SQL

- Based primarily on Relational Algebra with some features from Relational Calculus
- Components: Data definition language, Manipulation language
- Other SQL features
  - Transaction definition: end of query by default
  - Security, Views, Index
- Embedded SQL – embed SQL commands in a general purpose language
- Database connectivity packages allow queries to be passed to DB from applications – JDBC
SQL-Data Definition language

- We’ve seen the DDL component of SQL
  - CREATE TABLE
  - ALTER TABLE
  - Key constraints
    - PRIMARY KEY(<>)
    - UNIQUE (<>)
  - Foreign Key/Referential Integrity
    - FOREIGN KEY (<>) REFERENCES <table name>
  - Other constraints?
    - Will see later

Basic SQL Query

```
SELECT [DISTINCT] attribute-list
FROM relation-list
WHERE qualification/predicate
```

- `relation-list` A list of relation names (possibly with a range-variable, i.e., tuple variable, after each name).
- `attribute-list` A list of attributes of relations in `relation-list`
- `Qualification/predicate` Comparisons (Attr `op` const or Attr1 `op` Attr2, where `op` is one of `<`, `>`, `=`, `<=`, `>=`, `<>`) combined using AND, OR and NOT.
- `DISTINCT` is an optional keyword indicating that the answer should not contain duplicates. Default is that duplicates are not eliminated!
Conceptual Evaluation Strategy

- **Semantics** of an SQL query defined in terms of the following conceptual evaluation strategy:
  - Compute the cross-product of *relation-list*.
  - Discard resulting tuples if they fail *predicate qualifications*.
  - Delete attributes that are not in *target attribute-list*.
    - If `DISTINCT` is specified, eliminate duplicate rows.
    - SQL allows duplicates in relations (unlike Rel. Algebra)
  - This strategy is probably the least efficient way to compute a query! An optimizer will find more efficient strategies to compute *the same answers*.

Equivalence with Rel. Algebra

- **Select** `<attribute-list>` = Project `Π_{(attribute-list)}`
- **From** `<r_1,r_2,...>` = Cartesian Product `r_1 X r_2 ...`
- **Where** `<predicate>` = Select `σ_{(predicate)}`
  - consisting of one or more conditions connected by logical connectives (and, or, not)
  - If no condition then ALL tuples selected
- **SQL query** = `Π_{(attr-list)} σ_{(predicate condition)} (R ∘ S)`
- **NOTE**: SQL has no natural join operator
  - Need to build cross product and then specify predicate (qualifiers) that forces the join condition
**Simple SQL Query**

```
SELECT * FROM Product
WHERE category='Gadgets'
```

---

**Simple SQL Query**

```
SELECT PName, Price, Manufacturer
FROM Product
WHERE Price > 100
```

---
Eliminating Duplicates

SELECT DISTINCT category
FROM Product

Compare to:

SELECT category
FROM Product

Joins

Product (pname, price, category, manufacturer)
Company (cname, stockPrice, country)

Find all products under $200 manufactured in Japan; return their names and prices.

SELECT PName, Price
FROM Product, Company
WHERE Manufacturer=CName AND Country='Japan'
AND Price <= 200
Joins

<table>
<thead>
<tr>
<th>PName</th>
<th>Price</th>
<th>Category</th>
<th>Manufacturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>$19.99</td>
<td>Gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>Powergizmo</td>
<td>$29.99</td>
<td>Gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>SingleTouch</td>
<td>$149.99</td>
<td>Photography</td>
<td>Canon</td>
</tr>
<tr>
<td>MultiTouch</td>
<td>$203.99</td>
<td>Household</td>
<td>Hitachi</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cname</th>
<th>StockPrice</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>GizmoWorks</td>
<td>25</td>
<td>USA</td>
</tr>
<tr>
<td>Canon</td>
<td>65</td>
<td>Japan</td>
</tr>
<tr>
<td>Hitachi</td>
<td>15</td>
<td>Japan</td>
</tr>
</tbody>
</table>

SELECT PName, Price FROM Product, Company WHERE Manufacturer=CName AND Country='Japan' AND Price <= 200

A Join Subtlety

Product (pname, price, category, manufacturer)
Company (cname, stockPrice, country)

Find all countries that manufacture some product in the ‘Gadgets’ category.

SELECT Country FROM Product, Company WHERE Manufacturer=CName AND Category='Gadgets'
## A Join Subtlety

<table>
<thead>
<tr>
<th>Name</th>
<th>Price</th>
<th>Category</th>
<th>Manufacturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>$19.99</td>
<td>Gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>Powergizmo</td>
<td>$29.99</td>
<td>Gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>SingleTouch</td>
<td>$149.99</td>
<td>Photography</td>
<td>Canon</td>
</tr>
<tr>
<td>MultiTouch</td>
<td>$203.99</td>
<td>Household</td>
<td>Hitachi</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Company</th>
<th>StockPrice</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>GizmoWorks</td>
<td>25</td>
<td>USA</td>
</tr>
<tr>
<td>Canon</td>
<td>45</td>
<td>Japan</td>
</tr>
<tr>
<td>Hitachi</td>
<td>15</td>
<td>Japan</td>
</tr>
</tbody>
</table>

**SELECT**  
**FROM**  
**WHERE**  
Manufacturer=CName AND Category='Gadgets'

What is the problem? What’s the solution?

---

### Tuple/Range variables

- Concept of tuple/range variables borrowed from relational calculus
  - Tuple \( t \) of type \( R: t \in R \)
  - *What about* \( x \in R, y \in R \)
- It performs the job of the rename operator in relational algebra
  - One variable with name \( x \) and one with name \( y \)
- Need to worry about scope of tuple variables when we have nested queries
**Tuple Variables**

Person(pname, address, worksfor)
Company(cname, address)

```sql
SELECT DISTINCT pname, address
FROM Person, Company
WHERE worksfor = cname
```

```sql
SELECT DISTINCT Person.pname, Company.address
FROM Person, Company
WHERE Person.worksfor = Company.cname
```

```sql
SELECT DISTINCT x.pname, y.address
FROM Person AS x, Company AS y
WHERE x.worksfor = y.cname
```

---

**Meaning (Semantics) of SQL Queries**

```sql
SELECT a_1, a_2, ..., a_k
FROM R_1 x_1, R_2 x_2, ..., R_n x_n
WHERE Conditions
```

```
Answer = {}
for x_1 in R_1 do
  for x_2 in R_2 do
    ..... 
    for x_n in R_n do
      if Conditions
        then Answer = Answer \cup \{(a_1,..,a_k)\}
  return Answer
```
A Note on Tuple/Range Variables

- Really needed only if the same relation appears twice in the FROM clause. The query can be written as:

```
SELECT C.name
FROM Customer C, Deposit D
WHERE C.custid=D.custid
```

OR

```
SELECT name
FROM Customer, Deposit
WHERE Customer.custid=Deposit.custid
```

It is good style, however, to use range variables always!

Schema of Bank DB

- Customer (CustID, Name, street, city, zip)
  - Customer ID, Name, and Address info: street, city, zip

- Deposit (CustID, Acct-num, balance, Branch-name)
  - Customer ID, Account number, Balance in account, name of branch where account is held

- Loan (CustID, Loan-num, Amount, Branch-name)
  - Customer ID, loan number, amount of loan...

- Branch (Branch-name, assets, Branch-city)
Tuple variables

- Find customers (ID) who have an account at a branch where CustID 6666 has an account.
- Need to access Deposit table twice
  - Once to extract branch X of CustID 6666
  - Second time to find accounts at these branches X
  - Define two “variables” A, B of ‘type’ Deposit
  - B is variable that corresponds to CustID 6666 and its branch-name field is equal to “X”
  - A is a variable whose branch-name is equal to “X”

SELECT A.CustID
FROM Deposit A, Deposit B
WHERE B.CustID=6666
AND A.branchname=B.branchname

B is a variable that refers to Customer 6666
Variable A can take on any values that satisfy the query
Find customers who have an account and live in NY

SELECT C.Custid
FROM Customer C, Deposit D
WHERE C.custid=D.custid
AND C.city=NY

- Would adding DISTINCT to this query make a difference?
- What is the effect of replacing C.custid by C.name in the SELECT clause? Would adding DISTINCT to this variant of the query make a difference?
Expressions and Strings

```sql
SELECT C.CustID AS cid
FROM Customer C
WHERE C.street LIKE '%Main%'
```

- Illustrates use of arithmetic expressions and string pattern matching: *Find customers whose street name includes ‘Main’.*
- `AS` and `=` are two ways to name fields in result.
- `LIKE` is used for string matching. `_` stands for any one character and `%` stands for 0 or more arbitrary characters.

Set Operations in SQL

*Find all customers (IDs) who have a loan or an account at the Downtown branch*

- **UNION:** Can be used to compute the union of any two *union-compatible* sets of tuples (which are themselves the result of SQL queries).
- **IMPORTANT NOTE:** In set operations, SQL removes duplicates
- Also available: **EXCEPT** (What do we get if we replace `UNION` by `EXCEPT`?)
Set Operations in SQL

Find all customers (IDs) who have a loan or an account at the Downtown branch

- **UNION**: Can be used to compute the union of any two *union-compatible* sets of tuples (which are themselves the result of SQL queries).

- **IMPORTANT NOTE**: In set operations, SQL removes duplicates.

- Also available: **EXCEPT** (What do we get if we replace **UNION** by **EXCEPT**?)

```
SELECT CustID
FROM  Loan
WHERE branchname='Downtown'
UNION
SELECT CustID
FROM Deposit
WHERE branchname='Downtown';
```

Find all customers (IDs) who have a loan and an account at the Downtown branch

- **INTERSECT**: Can be used to compute the intersection of any two *union-compatible* sets of tuples.

- Included in the SQL/92 standard, but some systems don’t support it.

```
SELECT D.CustID
FROM Deposit D, Loan L
WHERE D.CustID=L.CustID
AND  D.branchname='Downtown'
AND D.branchname=L.branchname;
```

```
SELECT CustID
FROM Loan
WHERE branchname='Downtown'
INTERSECT
SELECT CustID
FROM Deposit
WHERE branchname='Downtown';
```
SQL…What we have seen thus far

- "basic" SQL
  - basic SELECT statement
  - JOINS
  - SET operations
  - Tuple variables

- Next: Advanced SQL
  - Mix in Relational calculus operators
  - Add in features
  - SET membership operations
    - IN, <ALL, CONTAINS, etc.
  - Nested Queries
  - Tuple Variables
  - Aggregate Operators

Nested Queries

- Nested query has a sub-query nested in the WHERE clause
  - When query needs to refer to a set of tuples that need to be computed (and are not stored as a relation)
  - Can also appear in FROM clause
  - Need to be careful about scope of tuple variables
    - Scoping rules: local definition and then global
    - In subquery – legal to use only tuple variables defined in subquery itself or in any query that contains the subquery
**Nested Queries: Semantics**

- Evaluate subquery at each reference
  - Construct cross product of tables in FROM clause
  - For each row when testing predicate conditions in WHERE clause
    - Recompute subquery
      - Is this really necessary?
    - If subquery contains another subquery then apply same principle

**Subqueries Returning Relations and Set Membership operators**

Company(name, city)
Product pname, maker)
Purchase(id, product, buyer)

Return cities of companies that manufacture products bought by Joe Blow

```sql
SELECT Company.city
FROM Company
WHERE Company.name IN
  (Set of Companies that manufacture products bought by Joe Blow);
/* write a SELECT query to obtain this set */
```
Subqueries Returning Relations

Company(name, city)
Product(pname, maker)
Purchase(id, product, buyer)

Return cities of companies that manufacture products bought by Joe Blow

```sql
SELECT Company.city
FROM Company
WHERE Company.name IN
  (SELECT Product.maker
   FROM Purchase, Product
   WHERE Product.pname = Purchase.product
     AND Purchase.buyer = 'Joe Blow');
```

Example Bank Database

- Customer(CustID, Name, Street, City, Zip)
- Deposit(CustID, Acct-Num, branch-name, balance)
- Loan(CustID, Acct-num, branch-name, balance)
- Branch(branch-name, assets, city)
Set Membership Operations: (a)

- Can check for set membership using **IN** and **NOTIN**
  - \( x \ \text{IN} \ A \text{ or } x \ \text{NOTIN} \ A \)
  - Implements Relational Calculus operators
  - **IN** connective tests for membership in the set \( A \)
    - Set \( A \) may be produced by a **SELECT**
  - **NOTIN** tests for absence of tuples
  - Can test using multiple attribute element
- Set existence using **EXISTS**
  - Returns true if the argument subquery is nonempty (the converse for the **NOT EXISTS**) thus checking for empty relations

Find all customers who have both a loan and an account at the Downtown branch

- Find all account holders at Downtown branch
  - Call this set \( A \)
  - **SELECT** CustID
  - **FROM** Deposit
  - **WHERE** branch-name="Downtown"
- From loan relation, select those customers who also appear in set of account holders
  - If customer is in set \( A \), then select in result
    - Branch-name="Downtown" and CustID IN \( A \)
Find all customers who have both a loan and an account at the Downtown branch

- Find all account holders at Downtown branch
- From loan relation, select those customers who also appear in set of account holders
- The embedded/nested select selects customers with accounts
- Note that this query was written earlier without using nested queries

```
SELECT distinct CustID
FROM Loan
WHERE branchname='Downtown'
AND CustID IN
(SELECT CustID
FROM Deposit
WHERE branchname='Downtown');
```

Set Membership: Quantifiers

Product (pname, price, company)
Company (cname, city)

Find all companies that make some products with price < 100

Existential: easy ! ☺️
Set Membership: Quantifiers

Product (pname, price, company)
Company (cname, city)

Find all companies that make some products with price < 100

```
SELECT DISTINCT Company.cname
FROM Company, Product
WHERE Company.cname = Product.company and Product.price < 100
```

Existential: easy ! 😊

Set Membership: Quantifiers

Product (pname, price, company)
Company (cname, city)

Find all companies that make only products with price < 100

same as:

Find all companies such that all of their products have price < 100

Recall equivalence: For all x P(x) = Not Exists x (Not P(x))

Universal: hard ! 😞
**Set Membership: Quantifiers**

1. Find *the other* companies: i.e. s.t. *some* product \( \geq 100 \)

\[
\text{SELECT DISTINCT Company.cname} \\
\text{FROM Company} \\
\text{WHERE Company.cname IN (SELECT Product.company} \\
\text{FROM Product} \\
\text{WHERE Product.price} \geq 100)
\]

2. Find all companies s.t. *all* their products have price < 100

\[
\text{SELECT DISTINCT Company.cname} \\
\text{FROM Company} \\
\text{WHERE Company.cname NOT IN (SELECT Product.company} \\
\text{FROM Product} \\
\text{WHERE Product.price} \geq 100)
\]
More Set Membership Operations

- Previous operators allowed checking for existence
- SQL provides operators to test elements of one set $A$ with elements on another set $B$
  - **SOME**: $op$ SOME
    - Also called as ANY in some versions
  - **ALL**: $op$ ALL
    - $op$ can be $\geq$, $>$, $<$, $\leq$, $=$, not=
- Test single value against members of an entire set
  - $X > ALL \ (R)$

Find branches that have greater assets than some branch located in Brooklyn

- $A=$Set of branches in Brooklyn $\quad$ Branch.assets $>$some
- Compare assets of branch to members of set $A$
  - Branch is selected if its assets is greater than assets of SOME branch in Brooklyn
Find branches that have greater assets than some branch located in Brooklyn

```sql
SELECT branchname
FROM Branch
WHERE assets >
    (SELECT assets
     FROM Branch
     WHERE branchcity='Brooklyn');
```

- Inner select has set of assets of branches in Brooklyn
- For branches that have assets greater than all branches in Brooklyn replace `>some` with `>all`

---

Set Comparison Operations: subset

- Check if one set (query result) contains another set (query result)
  - Is A subset of B?
  - Is A not a subset of B?
- contains and not contains operators
Set Membership Examples:

Find all customers who have an account at all branches located in Brooklyn

SELECT CustID
FROM Deposit S
WHERE
  (set of branchnames where customer has an account)
  contains
  (set of all branchnames located in Brooklyn)
Set existence example

- Test if subquery is empty
  - Exists returns true if argument is nonempty
- Find all customers who have both an account and a loan at Downtown branch
- First test if customer has account and second test if customer has loan
  - Exists in Deposit and Exists in Loan

SELECT C.CustID
FROM Customer C
WHERE exists (SELECT *
               FROM Deposit D
               WHERE D.CustID=C.CustID
               AND D.branchname='Downtown')
AND exists (SELECT *
            FROM Loan L
            WHERE L.CustID= C. CustID
            AND L.branchname='Downtown');
Null Values

- Field values in a tuple are sometimes unknown (e.g., a rating has not been assigned) or inapplicable (e.g., no spouse’s name).
  - SQL provides a special value null for such situations.
- The presence of null complicates many issues. E.g.:
  - Special operators needed to check if value is/is not null.
  - Is rating>8 true or false when rating is equal to null? What about AND, OR and NOT connectives?
  - We need a 3-valued logic (true, false and unknown).
  - Meaning of constructs must be defined carefully. (e.g., WHERE clause eliminates rows that don’t evaluate to true.)
  - New operators (in particular, outer joins) possible/needed.

NULLS in SQL

- Whenever we don’t have a value, we can put a NULL
- Can mean many things:
  - Value does not exist
  - Value exists but is unknown
  - Value not applicable
  - Etc.
- The schema specifies for each attribute if it can be null (nullable attribute) or not
  - NOT NULL after declaring attribute domain
- How does SQL cope with tables that have NULLs?
Null Values

- If x = NULL then \(4 \times (3-x)/7\) is still NULL
- If x = NULL then x = “Joe” is UNKNOWN
- In SQL there are three boolean values:
  - FALSE = 0
  - UNKNOWN = 0.5
  - TRUE = 1

Null Values

- C1 AND C2 = min(C1, C2)
- C1 OR C2 = max(C1, C2)
- NOT C1 = 1 – C1

```
SELECT *
FROM Person
WHERE (age < 25) AND (height > 6 OR weight > 190)
```

E.g.
- age=20
- height=NULL
- weight=200

Rule in SQL: include only tuples that yield TRUE
Null Values

Unexpected behavior:

```
SELECT *
FROM Person
WHERE age < 25 OR age >= 25
```

Some Persons are not included!

---

Null Values

Can test for NULL explicitly:
- x IS NULL
- x IS NOT NULL

```
SELECT *
FROM Person
WHERE age < 25 OR age >= 25 OR age IS NULL
```

Now it includes all Persons
**Outerjoins**

Explicit joins in SQL = ‘inner joins’:
Product(name, category)
Purchase(prodName, store)

```
SELECT Product.name, Purchase.store
FROM Product JOIN Purchase ON
  Product.name = Purchase.prodName
```

Same as:
```
SELECT Product.name, Purchase.store
FROM Product, Purchase
WHERE Product.name = Purchase.prodName
```

But Products that never sold will be lost!

---

Left outer joins in SQL:
Product(name, category)
Purchase(prodName, store)

```
SELECT Product.name, Purchase.store
FROM Product LEFT OUTER JOIN Purchase ON
  Product.name = Purchase.prodName
```

---

CS 2441
### Outer Joins

- **Left outer join:**
  - Include the left tuple even if there's no match
- **Right outer join:**
  - Include the right tuple even if there's no match
- **Full outer join:**
  - Include the both left and right tuples even if there's no match
Next: Advanced SQL

- We’ve done basic SQL
  - Returns data stored in database
    - Combination of Rel algebra and calculus

- More advanced SQL:
  - Aggregate and Group-by operations
    - Return result of applying some operations on the data
  - General constraints on schema
    - Constraints, CHECKs, Assertions, Triggers
  - Stored procedures