make conceptual cylinder

- A block is a unit of transfer
  - made up of one or more sectors

Disk Access Time

- Time to read/write (an Input/Output or I/O) a block
  - Seek time
  - Rotational Delay
  - Transfer time

- **Seek** time and rotational delay dominate (stats: wikipedia)
  - Seek time: about 4 to 15msec
  - Rotational: avg 4msec (7200rpm)
  - Transfer rate: < 0.1msec per 8KB block

*Reduce I/O cost by reducing seek and rotation*

Jim Gray’s Latency Analogy

Modified figure from Alpha Sort paper

Jim Gray: http://news.microsoft.com/1999/05/14/microsoft-researcher-jim-gray-receives-turing-award-for-helping-to-transform-databases-into-dynamic-tools-used-by-millions
Random vs. Sequential Access

If data can be read/written sequentially, have zero seek time and rotational delay!


Exercise 3: Counting I/Os

• Query: joining relations Students and Enrolled
  
  SELECT S.name, E.CID
  FROM Students S, Enrolled E
  WHERE S.sid=E.sid;

• [Simple] join pseudocode:
  For each tuple i of outer relation
  For each tuple j of inner relation
    Check if i.sid == j.sid

• Relation info
  – Students: 20 pages, 1000 total tuples
  – Enrolled: 50 pages, 6000 total tuples
  – For a given relation, pages on disk sequentially

Exercise: Counting I/Os

• Think of the simple algorithm as a nested for-loop like this:

  For each page of Outer relation
    Load that page // one I/O
    For each tuple of Outer on that page
      For each page of Inner relation
        Load that page // one I/O
        For each tuple of Inner on that page
          // do tuple comparison

Exercise: Counting I/Os

• Total I/Os = (# pages in outer) +
  (# tuples in outer) * (# pages in inner)
  – Students outer: 20 + 1000*50 = 50,020
  – Enrolled outer: 50 + 6000*20 = 120,050

• # Random I/Os =
  (# pages in Outer) + (# tuples in Outer)(1)

• # Sequential I/Os =
  (# pages in Inner − 1) (# tuples in Outer)
Simplified RDBMS Architecture

Next: the buffer manager

Disk Space Manager
- Manages space on disk
- Higher levels call on it to allocate/de-allocate, and read/write *pages*

Concerned with concurrency control and recovery

Data records

Query optimizer

Query executor

Access methods

Buffer management

Disk management

Application

The Buffer Manager

- Data must be RAM for DBMS to operate on it
  - Too costly to keep all data in RAM

- Buffer manager
  - Maintain a pool of space in RAM
  - Talks to disk space manager to read/write pages
  - Higher levels do not know what is in RAM or not

Buffer Pool

**Page Requests from Higher Levels**

**BUFFER POOL**

**MAIN MEMORY**

**DISK**

**DB**

choice of frame dictated by replacement policy

**Buffer Pool**: collection of frames used to temporarily keep data for query processor.

Important Terms

- **Disk page**: unit of transfer between disk and memory. Size is DBMS configuration parameter (e.g., 4-32 KB).

- **Frame**: unit of memory. Typically same size as disk page size.
When a Request Comes in...

- If requested page is *in* the buffer pool
  - *Pin* the page to mark as in use

- Else, if requested page is *not* in buffer pool
  - If there is an available frame, put the page in that frame
  - Else, select a frame for replacement using a replacement policy
    (only un-pinned pages are eligible for replacement)
    - If selected frame is *dirty*, write it back to disk
    - Read requested page into the selected frame
    - Pin the page

Buffer Replacement Policy

- When no available frames in buffer pool, need to *evict* one based on a replacement policy
  - Choice of policy impacts number of disk I/Os
  - Efficacy depends on *access pattern* of pages

What would an optimal policy do?

LRU Policy (Least Recently Used)

- Evict the page that was accessed (pinned) furthest in the past, i.e., the *least recently used* of the pages in the pool

- Example:
  - Buffer pool with 4 frames
  - Assume pages are immediately unpinned after use

<table>
<thead>
<tr>
<th>Access pattern:</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>A</th>
<th>D</th>
<th>E</th>
<th>C</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame 1</td>
<td>A</td>
<td></td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Frame 2</td>
<td>B</td>
<td></td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Frame 3</td>
<td></td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Frame 4</td>
<td></td>
<td></td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

  # hits: 2
  # misses: 6

Intuition: if a page has not been used in a while, it probably won’t again soon

Issues with LRU

- Sequential flooding
  - # buffer frames < # pages in file
  - each request causes an I/O

  - E.g., repeated sequential scans

  - MRU (most recently used) policy can be a better policy in this case