CS 232A: Database System Principles

Introduction: Prerequisites checklist & Course Overview

Introduction

- (Prereq) Applications' View of a Relational Database Management System (RDBMS)
- The Big Picture of UCSD's DB program
- (Prereq) Relational Model Quick Overview
- (Prereq) SQL Quick Overview

Applications' View of a Relational Database Management (RDBMS) System

- Applications: ...........
- Persistent data structure
  - Large volume of data
  - "Independent" from processes using the data
- SQL high-level programming interface for access & modification
  - Automatically optimized
- Transaction management (ACID)
  - Atomicity: all or none happens, despite failures & errors
  - Concurrency
  - Isolation: appearance of "one at a time"
  - Durability: recovery from failures and other errors
CSE232A and the rest of UCSD’s database course program

- **CSE132A**: Basics of relational database systems
  - Application view orientation
  - Basics on algebra, query processing
- **CSE132B**: Application-oriented project course
  - How to design and use in applications complex databases
  - Active database aspects and materialized views
  - JDBC issues
- **CSE135**: Online Analytics Applications
  - Data cubes
  - Live analytics dashboards
  - Application server aspects pertaining to JDBC

CSE232A and the rest of UCSD’s database course program

- **CSE232** is about how databases work internally
  - rather than how to make databases for applications
  - yet, knowing internals makes you a master database programmer
- **CSE233**: Database Theory
  - Theory of query languages
  - Deductive and Object-Oriented databases
- **CSE232B**: Advanced Database Systems
  - Non-conventional database systems, such as
    - mediators & distributed query processing
    - object-oriented and XML databases
    - Deductive databases and recursive query processing

Data Structure: Relational Model

- Relational databases: Schema + Data
- **Schema** (also called scheme):
  - collection of tables (also called relations)
  - each table has a set of attributes
  - no repeating relation names, no repeating attributes in one table
- **Data** (also called instance):
  - set of tuples
  - tuples have one value for each attribute of the table they belong

<table>
<thead>
<tr>
<th>Title</th>
<th>Director</th>
<th>Actor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wild</td>
<td>Lynch</td>
<td>Winger</td>
</tr>
<tr>
<td>Sky</td>
<td>Berto</td>
<td>Winger</td>
</tr>
<tr>
<td>Reds</td>
<td>Brando</td>
<td></td>
</tr>
<tr>
<td>Tango</td>
<td>Berto</td>
<td>Brando</td>
</tr>
<tr>
<td>Tango</td>
<td>Berto</td>
<td>Winger</td>
</tr>
<tr>
<td>Tango</td>
<td>Berto</td>
<td>Snyder</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Odeon</td>
</tr>
<tr>
<td>Forum</td>
</tr>
<tr>
<td>Forum</td>
</tr>
</tbody>
</table>
Relational Model: Primary and Foreign Keys

• "Theater is primary key of Schedule" means its value is unique in Schedule.Theater

• "Title of Schedule references Movie.Title" means every Title value of Schedule also appears as Movie.Title

• If attribute R.A references primary key S.B then we say that "R.A is a foreign key that references S.B"
  – Most common reference case
  – See NorthWind

<table>
<thead>
<tr>
<th>Movie</th>
<th>Title</th>
<th>Director</th>
<th>Actor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wild</td>
<td>Lynch</td>
<td>Winger</td>
<td></td>
</tr>
<tr>
<td>Sky</td>
<td>Berto</td>
<td>Winger</td>
<td></td>
</tr>
<tr>
<td>Reds</td>
<td>Beatty</td>
<td>Beatty</td>
<td></td>
</tr>
<tr>
<td>Tango</td>
<td>Berto</td>
<td>Brando</td>
<td></td>
</tr>
<tr>
<td>Tango</td>
<td>Berto</td>
<td>Winger</td>
<td></td>
</tr>
<tr>
<td>Tango</td>
<td>Berto</td>
<td>Snyder</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Theater</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Odeon</td>
<td>Wild</td>
<td></td>
</tr>
<tr>
<td>Forum</td>
<td>Reds</td>
<td></td>
</tr>
<tr>
<td>Forum</td>
<td>Sky</td>
<td></td>
</tr>
</tbody>
</table>

Lack of conventional primary, foreign keys and violation of normalization rules makes this a practically unlikely schema

Programming Interface:
JDBC/ODBC

• How client opens connection with server
• How access & modification commands are issued

Access (Query) & Modification Language: SQL

• SQL
  – used by the database application
  – declarative: we only describe what we want to retrieve
  – based on tuple relational calculus
    • Important in logic-based optimizations
• The result of a query is always a table
• Internal Equivalent of SQL: Relational Algebra
  – used internally by the database system
  – procedural: we describe how we retrieve
    • Important in query processing and optimization
  – often useful in explaining the semantics of SQL in an indirect way
  – Confusing point: Set (in theory) vs Bag (in practice) semantics
Basic Relational Algebra
Operators (Set)

- **Selection (σ)**
  - \( \sigma_c R \) selects tuples of the argument relation \( R \) that satisfy the condition \( c \).
  - The condition \( c \) consists of atomic predicates of the form:
    - \( \text{attr} = \text{value} \) (\( \text{attr} \) is attribute of \( R \))
    - \( \text{attr}1 = \text{attr}2 \)
    - other operators possible (e.g., \( >, <, =, \neq, \text{LIKE} \))
  - Bigger conditions constructed by conjunctions (AND) and disjunctions (OR) of atomic predicates

- **Projection (π)**
  - \( \pi_{\text{attr}1, \ldots, \text{attr}N} R \) returns a table that has only the attributes \( \text{attr}1, \ldots, \text{attr}N \) of \( R \).
  - Set version: no duplicate tuples in the result (notice the example has only one \( (\text{Tango, Berto}) \) tuple).
  - Bag version: allows duplicates

- **Cartesian Product (x)**
  - the schema of the result has all attributes of both \( R \) and \( S \).
  - for every pair of tuples \( r \) from \( R \) and \( s \) from \( S \) there is a result tuple that consists of \( r \) and \( s \).
  - if both \( R \) and \( S \) have an attribute \( A \) then rename to \( R.A \) and \( S.A \)

### SQL Queries: The Basic From

- **Basic form**
  - \( \text{SELECT DISTINCT a}_1, \ldots, a_N \text{ FROM R}_1, \ldots, R_M \text{ WHERE condition} \)
  - Equivalent relational algebra expression
  - \( \pi_{a_1, a_2, \ldots, a_N, \text{condition}(R_1 \times \ldots \times R_M)} R \)
  - WHERE clause is optional
  - When more than one relations of the FROM have an attribute named \( A \) we refer to a specific \( A \) attribute as \( <\text{RelationName}>.A \)

**Find titles of currently playing movies**

```sql
SELECT Title
FROM Schedule
```

**Find the titles of all movies by “Berto”**

```sql
SELECT Title
FROM Schedule
WHERE Director="Berto"
```

**Find the titles and the directors of all currently playing movies**

```sql
SELECT Movie.Title, Director
FROM Movie, Schedule
WHERE Movie.Title=Schedule.Title
```
Duplicates and Nulls

- **Duplicate elimination** must be explicitly requested
  - `SELECT DISTINCT ... FROM ... WHERE ...`
- **Null values**
  - All comparisons involving NULL are ½ by definition
  - Simplification: ½ -> false
  - All aggregation operations, except count, ignore NULL values

<table>
<thead>
<tr>
<th>Title</th>
<th>Director</th>
<th>Actor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wild</td>
<td>Lynch</td>
<td>Winger</td>
</tr>
<tr>
<td>Reds</td>
<td>NULL</td>
<td>Beatty</td>
</tr>
<tr>
<td>Tango</td>
<td>Berto</td>
<td>Brando</td>
</tr>
<tr>
<td>Tango</td>
<td>Berto</td>
<td>Winger</td>
</tr>
<tr>
<td>Tango</td>
<td>Berto</td>
<td>NULL</td>
</tr>
</tbody>
</table>

SELECT Title FROM Movie

SELECT DISTINCT Title FROM Movie

SQL Queries: Aliases

- Use the same relation more than once in the FROM clause
- By introducing tuple variables
- Example: find actors who are also directors

```
SELECT t.Actor
FROM Movie t, Movie s
WHERE t.Actor=s.Director
```

Example on Aliases and Long Primary/Foreign Key Join Chains

```
SELECT DISTINCT Customers.ContactName
FROM Customers, Customers AS Customers_1, Orders, Orders AS Orders_1, 
[Order Details], [Order Details] AS [Order Details_1], Products
WHERE Customers.CustomerID=Orders.CustomerID
AND Orders.OrderID=[Order Details].OrderID
AND [Order Details].ProductID=Products.ProductID
AND Products.ProductID=[Order Details_1].ProductID
AND [Order Details_1].OrderID=Orders_1.OrderID
AND Orders_1.CustomerID=Customers_1.CustomerID
AND Customers_1.City="London";
```
SQL Queries: Nesting

- The WHERE clause can contain predicates of the form
  - `attr/value IN <SQL query>`
  - `attr/value NOT IN <SQL query>`
- The predicate is satisfied if the `attr` or `value` appears in the result of the nested `<SQL query>`
- Queries involving nesting but no negation can always be un-nested, unlike queries with nesting and negation

Another Form of the “Long Join” Query

```sql
SELECT DISTINCT Customers.ContactName
FROM Customers
WHERE Customers.CustomerID IN (
  SELECT Orders.CustomerID
  FROM Customers AS Customers_1, Orders, Orders AS Orders_1, [Order Details], [Order Details] AS [Order Details_1], Products
  WHERE Orders.OrderID=[Order Details].OrderID
  AND [Order Details].ProductID=Products.ProductID
  AND Products.ProductID=[Order Details_1].ProductID
  AND [Order Details_1].OrderID=Orders_1.OrderID
  AND Orders_1.CustomerID=Customers_1.CustomerID
  AND Customers_1.City="London"
)
```
**Query Expressing Negation with NOT IN**

Find the contact names of customers who do not have orders of products also ordered by London customers.

```
SELECT DISTINCT Customers.ContactName
FROM Customers
WHERE Customers.CustomerID NOT IN (
  SELECT Orders.CustomerID
  FROM Customers AS Customers_1, Orders, Orders AS Orders_1, [Order Details], [Order Details] AS [Order Details_1], Products
  WHERE Orders.OrderID=[Order Details].OrderID
  AND [Order Details].ProductID=Products.ProductID
  AND Products.ProductID=[Order Details_1].ProductID
  AND [Order Details_1].OrderID=Orders_1.OrderID
  AND Orders_1.CustomerID=Customers_1.CustomerID
  AND Customers_1.City="London"
);}
```

---

**SQL Queries: Aggregation and Grouping**

- There is no relational algebra equivalent for aggregation and grouping
- Aggregate functions: AVG, COUNT, MIN, MAX, SUM, and recently user defined functions as well
- Group-by

<table>
<thead>
<tr>
<th>Name</th>
<th>Dept</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Joe</td>
<td>Toys</td>
<td>45</td>
</tr>
<tr>
<td>Nick</td>
<td>PCs</td>
<td>50</td>
</tr>
<tr>
<td>Jim</td>
<td>Toys</td>
<td>35</td>
</tr>
<tr>
<td>Jack</td>
<td>PCs</td>
<td>40</td>
</tr>
</tbody>
</table>

Find the average salary for each department.

```
SELECT Dept, Avg(Salary) AS AvgSal
FROM Employee
GROUP-BY Dept
```

<table>
<thead>
<tr>
<th>Dept</th>
<th>AvgSal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toys</td>
<td>40</td>
</tr>
<tr>
<td>PCs</td>
<td>45</td>
</tr>
</tbody>
</table>

**SQL Grouping: Conditions that Apply on Groups**

- **HAVING** clause

Find the average salary of for each department that has more than 1 employee.

```
SELECT Dept, Avg(Salary) AS AvgSal
FROM Employee
GROUP-BY Dept
HAVING COUNT(Name)>1
```
SQL as a Data Manipulation Language:

Insertions

- inserting tuples
  - `INSERT INTO R VALUES (v1, ..., vk)`;
- some values may be left NULL
- use results of queries for insertion
  - `INSERT INTO R SELECT ... FROM ... WHERE`  
    - `INSERT INTO Movie
      VALUES ("Brave", "Gibson", "Gibson");`
    - `INSERT INTO Movie(Title,Director)
      VALUES ("Brave", "Gibson");`
    - `INSERT INTO EuroMovie
      SELECT * FROM Movie
      WHERE Director = "Berto";`

SQL as a Data Manipulation Language:

Updates and Deletions

- **Deletion** basic form:
  delete every tuple that satisfies `<cond>`
  - `DELETE FROM R WHERE <cond>`
- **Update** basic form:
  update every tuple that satisfies `<cond>` in the way specified by the SET clause
  - `UPDATE R
    SET A1=<exp1>, ..., Ak=<expk>
    WHERE <cond>`
    - `DELETE FROM Movie
      WHERE Title NOT IN (SELECT Title
      FROM Schedule)`
    - `UPDATE Movie
      SET Director="Bertoluci"
      WHERE Director="Berto"`
    - `UPDATE Employee
      SET Salary = 1.1 * Salary
      WHERE Dept = "Toys"`
    - `Increase by 10% the salary of the employee with the highest salary`

Transaction Management

- **Transaction**: Collection of actions that maintain the consistency of the database if ran to completion & isolated
- **Goal**: Guarantee integrity and consistency of data despite
  - Concurrency
  - Failures
- **Concurrency Control**
- **Recovery**
Example Concurrency & Failure Problems

- Consider the "John & Mary" checking & savings account
  - C: checking account balance
  - S: savings' account balance
- Check-to-Savings transfer transaction moves $X from C to S
  - If it runs in the system alone and to completion the total sum of C and S stays the same

```
C2S(X=100)
Read(C);
C:=C-100
Write(C)
Read(S)
S:=S+100
Write(S)
```

Example Failure Problem & Recovery Module’s Goal

- Database is in inconsistent state after machine restarts
- It is not the developer’s problem to account for crashes
- Recovery module guarantees that all or none of transaction happens and its effects become “durable”

```
C2S(X=100)
Read(C);
C:=C-100
Write(C)
CPU HALTS
Read(S)
S:=S+100
Write(S)
```

Example Concurrency Problem & Concurrency Control Module’s Goals

- If multiple transactions run in sequence the resulting database is consistent
- Serial schedules
  - De facto correct

```
Serial Schedule
Read(C);
C:=C+100
Write(C)
Read(S)
S:=S-100
Write(S)
Read(C)
C:=C+50
Write(C)
Read(S)
S:=S-50
Write(S)
```
Example Concurrency Problem & Concurrency Control Module’s Goals

Good Schedule w/ Concurrency
Read(C); C:=C+100
Write(C) Read(C)
C:=C+50 Write(C)
Read(S)
S:=S-100
Write(S) Read(S)
S:=S-50
Write(S)

- Databases allow transactions to run in parallel

Example Concurrency Problem & Concurrency Control Module’s Goals

Bad Schedule w/ Concurrency
Read(C); C:=C+100
Write(C) Read(C)
C:=C+50 Write(C)
Read(S)
S:=S-100
Write(S) Read(S)
S:=S-50
Write(S)

- “Bad” interleaved schedules may leave database in inconsistent state
- Developer should not have to account for parallelism
- Concurrency control module guarantees serializability
  - only schedules equivalent to serial ones happen

Course Topics

- Hardware aspects (very brief)
- Physical Organization Structure (very brief)
  Records in blocks, dictionary, buffer management, ...
- Indexing
  B-Trees, hashing, ...
- Query Processing
  rewriting, physical operators, cost-based optimization, semantic optimization ...
- Crash Recovery
Course Topics

- Concurrency Control
  Correctness, locks, deadlocks...
- Materialized views
  Incremental view maintenance, answering queries using views
- Federated databases
  Distributed query optimization
- Parallel query processing
- Column databases

Database System Architecture

Query Processing

SQL query

Parser

Relational algebra

Query Rewriter and Optimizer

View definitions

Statistics & Catalogs & System Data

Execution Engine

Buffer Manager

Data + Indexes

Transaction Management

Calls from Transactions (read, write)

Transaction Manager

Concurrency Controller

Lock Table

Recovery Manager

Log