CMSC424: Database Design

Instructor: Amol Deshpande
amol@cs.umd.edu

Databases

- **Data Models**
  - Conceptual representation of the data

- **Data Retrieval**
  - How to ask questions of the database
  - How to answer those questions

- **Data Storage**
  - How/where to store data, how to access it

- **Data Integrity**
  - Manage crashes, concurrency
  - Manage semantic inconsistencies
Outline

- SQL
- Relational Algebra
- Formal Semantics of SQL
- More SQL

SQL Query Examples

- Movie(title, year, length, inColor, studioName, producerC#)
- StarsIn(movieTitle, movieYear, starName)
- MovieStar(name, address, gender, birthdate)
- MovieExec(name, address, cert#, netWorth)
- Studio(name, address, presC#)
SQL Constructs: Data Definition Language

- Data Definition Language
  - Create table, insert into <> values() etc...
SQL Constructs: Single Table Queries

- Types of queries/constructs
  - Many different types of predicates
  - Aggregates
  - Set operations
  - Nested subqueries
  - Finding max tuple
    - Key observation: A tuple has the maximum value if its value equals the maximum value
  - Ranking
    - Key observation: A tuple is ranked 5th or lower by “score” iff the number of tuples with “score” larger than it is less than 5
  - Set comparisons

SQL Constructs: Multi-table Queries

- Key:
  - Do a join to get an appropriate table
  - Use the constructs for single-table queries
- You will get used to doing all at once

- Different types of joins
  - Inner joins/natural joins
    - Only tuples that have matches
  - Outer joins
    - Left/right outer join
    - Full outer join
  - Semi-join, anti-join, theta-join (not SQL operations)
    - Important --- we will discuss them later
Using FROM clause with WHERE clause

```sql
select title, year, me.name as producerName
from movies m, movieexec me
where m.producerC# = me.cert#;
```

“inner join”

```sql
select title, year, me.name as producerName
from movies m inner join movieexec me
on m.producerC# = me.cert#;
```

“natural join”

- Finds the common attributes and does equality join
- Useless on the movies database (common attributes named differently)

```sql
select * from olympics natural join countries
```

Consider the query:

```sql
select title, year, producerC#, count(starName)
from movies, starsIn
where title = starsIn.movieTitle and year = starsIn.movieYear
group by title, year, producerC#
```

You expect:
- A list of movies with the number of stars in them

But:
- What about movies with no stars?
- We would like to get “(movie, year, producerC#, 0)” for them
Outer Joins

- **Left outer join**
  
  ```sql
  select title, year, producerC#, count(starName)
  from movies
  left outer join starsIn
  on title = starsIn.movieTitle and year = starsIn.movieYear
  group by title, year, producerC#
  ```

  - All tuples from “movies” that have no matches in starsIn are included with NULLs
    - So if a tuple (m1, 1990) has no match in starsIn, we get (m1, 1990, NULL) in the result
  - The count(starName) works correctly then.
    - Note: count(*) would not work correctly.

- **Extensions: Right outer join, and Full outer join**

---

Set Comparisons

- **Set comparisons in SQL; why?**
  
  - > some: The subquery may not return something that is ordinal (e.g. it may return strings)
  - > all: seems it would be useful to simplify some queries that involve finding maximums among groups
    - Won’t have to create temporary tables
    - E.g....
Example

- Find the producer with max average length movie
  - We used:
    ```
    create table temp as
    select name, avg(length) as aveL
    from movieExec me, movies m
    where me.cert# = m.producer#
    group by name;
    select name
    from temp
    where aveL = (select max(aveL) from temp);
    ```
  - Alternative
    ```
    select name
    from movieExec me, movie m
    where me.cert# = m.producer#
    group by name
    having avg(length) >= all (select avg(length)
    from movie m
    group by producer#)
    ```
    - Note: No need to do the join here;
    - Grouping by producer# is identical to grouping by name

SQL Query Examples

- Find movie titles that appear more than once
  - Option 1: Use group by/having clause
    ```
    select title from movies group by title having count(year) > 1;
    ```
  - Option 2: Use set operations
    ```
    select title from movies old where year <
    any (select year from movie where title = old.title)
    ```
SQL Query Examples

Find actors/actresses 3 degrees away from H. Ford

An actor is one degree away from H. Ford if they starred in a movie together

First create a costars table

create table costars as
select distinct s1.starName as star1, s2.starname as star2
from starsIn s1, starsIn s2
where s1.movieTitle = s2.movieTitle and s1.movieYear = s2.movieYear;

Join multiple of these together appropriately

select c3.star2
from costars c1, costars c2, costars c3
where c1.star1 = “H. Ford” and c1.star2 = c2.star1 and c2.star2 = c3.star1

Different answers depending on whether (H. Ford, H.Ford) exists in this table (it does in the above table)
Outline

- SQL
- Relational Algebra
- Formal Semantics of SQL
- More SQL

Relational Algebra

- Procedural language

- Six basic operators
  - select
  - project
  - union
  - set difference
  - Cartesian product
  - rename

- The operators take one or more relations as inputs and give a new relation as a result.
### Select Operation

Relation `r`:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>α</td>
<td>α</td>
<td>1</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>α</td>
<td>β</td>
<td>5</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>β</td>
<td>β</td>
<td>12</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>β</td>
<td>β</td>
<td>23</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

\[ \sigma_{A=B \land D > 5} (r) \]

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>α</td>
<td>α</td>
<td>1</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>β</td>
<td>β</td>
<td>23</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

**Unfortunate naming confusion**

**SQL Equivalent:**

```sql
select *
from r
where A = B and D > 5
```

### Project

Relation `r`:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>α</td>
<td>α</td>
<td>1</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>α</td>
<td>β</td>
<td>5</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>β</td>
<td>β</td>
<td>12</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>β</td>
<td>β</td>
<td>23</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

\[ \Pi_{A,D} (r) \]

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>α</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>α</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>β</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>β</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

**SQL Equivalent:**

```sql
select distinct A, D
from r
```
**Set Union, Difference**

Relation r, s

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>α</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>α</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>β</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>r</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>α</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>β</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>s</td>
<td></td>
</tr>
</tbody>
</table>

r ∪ s: 

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>α</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>α</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>β</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

r − s:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>α</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>β</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Must be compatible schemas

SQL Equivalent:

```
select * from r
union/except/intersect
select * from s;
```

What about intersection?

Can be derived

r ∩ s = r − (r − s);

This is one case where duplicates are removed.

**Cartesian Product**

Relation r, s

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>α</td>
<td>1</td>
<td></td>
<td>10</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>β</td>
<td>2</td>
<td></td>
<td>20</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>γ</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>r</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>α</td>
<td>1</td>
<td>10</td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>β</td>
<td>2</td>
<td>20</td>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>γ</td>
<td>3</td>
<td>10</td>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>s</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

r × s:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>α</td>
<td>1</td>
<td>10</td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>β</td>
<td>2</td>
<td>10</td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>β</td>
<td>2</td>
<td>20</td>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>β</td>
<td>2</td>
<td>10</td>
<td>a</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SQL Equivalent:

```
select distinct * from r, s
```

Does not remove duplicates.
Rename Operation

- Allows us to name, and therefore to refer to, the results of relational-algebra expressions.
- Allows us to refer to a relation by more than one name.
  Example:
  \[ \rho_X(E) \]
  returns the expression \( E \) under the name \( X \)

If a relational-algebra expression \( E \) has arity \( n \), then
\[ \rho_{X(A_1, A_2, ..., A_n)}(E) \]
returns the result of expression \( E \) under the name \( X \), and with the attributes renamed to \( A_1, A_2, ..., A_n \).

Relational Algebra

- Those are the basic operations

- What about SQL Joins?
  - Compose multiple operators together
    \[ \sigma_{A=C}(r \times s) \]

- Additional Operations
  - Set intersection
  - Natural join
  - Division
  - Assignment
Additional Operators

- Set intersection (\(\cap\))
  - \(r \cap s = r - (r - s)\)
  - SQL Equivalent: intersect

- Assignment (\(\leftarrow\))
  - A convenient way to right complex RA expressions
  - Essentially for creating "temporary" relations
  - \(temp1 \leftarrow \prod_{R-S}(r)\)
  - SQL Equivalent: “create table as…”

Additional Operators: Joins

- Natural join (\(\bowtie\))
  - A Cartesian product with equality condition on common attributes
  - Example:
    - if \(r\) has schema \(R(A, B, C, D)\), and if \(s\) has schema \(S(E, B, D)\)
    - Common attributes: \(B\) and \(D\)
    - Then:
      \[
      r \bowtie s = \prod_{r.A, r.B, r.C, r.D, s.E} (\sigma_{r.B = s.B \land r.D = s.D} (r \times s))
      \]
  - SQL Equivalent:
    - select \(r.A, r.B, r.C, r.D, s.E\) from \(r, s\) where \(r.B = s.B\) and \(r.D = s.D\)
    - OR
    - select * from \(r\) natural join \(s\)
Additional Operators: Joins

- **Equi-join**
  - A join that only has equality conditions

- **Theta-join (⋈θ)**
  - \( r ⋈θ s = \sigma_θ(r \times s) \)

- **Left outer join (⟕)**
  - Say \( r(A, B), s(B, C) \)
  - We need to somehow find the tuples in \( r \) that have no match in \( s \)
  - Consider: \( (r - \pi_{r.A, r.B}(r ⋈ s)) \)
  - We are done:
    \[
    (r ⋈ s) \cup \rho_{\text{temp}(A, B, C)} \left( (r - \pi_{r.A, r.B}(r ⋈ s)) \times \{(\text{NULL})\} \right)
    \]

### Additional Operators: Join Variations

**Tables:** \( r(A, B), s(B, C) \)

<table>
<thead>
<tr>
<th>name</th>
<th>Symbol</th>
<th>SQL Equivalent</th>
<th>RA expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>cross product</td>
<td>×</td>
<td>select * from r, s;</td>
<td>( r \times s )</td>
</tr>
<tr>
<td>natural join</td>
<td>⋈</td>
<td>natural join</td>
<td>( \pi_{r.A, r.B, s.C}(r \times s) )</td>
</tr>
<tr>
<td>theta join</td>
<td>⋈θ</td>
<td>from .. where θ;</td>
<td>( \sigma_θ(r \times s) )</td>
</tr>
<tr>
<td>equi-join</td>
<td>⋈θ</td>
<td>(theta must be equality)</td>
<td></td>
</tr>
<tr>
<td>left outer join</td>
<td>⋈</td>
<td>left outer join (with &quot;on&quot;)</td>
<td>(see previous slide)</td>
</tr>
<tr>
<td>full outer join</td>
<td>≡</td>
<td>full outer join (with &quot;on&quot;)</td>
<td>–</td>
</tr>
<tr>
<td>(left) semijoin</td>
<td>⋉</td>
<td>none</td>
<td>( \pi_{r.A, r.B}(r ⋈ s) )</td>
</tr>
<tr>
<td>(left) antijoin</td>
<td>⊲</td>
<td>none</td>
<td>( r - \pi_{r.A, r.B}(r ⋈ s) )</td>
</tr>
</tbody>
</table>
Additional Operators: Division

- Suitable for queries that have “for all”
  - \( r \div s \)
- Think of it as “opposite of Cartesian product”
  - \( r \div s = t \iff t \times s \subseteq r \)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\alpha)</td>
<td>1</td>
<td>(\alpha)</td>
<td>10</td>
<td>a</td>
</tr>
<tr>
<td>(\alpha)</td>
<td>1</td>
<td>(\beta)</td>
<td>10</td>
<td>a</td>
</tr>
<tr>
<td>(\alpha)</td>
<td>1</td>
<td>(\beta)</td>
<td>20</td>
<td>b</td>
</tr>
<tr>
<td>(\beta)</td>
<td>2</td>
<td>(\alpha)</td>
<td>10</td>
<td>a</td>
</tr>
<tr>
<td>(\beta)</td>
<td>2</td>
<td>(\beta)</td>
<td>10</td>
<td>a</td>
</tr>
<tr>
<td>(\beta)</td>
<td>2</td>
<td>(\gamma)</td>
<td>10</td>
<td>b</td>
</tr>
</tbody>
</table>

\[ \frac{A \quad B}{\alpha \quad 1 \quad \beta \quad 2} = \frac{C \quad D \quad E}{\alpha \quad 10 \quad a \quad \beta \quad 10 \quad a \quad \beta \quad 20 \quad b \quad \gamma \quad 10 \quad b} \]

Relational Algebra Examples

- \(\text{branch} \ (\text{branch-name, branch-city, assets})\)
- \(\text{customer} \ (\text{customer-name, customer-street, ...})\)
- \(\text{account} \ (\text{account-number, branch-name, balance})\)
- \(\text{loan} \ (\text{loan-number, branch-name, amount})\)
- \(\text{depositor} \ (\text{customer-name, account-number})\)
- \(\text{borrower} \ (\text{customer-name, loan-number})\)
Find all loans of over $1200

\[ \sigma_{\text{amount} > 1200} (\text{loan}) \]

Find the loan number for each loan of an amount greater than $1200

\[ \Pi_{\text{loan-number}} (\sigma_{\text{amount} > 1200} (\text{loan})) \]

Find the names of all customers who have a loan, an account, or both, from the bank

\[ \Pi_{\text{customer-name}} (\text{borrower}) \cup \Pi_{\text{customer-name}} (\text{depositor}) \]
Relational Algebra Examples

Find the names of all customers who have a loan and an account at bank.
\[ \pi_{\text{customer-name}}(\text{borrower}) \cap \pi_{\text{customer-name}}(\text{depositor}) \]

Find the names of all customers who have a loan at the Perryridge branch.
\[ \pi_{\text{customer-name}}(\sigma_{\text{branch-name} = \text{Perryridge}}(\sigma_{\text{borrower.loan-number} = \text{loan.loan-number}}(\text{borrower} \times \text{loan}))) \]

Find the largest account balance
1. Rename the account relation a \( d \)
\[ \pi_{\text{balance}(\text{account})} - \pi_{\text{account.balance}}(\sigma_{\text{account.balance} < \text{d.balance}}(\text{account} \times \rho_{d}(\text{account}))) \]

Outline

- SQL
- Relational Algebra
- Formal Semantics of SQL
- More SQL
Duplicates

- By definition, relations are sets
  - So → No duplicates allowed
- Problem:
  - Not practical to remove duplicates after every operation
  - Why?
- So...
  - SQL by default does not remove duplicates
- SQL follows bag semantics, not set semantics
  - Implicitly we keep count of number of copies of each tuple

Formal Semantics of SQL

- RA can only express `SELECT DISTINCT` queries
- To express SQL, must extend RA to a bag algebra
  → Bags (aka: multisets) like sets, but can have duplicates

\[ \{5, 3, 3\} \]
\[ e.g: \text{homes} = \begin{array}{|c|c|} \hline \text{cname} & \text{ccity} \\ \hline \text{Johnson} & \text{Brighton} \\ \text{Smith} & \text{Perry} \\ \text{Johnson} & \text{Brighton} \\ \text{Smith} & \text{R.H.} \\ \hline \end{array} \]

- Next: will define RA*: a bag version of RA
Formal Semantics of SQL: RA*

1. $\sigma^*_{p_1}(r)$: preserves copies in r
   
   e.g: $\sigma^*_{\text{city} = \text{Brighton}} (\text{homes}) = \begin{array}{|c|c|} \hline \text{cname} & \text{ccity} \\ \hline Johnson & Brighton \\ Johnson & Brighton \\ \hline \end{array}$

2. $\pi^*_{A_1, \ldots, A_n}(r)$: no duplicate elimination
   
   e.g: $\pi^*_{\text{cname}} (\text{homes}) = \begin{array}{|c|} \hline \text{cname} \\ \hline Johnson \\ Smith \\ Johnson \\ Smith \\ \hline \end{array}$

3. $r \cup^* s$: additive union
   
   $\begin{array}{|c|c|} \hline A & B \\ \hline 1 & \alpha \\ 1 & \alpha \\ 2 & \beta \\ \hline \end{array} \cup^* \begin{array}{|c|c|} \hline A & B \\ \hline 2 & \beta \\ 3 & \alpha \\ 1 & \alpha \\ \hline \end{array} = \begin{array}{|c|c|} \hline A & B \\ \hline 1 & \alpha \\ 1 & \alpha \\ 2 & \beta \\ 2 & \beta \\ 3 & \alpha \\ 1 & \alpha \\ \hline \end{array}$

4. $r -^* s$: bag difference
   
   e.g: $r -^* s = \begin{array}{|c|c|} \hline A & B \\ \hline 1 & \alpha \\ \hline \end{array}$
   
   $s -^* r = \begin{array}{|c|c|} \hline A & B \\ \hline 3 & \alpha \\ \hline \end{array}$
Formal Semantics of SQL: RA*

5. \( r \times^* s: \) cartesian product

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>+</td>
</tr>
<tr>
<td>1</td>
<td>α</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>α</td>
<td>+</td>
</tr>
<tr>
<td>1</td>
<td>β</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>β</td>
<td>+</td>
</tr>
</tbody>
</table>

Query:

```
SELECT a_1, \ldots, a_n
FROM r_1, \ldots, r_m
WHERE p
```

Semantics:

\[ \pi^*_{A_1, \ldots, A_n} (\sigma^*_{p} (r_1 \times^* \ldots \times^* r_m)) \] (1)

Query:

```
SELECT DISTINCT a_1, \ldots, a_n
FROM r_1, \ldots, r_m
WHERE p
```

Semantics: What is the only operator to change in (1)?

\[ \pi_{A_1, \ldots, A_n} (\sigma^*_{p} (r_1 \times \ldots \times r_m)) \] (2)
Set/Bag Operations Revisited

- **Set Operations**
  - UNION $\equiv U$
  - INTERSECT $\equiv \cap$
  - EXCEPT $\equiv -$

- **Bag Operations**
  - UNION ALL $\equiv U^*$
  - INTERSECT ALL $\equiv \cap^*$
  - EXCEPT ALL $\equiv -^*$

**Duplicate Counting:**

Given $m$ copies of $t$ in $r$, $n$ copies of $t$ in $s$, how many copies of $t$ in:

- $r$ UNION ALL $s$?
  - $A$: $m + n$

- $r$ INTERSECT ALL $s$?
  - $A$: min$(m, n)$

- $r$ EXCEPT ALL $s$?
  - $A$: max$(0, m-n)$

### SQL: Summary

<table>
<thead>
<tr>
<th>Clause</th>
<th>Eval Order</th>
<th>Semantics (RA/RA*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT [(DISTINCT)]</td>
<td>4</td>
<td>$\pi$ (or $\pi^*$)</td>
</tr>
<tr>
<td>FROM</td>
<td>1</td>
<td>$\times^*$</td>
</tr>
<tr>
<td>WHERE</td>
<td>2</td>
<td>$\sigma^*$</td>
</tr>
<tr>
<td>INTO</td>
<td>7</td>
<td>$\leftarrow$</td>
</tr>
<tr>
<td>GROUP BY</td>
<td>3</td>
<td>Extended relational operator $g$</td>
</tr>
<tr>
<td>HAVING</td>
<td>5</td>
<td>$\sigma^*$</td>
</tr>
<tr>
<td>ORDER BY</td>
<td>6</td>
<td>Can’t express: requires ordered sets, bags</td>
</tr>
<tr>
<td>AS</td>
<td>-</td>
<td>$\rho$</td>
</tr>
<tr>
<td>UNION ALL</td>
<td>8</td>
<td>$U^*$</td>
</tr>
<tr>
<td>UNION</td>
<td></td>
<td>$U$</td>
</tr>
<tr>
<td>(similarly intersection, except)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Outline

- SQL
- Relational Algebra
- Formal Semantics of SQL

- More SQL
  - Nulls
  - Updates
  - Views
  - Integrity Constraints
  - Transactions
  - Triggers

Relational Algebra Examples

```
branch
  | branch-name
  | branch-city
  | assets

account
  | account-number
  | branch-name
  | balance

depositor
  | customer-name
  | account-number

customer
  | customer-name
  | customer-street
  | customer-city

loan
  | loan-number
  | branch-name
  | amount

borrower
  | customer-name
  | loan-number
```
Modification of the Database – Deletion

- Delete all account records at the Perryridge branch
  
  ```sql
  delete from account
  where branch-name = 'Perryridge'
  ```

- Delete all accounts at every branch located in Needham city.

  ```sql
  delete from depositor
  where account-number in
  (select account-number
   from branch, account
   where branch-city = 'Needham'
   and branch.branch-name = account.branch-name);

  delete from account
  where branch-name in (select branch-name
                        from branch
                        where branch-city = 'Needham');
  ```

Example Query

- Delete the record of all accounts with balances below the average at the bank.

  ```sql
  delete from account
  where balance < (select avg (balance)
      from account)
  ```

  Problem: as we delete tuples from `deposit`, the average balance changes

  Solution used in SQL:

  1. First, compute `avg` balance and find all tuples to delete
  2. Next, delete all tuples found above (without recomputing `avg` or retesting the tuples)
Modification of the Database – Insertion

- Add a new tuple to account
  ```sql
  insert into account
  values ('A-9732', 'Perryridge', 1200)
  ```

- or equivalently
  ```sql
  insert into account (branch-name, balance, account-number)
  values ('Perryridge', 1200, 'A-9732')
  ```

- Add a new tuple to account with balance set to null
  ```sql
  insert into account
  values ('A-777', 'Perryridge', null)
  ```

Modification of the Database – Updates

- Increase all accounts with balances over $10,000 by 6%, all other accounts receive 5%.
  - Write two update statements:
    ```sql
    update account
    set balance = balance * 1.06
    where balance > 10000;
    ```
    ```sql
    update account
    set balance = balance * 1.05
    where balance <= 10000;
    ```
  - The order is important
  - Can be done better using the `case` statement
Modification of the Database – Updates

- Increase all accounts with balances over $10,000 by 6%, all other accounts receive 5%.
  - Can be done better using the `case` statement
    ```sql
    update account
    set balance =
    case
      when balance > 10000
        then balance * 1.06
      when balance <= 10000
        then balance * 1.05
    end;
    ```

Outline

- SQL
- Relational Algebra
- Formal Semantics of SQL

- More SQL
  - Nulls
  - Updates
  - Views
  - Integrity Constraints
  - Transactions
  - Triggers
Views

- Provide a mechanism to hide certain data from the view of certain users. To create a view we use the command:

  ```sql
  create view v as <query expression>
  where:
  <query expression> is any legal expression
  The view name is represented by v
  ```

- Can be used in any place a normal table can be used
- For users, there is no distinction in terms of using it

Example Queries

- A view consisting of branches and their customers

  ```sql
  create view all-customers as
    (select branch-name, customer-name
     from depositor, account
     where depositor.account-number = account.account-number)
  union
    (select branch-name, customer-name
     from borrower, loan
     where borrower.loan-number = loan.loan-number)
  ```

  Find all customers of the Perryridge branch

  ```sql
  select customer-name
  from all-customers
  where branch-name = ‘Perryridge’
  ```
Views

- Is it different from DBMS's side?
  - Yes; a view may or may not be materialized
  - Pros/Cons?

- Updates into views have to be treated differently
  - In most cases, disallowed.

Views vs Tables

<table>
<thead>
<tr>
<th>Creating</th>
<th>Create view V as (select * from A, B where …)</th>
<th>Create table T as (select * from A, B where …)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Can be used</td>
<td>In any select query. Only some update queries.</td>
<td>It's a new table. You can do what you want.</td>
</tr>
<tr>
<td>Maintained as</td>
<td>1. Evaluate the query and store it on disk as if a table. 2. Don't store. Substitute in queries when referenced.</td>
<td>It's a new table. Stored on disk.</td>
</tr>
<tr>
<td>What if a tuple inserted in A?</td>
<td>1. If stored on disk, the stored table is automatically updated to be accurate. 2. If we are just substituting, there is no need to do anything.</td>
<td>T is a separate table; there is no reason why DBMS should keep it updated. If you want that, you must define a trigger.</td>
</tr>
</tbody>
</table>
Views vs Tables

- Views strictly supercede “create a table and define a trigger to keep it updated”
- Two main reasons for using them:
  - Security/authorization
  - Ease of writing queries
    - E.g. IndividualMedals table
    - The way we are doing it, the IndividualMedals table is an instance of “creating table”, and not “creating view”
    - Creating a view might have been better.
- Perhaps the only reason to create a table is to force the DBMS to choose the option of “materializing”
  - That has efficiency advantages in some cases
  - Especially if the underlying tables don’t change

Update of a View

- Create a view of all loan data in loan relation, hiding the amount attribute
  
```
  create view branch-loan as
  select branch-name, loan-number
  from loan
  ```
- Add a new tuple to branch-loan
  
```
  insert into branch-loan
  values ('Perryridge', 'L-307')
  ```
- This insertion must be represented by the insertion of the tuple
  
```
  ('L-307', 'Perryridge', null)
  ```
  into the loan relation

- Updates on more complex views are difficult or impossible to translate, and hence are disallowed.
- Many SQL implementations allow updates only on simple views (without aggregates) defined on a single relation
Outline

- SQL
- Relational Algebra
- Formal Semantics of SQL

- More SQL
  - Nulls
  - Updates
  - Views
  - Integrity Constraints
  - Transactions
  - Triggers

SQL: Nulls

The “dirty little secret” of SQL  
(major headache for query optimization)

Can be a value of any attribute

 e.g: branch =

<table>
<thead>
<tr>
<th>bname</th>
<th>bcity</th>
<th>assets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>Boston</td>
<td>9M</td>
</tr>
<tr>
<td>Perry</td>
<td>Horseneck</td>
<td>1.7M</td>
</tr>
<tr>
<td>Mianus</td>
<td>Horseneck</td>
<td>.4M</td>
</tr>
<tr>
<td>Waltham</td>
<td>Boston</td>
<td>NULL</td>
</tr>
</tbody>
</table>

What does this mean?

(unknown) We don’t know Waltham’s assets?  
(inapplicable) Waltham has a special kind of account without assets  
(withheld) We are not allowed to know
SQL: Nulls

Arithmetic Operations with Null

\[ n + \text{NULL} = \text{NULL} \]  
(similarly for all arithmetic ops: \(+\), \(-\), \(*\), /, mod, ...)

e.g: \( \text{branch} = \)

<table>
<thead>
<tr>
<th>bname</th>
<th>bcity</th>
<th>assets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>Boston</td>
<td>9M</td>
</tr>
<tr>
<td>Perry</td>
<td>Horseneck</td>
<td>1.7M</td>
</tr>
<tr>
<td>Mianus</td>
<td>Horseneck</td>
<td>.4M</td>
</tr>
<tr>
<td>Waltham</td>
<td>Boston</td>
<td>NULL</td>
</tr>
</tbody>
</table>

```
SELECT bname, assets * 2 as a2
FROM branch
```

<table>
<thead>
<tr>
<th>bname</th>
<th>a2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>18M</td>
</tr>
<tr>
<td>Perry</td>
<td>3.4M</td>
</tr>
<tr>
<td>Mianus</td>
<td>.8M</td>
</tr>
<tr>
<td>Waltham</td>
<td>NULL</td>
</tr>
</tbody>
</table>

SQL: Nulls

Boolean Operations with Null

\[ n < \text{NULL} = \text{UNKNOWN} \]  
(similarly for all boolean ops: \(>\), \(<=\), \(>=\), \(<>\), =, ...)

e.g: \( \text{branch} = \)

```
SELECT *
FROM branch
WHERE assets = NULL
```

Counter-intuitive: NULL * 0 = NULL

Counter-intuitive: select * from movies
where length >= 120 or length <= 120
**SQL: Nulls**

**Boolean Operations with** Null

\[ n < \text{NULL} = \text{UNKNOWN} \]  
(similarly for all boolean ops: >, \(<=\), \(>=\), \(<\), \(\neq\), \(=\), \(\ldots\))

e.g: branch =

<table>
<thead>
<tr>
<th>bname</th>
<th>bcity</th>
<th>assets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>Boston</td>
<td>9M</td>
</tr>
<tr>
<td>Perry</td>
<td>Horseneck</td>
<td>1.7M</td>
</tr>
<tr>
<td>Mianus</td>
<td>Horseneck</td>
<td>.4M</td>
</tr>
<tr>
<td>Waltham</td>
<td>Boston</td>
<td>NULL</td>
</tr>
</tbody>
</table>

SELECT *  
FROM branch  
WHERE assets IS NULL

**SQL: Unknown**

**Boolean Operations with** Unknown

\[ n < \text{NULL} = \text{UNKNOWN} \]  
(similarly for all boolean ops: >, \(<=\), \(>=\), \(<\), \(\neq\), \(=\), \(\ldots\))

FALSE OR UNKNOWN = UNKNOWN  
TRUE AND UNKNOWN = UNKNOWN

Intuition: substitute each of TRUE, FALSE for unknown. If different answer results, results is unknown

UNKNOWN OR UNKNOWN = UNKNOWN  
UNKNOWN AND UNKNOWN = UNKNOWN  
NOT (UNKNOWN) = UNKNOWN

Can write:

SELECT ...
FROM ...
WHERE booleanexp IS UNKNOWN

UNKNOWN tuples are not included in final result
Given

branch =

<table>
<thead>
<tr>
<th>bname</th>
<th>bcity</th>
<th>assets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>Boston</td>
<td>9M</td>
</tr>
<tr>
<td>Perry</td>
<td>Horseneck</td>
<td>1.7M</td>
</tr>
<tr>
<td>Mianus</td>
<td>Horseneck</td>
<td>.4M</td>
</tr>
<tr>
<td>Waltham</td>
<td>Boston</td>
<td>NULL</td>
</tr>
</tbody>
</table>

### Aggregate Operations

```sql
SELECT SUM (assets) = SUM 11.1 M
FROM branch
NULL is ignored for SUM
```

*Same for AVG (3.7M), MIN (0.4M), MAX (9M)*

Also for COUNT(assets) -- returns 3

---

Given

branch =

<table>
<thead>
<tr>
<th>bname</th>
<th>bcity</th>
<th>assets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>Boston</td>
<td>9M</td>
</tr>
<tr>
<td>Perry</td>
<td>Horseneck</td>
<td>1.7M</td>
</tr>
<tr>
<td>Mianus</td>
<td>Horseneck</td>
<td>.4M</td>
</tr>
<tr>
<td>Waltham</td>
<td>Boston</td>
<td>NULL</td>
</tr>
</tbody>
</table>

### Aggregate Operations

```sql
SELECT SUM (assets) = SUM NULL
FROM branch
```

*But COUNT (*) returns*

- Same as AVG, MIN, MAX
- But COUNT (assets) returns COUNT 0

---
Transactions

A transaction is a sequence of queries and update statements executed as a single unit
- Transactions are started implicitly and terminated by one of
  - commit work: makes all updates of the transaction permanent in the database
  - rollback work: undoes all updates performed by the transaction.
- Motivating example
  - Transfer of money from one account to another involves two steps:
    - deduct from one account and credit to another
  - If one steps succeeds and the other fails, database is in an inconsistent state
  - Therefore, either both steps should succeed or neither should
- If any step of a transaction fails, all work done by the transaction can be undone by rollback work.
- Rollback of incomplete transactions is done automatically, in case of system failures
Transactions (Cont.)

- In most database systems, each SQL statement that executes successfully is automatically committed.
  - Each transaction would then consist of only a single statement
  - Automatic commit can usually be turned off, allowing multi-statement transactions, but how to do so depends on the database system
  - Another option in SQL:1999: enclose statements within
    ```
    begin atomic
    ...
    end
    ```

Outline

- SQL
- Relational Algebra
- Formal Semantics of SQL

More SQL

- Nulls
- Updates
- Views
- Integrity Constraints
- Transactions
- Triggers
Triggers

- A trigger is a statement that is executed automatically by the system as a side effect of a modification to the database.

- Suppose that instead of allowing negative account balances, the bank deals with overdrafts by
  1. setting the account balance to zero
  2. creating a loan in the amount of the overdraft
  3. giving this loan a loan number identical to the account number of the overdrawn account

Trigger Example in SQL:1999

```sql
create trigger overdraft_trigger after update on account
    referencing new row as nrow
    for each row
    when nrow.balance < 0
    begin atomic
        actions to be taken
    end
```
Trigger Example in SQL:1999

```sql
create trigger overdraft-trigger after update on account
  referencing new row as nrow
  for each row
  when nrow.balance < 0
  begin atomic
    insert into borrower
    (select customer-name, account-number
     from depositor
     where nrow.account-number = depositor.account-number);
    insert into loan values
    (nrow.account-number, nrow.branch-name, nrow.balance);
    update account set balance = 0
    where account.account-number = nrow.account-number
  end
```

Triggers...

- External World Actions
  - How does the DB order something if the inventory is low?

- Syntax
  - Every system has its own syntax

- Careful with triggers
  - Cascading triggers, Infinite Sequences...

- More Info/Examples:
  - Google: “create trigger” oracle download-uk
Outline

- SQL
- Relational Algebra
- Formal Semantics of SQL

- More SQL
  - Nulls
  - Updates
  - Views
  - Integrity Constraints
  - Transactions
  - Triggers

Next:

- Integrity constraints
- ??
- Prevent semantic inconsistencies
IC’s

- Predicates on the database
- Must always be true (checked whenever db gets updated)

- There are the following 4 types of IC’s:
  - **Key constraints** (1 table)
    - e.g., 2 accts can’t share the same acct_no
  - **Attribute constraints** (1 table)
    - e.g., accts must have nonnegative balance
  - **Referential Integrity constraints** (2 tables)
    - E.g. bnames associated w/ loans must be names of real branches
  - **Global Constraints** (n tables)
    - E.g., all loans must be carried by at least 1 customer with a savings acct

---

Key Constraints

Idea: specifies that a relation is a set, not a bag

SQL examples:

1. **Primary Key:**
   ```sql
   CREATE TABLE branch(
     bname  CHAR(15) PRIMARY KEY,
     bcity      CHAR(20),
     assets    INT);
   or
   CREATE TABLE depositor(
     cname   CHAR(15),
     acct_no  CHAR(5),
     PRIMARY KEY(cname, acct_no));
   ```

2. **Candidate Keys:**
   ```sql
   CREATE TABLE customer ( 
     ssn     CHAR(9) PRIMARY KEY, 
     cname   CHAR(15),
     address CHAR(30),
     city     CHAR(10),
     UNIQUE (cname, address, city));
   ```
Key Constraints

Effect of SQL Key declarations
- PRIMARY (A1, A2, .., An) or
- UNIQUE (A1, A2, ..., An)

Insertions: check if any tuple has same values for A1, A2, .., An as any inserted tuple. If found, **reject insertion**

Updates to any of A1, A2, ..., An: treat as insertion of entire tuple

Primary vs Unique (candidate)
1. 1 primary key per table, several unique keys allowed.
2. Only primary key can be referenced by “foreign key” (ref integrity)
3. DBMS may treat primary key differently (e.g.: create an index on PK)

How would you implement something like this?

Attribute Constraints

- Idea:
  - Attach constraints to values of attributes
  - Enhances types system (e.g.: >= 0 rather than integer)
- In SQL:
  1. **NOT NULL**
     
     ```sql
     CREATE TABLE branch(
         bname  CHAR(15) NOT NULL,
         ....
     )
     ```
     
     Note: declaring bname as primary key also prevents null values
  2. **CHECK**

     ```sql
     CREATE TABLE depositor(
         ....
         balance int NOT NULL,
         CHECK( balance >= 0),
         ....
     )
     ```

     affect insertions, update in affected columns
**Attribute Constraints**

**Domains:** can associate constraints with DOMAINS rather than attributes

e.g: instead of:

```sql
CREATE TABLE depositor(
    ....
    balance INT NOT NULL,
    CHECK (balance >= 0)
)
```

One can write:

```sql
CREATE DOMAIN bank-balance INT (
    CONSTRAINT not-overdrawn CHECK (value >= 0),
    CONSTRAINT not-null-value CHECK( value NOT NULL));
```

```sql
CREATE TABLE depositor (
    ....
    balance    bank-balance,
)
```

Advantages?

---

**Advantages of associating constraints with domains:**

1. can avoid repeating specification of same constraint for multiple columns

2. can name constraints
e.g.: CREATE DOMAIN bank-balance INT ( CONSTRAINT not-overdrawn CHECK (value >= 0),
     CONSTRAINT not-null-value CHECK( value NOT NULL));

allows one to:

1. add or remove:
   ```sql
   ALTER DOMAIN bank-balance
   ADD CONSTRAINT capped
   CHECK( value <= 10000)
   ```

2. report better errors (know which constraint violated)
Referential Integrity Constraints

Idea: prevent “dangling tuples” (e.g.: a loan with a bname, Kenmore, when no Kenmore tuple in branch)

Referencing Relation (e.g. loan) \[ \rightarrow \] Referenced Relation (e.g. branch)

“foreign key” bname \[ \rightarrow \] primary key bname

Ref Integrity: ensure that:

foreign key value \rightarrow primary key value

(note: don’t need to ensure \[ \leftarrow \], i.e., not all branches have to have loans)

In SQL:

CREATE TABLE branch(  
  bname  CHAR(15) PRIMARY KEY  
  ....)

CREATE TABLE loan (  
  .......  
  FOREIGN KEY bname REFERENCES branch);

Affects:
1) Insertions, updates of referencing relation
2) Deletions, updates of referenced relation
what happens when we try to delete this tuple?

Ans: 3 possibilities
1) reject deletion/update
2) set \(t_i[c], t_j[c] = \text{NULL}\)
3) propagate deletion/update
   - DELETE: delete \(t_i, t_j\)
   - UPDATE: set \(t_i[c], t_j[c]\) to updated values

CREATE TABLE A (   ......
FOREIGN KEY c REFERENCES B \text{action}  
........... )

Action:  1) left blank (deletion/update rejected)
2) ON DELETE SET NULL/ ON UPDATE SET NULL
   sets \(t_i[c] = \text{NULL}, t_j[c] = \text{NULL}\)
3) ON DELETE CASCADE
   deletes \(t_i, t_j\)
   ON UPDATE CASCADE
   sets \(t_i[c], t_j[c]\) to new key values
Global Constraints

Idea: two kinds
1) single relation (constraints spans multiple columns)
   - E.g.: CHECK (total = svngs + check) declared in the CREATE TABLE
2) multiple relations: CREATE ASSERTION

SQL examples:
1) single relation: All BkIn branches must have assets > 5M

   CREATE TABLE branch (  
       ........
       bcity  CHAR(15),
       assets INT,
       CHECK (NOT(bcity = 'Bkln') OR assets > 5M))

   Affects:
   insertions into branch
   updates of bcity or assets in branch

Global Constraints

SQL example:
2) Multiple relations: every loan has a borrower with a savings account

   CHECK (NOT EXISTS (     
       SELECT *  
       FROM loan AS L  
       WHERE NOT EXISTS(       
           SELECT *  
           FROM borrower B, depositor D, account A  
           WHERE B.cname = D.cname AND       
           D.acct_no = A.acct_no AND       
           L.loan = B.loan)))

   Problem: Where to put this constraint? At depositor? Loan? ....

   Ans: None of the above:
       CREATE ASSERTION loan-constraint
       CHECK( ...... )

   Checked with EVERY DB update!
   very expensive.....
Summary: Integrity Constraints

<table>
<thead>
<tr>
<th>Constraint Type</th>
<th>Where declared</th>
<th>Affects...</th>
<th>Expense</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Constraints</td>
<td>CREATE TABLE (PRIMARY KEY, UNIQUE)</td>
<td>Insertions, Updates</td>
<td>Moderate</td>
</tr>
<tr>
<td>Attribute Constraints</td>
<td>CREATE TABLE CREATE DOMAIN</td>
<td>Insertions, Updates</td>
<td>Cheap</td>
</tr>
<tr>
<td></td>
<td>(Not NULL, CHECK)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Referential Integrity</td>
<td>Table Tag (FOREIGN KEY .... REFERENCES ....)</td>
<td>1. Insertions into referencing rel’n</td>
<td>1.2: like key constraints. Another reason to index/sort on the primary keys 3,4: depends on a. update/delete policy chosen b. existence of indexes on foreign key</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. Updates of referencing rel’n of relevant attrs</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3. Deletions from referenced rel’n</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>4. Update of referenced rel’n</td>
<td></td>
</tr>
<tr>
<td>Global Constraints</td>
<td>Table Tag (CHECK) or outside table (CREATE ASSERTION)</td>
<td>1. For single rel’n constraint, with insertion, deletion of relevant attrs</td>
<td>1. cheap</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. For assertions w/ every db modification</td>
<td>2. very expensive</td>
</tr>
</tbody>
</table>

SQL

- Is that it?
  - Unfortunately No
  - SQL 3 standard is several hundreds of pages (if not several thousands)
  - And expensive too..

- We will discuss a few more constructs along the way
  - E.g. Embedded SQL, creating indexes etc

- Again, this is what the reference books are for; you just need to know where to look in the reference book