

Valued Dictionary Abstract Data Types: Databases - SQL

CS240: Data Structures and Data Management
Slide Set 14

Jérémy Barbay

March 7-9 2006

Outline

SQL

Implementation

Outline

- ▶ We will spend about two lectures on databases
 1. Overview of databases
 2. Terminology
 3. SQL
 4. Relational Algebra
 5. Implementation

Overview

- ▶ A **large** amount of **structured** data stored for a **long** time
 - ▶ Gigabytes and Terabytes of information
 - ▶ Generally outlasts the original technology
- ▶ Updates are frequent
- ▶ Most of the data exists indefinitely
- ▶ Nature of searches change over time
- ▶ Utilize a **DBMS** – Database Management System
- ▶ Only consider **relational databases**

Peanuts Database

► Students Relation:

StudentID	Name	Address	Phone
12345	C. Brown	12 Apple St.	555-1234
67890	L. Van Pelt	34 Pear Ave.	555-5678
22222	P. Patty	56 Grape Blvd.	555-9999

► Grades Relation:

Course	StudentId	Grade
CS101	12345	85
CS101	67890	70
EE200	12345	65
EE200	22222	75
CS101	33333	83
PH100	67890	69

Peanuts Database

► Courses Relation:

Course	Prerequisite
CS101	CS100
EE200	EE005
EE200	CS100
CS120	CS101
CS121	CS120
CS205	CS101
CS206	CS121
CS206	CS205

• Times Relation:

Course	Day	Hour
CS101	M	9AM
CS101	W	9AM
CS101	F	9AM
EE200	T	10AM
EE200	W	1PM
EE200	R	10AM

► Rooms Relation:

Course	Room
CS101	Turing Aud.
EE200	25 Ohm Hall
PH100	Newton Lab

Terminology

- **Attribute** – Name of a set
 - Column name
- **Tuple** – Ordered set; maps attributes to values
 - One row of a table
- **Relation** – Finite set of tuples
 - The table
 - This is the actual information
- **Relation Scheme** – Set of attribute names in a relation
 - Set of column names for a table
- **Database** – Set of relations
 - Set of tables
- **Database Scheme** – Set of relation schemes in a database
 - All the attributes and their various relationships

Assumptions

- No duplicate rows
- Exactly one entry in any row i and column j of a relation
- Rows **are not** ordered
- Columns **are** ordered

Queries

- ▶ We will focus on lookups/queries primarily
- ▶ More powerful requests than standard Find(key)
 - ▶ **Random Access** – find tuple with key K
 - ▶ **Sequential Access** – a list of all the tuples
 - ▶ **Value** – find tuple with attribute $A = v$
 - ▶ **Range** – find all tuples with attributes in given range
 - ▶ **Function** – compute function f for an attribute over a relation
 - ▶ **Boolean** – find all tuples satisfying a boolean expression
 - ▶ **Quantified** – give a yes or no answer
 - ▶ **Similarity** – tuples matching a regular expression

SQL

- ▶ General form of the SQL SELECT statement:

```
SELECT
( DISTINCT          —  $\varepsilon$  )
( columns — * — |aggregate function| |column| )
FROM relations
( WHERE conditions —  $\varepsilon$  )
( GROUP BY columns —  $\varepsilon$  )
( ORDER BY columns —  $\varepsilon$  )
```
- ▶ *conditions* consists of attribute names connected by logical and arithmetic operators
- ▶ *aggregate function* can be:
 - ▶ AVG()
 - ▶ MIN()
 - ▶ MAX()
 - ▶ COUNT()
 - ▶ SUM()

Queries

- ▶ **Note:** The result of a query is a new (unnamed) relation
- ▶ List the personal information for all of the students sorted by student number:

Solution

```
SELECT Name, Address, Phone FROM Students
ORDER BY StudentID;
```

- ▶ List all student numbers for which a grade has been recorded:

Solution

```
SELECT StudentID FROM Grades;
```

- ▶ List all unique student numbers for which a grade has been recorded:

Solution

```
SELECT DISTINCT Student ID FROM Grades;
```

Queries

- ▶ What is the highest student number?

Solution

```
SELECT MAX(StudentID) FROM Students;
```

- ▶ Provide a nicer column name for the above query.

Solution

- ▶ What is Charlie Brown's phone number?

Solution

```
SELECT Phone FROM Students WHERE Name = 'C.Brown';
```

- ▶ What is the average grade in each course?

Solution

```
SELECT Course, AVG(Grade) FROM Grades GROUP BY
Course;
```

Cartesian Product

- ▶ We saw the set operators **union**, **intersection** and **difference**
- ▶ We can also take the **Cartesian product**
- ▶ Every tuple in first set is combined with every tuple in the second set

$R =$	<table><tr><th>A</th><th>B</th></tr><tr><td>0</td><td>1</td></tr><tr><td>2</td><td>3</td></tr></table>	A	B	0	1	2	3	$S =$	<table><tr><th>A</th><th>B</th></tr><tr><td>0</td><td>1</td></tr><tr><td>4</td><td>5</td></tr></table>	A	B	0	1	4	5	$T =$	<table><tr><th>C</th><th>D</th><th>E</th></tr><tr><td>6</td><td>7</td><td>8</td></tr></table>	C	D	E	6	7	8
A	B																						
0	1																						
2	3																						
A	B																						
0	1																						
4	5																						
C	D	E																					
6	7	8																					

▶ $R \times T$:

$R \times S$:

Queries

- ▶ What grade did each student receive (using their name, not their student number)?

Solution

```
SELECT R.Name, S.Course, S.Grade FROM Students R,  
Grades S  
WHERE R.StudentId=S.StudentId;
```

- ▶ What prerequisites must Charlie Brown have taken?

Solution

```
SELECT DISTINCT C.Prerequisite  
FROM Students S, Grades G, Courses C  
WHERE S.name = 'C. Brown'  
AND S.StudentID = G.StudentID  
AND G.course=C.course;
```

Queries

- ▶ Suppose we had the following **CS240Grades Relation**:

StudentId	Grade
98765	85
⋮	⋮

- ▶ What student numbers got above average in the course?

Solution

```
SELECT StudentId  
FROM CS240Grades C1,  
(SELECT AVG(Grade) AS A  
FROM CS240Grades) C2  
WHERE C1.Grade > C2.A
```

Summary

- ▶ Databases are one of the **main motivation** for efficient Data Structures.
- ▶ **Sorted data** is helping, but not always available (or is sorted in another order).

Outline

SQL

Implementation

Relational Algebra

- ▶ Allows us to build mathematical expressions for relational queries
- ▶ Express query without giving details of how to implement
- ▶ Allows for optimizations
 - ▶ One SQL statement may correspond to numerous expressions
 - ▶ Use algebraic laws to convert expressions
 - ▶ Apply rules to (hopefully) minimize total work

Select

- ▶ $\sigma_C(R)$
- ▶ Select tuples from relation R satisfying condition C
- ▶ C can be connected by arithmetic and boolean operators
- ▶ Horizontal subset
- ▶ Like the `WHERE` clause in SQL
- ▶ **Example:** List all the tuples for grades reported in course CS101

Solution

$\sigma_{Course="CS101"}(Grades)$

Project

- ▶ $\pi_A(R)$
- ▶ Delete from relation R all attributes not in A
- ▶ Implicitly removes duplicate rows
- ▶ Vertical subset
- ▶ Like the `SELECT DISTINCT` clause in SQL
- ▶ **Example:** List all the students and their phone numbers

Solution

$\pi_{Name,Phone}(Students)$

- ▶ We can perform composition between various operators
- ▶ **Example:** List all student numbers having taken CS101

Solution

$\pi_{StudentID}(\sigma_{Course="CS101"}(Grades))$

Join

- ▶ $R \bowtie_a S$
- ▶ Concatenate all tuples from R and S where $R.a = S.a$
- ▶ Removes duplicate columns
- ▶ Natural Join – $R \bowtie S$
 - ▶ R and S have only one column in common
- ▶ Combines FROM and WHERE clause in SQL
- ▶ **Example:** List the time and room for each class

Solution

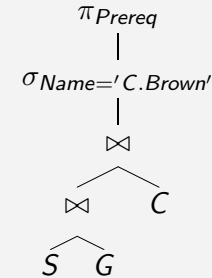
$Rooms \bowtie Times$

Representation

- ▶ What prerequisites must Charlie Brown have taken?
- ▶ Directly translate the SQL from last lecture:

$$\pi_{Prereq} \left(\sigma_{Name='C.Brown'} ((S \bowtie G) \bowtie C) \right)$$

- ▶ We can draw an expression tree:

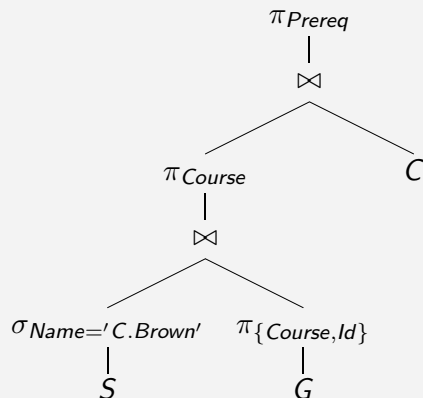


An Optimization

- ▶ An equivalent representation:

$$\pi_{Prereq} \left(\pi_{Course} \left(\sigma_{Name='C.Brown'}(S) \bowtie \pi_{\{Course, Id\}}(G) \right) \bowtie C \right)$$

- ▶ Corresponding expression tree:



How to Optimize

1. Optimize the expression tree

- ▶ Usually, $\text{COST}(\bowtie) \geq \text{COST}(\pi) \geq \text{COST}(\sigma)$
- ▶ We want to push selection and projection as far down the tree as possible
- ▶ Apply laws such as:
 - ▶ $\sigma_c(R \bowtie S) \equiv \sigma_c(R) \bowtie S$ if $c \notin S$
 - ▶ $\pi_A(R \bowtie S) \equiv \pi_A(\pi_{A'}(R) \bowtie \pi_{A''}(S))$
- ▶ Duplicate detection should be left until the end
- ▶ Use heuristics to decide when to apply rules

2. Optimize the implementation of each operator

- ▶ Our focus

Database Implementation

- ▶ Still treat a database as a dictionary ADT
- ▶ **Primary Key** – One or more fields whose value(s) uniquely identify each tuple in a relation
- ▶ **Primary Index** – Main data structure organized by primary keys
 - ▶ B⁺-Tree
 - ▶ Extendible Hashing
 - ▶ What is 12345's phone number?
- ▶ **Secondary Index** – An index based on another attribute
 - ▶ What is Charlie Brown's phone number?
- ▶ Updates are more expensive, but certain queries are much faster

Indices

- ▶ The secondary index does not store data
- ▶ Points to where the data is in the primary index
- ▶ Example:
 - ▶ Id is the primary key with the index stored in a hash table
 - ▶ Name is the secondary index stored in a B-Tree

Implementing Operators

- ▶ Generally either sorting based or hashing based
- ▶ Choice depends on size of relation and existing indices
- ▶ **Union**
 - ▶ Sort both relations
 - ▶ Merge removing duplicates
 - ▶ What if you know there are no duplicates?
- ▶ **Difference**
 - ▶ Identical

Simple Operators

- ▶ **Intersection**
 - ▶ Still much the same
 - ▶ Usually implemented as special case of the join operator
- ▶ **Projection**
 - ▶ Scan each tuple
 - ▶ Removing duplicates is primary problem
 - ▶ Sort as you perform the scan
 - ▶ Could also utilize a hash table

Selection

- ▶ Duplicate detection not an issue
- ▶ **Equality Selection**
 - ▶ $\sigma_{A=v}$
 - ▶ Use (extendible) hashing
 - ▶ Consider Students with Id as primary index
 - Good:** $Id = 12345$
 - Bad:** $Id > 12345$
 - Bad:** $Name = 'C.Brown'$

Selection

- ▶ **Range Selection**
 - ▶ $\sigma_{v \leq A \leq w}$
 - ▶ Use B^+ -Tree
 - ▶ Consider Grades with (Course, Id) as primary index
 - Good:** $Course = CS101 \ \& \ Id = 12345$
 - Good:** $Course = CS101$
 - Bad:** $Id = 12345$
- ▶ Chose the appropriate index based on particular request
- ▶ Create indices based on the queries users tend to ask

Join

- ▶ Suppose we have the following relations (with sample data):

▶ **R:**

A	B
0	6
2	1
4	3

• **S:**

B	C
6	4
3	8
9	12
6	15

- ▶ Perform $R \bowtie S$

A	B	C
0	6	4
0	6	15
4	3	8

- ▶ Let $r = |R|$ and $s = |S|$

Join

There are several ways to implement Joins:

- ▶ **Nested Loop**
- ▶ **Sorted Join**
- ▶ **Index Join**

What are the advantages of each implementation?

Join

nested loop: very costly ($\text{Cost} = rs$)

```
for each tuple a in R
  for each tuple b in S
    if a.B = b.B then output a, b
```

Join

sort join: merge, mark as R or S and sort on B

merge R and S
sort on B attribute
go through linearly, joining on grouped elements (since they will be sorted together)

$(\text{Cost} = (r + s) \log(r + s))$

Join

index join: if index on join element B ($\text{Cost} = r + s$ if index is $O(1)$ time to find tuples (hash table). $\text{Cost} = r + s \log s$ if B-tree used on index)

```
for each tuple a in R
  use index of S to find tuples b where b.B=a.B
  for each tuple b in S
    output a, b
```

Summary

- ▶ Database is a **whole field of research**, we barely scratched the surface here.
- ▶ Database is one of the main motivations for efficient data-structures: how to manage **very large** amount of data.
- ▶ How to compute things is not the whole story, but also in **which order** to compute it.