Review of Basic Database Concepts

CPS 296.1
Topics in Database Systems

What’s a database system?
• According to Oxford Dictionary
  – Database: an organized body of related information
  – Database system, DataBase Management System, or DBMS: a software system that facilitates the creation and maintenance and use of an electronic database
• More precisely, a DBMS should support
  – Efficient and convenient querying and updating of large amounts of persistent data
  – Safe, multi-user access

Two important questions
• What is the right API for a DBMS?
  – Data model
  • How is the data structured conceptually?
  – Query language
  • How do users ask queries about the data?
• How does the DBMS support the API?
  – Query processing and optimization
    • What is the most efficient way to answer a query?
  – Transaction processing
    • How are atomicity, consistency, isolation, and durability of transaction ensured?

Entity-relationship (E/R) diagram
• Entities: students and courses
  • Relationships: students enroll in courses
  
Before the relational “revolution”
• Hierarchical and network data models
  – Relationships are modeled as pointers
  – Queries require explicit pointer following
• Example: a simplified CODASYL query

Before the relational “revolution”

Physical data independence
• Problems with hierarchical and network data models
  – Access to data is not declarative
  – Whenever data is reorganized, applications must be reprogrammed!
• Physical data independence
  – Applications should not need to worry about how data is physically structured and stored
  – Applications should work with a logical data model and declarative query language
  – Leave the implementation details and optimization to DBMS
Relational data model

- A database is a collection of relations (or tables)
- Each relation has a list of attributes (or columns)
- Each relation contains a set of tuples (or rows)
  - Duplicates not allowed

<table>
<thead>
<tr>
<th>Student</th>
<th>Course</th>
<th>Enroll</th>
</tr>
</thead>
<tbody>
<tr>
<td>SID</td>
<td>name</td>
<td>age</td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td>142</td>
<td>Bart</td>
<td>10</td>
</tr>
<tr>
<td>123</td>
<td>Milhouse</td>
<td>10</td>
</tr>
<tr>
<td>857</td>
<td>Lisa</td>
<td>8</td>
</tr>
<tr>
<td>456</td>
<td>Ralph</td>
<td>8</td>
</tr>
</tbody>
</table>

Schema versus instance

- Schema (metadata)
  - Structure and constraints over data
    - Student (SID integer, name string, age integer, GPA float)
    - Course (CID string, title string)
    - Enroll (SID integer, CID integer)
    - Student.SID is a key, Enroll.SID is a foreign key referencing Student.SID, etc.
  - Changes infrequently
- Instance
  - Actual contents that conform to the schema
    - { <142, Bart, 10, 2.3>, <123, Milhouse, 10, 3.1>, ... }
    - { <CPS 296, Topics in Database Systems>, ... }
    - { <142, CPS 296>, <142, CPS 216>, ... }
  - Changes frequently

Relational algebra

- Core set of operators:
  - Selection, projection, cross product, union, difference, and renaming
- Additional, derived operators:
  - Join, etc.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operator</th>
</tr>
</thead>
</table>

Selection

- Notation: $\sigma_p (R)$
  - $p$ is called a selection condition/predicate
- Output: only rows that satisfy $p$
- Example: Students with GPA higher than 3.0
  $$\sigma_{\text{GPA} > 3.0} (\text{Student})$$

Projection

- Notation: $\pi_L (R)$
  - $L$ is a list of columns in $R$
- Output: only the columns in $L$
  - Duplicate rows are removed
- Example: age distribution of students
  $$\pi_{\text{age}} (\text{Student})$$

Cross product

- Notation: $R \times S$
- Output: for each row $r$ in $R$ and each row $s$ in $S$, output a row $rs$ (concatenation of $r$ and $s$)
- Example: $\text{Student} \times \text{Enroll}$
Derived operator: join
- Notation: $R \bowtie_p S$ (shorthand for $\sigma_p (R \times S)$)
  - $p$ is called a join condition/predicate
- Example: students and CIDs of their courses
  $\text{Student} \bowtie \text{Enroll.SID} = \text{Enroll.SID}$

<table>
<thead>
<tr>
<th>SID</th>
<th>Name</th>
<th>Age</th>
<th>GPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>142</td>
<td>Bart</td>
<td>10</td>
<td>2.3</td>
</tr>
<tr>
<td>123</td>
<td>Milhouse</td>
<td>10</td>
<td>3.1</td>
</tr>
</tbody>
</table>

Union and difference
- Notation: $R \cup S$
  - $R$ and $S$ must have identical schema
- Output:
  - Same schema as $R$ and $S$
  - Contains all rows in $R$ and all rows in $S$, with duplicates eliminated
- Notation: $R \setminus S$
  - $R$ and $S$ must have identical schema
- Output:
  - Same schema as $R$ and $S$
  - Contains all rows in $R$ that are not found in $S$

Renaming
- Notation: $\rho_S (R)$, or $\rho_S (A_1, A_2, \ldots, A_n) (R)$
- Purpose: rename a table and/or its columns
  - No real processing involved
  - Used to avoid confusion caused by identical column names
- Example: all pairs of (different) students
  $\rho_{\text{Student1}} (\text{SID}, \text{name}, \text{age}, \text{GPA})$
  $\rho_{\text{Student2}} (\text{SID2}, \text{name2}, \text{age2}, \text{GPA2})$

Relational algebra example
- Names of students in CPS 296 with 4.0 GPA
  $\sigma_{\text{GPA} = 4.0} (\text{Student.SID} = \text{Enroll.SID})$

SQL
- Pronounced “S-Q-L” or “sequel”
- The query language of every commercial DBMS
- Simplest form: `SELECT A_1, A_2, \ldots, A_n FROM R_1, R_2, \ldots, R_m WHERE condition;`
  - Also called an SPJ (select-project-join) query
  - Equivalent (more or less) to relational algebra query
  - Unlike relational algebra, SQL preserves duplicates by default

SQL example
- Names of students in CPS 296 with 4.0 GPA
  ```sql
  SELECT Student.name
  FROM Student, Enroll
  WHERE Enroll.CID = 'CPS 296'
  AND Enroll.SID = Student.SID
  AND Student.GPA = 4.0;
  ```
More SQL features

SELECT [DISTINCT] list_of_output_columns
FROM list_of_tables
WHERE where_condition
GROUP BY list_of_group_by_columns
HAVING having_condition
ORDER BY list_of_order_by_columns;

Operational semantics
- FROM: take the cross product of list_of_tables
- WHERE: apply \( \sigma \) where_condition
- GROUP BY: group result tuples according to list_of_group_by_columns
- HAVING: apply \( \sigma \) having_condition to the groups
- SELECT: apply \( \pi \) list_of_output_columns (preserve duplicates)
- DISTINCT: eliminate duplicates
- ORDER BY: sort the result by list_of_order_by_columns

SQL example with aggregation
- Find the average GPA for each age group with at least three students

SELECT age, AVG(GPA)
FROM Student
GROUP BY age
HAVING COUNT(*) >= 3;

<table>
<thead>
<tr>
<th>SID</th>
<th>name</th>
<th>age</th>
<th>GPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>142</td>
<td>Bart</td>
<td>10</td>
<td>2.3</td>
</tr>
<tr>
<td>789</td>
<td>Jessica</td>
<td>10</td>
<td>4.2</td>
</tr>
<tr>
<td>456</td>
<td>Ralph</td>
<td>8</td>
<td>2.3</td>
</tr>
<tr>
<td>123</td>
<td>Milhouse</td>
<td>10</td>
<td>3.1</td>
</tr>
<tr>
<td>857</td>
<td>Lisa</td>
<td>8</td>
<td>4.3</td>
</tr>
</tbody>
</table>

 Summary: relational query languages
- Not your general-purpose programming language
  - Not expected to be Turing-complete
  - Not intended to be used for complex calculations
  - Amenable to much optimization
- More declarative than languages for hierarchical and network data models
  - No explicit pointer following
    - Replaced by joins that can be easily reordered
- Next: How do we support relational query languages efficiently?

Access paths
- Store data in ways to speed up queries
  - Heap file: unordered set of records
  - B+-tree index: disk-based balanced search tree with logarithmic lookup and update
  - Linear/extensible hashing: disk-based hash tables that can grow dynamically
  - Bitmap indexes: potentially much more compact
  - And many more...
- One table may have multiple access paths
  - One primary index that stores records directly
  - Multiple secondary indexes that store pointers to records

Query processing methods
- The same query operator can be implemented in many different ways
- Example: \( R \bowtie_{E.A=S.B} S \)
  - Nested-loop join: for each tuple of \( R \), and for each tuple of \( S \), join
  - Index nested-loop join: for each tuple of \( R \), use the index on \( S.B \) to find joining \( S \) tuples
  - Sort-merge join: sort \( R \) by \( R.A \), sort \( S \) by \( S.B \), and merge-join
  - Hash join: partition \( R \) and \( S \) by hashing \( R.A \) and \( S.B \), and join corresponding partitions
  - And many more...

Motivation for query optimization
- The same query can have many different execution plans
- Example: SELECT Student.name
  FROM Student, Enroll
  WHERE Enroll.CID = 'CPS 296'
  AND Enroll.SID = Student.SID
  AND Student.GPA = 4.0;
  - Plan 1: evaluate \( \sigma_{GPA=4.0}(Student) \), for each result \( SID \), find the Enroll tuples with this \( SID \) and check if \( CID \) is CPS 296
  - Plan 2: evaluate \( \sigma_{CID=CPS 296}(Enroll) \), for each result \( SID \), find the Student tuple with this \( SID \) and check if GPA is 4.0
  - Plan 3: evaluate both \( \sigma_{GPA=4.0}(Student) \) and \( \sigma_{CID=CPS 296}(Enroll) \), and join them on \( SID \)
  - Any many more...
Query optimization

• A huge number of possible execution plans
  − With different access methods, join order, join methods, etc.
• Query optimizer’s job
  − Enumerate candidate plans
    • Query rewrite: transform queries or query plans into equivalent ones
  − Estimate costs of plans
    • Use statistics such as histograms
  − Pick a plan with reasonably low cost
    • Dynamic programming
    • Randomized search

Optimizing for I/O

<table>
<thead>
<tr>
<th>Location</th>
<th>Cycles</th>
<th>Location</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>1</td>
<td>My head</td>
<td>1 min.</td>
</tr>
<tr>
<td>Memory</td>
<td>100</td>
<td>Washington D.C.</td>
<td>1.5 hr.</td>
</tr>
<tr>
<td>Disk</td>
<td>10^6</td>
<td>Pluto</td>
<td>2 yr.</td>
</tr>
</tbody>
</table>

(source: AlphaSort paper, 1995)

➢ I/O costs dominate database operations
  − DBMS typically optimizes the number of I/O’s
• Example: Which of the following is a more efficient way to process `SELECT * FROM R ORDER BY R.A;`?
  − Use an available secondary B+-tree index on R.A: follow leaf pointers, which are already ordered by R.A
  − Just sort the table