Introduction

• Applications’ View of a Relational Database Management System (RDBMS)
• The Big Picture of UCSD’s DB program
• Relational Model Quick Overview
• SQL Quick Overview
• Transaction Management Quick Overview
• What is Hard about building a RDBMS?

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
  - Materialized views, decision support queries
  - JDBC issues
- CSE135: Web application programming
  - Application server aspects pertaining to JDBC

CSE232A and the rest of UCSD’s database course program

- CSE232A variant by Victor and Alin
  - More theory-oriented
  - More time on learning SQL
- CSE233: Database Theory
  - Theory of query languages
  - Deductive and Object-Oriented databases
- CSE232B: Advanced Database Systems
  - The structure and operation of non-conventional database systems, such as
    - data warehouses & OLAP systems
    - 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>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>
</tr>
</thead>
<tbody>
<tr>
<td>Theater</td>
</tr>
<tr>
<td>Odeon</td>
</tr>
<tr>
<td>Forum</td>
</tr>
<tr>
<td>Forum</td>
</tr>
</tbody>
</table>

Review Slide from Victor Vianu's 132A
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

### Movie
<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>Beatty</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>Snyder</td>
</tr>
</tbody>
</table>

### Schedule
<table>
<thead>
<tr>
<th>Theater</th>
<th>Title</th>
<th>Schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forum</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>

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

- **Selection ($\sigma$)**
  - $\sigma_c R$ selects tuples of the argument relation $R$ that satisfy the condition $c$.
  - The condition $c$ consists of atomic predicates of the form:
    - $\text{attr} = \text{value}$ (attr is attribute of $R$)
    - $\text{attr}_1 = \text{attr}_2$
    - Other operators possible (e.g., $>$, $<$, $\leq$, $\geq$, $\neq$, LIKE)
  - Bigger conditions constructed by conjunctions (AND) and disjunctions (OR) of atomic predicates.

#### Examples

<table>
<thead>
<tr>
<th>Title</th>
<th>Director</th>
<th>Actor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sky</td>
<td>Berto</td>
<td>Winger</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>

- $\sigma_{\text{Director}=\text{Actor}} R$

<table>
<thead>
<tr>
<th>Title</th>
<th>Director</th>
<th>Actor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sky</td>
<td>Berto</td>
<td>Winger</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>

### Basic Relational Algebra

#### Operators

- **Projection ($\pi$)**
  - $\pi_{\text{attr}_1, \ldots, \text{attr}_n} R$ 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 (Tango, Berto) tuple).
  - Bag version: allows duplicates.

- **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 $S.A$.

#### Examples

<table>
<thead>
<tr>
<th>$R$</th>
<th>$S$</th>
<th>$R \times S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0 a</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>1 b</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>2 c d</td>
</tr>
</tbody>
</table>

### Basic Relational Algebra

#### Operations

- **Rename ($\rho$)**
  - $\rho_{\text{attr}_A \rightarrow \text{attr}_B} R$ renames attribute $A$ of relation $R$ into $B$.
  - $\rho_{\text{attr}} R$ renames relation $R$ into $S$.

- **Union ($U$)**
  - Applies to two tables $R$ and $S$ with same schema.
  - $R U S$ is the set/bag of tuples that are in $R$ or $S$ or both.

- **Difference ($-$)**
  - Applies to two tables $R$ and $S$ with same schema.
  - $R - S$ is the set of tuples in $R$ but not in $S$.

#### Examples

- Find all people, i.e., actors and directors of the table Movie
  - $\pi_{\text{people}} \rho_{\text{Actor} \rightarrow \text{people.Movie}} U \pi_{\text{people}} \rho_{\text{Director} \rightarrow \text{people.Movie}}$

- Find all directors who are not actors
  - $\pi_{\text{director}} \rho_{\text{Actor} \rightarrow \text{director.Movie}}$
SQL Queries: The Basic From

- Basic form
  \[
  \text{SELECT } a_1, \ldots, a_N \\
  \text{FROM } R_1, \ldots, R_M \\
  \text{WHERE condition}
  \]
- Equivalent relational algebra expression
  \[
  \pi_{a_1, \ldots, a_N} \sigma_{\text{condition}}(R_1 \times \ldots \times R_M)
  \]
- 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
  - \( \text{SELECT DISTINCT ... FROM ... WHERE ...} \)
- Null values
  - all comparisons involving NULL are \textit{false} by definition
  - all aggregation operations, except \textit{count}, ignore NULL values

| Title | Title
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Tango</td>
<td>Tango</td>
</tr>
<tr>
<td>Tango</td>
<td></td>
</tr>
</tbody>
</table>

<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>Reda</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>

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

```sql
SELECT t.Actor \\
FROM Movie t, Movie s \\
WHERE t.Actor=s.Director
```
Example on Aliases and Long Primary/Foreign Key Join Chains

```sql
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 [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

```sql
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"
);
```

Nested Queries: Existential and Universal Quantification

- $A \text{ op ANY } <\text{nested query}>$ is satisfied if there is a value $X$ in the result of the $<\text{nested query}>$ and the condition $A \text{ op } X$ is satisfied — ANY aka SOME
- $A \text{ op ALL } <\text{nested query}>$ is satisfied if for every value $X$ in the result of the $<\text{nested query}>$ the condition $A \text{ op } X$ is satisfied

Find directors of currently playing movies
```sql
SELECT Director
FROM Movie
WHERE Title = ANY
    SELECT Title
    FROM Schedule
```

Find the employees with the highest salary
```sql
SELECT Name
FROM Employee
WHERE Salary >= ALL
    SELECT Salary
    FROM Employee
```
SQL: Union, Intersection, Difference

- **Union**
  - `<SQL query 1>` UNION `<SQL query 2>`
- **Intersection**
  - `<SQL query 1>` INTERSECT `<SQL query 2>`
- **Difference**
  - `<SQL query 1>` MINUS `<SQL query 2>`

Find all actors or directors
(Select Actor FROM Movie)
UNION
(SELECT Director FROM Movie)

Find all actors who are not directors
(SELECT Actor FROM Movie)
MINUS
(SELECT Director FROM Movie)

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 Avg(Salary) AS AvgSal
FROM Employee

Find the average salary for each department
SELECT Dept, Avg(Salary) AS AvgSal
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
SELECT Dept, Avg(Salary) AS AvgSal
FROM Employee
GROUP-BY Dept
HAVING COUNT(Name)>1
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.Title=Schedule.Title

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
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 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

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

- 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

- Databases allow transactions to run in parallel

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)

Example Concurrency Problem & Concurrency Control Module’s Goals

- “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

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

• Applications’ View of a Relational Database Management System (RDBMS)
• The Big Picture of UCSD’s DB program
• Relational Model Quick Overview
• SQL Quick Overview
• Transaction Management Quick Overview
  What is Hard about building a RDBMS?

Isn’t Implementing a Database System Simple?

Introducing the MEGATRON 3000 Database Management System

• The latest from Megatron Labs
• Incorporates latest relational technology
• UNIX compatible
• Lightweight & cheap!

Megatron 3000 Implementation Details

• Relations stored in files (ASCII)
  – e.g., relation Movie is in /usr/db/Movie
• Directory file (ASCII) in /usr/db/directory

| Movie# | Title# | STR# | Director# | Actor# | STR# | ...
|--------|--------|------|-----------|--------|------| ...
| Schedule# | Theater# | STR# | Title# | STR# | ...
| ... |
Megatron 3000
Sample Sessions

% MEGATRON3000
   Welcome to MEGATRON 3000!
&
...
& quit
%

Megatron 3000
Sample Sessions

& select *
   from Movie #
Title       Director  Actor
Wild        Lynch     Winger
Sky         Berto     Winger
Reds        Beatty    Beatty
Tango       Berto     Brando
Tango       Berto     Winger
Tango       Berto     Snyder
&

Megatron 3000
Sample Sessions

& select Theater, Movie.Title
   from Movie, Schedule
   where Movie.Title=Schedule.Title
   AND Actor = "Winger" #
Theater Title
Odeon    Wild
Forum    Sky
&
Megatron 3000
Sample Sessions

& select *
from Movie | LPR #
&

Result sent to LPR (printer).

Megatron 3000
Sample Sessions

& select *
from Movie
where Actor = "Winger" | T #
&

New relation T created.

Megatron 3000

• To execute
  select * from Movie where Actor="Winger"
  (1) Read dictionary to get Movie attributes
  (2) Read Movie file, for each line:
      (a) Check condition
      (b) If OK, display
Megatron 3000

- To execute
  
  ```
  select Theater, Movie.Title
  from Movie, Schedule
  where Movie.Title=Schedule.Title
  AND optional condition
  ```

  1. Read dictionary to get Movie, Schedule attributes
  2. Read Movie file, for each line:
     a. Read Schedule file, for each line:
        i. Create join tuple
        ii. Check condition
        iii. Display if OK

What’s wrong with the Megatron 3000 DBMS?

- Tuple layout on disk
  e.g., - Change string from 'Cat' to 'Cats' and we have to rewrite file
        - ASCII storage is expensive
        - Deletions are expensive

What’s wrong with the Megatron 3000 DBMS?

- Search expensive; no indexes
  e.g., - Cannot find tuple with given key quickly
        - Always have to read full relation
What’s wrong with the Megatron 3000 DBMS?

• Brute force query processing
e.g.,
   select Theater, Movie.Title
   from Movie, Schedule
   where Movie.Title=Schedule.Title
   AND optional condition
• Much better if
  – (when selective) Use index to select tuples that satisfy condition
  – Use index to find theaters where qualified titles play
• Or (when optional condition not selective)
  – Sort both relations on title and merge
• Exploit caches and buffers

What’s wrong with the Megatron 3000 DBMS?

• Concurrency control & recovery
• No reliability
e.g., - Can lose data
  - Can leave operations half done

What’s wrong with the Megatron 3000 DBMS?

• Security
• Interoperation with other systems
• Consistency enforcement
Course Topics

- Hardware aspects
- Physical Organization Structure
  - Records in blocks, dictionary, buffer management,...
- Indexing
  - B-Trees, hashing,...
- Indexing
  - B-Trees, hashing,...
- Query Processing
  - Hashing, physical operators, cost-based optimization, semantic optimization,...
- Crash Recovery
  - Failures, stable storage,...

Course Topics

- Concurrency Control
  - Correctness, locks, deadlocks,...
- Miscellaneous topics, as time permits
  - Distributed databases, warehousing, etc

Database System Architecture

Query Processing
- SQL query
  - Parser
  - Relational algebra
  - Query Rewriter and Optimizer
  - View definitions
  - Statistics & Catalogs & System Data
  - Query execution plan
  - Execution Engine
  - Data + Indexes

Transaction Management
- Calls from Transactions (read,write)
  - Transaction Manager
  - Concurrency Controller
  - Recovery Manager
  - Log
The Journey of a Query (Example)

```
SELECT t.Actor
FROM Movie t, s
WHERE t.Title=s.Title
AND s.Actor="Winger"
```

The Journey of a Query (cont’d)

What is the algebra used?
What are the rules for transforming algebraic expressions?

What algorithms can be used for each operator (e.g., join, aggregation), i.e., how does the logical algebra turn into a physical one?
How do we evaluate the cost of a possible execution plan?
How do we explore the space of options?

The Journey of a Query (cont’d)

How is the table arranged on the disk?
Are tuples with the same actor value clustered (consecutive)?
What is the exact structure of the index (tree, hash table, …)?

How is the table arranged on the disk?
Are tuples with the same actor value clustered (consecutive)?
What is the exact structure of the index (tree, hash table, …)?