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

- Why applications use DBs?
- Persistent data structure
  - "Independent" from processes using the data
  - Also, large volume of data
- 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

Java-based Web Application

RDBMS Client

JDBC/ODBC

SQL commands

RDBMS Server

Relational Database

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):
  - Bag (set in 132A) of tuples
  - tuples have one value for each attribute of the table they belong

<table>
<thead>
<tr>
<th>Movie</th>
<th>Title</th>
<th>Director</th>
<th>Studio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wild</td>
<td>Lynch</td>
<td>MGM</td>
<td></td>
</tr>
<tr>
<td>Sky</td>
<td>Berto</td>
<td>MGM</td>
<td></td>
</tr>
<tr>
<td>Reds</td>
<td>Beatty</td>
<td>Disney</td>
<td></td>
</tr>
<tr>
<td>Tango</td>
<td>Berto</td>
<td>Disney</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>

Relational Model: Primary and Foreign Keys

- "Title is key of Movie" means every title value is unique in column Movie.Title
- Primary key – Movie.ID
  - typically autogenerated
- "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
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 user
  – declarative: we only describe what we want to retrieve
  – based on tuple relational calculus
• The result of a query is always a table (regardless of the query language used)
• Internal Equivalent of SQL: Relational Algebra
  – used internally by the database system
  – procedural (operational): we describe how we retrieve
  – often useful in explaining the semantics of SQL in an indirect way

```
σ
```

Basic Relational Algebra Operators

• Selection (σ)
  – σR (condition) selects tuples of the argument relation R that satisfy the condition c.
  – The condition c consists of atomic predicates of the form
    • attr = value (attr is attribute of R)
    • attr1 = attr2
    • other operators possible (e.g., >, <, IN, LIKE)
  – Bigger conditions constructed by conjunctions (AND) and disjunctions (OR) of atomic predicates

Find movies where director="Berto"
σDirector="Berto" Movie

<table>
<thead>
<tr>
<th>ID</th>
<th>Title</th>
<th>Director</th>
<th>Studio</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Sky</td>
<td>Berto</td>
<td>MGM</td>
</tr>
<tr>
<td>4</td>
<td>Tango</td>
<td>Berto</td>
<td>Disney</td>
</tr>
</tbody>
</table>
Basic Relational Algebra Operators
- Projection ($\pi$)
  \(\pi_{attr1, ..., attrN}(R)\) returns a table that has only the attributes attr1, ..., attrN of R
- Cartesian Product ($\times$)
  - 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

<table>
<thead>
<tr>
<th>MovieTitle</th>
<th>Director</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wild</td>
<td>Lynch</td>
</tr>
<tr>
<td>Sky</td>
<td>Berto</td>
</tr>
<tr>
<td>Reds</td>
<td>Beatty</td>
</tr>
<tr>
<td>Tango</td>
<td>Berto</td>
</tr>
</tbody>
</table>

Project the title and director of Movie

<table>
<thead>
<tr>
<th>R</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
</tr>
</tbody>
</table>

\(\pi_{\text{MovieTitle}, \text{Director}}(R)\)

SQL Queries: The Basic From
- Basic form
  SELECT a1, ..., aN
  FROM R1, ..., RM
  WHERE condition
- Equivalent relational algebra expression
  \(\pi_{a1, ..., aN}(\sigma_{\text{condition}}(R1 \times ... \times RM))\)
- 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 <RelationName>.A

Find all theaters
SELECT Theater
FROM Schedule

Find the titles of all movies by “Berto”
SELECT Title
FROM Movie
WHERE Director="Berto"

Find the titles, directors and theaters of all currently playing movies
SELECT Title, Director, Theater
FROM Movie, Schedule
WHERE Movie.ID=Schedule.MID

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

Find the average salary of all employees
SELECT AvgSal=Avg(Salary)
FROM Employee

Find the average salary for each department
SELECT Dept, AvgSal=Avg(Salary)
FROM Employee
GROUP-BY Dept
SQL Grouping: Conditions that Apply on Groups

- **HAVING clause**

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

```sql
SELECT Dept, AvgSal=(Avg(Salary))
FROM Employee
GROUP-BY Dept
HAVING COUNT(Name)>1
```

...and a Few “Tricky” Points: Duplicates and their Elimination

- **Duplicate elimination** must be explicitly requested
  - `SELECT DISTINCT ... FROM ... WHERE ...`

Find the titles and the directors of all currently playing movies

```sql
SELECT Title, Director
FROM Movie, Schedule
WHERE Movie.ID=Schedule.MID
```

```plaintext
<table>
<thead>
<tr>
<th>ID</th>
<th>Title</th>
<th>Director</th>
<th>Studio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Wild</td>
<td>Lynch</td>
<td>MGM</td>
</tr>
<tr>
<td>2</td>
<td>Sky</td>
<td>Berto</td>
<td>MGM</td>
</tr>
<tr>
<td>3</td>
<td>Reds</td>
<td>Beatty</td>
<td>Disney</td>
</tr>
<tr>
<td>4</td>
<td>Tango</td>
<td>Berto</td>
<td>Disney</td>
</tr>
</tbody>
</table>
```

```sql
SELECT DISTINCT Title, Director
FROM Movie, Schedule
WHERE Movie.ID=Schedule.MID
```

...and a Few “Tricky” Points: Null Values

- **Null values**
  - all comparisons involving NULL are false by definition
  - all aggregation operations, except count, ignore NULL values

Find MGM movies

```sql
SELECT *
FROM Movie
WHERE Studio="MGM"
```

Find movies of unknown studios

```sql
SELECT *
FROM Movie
WHERE ISNULL(Studio)
```

```plaintext
<table>
<thead>
<tr>
<th>ID</th>
<th>Title</th>
<th>Director</th>
<th>Studio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Wild</td>
<td>Lynch</td>
<td>MGM</td>
</tr>
<tr>
<td>2</td>
<td>Sky</td>
<td>Berto</td>
<td>MGM</td>
</tr>
<tr>
<td>3</td>
<td>Reds</td>
<td>Beatty</td>
<td>NULL</td>
</tr>
<tr>
<td>4</td>
<td>Tango</td>
<td>Berto</td>
<td>Disney</td>
</tr>
</tbody>
</table>
```

```sql
SELECT *
FROM Movie
WHERE ISNULL(Studio)
```
SQL: More Bells and Whistles ...

- Select all attributes using *
- Pattern matching conditions
  - `<attr>` LIKE `<pattern>`

Retrieve all movie attributes of currently playing movies
```
SELECT Movie.*
FROM Movie, Schedule
WHERE Movie.ID=Schedule.MID
```

Retrieve all movies where the title starts with “Ta”
```
SELECT *
FROM Movie
WHERE Title LIKE "Ta%"
```

SQL as a Data Manipulation Language:

Insertions

- inserting tuples
  - `INSERT INTO R VALUES (v1,...,vn);`
- some values may be left NULL
- use results of queries for insertion
  - `INSERT INTO R SELECT ... FROM ... WHERE`

Insert into `Movie`
```
INSERT INTO Movie
VALUES ("Brave", "Gibson", "Gibson");
```

Insert into `Movie` as a result of a query
```
INSERT INTO Movie
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 the movies that are not currently playing
```
DELETE FROM Movie
WHERE Title NOT IN (SELECT Title
    FROM Schedule)
```

Change all “Berto” entries to “Bertoluci”
```
UPDATE Movie
    SET Director="Bertoluci"
WHERE Director="Berto"
```

Increase all salaries in the Toys dept by 10%
```
UPDATE Employee
    SET Salary = 1.1 * Salary
WHERE Dept = "Toys"
```

The “rich get richer” exercise:
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”
Example Concurrency Problem & Concurrency Control Module’s Goals

Serial Schedule
Read(C);
C:=C+100
Write(C)
Read(S)
S:=S-100
Write(S)

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

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