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
- 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
  - JDBC issues
- **CSE135**: Online Analytics Applications
  - Data cubes
  - Live analytics dashboards
  - NoSQL
  - 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
  - allows you to choose the right kind of database
- **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>Movie</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>

<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>

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

<table>
<thead>
<tr>
<th>Movie</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>

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

• **Selection** (σ)
  - \( c \) 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., \( >, <, \leq, \geq, \neq, \text{LIKE} \))
  - Bigger conditions constructed by conjunctions (\( \land \)) and disjunctions (\( \lor \)) of atomic predicates

<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>

Find tuples where director=“Berto”

\( \sigma_{\text{Director} = \text{Berto}} \text{Movie} \)

<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>

Find tuples where director=actor

\( \sigma_{\text{Director} = \text{Actor}} \text{Movie} \)

<table>
<thead>
<tr>
<th>Title</th>
<th>Director</th>
<th>Actor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reds</td>
<td>Beatty</td>
<td>Beatty</td>
</tr>
</tbody>
</table>

Find tuples where director=“Berto” OR director=actor

\( \sigma_{\text{Director} = \text{Berto} \lor \text{Director} = \text{Actor}} \text{Movie} \)

<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>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>

Basic Relational Algebra
Operators

• **Projection** (\( \pi \))
  - \( \pi_{\text{attr}_1, \ldots, \text{attr}_n} \text{R} \) 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

<table>
<thead>
<tr>
<th>Title</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

\( \pi_{\text{Title, Director}} \text{Movie} \)

<table>
<thead>
<tr>
<th>Title</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>

Basic Relational Algebra
Operations

• **Rename** (:)
  - \( \rho \text{A} \rightarrow \text{B} \text{R} \) renames attribute \( A \) of relation \( R \) into \( B \)
  - \( \rho \text{R} \text{S} \) renames relation \( R \) into \( S \)

<table>
<thead>
<tr>
<th>R x S</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R x S</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>c</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4</td>
<td>c</td>
</tr>
</tbody>
</table>

Find all people, ie, actors and directors of the table Movie

\( \rho_{\text{Actor} \rightarrow \text{People, Movie}} \text{Movie} \)

Find all directors who are not actors

\( \rho_{\text{Director} \rightarrow \text{People, Movie}} \text{Movie} \)

Find all people, ie, actors and directors of the table Movie

\( U \rho_{\text{Actor} \rightarrow \text{People, Movie}} \text{Movie} \)

Find all people, ie, actors and directors of the table Movie

\( \rho_{\text{Director} \rightarrow \text{People, Movie}} \text{Movie} \)
SQL Queries: The Basic From

- **Basic form**
  
  \[
  \text{SELECT } a_1, ..., a_N \\
  \text{FROM } R_1, ..., R_M \\
  \text{WHERE condition}
  \]

- **Equivalent relational algebra expression**
  
  \[
  \pi_{a_1, ..., a_N}(\sigma_{\text{condition}}(R_1 \times ... \times R_M))
  \]

- **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 | SELECT Title \\
|----------------------------------------| FROM Schedule |
| Find the titles of all movies by “Berto” | SELECT Title \\
|----------------------------------------| FROM Schedule  \\
| WHERE Director=“Berto” | |
| Find the titles and the directors of all currently playing movies | 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></th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT Title</td>
<td></td>
</tr>
<tr>
<td>FROM Movie</td>
<td></td>
</tr>
<tr>
<td>Tango</td>
<td></td>
</tr>
<tr>
<td>Tango</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT DISTINCT Title</td>
<td></td>
</tr>
<tr>
<td>FROM Movie</td>
<td></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>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>

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"
)
);
```

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

```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"
)
);
```

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>Employee</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 of all employees

```sql
SELECT Avg(Salary) AS AvgSal
FROM Employee
```

Find the average salary for each department

```sql
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
  
  ```sql
  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`

  ```sql
  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>`

  ```sql
  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"

  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”

```
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

<table>
<thead>
<tr>
<th>Serial Schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Read(C);</code></td>
</tr>
<tr>
<td><code>C:=C+100</code></td>
</tr>
<tr>
<td><code>Write(C)</code></td>
</tr>
<tr>
<td><code>Read(S)</code></td>
</tr>
<tr>
<td><code>S:=S-100</code></td>
</tr>
<tr>
<td><code>Write(S)</code></td>
</tr>
</tbody>
</table>

Example Concurrency Problem & Concurrency Control Module’s Goals

- Databases allow transactions to run in parallel

<table>
<thead>
<tr>
<th>Good Schedule w/ Concurrency</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Read(C);</code></td>
</tr>
<tr>
<td><code>C:=C+100</code></td>
</tr>
<tr>
<td><code>Write(C)</code></td>
</tr>
<tr>
<td><code>Read(C)</code></td>
</tr>
<tr>
<td><code>C:=C+50</code></td>
</tr>
<tr>
<td><code>Write(C)</code></td>
</tr>
<tr>
<td><code>Read(S)</code></td>
</tr>
<tr>
<td><code>S:=S-100</code></td>
</tr>
<tr>
<td><code>Write(S)</code></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Bad Schedule w/ Concurrency</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Read(C);</code></td>
</tr>
<tr>
<td><code>C:=C+100</code></td>
</tr>
<tr>
<td><code>Write(C)</code></td>
</tr>
<tr>
<td><code>Read(C)</code></td>
</tr>
<tr>
<td><code>C:=C+50</code></td>
</tr>
<tr>
<td><code>Write(C)</code></td>
</tr>
<tr>
<td><code>Read(S)</code></td>
</tr>
<tr>
<td><code>S:=S-50</code></td>
</tr>
<tr>
<td><code>Write(S)</code></td>
</tr>
<tr>
<td><code>Read(S)</code></td>
</tr>
<tr>
<td><code>S:=S-100</code></td>
</tr>
<tr>
<td><code>Write(S)</code></td>
</tr>
</tbody>
</table>
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
- 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

<table>
<thead>
<tr>
<th>Movie#</th>
<th>Title#</th>
<th>Director#</th>
<th>STR#</th>
<th>Actor#</th>
<th>STR#</th>
<th>…</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Schedule#</td>
<td>Theater#</td>
<td>STR</td>
<td>Title#</td>
<td>STR</td>
<td>…</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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:
  
  ```sql
  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:
     1. Read Schedule file, for each line:
        a. Create join tuple
        b. Check condition
        c. 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.,
  
  ```sql
  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

- Scalability, parallelism

- Smart use of hardware, memory hierarchy
Course Topics

- Hardware aspects (very brief)
- Physical Organization Structure (very brief)
  Records in blocks, dictionary, buffer management, ...
- Indexing
  B-Trees, hashing, ...
- Query Processing
  rewritting, 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 & data services
  Distributed query optimization, capabilities-based rewriting
- Parallel query processing

Database System Architecture

Query Processing

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

Transaction Management

- Calls from Transactions (read, write)
  - Transaction Manager
  - Concurrency Controller
  - Recovery Manager
  - Log