More SQL

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
  - Views
  - Constraints, triggers
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');
```
Set Membership Operations: (a)

- Can check for set membership using \textbf{IN} and \textbf{NOTIN}
  - $x \text{ IN } A$ or $x \text{ NOTIN } A$
    - Implements Relational Calculus operators
  - \textbf{IN} connective tests for membership in the set $A$
    - Set $A$ may be produced by a \textbf{SELECT}
  - \textbf{NOTIN} tests for absence of tuples
  - Can test using multiple attribute element

- Set existence using \textbf{EXISTS}
  - Returns true if the argument subquery is nonempty (the converse for the \textbf{NOT EXISTS}) thus checking for empty relations

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)
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 customerID is in set A, then select in result
    - Predicate condition is: 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 only products with price < 100

same as:

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

Recall equivalence: Forall x P(x) = Not Exists x (Not P(x))

Universal: hard ! 😞

1. Find the other companies: i.e. s.t. some product ≥ 100

```sql
SELECT DISTINCT Company.cname
FROM Company
WHERE Company.cname IN (SELECT Product.company
FROM Product
WHERE Product.price >= 100)
```
**Set Membership: Quantifiers**

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

```
SELECT DISTINCT Company.cname
FROM Company
WHERE Company.cname IN (SELECT Product.company
FROM Product
WHERE Product.price >= 100)
```

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

```
SELECT DISTINCT Company.cname
FROM Company
WHERE Company.cname NOT IN (SELECT Product.company
FROM Product
WHERE Product.price >= 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

- \( A = \text{assets of branches in Brooklyn} \)
- \( \text{Branch.assets > some (assets of branches in Brooklyn)} \)

- 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 > some
  (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)

SELECT S.CustID
FROM Deposit S
WHERE (SELECT T.branchname
        FROM Deposit T
        WHERE S.CustID=T.CustID)
    CONTAINS
  (SELECT branchname
     FROM Branch
     WHERE branchcity=‘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 grade 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 \( x = \text{NULL} \) then \( 4(3-x)/7 \) is still \( \text{NULL} \)
- If \( x = \text{NULL} \) then \( x = \text{“Joe”} \) is \( \text{UNKNOWN} \)
- In SQL there are three boolean values:
  - FALSE = 0
  - UNKNOWN = 0.5
  - TRUE = 1

Null Values

- \( C1 \text{ AND } C2 = \min(C1, C2) \)
- \( C1 \text{ OR } C2 = \max(C1, C2) \)
- \( \neg C1 = 1 - C1 \)

Rule in SQL: include only tuples that yield \( \text{TRUE} \)

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

E.g.
age=20
height=NULL
weight=200
**Null Values**

Unexpected behavior:

```sql
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`

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

Now it includes all Persons
Joins and NULLs

List purchases at each store for each product

Product(name, category)
Purchase(prodName, store)

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!
### Outerjoins

Left outer joins in SQL:

Product(name, category)
Purchase(prodName, store)

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

Course Administrivia...

- Homworks on SQL posted
  - Broken into two homeworks HW 3, HW 4
- Work on writing a lot of SQL queries using MySQL
  - Next class will be in-class queries “exercises”
- Read textbook or other book
  - Notes are meant to substitute for book
- Next topic: Design of relational schemas
  - Project phase 1 will be posted end of next week
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

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!
SQL– Aggregate Operations

- Thus far SQL (and Relational Algebra/Calculus) only fetched data stored in database tables
- What if we need some basic ‘statistics’ on the data?
  - Number of rows?
  - Maximum value in a field?
- Aggregate Operators: apply a function to a set of tuples
  - Function defined on one (or more) field

Aggregate Operators

- Compute functions on set of tuples selected by WHERE clause
- Semantics: if SELECT clause contains aggregate operations then it can contain only aggregate operations
  - Except when groupby construct is used
  - Functions on sets of values but result is single value
  - Average, minimum, maximum, sum, count(size)
Aggregate Operators

- **Sum(Attribute)**
  - Computes sum of the values in the relation/set
- **Max(attribute)**
  - Computes the maximum among all values in the set
- **Average(attribute)**
  - Computes average
- **Count(attribute) or count(*)**
  - Counts number of tuples in the set

**Significant extension of relational algebra.**

```
SELECT AVG(balance) 
FROM Deposit;
```

```
SELECT AVG(D.balance) 
FROM Deposit D 
WHERE D.branchname='GW';
```

```
SELECT COUNT(Distinct CustID) 
FROM Deposit;
```

```
SELECT AVG (D.balance) 
FROM Deposit D 
WHERE D.balance= (SELECT MAX(S.balance) 
FROM Deposit S)
```

```
SELECT COUNT ( [DISTINCT] A) 
SUM ( [DISTINCT] A) 
AVG ( [DISTINCT] A) 
MAX (A) 
MIN (A) 
```

**single column**
More Examples

Can provide arithmetic expression as argument to functions
Purchase(product, date, price, quantity)

\[
\text{SELECT } \text{Sum(price * quantity)} \\
\text{FROM } Purchase
\]

What do they mean?

Simple Aggregations

Purchase

<table>
<thead>
<tr>
<th>Product</th>
<th>Date</th>
<th>Price</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>10/21</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>Banana</td>
<td>10/3</td>
<td>0.5</td>
<td>10</td>
</tr>
<tr>
<td>Banana</td>
<td>10/10</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>Bagel</td>
<td>10/25</td>
<td>1.50</td>
<td>20</td>
</tr>
</tbody>
</table>

\[
\text{SELECT } \text{Sum(price * quantity)} \\
\text{FROM } Purchase \\
\text{WHERE} \text{ product = ‘bagel’}
\]
Motivation for Grouping

- So far, we’ve applied aggregate operators to all (qualifying) tuples. Sometimes, we want to apply them to each of several groups of tuples.
- Consider: Find the average balance for each branch in the bank.
  - In general, we don’t know how many branches exist, and what the balances are!
  - Suppose we know that 10 branchnames exists; then we can write 10 queries that look like this (!):
    
    \[
    \text{SELECT AVG(balance) FROM Deposit D WHERE D.branchname='x'}
    \]
    
    For \( x = 1, 2, \ldots, 10 \):

Grouping and Aggregation

Purchase(product, date, price, quantity)

Find total sales after 10/1/2005 per product.

\[
\text{SELECT product, Sum(price*quantity) AS TotalSales FROM Purchase WHERE date > '10/1/2005'}
\]

GROUP BY product

Let’s see what this means…
**Grouping and Aggregation**

1. Compute the FROM and WHERE clauses.

2. Group by the attributes in the GROUPBY

3. Compute the SELECT clause: grouped attributes and aggregates.

**1&2. FROM-WHERE-GROUPBY**

<table>
<thead>
<tr>
<th>Product</th>
<th>Date</th>
<th>Price</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>10/21</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>Bagel</td>
<td>10/25</td>
<td>1.50</td>
<td>20</td>
</tr>
<tr>
<td>Banana</td>
<td>10/3</td>
<td>0.5</td>
<td>10</td>
</tr>
<tr>
<td>Banana</td>
<td>10/10</td>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>
3. **SELECT**

<table>
<thead>
<tr>
<th>Product</th>
<th>Date</th>
<th>Price</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>10/21</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>Bagel</td>
<td>10/25</td>
<td>1.50</td>
<td>20</td>
</tr>
<tr>
<td>Banana</td>
<td>10/3</td>
<td>0.5</td>
<td>10</td>
</tr>
<tr>
<td>Banana</td>
<td>10/10</td>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Product</th>
<th>TotalSales</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>50</td>
</tr>
<tr>
<td>Banana</td>
<td>15</td>
</tr>
</tbody>
</table>

```sql
SELECT product, Sum(price*quantity) AS TotalSales
FROM Purchase
WHERE date > '10/1/2005'
GROUP BY product
```

(a) Find average balance at each branch and (b) find number of customers with account at each branch

(a) `SELECT branchname, avg(balance) FROM Deposit GROUP BY branchname;`

(b) Query (a) keeps duplicates
```
SELECT branchname, count(distinct CustID)
FROM Deposit
GROUP BY branchname;
```
**Condition on the Groups**

- What if we are only interested in groups that satisfy a condition?

**Grouping and Aggregation**

Purchase(product, date, price, quantity)

Find total sales after 10/1/2005 per product for products that sold at least 30 quantity.

```
SELECT product, Sum(price*quantity) AS TotalSales
FROM Purchase
WHERE date > '10/1/2005'
GROUP BY product
```
Grouping and Aggregation

<table>
<thead>
<tr>
<th>Product</th>
<th>Date</th>
<th>Price</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>10/21</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>Bagel</td>
<td>10/25</td>
<td>1.50</td>
<td>20</td>
</tr>
<tr>
<td>Banana</td>
<td>10/3</td>
<td>0.5</td>
<td>10</td>
</tr>
<tr>
<td>Banana</td>
<td>10/10</td>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>

What if we are only interested in products that sold quantity >30?

```sql
SELECT product, Sum(price*quantity) AS TotalSales
FROM Purchase
WHERE date > '10/1/2005'
GROUP BY product
```

HAVING Clause

Purchase(product, date, price, quantity)
Find total sales after 10/1/2005 per product, except that we consider only products that had at least 30 buyers.

```sql
SELECT product, Sum(price * quantity)
FROM Purchase
WHERE date > '10/1/2005'
GROUP BY product
HAVING Sum(quantity) > 30
```

HAVING clause contains conditions on aggregates.
**General form of Grouping and Aggregation**

```
SELECT  S
FROM    R_1,...,R_n
WHERE   C_1
GROUP BY a_1,...,a_k
HAVING  C_2
```

S = may contain attributes $a_1,...,a_k$ and/or any aggregates but NO OTHER ATTRIBUTES
C_1 = is any condition on the attributes in $R_1,...,R_n$
C_2 = is any condition on aggregate expressions

---

**Generalized SELECT: Queries With GROUP BY and HAVING**

```
SELECT [DISTINCT] attribute-list
FROM relation-list
WHERE qualification/predicate
GROUP BY grouping-list
HAVING group-qualification/predicate
```

- The attribute-list contains (i) attribute names (ii) terms with aggregate operations (e.g., MIN (balance)).
  - The attribute list must be a subset of grouping-list.
    Intuitively, each answer tuple corresponds to a group, and these attributes must have a single value per group. (A group is a set of tuples that have the same value for all attributes in grouping-list.)
Conceptual Evaluation

- The cross-product of relation-list is computed, tuples that fail qualification in WHERE clause are discarded, 'unnecessary' fields are deleted, and the remaining tuples are partitioned into groups by the value of attributes in grouping-list.
- The group-qualification specified in the HAVING clause is then applied to eliminate some groups. Expressions in HAVING clause must have a single value per group!
  - In effect, an attribute in group-qualification that is not an argument of an aggregate op also appears in grouping-list. (SQL does not exploit primary key semantics here!)
- One answer tuple is generated per qualifying group.
- Any aggregate function can be applied to a group
  - Final SELECT can have function over each selected group

Find branches where average account balance is greater than $1200
Find branches where average account balance is greater than $1200

SELECT branchname, avg(balance)
FROM Deposit
GROUP BY branchname
HAVING avg(balance) > 1200

Find those branches with the highest average balance

- Aggregate functions cannot be composed
  - Max(min(..)) not allowed
- Select groups that have average balance greater than or equal to ALL average balances at all branches.
  - Need nested query to compute set of balances at each branch
SELECT branchname
FROM Deposit
GROUP BY branchname
HAVING AVG(balance) >= ALL
(SELECT AVG(balance)
FROM Deposit
GROUP BY branchname);

Find average balance of all depositors (account holders) who live in New York and have at least three accounts

- Select tuples where Customer lives in New York
- Form groups in Deposit based on Customer ID
  - GROUPBY CustID
- Select only those groups with 3 or more tuples
  - COUNT number of tuples in each group
SELECT AVG(balance)  
FROM Deposit D, Customer C  
WHERE D.CustID=C.CustID AND  
  C.City='New York'  
GROUP BY D.CustID  
HAVING COUNT(DISTINCT account-number) >= 3

---

**A quick Note:**

*Group-by v.s. Nested Query*

**Author(login, name)**

**Wrote(login, url)**

- Find authors who wrote ≥ 10 documents:
- Attempt 1: with nested queries

```sql
SELECT DISTINCT Author.name  
FROM Author  
WHERE count(SELECT Wrote.url  
            FROM Wrote  
            WHERE Author.login=Wrote.login) > 10
```

This is SQL by a novice
**Group-by v.s. Nested Query**

- Find all authors who wrote at least 10 documents:
- Attempt 2: SQL style (with GROUP BY)

```sql
SELECT Author.name
FROM Author, Wrote
WHERE Author.login=Wrote.login
GROUP BY Author.name
HAVING count(wrote.url) > 10
```

No need for DISTINCT: automatically from GROUP BY

---

**Quick Note: Outer Joins Application**

Compute, for each product, the total number of sales in ‘September’

```
Product(name, category)
Purchase(prodName, month, store)
```

```sql
SELECT Product.name, count(*)
FROM Product, Purchase
WHERE Product.name = Purchase.prodName
    and Purchase.month = 'September'
GROUP BY Product.name
```

What’s wrong?
**Outer Join Application**

Compute, for each product, the total number of sales in ‘September’
Product(name, category)
Purchase(prodName, month, store)

```sql
SELECT Product.name, count(*)
FROM Product LEFT OUTER JOIN Purchase ON
    Product.name = Purchase.prodName
    and Purchase.month = ‘September’
GROUP BY Product.name
```

Now we also get the products who sold in 0 quantity

---

**SQL … a little more**
Generalized SELECT: Queries With GROUP BY and HAVING

\[
\text{SELECT [DISTINCT] attribute-list} \\
\text{FROM relation-list} \\
\text{WHERE qualification/predicate} \\
\text{GROUP BY grouping-list} \\
\text{HAVING group-qualification/predicate}
\]

- The *attribute-list* contains (i) attribute names (ii) terms with aggregate operations (e.g., MIN (balance)).
  - The *attribute* list must be a subset of *grouping-list*. Intuitively, each answer tuple corresponds to a *group*, and these attributes must have a single value per group. (A *group* is a set of tuples that have the same value for all attributes in *grouping-list*.)

Conceptual Evaluation

- The cross-product of *relation-list* is computed, tuples that fail qualification in WHERE clause are discarded, ‘unnecessary’ fields are deleted, and the remaining tuples are partitioned into groups by the value of attributes in *grouping-list*.
- The *group-qualification* specified in the HAVING clause is then applied to eliminate some groups. Expressions in HAVING clause must have a single value per group!
  - In effect, an attribute in *group-qualification* that is not an argument of an aggregate op also appears in *grouping-list*. (SQL does not exploit primary key semantics here!)
- One answer tuple is generated per qualifying group.
- Any aggregate function can be applied to a group
  - Final SELECT can have function over each selected group
Next: Modifying the Database

- INSERT
- UPDATE
- DELETE

DELETE r
WHERE P
Predicate in P can be as complex as any select clause
Delete all accounts located in New York
DELETE Deposit
WHERE branchname in (SELECT branchname
FROM Branch
WHERE branchcity=‘New York’);
How about this query?

DELETE Deposit
WHERE balance < (SELECT avg(balance)
            FROM Deposit);

Delete anomalies

- If delete/update request contains embedded select (sub-query) that references relation where deletions/update take place
- SQL standard disallows such requests
  - Alternate implementation: mark tuples in first round, and actual delete in second round
- **INSERT**
  - Can insert tuple with specified values
  - Can insert set of tuples resulting from query
- **UPDATE**
  - Change a value in tuple without changing all values in the tuple
  - Can update set of tuples by using query to select the set

```
INSERT INTO Deposit VALUES (9732, 'Downtown', 1234, 1000);
```

Also can specify attributes

```
INSERT INTO Deposit(Account-Num, CustID, Branch-name, Balance) VALUES (1234, 9732, 'Downtown', 1000);
```
**INSERT**

- Give all customers with a Loan at Downtown branch a $200 savings account with same account number as Loan number

```
INSERT INTO Deposit
SELECT CustID, Loan-number, Branch-name, 200
FROM Loan
WHERE branch-name = 'Downtown';
```

**UPDATE**

- Update balances of all accounts over $10,000 to give 5% interest

```
UPDATE Deposit
SET balance = 1.05*balance
WHERE balance > 10,000;
```
**SQL: Views**

- Create a virtual relation which is result of a query
  - View can be referenced by other queries
- CREATE VIEW AS <query-expression>
  - Query expression is any legal expression

---

**Defining Views**

Views are relations, except that they are not physically stored.

For presenting different information to different users

**Employee**(ssn, name, department, project, salary)

CREATE VIEW Developers AS
SELECT name, project
FROM Employee
WHERE department = "Development"

Payroll has access to Employee, others only to Developers
Types of Views

- **Virtual views:**
  - Used in databases
  - Computed only on-demand – slow at runtime
  - Always up to date
- **Materialized views**
  - Used in data warehouses
  - Pre-computed offline – fast at runtime
  - May have stale data

Views: Example

- Create view of all branch names and IDs of customers with loan or deposit

CREATE VIEW all-customer AS
(SELECT branchname, CustID
FROM Deposit)
UNION
(SELECT branchname, CustID
FROM Loan);
Example

Person(name, city)
Purchase(buyer, seller, product, store)
Product(name, maker, price, category)

CREATE VIEW Seattle-Purchase AS

SELECT y.buyer, y.seller, y.product, y.store
FROM Person x, Purchase y
WHERE x.city = ‘Seattle’ AND
    x.name = y.buyer

Seattle-Purchase(buyer, seller, product, store) “virtual table”

Querying a View

We can later use the view:

SELECT v.name, u.store
FROM Seattle-Purchase u, Product v
WHERE u.product = v.name AND
    v.category = ‘shoes’
What Happens When We Query a View?

```
SELECT v.name, u.store
FROM Seattle-Purchase u, Product v
WHERE u.product = v.name AND v.category = 'shoes'
```

```
SELECT v.name, y.store
FROM Person x, Purchase y, Product v
WHERE x.city = 'Seattle' AND x.name = y.buyer AND y.product = v.name AND v.category = 'shoes'
```

Updating Views: Part 1

Purchase(buyer, seller, product, store)
Product(name, maker, price, category)

```
CREATE VIEW Expensive-Product AS
SELECT name, maker
FROM Product
WHERE price > 100
```

```
INSERT INTO Expensive-Product
VALUES('Gizmo', 'Gadgets INC.')
```

```
INSERT INTO Product
VALUES('Gizmo', 'Gadgets INC.', NULL, NULL)
```
Updating Views: Part 2

Purchase(buyer, seller, product, store)
Product(name, maker, price, category)

CREATE VIEW Toy-Product AS
SELECT price, maker
FROM Product
WHERE category = ‘Toys’

INSERT INTO Toy-Product
VALUES(‘Gadgets INC.’,$100)

INSERT INTO Product
VALUES(NULL, ‘Gadgets INC.’, 100, NULL)

Updating Views: Part 3

Purchase(buyer, seller, product, store)
Product(name, maker, price, category)

CREATE VIEW Buyer-Maker AS
SELECT x.buyer, y.maker
FROM Purchase x, Product y
WHERE x.product = y.name

INSERT INTO Buyer-Maker
VALUES(‘John Smith’, ‘Gadgets INC.’)

Non-updateable view

Most views are non-updateable
Updating Views

- Non-updatable views
  - Multiple relation views
  - Primary key NULL
  - Later versions of SQL allow some updates on multiple relation views
  - Security/privacy and bad use of views?
- Interesting “side-effects” on view authorization
  - More when we discuss security/authorization

Integrity Constraints (Review)

- An IC describes conditions that every legal instance of a relation must satisfy.
  - Inserts/deletes/updates that violate IC’s are disallowed.
  - Can be used to ensure application semantics (e.g., Custid is a key), or prevent inconsistencies (e.g., name has to be a string, balance must be > 100)
- Types of IC’s: Domain constraints, primary key constraints, foreign key constraints, general constraints.
  - Domain constraints: Field values must be of right type. Always enforced.
Constraints in SQL

- A constraint = a property that we’d like our database to hold
- The system will enforce the constraint by taking some actions:
  - forbid an update
  - or perform compensating updates

Constraints in SQL:
- Keys, foreign keys
- Attribute-level constraints
- Tuple-level constraints
- Global constraints: assertions

The more complex the constraint, the harder it is to check and to enforce
Constraints on Attributes and Tuples

- Constraints on attributes:
  - NOT NULL -- obvious meaning...
  - CHECK condition -- any condition!
- Constraints on tuples
  - CHECK condition

Domain Constraints – user defined types

- User can define a new domain
  - Restriction of domain types supported by DBMS
  - Example: no grade below C

```
CREATE DOMAIN nice-grade INTEGER DEFAULT 1
    CHECK (VALUE >1 AND VALUE <=4)
```
- In schema definition, attribute grade is defined to be of this type
  - grade nice-grade
- Remember domain constraints
  - These are very useful when designing real applications!!!
Constraints: Examples

- Define a minimum balance in an account

```sql
CREATE TABLE Deposit(
    CustID integer,
    Branch-name CHAR(20),
    Account-Num integer,
    Balance Real
)

PRIMARY KEY (CustID, Account-Num)
FOREIGN KEY (CustID) REFERENCES Customer
FOREIGN KEY (Branch-name) REFERENCES Branch
CHECK (Balance >= 100);
```

Constraints over multiple tables

- Account-number cannot be same as zip code

```sql
CREATE TABLE Deposit(
    CustID integer,
    Branch-name CHAR(20),
    Account-Num integer,
    Balance Real
)

PRIMARY KEY (CustID, Account-Num)
FOREIGN KEY (CustID) REFERENCES Customer
FOREIGN KEY (Branch-name) REFERENCES Branch
CHECK (Balance >= 100)
CHECK (Account-Num <>
    (SELECT zip
     FROM Customer));
```
General Constraints

- Useful when more general ICs than keys are involved.
- Can use queries to express constraint.
- Constraints can be named.
  - What is the advantage of this?

CREATE TABLE Sailors
( sid INTEGER,  
sname CHAR(10),
rating INTEGER,
age REAL,
PRIMARY KEY (sid),
CHECK ( rating >= 1  
AND rating <= 10 )
)

CREATE TABLE Reserves
( sname CHAR(10),  
bid INTEGER,  
day DATE,
PRIMARY KEY (bid,day),
CONSTRAINT noInterlakeRes
CHECK (`Interlake’ <>
( SELECT B.bname  
FROM Boats B  
WHERE B.bid=bid])))

CREATE TABLE Purchase
( prodName CHAR(30)  
CHECK (prodName IN  
SELECT Product.name  
FROM Product),  
date DATETIME NOT NULL)

What is the difference from Foreign-Key?
Constraints over Multiple Tables

CREATE TABLE Sailors
(sid INTEGER,
sname CHAR(10),
rating INTEGER,
age REAL,
PRIMARY KEY (sid),
CHECK ((SELECT COUNT (S.sid) FROM Sailors S) + (SELECT COUNT (B.bid) FROM Boats B) < 100)
)

- Awkward and wrong!
- If Sailors is empty, the number of Boats tuples can be anything!
- Constraint should be defined over entire database

Constraints Over Multiple Relations

Number of boats plus number of sailors is < 100
Assertions: Constraints Over Multiple Relations

CREATE TABLE Sailors
    ( sid INTEGER,
    sname CHAR(10),
    rating INTEGER,
    age REAL,
    PRIMARY KEY (sid),
    CHECK
    ( (SELECT COUNT (S.sid) FROM Sailors S) + (SELECT COUNT (B.bid) FROM Boats B) < 100 )

AWKWARD and wrong!
- If Sailors is empty, the number of Boats tuples can be anything!
- ASSERTION is the right solution; not associated with either table.

CREATE ASSERTION smallClub
    CHECK
    ( (SELECT COUNT (S.sid) FROM Sailors S) + (SELECT COUNT (B.bid) FROM Boats B) < 100 )

Assertions

- Assertion is a predicate expressing a condition that the database must satisfy
- Assertion must check to make sure an update is good
  - on violation it rejects the update
- Allows constraints over multiple tables
Final Comments on Constraints

- Can give them names, and alter later
- We need to understand exactly *when* they are checked
- We need to understand exactly *what* actions are taken if they fail

Triggers

- Trigger: procedure that starts automatically if specified changes occur to the DBMS
- Three parts:
  - Event (activates the trigger)
  - Condition (tests whether the triggers should run)
  - Action (what happens if the trigger runs)
Triggers: Example

- Overdraft transaction:
  - When balance is negative, create a Loan in the amount of the overdraft plus a fee
    - Loan number is same as account-number
- When to “act”?
  - On update of Deposit relation
- What is the action?
  - Add new tuple into Loan

```
DEFINE TRIGGER Overdraft
ON UPDATE OF Deposit T
If new.balance < 0
Then (  
  INSERT INTO Loan values
  (T.Branch-name, T.Cust-ID, T.account-num, new.balance+fine)
  UPDATE Deposit S
  SET S.balance=0
  WHERE T.account-num = S.account-num)
```
Summary

- SQL SELECT statements
  - Nested queries
  - Set operations
  - Aggregate operations
  - Grouping
- Constraints, Triggers, Assertions?
  - Will get to this after a week
- Next question: How to design a schema?

Relational Model: Current Summary

- Data driven not design driven
  - designed once; data changes over time without affecting applications
  - rules/constraints control how data defined and enforced
- SQL query language
  - Define schema
  - Query data
  - Constraints, triggers, assertions
- Next: how to design schema
  - changes to database scheme without affecting application?