SQL

- **SQL = Structured Query Language**
  - since early 1970s

- Combination of relational algebra and relational calculus constructs

- More acronyms:
  - DML: Data **M**anipulation **L**anguage
  - DDL: Data **D**efinition **L**anguage
    - Includes view definition, integrity constraints, authorization control

Relation Schema Example

- **Account** (account_number, branch_name, balance)
- **Branch** (branch_name, branch_city, assets)
- **Customer** (customer_name, customerStreet, customer_city)
  - For simplicity assume customer_name unique
- **Depositor** (customer_name, account_number)
- **Loan** (loan_number, branch_name, amount)
- **Borrower** (customer_name, loan_number)

Basic Structure

- A typical SQL query has the form:
  
  - **select** $A_1, A_2, ..., A_n$ → RA projection
  - **from** $r_1, r_2, ..., r_m$ → RA cartesian product
  - **where** $P$ → RA selection
  
  - $A_i$ represent attributes
  - $r_i$ represent relations
  - $P$ is a predicate

- Query is equivalent to the relational algebra expression:
  
  $$\Pi_{A_1, A_2, ..., A_n}(\sigma_P(r_1 \times r_2 \times ... \times r_m))$$

- **NOTE:** SQL results may contain duplicates

- The result of an SQL query is a relation.
**SQL – select**

- The `select` clause lists attributes desired in the result.
  - It corresponds to the projection operation of the relational algebra.
  - E.g., find the names of all branches in the `loan` relation:
    ```
    select branch_name
    from loan
    ```
  - Same query, in the “pure” relational algebra syntax:
    ```
    \( \prod_{\text{branch\_name}}(\text{loan}) \)
    ```

- **NOTE:** SQL does not permit the ‘-‘ character in names.
- **NOTE:** SQL names are case insensitive, i.e., you can use capital or small letters.

---

**SQL – select – II**

- SQL allows duplicates in relations and in query results.
  - To force elimination of duplicates, insert keyword `distinct` after `select`.
  - E.g., find the names of all branches in the `loan` relations, and remove duplicates:
    ```
    select distinct branch_name
    from loan
    ```
  - Keyword `all` specifies that duplicates not be removed:
    ```
    select all branch_name
    from loan
    ```

---

**SQL – select – III**

- An asterisk in the `select` clause denotes “all attributes”:
  ```
  select *
  from loan
  ```

- `select` can contain arithmetic expressions:
  - Similar to generalized projection from Relational Algebra.
  - Can involve the operation, +, -, *, and /,
  - Can operate on constants or attributes of tuples.

- Example:
  ```
  select loan_number, branch_name, amount * 100
  from loan
  ```

---

**SQL – where**

- The `where` clause specifies conditions the result must satisfy.
  - It corresponds to the selection predicate of the relational algebra.

- Example:
  ```
  find all loan number for loans made at the Perryridge branch with loan amounts greater than $1200:
  ```
  ```
  select loan_number:
  from loan
  where branch_name = 'Perryridge' and amount > 1200
  ```

- Comparison results can be combined using `and`, `or`, `not`.
- Comparisons can be applied to results of arithmetic expressions.
SQL – where – II

- SQL includes a *between* comparison operator.
- Example:
  - Find the loan number of those loans with loan amounts between $90,000 and $100,000 (that is, $\geq$90,000 and $\leq$100,000)
    
    ```sql
    select loan_number from loan
    where amount between 90000 and 100000
    ```

    is equivalent to:
    
    ```sql
    select loan_number from loan
    where amount >= 90000 and amount <= 100000
    ```

SQL – from

- The *from* clause lists the relations involved in the query.
  - Corresponds to the Cartesian product operation of the relational algebra.
- Examples:
  - Find the Cartesian product borrower x loan
    ```sql
    select * from borrower, loan
    ```
  - Find the name, loan number and loan amount of all customers having a loan at the Perryridge branch.
    ```sql
    select customer_name, borrower.loan_number, amount from borrower, loan
    where borrower.loan_number = loan.loan_number
    and branch_name = 'Perryridge'
    ```

SQL – rename

- SQL allows renaming relations and attributes using the *as* clause:
  
  ```sql
  old_name as new_name
  ```

- Example:
  - Find the name, loan number and loan amount of all customers; rename the column name loan_number as loan_id
    
    ```sql
    select customer_name, borrower.loan_number as loan_id, amount from borrower, loan
    where borrower.loan_number = loan.loan_number
    ```

SQL – tuple variables

- Tuple variables are defined in the *from* clause via the use of the *as* clause.
- Examples:
  - Find the customer names and their loan numbers for all customers having a loan at some branch.
    ```sql
    select customer_name, T.loan_number, S.amount from borrower as T, loan as S
    where T.loan_number = S.loan_number
    ```
  - Find the names of all branches that have greater assets than some branch located in Brooklyn
    ```sql
    select distinct T.branch_name from branch as T, branch as S
    where T.assets > S.assets and S.branch_city = 'Brooklyn'
    ```
SQL – like

- Like
  - String-matching operator for comparisons on character strings
- Patterns are described using two special characters:
  - Percent (%) matches any substring
  - Underscore (_) matches any single character
- Examples:
  - Find the names of all customers whose street includes "Main"
    ```sql
    select customer_name
    from customer
    where customer_street like '%Main%'
    ```
  - Match the name "twenty%" like "twenty\% escape '\""

SQL – order

- Order-by clause causes the tuples in the result to appear in sorted order
- Example:
  - List in alphabetic order the names of all customers having a loan in the Perryridge branch
    ```sql
    select distinct customer_name
    from borrower, loan
    where borrower.loan_number = loan.loan_number and branch_name = 'Perryridge'
    order by customer_name
    ```
- Specify desc for descending order
- Specify asc for ascending order (default)
  - E.g. order by customer_name desc

SQL – order – II

- It is possible to define two or more attributes to order by
- Example:
  - List entire loan relation in descending order of amount
  - If several loans have the same amount, order by loan_number
    ```sql
    select *
    from loan
    order by amount desc, loan_number asc
    ```

SQL Set Operations

- The set operations union, intersect, and except operate on relations
- Correspond to the relational algebra operations \( \cup, \cap, - \)
- Each of the above operations automatically eliminates duplicates
- To retain all duplicates use the corresponding multiset versions union all, intersect all and except all.
SQL Set Operations – II

- Depositor (customer_name, account_number)
- Borrower (customer_name, loan_number)

Assume set of customers who have an account:

```
SELECT customer_name FROM depositor
```

Assume set of customers who have a loan:

```
SELECT customer_name FROM borrower
```

SQL / Sets: Union

- Find all customers having an account, a loan, or both:
  
  ```
  (SELECT customer_name FROM depositor)
  UNION
  (SELECT customer_name FROM borrower)
  ```

- To retain all duplicates:
  
  ```
  (SELECT customer_name FROM depositor)
  UNION ALL
  (SELECT customer_name FROM borrower)
  ```

- If "Jones" has 3 accounts and 2 loans \( \Rightarrow \) appears 5 times
  
  \( \text{sum of a, b} \)

SQL / Sets: Intersect

- Find all customers having both an account and a loan:
  
  ```
  (SELECT customer_name FROM depositor)
  INTERSECT
  (SELECT customer_name FROM borrower)
  ```

- To retain all duplicates:
  
  ```
  (SELECT customer_name FROM depositor)
  INTERSECT ALL
  (SELECT customer_name FROM borrower)
  ```

- If "Jones" has 3 accounts and 2 loans \( \Rightarrow \) appears 2 times
  
  \( \text{min of a, b} \)

SQL / Sets: Except

- Find all customers having an account, but no a loan:
  
  ```
  (SELECT customer_name FROM depositor)
  EXCEPT
  (SELECT customer_name FROM borrower)
  ```

- To retain all duplicates:
  
  ```
  (SELECT customer_name FROM depositor)
  EXCEPT ALL
  (SELECT customer_name FROM borrower)
  ```

- If "Jones" has 3 accounts and 1 loans \( \Rightarrow \) appears 2 times
  
  \( a - b, \text{if } a>b, \text{or 0 otherwise} \)
Roadmap

- SQL Intro
  - Select, From, Where
  - Rename
  - Order-by
  - Union, Intersect, Except
- Aggregate Functions
- Null Values
- Nested Subqueries
- Views

SQL / Aggregate Functions

- These functions operate on a collection of values and return a single value
- SQL offers five aggregate functions:
  - `avg()`: average value
  - `min()`: minimum value
  - `max()`: maximum value
  - `sum()`: sum of values
  - `count()`: number of values

SQL / Aggregate Functions (Cont.)

- Find the average account balance at the Perryridge branch.
  ```sql
  select avg(balance) 
  from account 
  where branch_name = 'Perryridge'
  ```
- Find the number of depositors in the customer relation.
  ```sql
  select count(*) 
  from customer
  ```
- Find the number of depositors in the bank.
  ```sql
  select count(distinct customer_name) 
  from depositor
  ```

SQL Aggregation – Group By

- Find the number of depositors for each branch.
  ```sql
  select branch_name, count(distinct customer_name) 
  from depositor, account 
  where depositor.account_number = account.account_number 
  group by branch_name
  ```

Note: Attributes in `select` clause outside of aggregate functions must appear in `group by` list
Note: Similar to aggregation with grouping from Relational Algebra
**SQL Aggregation – Having Clause**

- **Example:**
  - Find the names of all branches where the average account balance is more than $1,200.

```
select branch_name, avg(balance)
from account
group by branch_name
having avg(balance) > 1200
```

**Note:** Predicates in the `having` clause are applied after the formation of groups, whereas predicates in the `where` clause are applied before forming groups.

---

**SQL – NULL Values**

- Tuples can have **NULL** value for some of their attributes
  - For an unknown value
  - For a value that does not exist

- **Predicate is null** can be used to check for null values.
  - Find all loan numbers which appear in the `loan` relation with NULL values for `amount`

```
select loan_number
from loan
where amount is null
```

- Any arithmetic expression involving NULL returns NULL
  - E.g. 5 + NULL returns NULL

---

**SQL – Logic with NULL Values**

- Any comparison with `null` returns `unknown`
  - E.g. 5 < null or null <> null or null = null

- Three-valued logic using the truth value `unknown`:

<table>
<thead>
<tr>
<th>AND</th>
<th>T</th>
<th>F</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>U</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OR</th>
<th>T</th>
<th>F</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>U</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>U</td>
<td>U</td>
<td>T</td>
<td>U</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NOT</th>
<th>T</th>
<th>F</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>U</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
</tbody>
</table>

- "P is unknown" \(\Rightarrow\) true if predicate \(P\) evaluates to `unknown`

- Result of `where` clause predicate is treated as `false` if it evaluates to `unknown`

---

**SQL – NULL Values in Aggregations**

- **Example:**
  - Find total of all loan amounts

```
select sum(amount)
from loan
```

- `sum()` operator ignores null amounts

- In general:
  - `sum()`, `avg()`, `min()`, `max()` ignore null values
  - `count(*)` considers null values

- Aggregation on an empty set:
  - `sum()`, `avg()`, `min()`, `max()` return null
  - `count(*)` returns 0

---
Roadmap

- SQL Intro
  - Select, From, Where
  - Rename
  - Order-by
  - Union, Intersect, Except

- Aggregate Functions
- Null Values
- Nested Subqueries
- Views

SQL – Nested Subqueries

- SQL provides a mechanism for nesting subqueries.

  - A subquery is a **select-from-where** expression that is included within another query.

  - A common use of subqueries is to perform tests for
    - set membership
    - set comparisons
    - set cardinality

Nested Subqueries – Examples /1

- Find all customers who have both an account and a loan at the bank:
  - ( `select` customer_name `from` depositor )
  - intersect
  - ( `select` customer_name `from` borrower )

- Find all customers who have both an account and a loan at the bank (version 2):
  - `select` distinct customer_name `from` borrower
  - where customer_name in ( `select` customer_name `from` depositor )

Nested Subqueries – Examples /2

- Find all customers who have a loan at the bank but do not have an account at the bank:
  - ( `select` customer_name `from` borrower )
  - except
  - ( `select` customer_name `from` depositor )

- Find all customers who have a loan at the bank but do not have an account at the bank (version 2):
  - `select` distinct customer_name `from` borrower
  - where customer_name not in ( `select` customer_name `from` depositor )
### Nested Subqueries – Examples

- Find all customers who have both an account and a loan at the Perryridge branch.
  ```sql
  select distinct customer_name
  from borrower, loan
  where borrower.loan_number = loan.loan_number and
  branch_name = "Perryridge" and
  (branch_name, customer_name) in
  (select branch_name, customer_name
   from depositor, account
   where depositor.account_number = account.account_number)
  ```

  **Note:** Above query can be written in a much simpler manner. The formulation above is simply to illustrate SQL features.

### Nested Subqueries – Set Comparison

- Find all branches that have greater assets than some branch located in Brooklyn.
  ```sql
  select branch_name
  from branch
  where assets > some (select assets
   from branch
   where branch_city = "Brooklyn")
  ```

  **Note:** Same query using > some clause

### Definition of SOME Clause

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$5 &lt; \text{some}$</td>
<td>0, 5</td>
<td>true</td>
</tr>
<tr>
<td>$5 &lt; \text{some}$</td>
<td>0, 5</td>
<td>false</td>
</tr>
<tr>
<td>$5 = \text{some}$</td>
<td>0, 5</td>
<td>true</td>
</tr>
<tr>
<td>$5 \neq \text{some}$</td>
<td>0, 5</td>
<td>true</td>
</tr>
</tbody>
</table>

**Note:** ($\neq \text{some}$) same as in ($\text{some}$) not the same as not in

### Definition of ALL Clause

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$5 &lt; \text{all}$</td>
<td>0, 5</td>
<td>false</td>
</tr>
<tr>
<td>$5 &lt; \text{all}$</td>
<td>0, 5</td>
<td>false</td>
</tr>
<tr>
<td>$5 = \text{all}$</td>
<td>0, 5</td>
<td>true</td>
</tr>
<tr>
<td>$5 = \text{all}$</td>
<td>0, 5</td>
<td>true</td>
</tr>
<tr>
<td>$5 \neq \text{all}$</td>
<td>0, 5</td>
<td>true</td>
</tr>
</tbody>
</table>

**Note:** ($\neq \text{all}$) same as not in
Nested Subqueries – Examples /4

- Find the names of all branches that have greater assets than all branches located in Brooklyn.
  
  ```sql
  select branch_name
  from branch
  where assets > all (select assets
                      from branch
                      where branch_city = 'Brooklyn')
  ```

Test for Empty Relations

- The `exists` construct returns the value `true` if the argument subquery is nonempty.

  - `exists r` equivalent to `r ≠ Ø`
  
  - `not exists r` equivalent to `r = Ø`

Nested Subqueries – Examples /5

- relation A contains relation B is equivalent to `exists (B except A)`

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

  ```sql
  select distinct S.customer_name
  from depositor as S
  where not exists ( (select branch_name
                      from branch
                      where branch_city = 'Brooklyn')
                   except (select R.branch_name
                           from depositor as T, account as R
                           where T.account_number = R.account_number
                           and S.customer_name = T.customer_name ) )
  ```

Relation Schema Example

- `Account (account_number, branch_name, balance)`
- `Branch (branch_name, branch_city, assets)`
- `Customer (customer_name, customer_street, customer_city)`
  
  - For simplicity assume `customer_name` unique
  
- `Depositor (customer_name, account_number)`
- `Loan (loan_number, branch_name, amount)`
- `Borrower (customer_name, loan_number)`
SQL Pattern Matching

LIKE:  
- percent (%) matches any substring
- underscore (_) matches any single character

Examples:
- Must end in 'base'  → like
- Must have at least two chars → like
- Must have at least two 'a's  → like
- Must include 'taba'  → like
- Must end in se end start in da  → like
- Must have exactly 8 chars  → like
- Must start in 'data'  → like

SQL Pattern Matching SOLUTIONS

LIKE:  
- percent (%) matches any substring
- underscore (_) matches any single character

Examples:
- Must end in 'base'  → like 'base'
- Must have at least two chars → like '_%
- Must have at least two 'a's  → like 'a%
- Must include 'taba'  → like 'taba'
- Must end in se end start in da  → like 'da%se'
- Must have exactly 8 chars  → like '______
- Must start in 'data'  → like 'data'

So Far

- Select, From, Where
- Rename
- Like
- Order-by
- Union, Intersect, Except
- Aggregation  
  - avg(), min(), max(), sum(), count()
- Group-by, Having
- Null Values
- Nested Subqueries  
  - A subquery is a select-from-where expression that is included within another query

Nested Subqueries

- New operators:
  - in, not in
  - >some, = >some, <=some, =some, >=some
  - >all, = >all, <=all, =all, >=all
  - exists, not exists

  Example:
  - Find all customers who have a loan at the bank but do not have an account at the bank
  - select distinct customer_name from borrower
  - where customer_name not in (select customer_name from depositor)
Test for Empty Relations

- The `exists` construct returns the value **true** if the argument subquery is nonempty.
  - `exists r` equivalent to `r ≠ Ø`
  - `not exists r` equivalent to `r = Ø`

- The following are equivalent:
  - relation A contains relation B
  - `(B − A) = Ø`
  - `not exists (B except A)`

Test for Absence of Duplicate Tuples

- The `unique` construct tests whether a subquery has any duplicate tuples in its result (opposite: `not unique`)

- Find all customers who have at most one account at the Perryridge branch

```
select T.customer_name
from depositor as T
where not unique (select R.customer_name
                         from account, depositor as R
                         where T.customer_name = R.customer_name
                           and R.account_number = account.account_number
                           and account.branch_name = 'Perryridge')
```

Views

- `create view` `v` as `<query expression>`
- `drop view` `v`

- Examples
  - `create view Oakland_accounts as
    select * from accounts
    where branch_name = 'Oakland'`

- Find all accounts in Oakland whose balance is over $200

```
select * from Oakland_accounts
where balance > 200
```

- `create view loan_info as
  select * from borrower, loan
  where borrower.account_number = loan.loan_number`
### View Examples

- A view consisting of branches and their customers

  ```sql
  create view all_customer 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_customer
  where branch_name = 'Perryridge'
  ```

### View Examples (cont)

- Explicitly define attribute names

  ```sql
  create view branch_loan_totals (branch_name, total_loan) as
  (select branch_name, sum(amount)
  from loan
  group-by branch_name)
  ```

  - Schema:
    - Loan (loan_number, branch_name, amount)
    - Borrower (customer_name, loan_number)

  ```sql
  create view loan_info_short as
  select customer_name, amount
  from borrower, loan
  where borrower.loan_number = loan.loan_number
  ```

### Complex Queries

- Find the average account balance of those branches where the average account balance is > $1200

  ```sql
  select branch_name, avg(balance)
  from account
  group by branch_name
  having avg(balance) > 1200
  ```

- Find the average account balance of those branches where the average account balance is > $1200

  ```sql
  select branch_name, avg_balance
  from (select branch_name, avg(balance)
  from account
  group by branch_name)
  as result (branch_name, avg_balance)
  where avg_balance > 1200
  ```

### So Far

- Basic SQL
- Rename, Like, Order-by
- Union, Intersect, Except
- Aggregation
  - avg(), min(), max(), sum(), count()
- Group-by, Having
- Null Values
- Nested Subqueries
- Views
- Complex Queries
Roadmap

- Database Modification
  - Deletions
  - Insertions
  - Updates
- Transactions
- Joined Relations

SQL – Deletion

- delete from R
  - where P
    - R is relation name
    - P is predicate

- Delete all accounts in the Perryridge Branch
  - delete from account
    - where branch_name = 'Perryridge'

- Delete all loans with amounts between $1300 and $1700
  - delete from loan
    - where amount between 1300 and 1700

SQL – Deletion /2

- Delete all accounts at every branch located in Pittsburgh
  - delete from account
    - where branch_name in
      - (select branch_name
         - from branch
         - where branch_city = 'Pittsburgh')

SQL – Deletion /3

- Delete all accounts with balances below the average
  - 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)
SQL – 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')
  ```
  or equivalently:
  ```sql
  insert into account (balance, branch_name, account_number)
  values (1200, 'Perryridge', 'A-9732')
  ```

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

SQL – Insertion /2

- Provide as a gift for all loan customers of the Perryridge branch, a $200 savings account
  ```sql
  insert into account
  values (loan_number, branch_name, 200)
  ```
  Let the loan number = account number for the savings account

- ```sql
  insert into account
  values (loan_number, branch_name, balance)
  ```
  where branch_name = 'Perryridge'

- ```sql
  insert into depositor
  select customer_name, loan_number
  from loan, borrower
  where branch_name = 'Perryridge' and loan.loan_number = borrower.loan_number
  ```

SQL – Insertion /3

- The select-from-where statement is fully evaluated before any of its results are inserted into the relation

- Otherwise, queries like
  ```sql
  insert into table1
  select *
  from table1
  ```
  would cause problems (what?)

SQL – Updates

- Increase all account balances by 5%
  ```sql
  update account
  set balance = balance * 1.05
  ```

- Increase all account balances by 6% if balance over 500
  ```sql
  update account
  set balance = balance * 1.06
  where balance > 500
  ```

- Increase all account balances by 7% if balance over avg
  ```sql
  update account
  set balance = balance * 1.07
  where balance > select avg (balance)
  from account
  ```
SQL – Updates /2

- Increase all accounts with balances over $10,000 by 6%, all other accounts receive 5%.
  - Write two update statements:
    
    ```
    update account
    set balance = balance * 1.06
    where balance > 10000
    
    update account
    set balance = balance * 1.05
    where balance <= 10000
    ```

- The order is important. Why?
- Can be done better using the case statement (next)

SQL – Updates / Case Statement

- Same query as before:
  Increase all accounts with balances over $10,000 by 6%, all other accounts receive 5%.
  
  ```
  update account
  set balance = case
  when balance <= 10000 then balance * 1.05
  else balance * 1.06
  end
  ```

SQL Updates / Views

- Create a view of all loan data in loan relation, hiding the amount:
  
  ```
  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 tuple
  
  ```
  ('L-307', 'Perryridge', null)
  ```

  into the loan relation

- Updates on more complex views are difficult or impossible to translate, and hence are disallowed.

Roadmap

- Database Modification
  - Deletions
  - Insertions
  - Updates

- Transactions

- Joined Relations
SQL – Transactions

A transaction is a sequence of queries and update statements executed as a single unit
- Started implicitly
- Terminated by:
  - Commit – makes all updates permanent
  - Rollback – undoes all updates performed by transaction

Motivating Example:
- Transfer money from account A to account B
  - Deduct $x from A
  - Credit $x to B
- If one of two steps fail → database is in inconsistent state
- Either both steps should succeed or none

SQL – 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

Another option in SQL:1999
- enclose statements within begin atomic ...
  end

Roadmap

- Database Modification
  - Deletions
  - Insertions
  - Updates

- Transactions

- Joined Relations

Joined Relations

- Join operation
  - take two relations and return as a result another relation
  - typically used as subquery expressions in from clause
- Join condition
  - defines which/how tuples in the two relations match
  - defines what attributes are present in the result of the join
- Join type
  - defines how to treat tuples that do not match

<table>
<thead>
<tr>
<th>Join Types</th>
<th>Join Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>inner join</td>
<td>natural</td>
</tr>
<tr>
<td>left outer join</td>
<td>on &lt;predicate&gt;</td>
</tr>
<tr>
<td>right outer join</td>
<td>using (A1, A2, ..., An)</td>
</tr>
<tr>
<td>full outer join</td>
<td></td>
</tr>
</tbody>
</table>
Joined Relations – II

- on <predicate>
  - join condition = predicate
  - schema includes all attributes (left relation first, right second)
  - SQL does not require attribute names to be unique

- natural join
  - natural join (from Relational Algebra) on all common attributes
  - schema:
    - common attributes first (same order as in left relation)
    - remaining attributes from left relation
    - remaining attributes from right relation

- using (<attribute_list>)
  - Same as natural join, BUT focus only on attributes in attribute_list (must be common and appear once in result)

Example Datasets

- Loan

<table>
<thead>
<tr>
<th>loan_number</th>
<th>branch_name</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-170</td>
<td>Downtown</td>
<td>3000</td>
</tr>
<tr>
<td>L-230</td>
<td>Redwood</td>
<td>4000</td>
</tr>
<tr>
<td>L-260</td>
<td>Perryridge</td>
<td>1700</td>
</tr>
</tbody>
</table>

- Borrower

<table>
<thead>
<tr>
<th>customer_name</th>
<th>loan_number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jones</td>
<td>L-170</td>
</tr>
<tr>
<td>Smith</td>
<td>L-230</td>
</tr>
<tr>
<td>Hayes</td>
<td>L-155</td>
</tr>
</tbody>
</table>

Note: borrower information missing for L-260 and loan information missing for L-155

Joined Relations – Examples

- loan inner join borrower on
  loan.loan_number = borrower.loan_number

<table>
<thead>
<tr>
<th>loan_number</th>
<th>branch_name</th>
<th>amount</th>
<th>customer_name</th>
<th>loan_number</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-170</td>
<td>Downtown</td>
<td>3000</td>
<td>Jones</td>
<td>L-170</td>
</tr>
<tr>
<td>L-230</td>
<td>Redwood</td>
<td>4000</td>
<td>Smith</td>
<td>L-230</td>
</tr>
</tbody>
</table>

- loan left outer join borrower on
  loan.loan_number = borrower.loan_number

<table>
<thead>
<tr>
<th>loan_number</th>
<th>branch_name</th>
<th>amount</th>
<th>customer_name</th>
<th>loan_number</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-170</td>
<td>Downtown</td>
<td>3000</td>
<td>Jones</td>
<td>L-170</td>
</tr>
<tr>
<td>L-230</td>
<td>Redwood</td>
<td>4000</td>
<td>Smith</td>
<td>L-230</td>
</tr>
<tr>
<td>L-260</td>
<td>Perryridge</td>
<td>1700</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>

- loan natural inner join borrower

<table>
<thead>
<tr>
<th>loan_number</th>
<th>branch_name</th>
<th>amount</th>
<th>customer_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-170</td>
<td>Downtown</td>
<td>3000</td>
<td>Jones</td>
</tr>
<tr>
<td>L-230</td>
<td>Redwood</td>
<td>4000</td>
<td>Smith</td>
</tr>
</tbody>
</table>

- loan natural right outer join borrower

<table>
<thead>
<tr>
<th>loan_number</th>
<th>branch_name</th>
<th>amount</th>
<th>customer_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-170</td>
<td>Downtown</td>
<td>3000</td>
<td>Jones</td>
</tr>
<tr>
<td>L-230</td>
<td>Redwood</td>
<td>4000</td>
<td>Smith</td>
</tr>
<tr>
<td>L-155</td>
<td>null</td>
<td>null</td>
<td>Hayes</td>
</tr>
</tbody>
</table>
**Joined Relations – Examples**

\[
\text{loan full outer join borrower using (loan_number)}
\]

<table>
<thead>
<tr>
<th>loan_number</th>
<th>branch_name</th>
<th>amount</th>
<th>customer_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-170</td>
<td>Downtown</td>
<td>3000</td>
<td>Jones</td>
</tr>
<tr>
<td>L-230</td>
<td>Redwood</td>
<td>4000</td>
<td>Smith</td>
</tr>
<tr>
<td>L-260</td>
<td>Perryridge</td>
<td>1700</td>
<td>null</td>
</tr>
<tr>
<td>L-155</td>
<td>null</td>
<td>null</td>
<td>Hayes</td>
</tr>
</tbody>
</table>

- Find all customers who have either an account or a loan (but not both) at the bank:

\[
\text{select customer_name from (depositor natural full outer join borrower)}
\text{where account_number is null or loan_number is null}
\]

**Roadmap**

- Joined Relations

**SQL – Data Definition Language**

- Domain Types and Definitions
- Domain Constraints
- Referential Integrity
- Schema Changes

**Data Definition Language (DDL)**

- Allows the specification of a set of relations and also information about each relation, including:
  - The schema for each relation.
  - The domain of values associated with each attribute.
  - Integrity constraints
  - The set of indices to be maintained for each relation.
  - Security and authorization information for each relation.
  - The physical storage structure of each relation on disk

**Domain Types in SQL**

- \text{char(n)}: fixed length character string, with user-specified length \( n \)
- \text{varchar(n)}: Variable length character strings, with user-specified maximum length \( n \).
- \text{int}: Integer (finite subset of the integers / machine-dependent).
- \text{smallint}: Small integer (a machine-dependent subset of the integer domain type).
- \text{numeric(p,n)}: Fixed point number, with user-specified precision of \( p \) digits, with \( n \) digits to the right of decimal point.
- \text{real, double precision}: Floating point and double-precision floating point numbers, with machine-dependent precision.
- \text{float(n)}: Floating point number, with user-specified precision of at least \( n \) digits.
Null values are allowed in all the domain types
- Declaring an attribute to be `not null` prohibits null values for that attribute.

**create domain** construct in SQL-92 creates user-defined domain types

Examples:
- `create domain person_name char(25) not null`
- `create domain customer_city varchar(50)`

Date/Time Types in SQL
- **date.** Dates, containing a (4 digit) year, month and date
  - E.g. `date '2001-7-27'`
- **time.** Time of day, in hours, minutes and seconds.
  - E.g. `time '09:00:30'` `time '09:00:30.75'`
- **timestamp:** date plus time of day
  - E.g. `timestamp '2001-7-27 09:00:30.75'`
- **Interval:** period of time
  - E.g. Interval ‘1’ day
  - Subtracting a date/time/timestamp value from another gives an interval value
  - Interval values can be added to date/time/timestamp values

Date/Time Types in SQL – II
- Operations on Date Types:
  - Datetime (+ or –) Interval = Datetime
  - Datetime – Datetime = Interval
  - Interval (* or /) Number = Interval
  - Interval (+ or –) Interval = Interval

  - Can compare data/time/timestamp values
  - Can extract values of individual fields from date/time/timestamp
    - E.g. `extract (year from r.start_time)`
  - Can cast string types to date/time/timestamp
    - E.g. `cast <string-valued-expression> as date`

Create Table Construct
- An SQL relation is defined using the **create table** command:
  ```sql
  create table r (A_1 D_1, A_2 D_2, ..., A_n D_n, (integrity-constraint_1), ..., (integrity-constraint_k))
  ```
  - `r` is the name of the relation
  - each `A_i` is an attribute name in the schema of relation `r`
  - `D_i` is the data type of values in the domain of attribute `A_i`

  - Example:
    ```sql
    create table branch
    (branch_name char(15) not null, branch_city char(30), assets integer)
    ```
Roadmap

- Joined Relations
- SQL – Data Definition Language
  - Domain Types and Definitions
  - Domain Constraints
  - Referential Integrity
  - Schema Changes

Domain Constraints

- The most elementary form of integrity constraint.
- They test values inserted in the database, and test queries to ensure that the comparisons make sense.
- New domains can be created from existing data types
  - E.g. `create domain Dollars numeric(12, 2)`
  - `create domain Pounds numeric(12, 2)`
- We cannot assign or compare a value of type Dollars to a value of type Pounds.
  - However, we can convert type:
    - `(cast r.A as Pounds)`
    - (Should also multiply by the dollar-to-pound conversion-rate)

Domain Constraints – check clause

- `check` clause in SQL permits domains to be restricted:

Examples:
- Use `check` clause to ensure that an hourly-wage domain allows only values greater than a specified value.
  - `create domain hourly-wage numeric(5,2) constraint value-test check(value > = 4.00)`
  - The domain has a constraint that ensures that the hourly-wage is greater than 4.00
  - The clause `constraint value-test` is optional; useful to indicate which constraint an update violated.

Domain Constraints – check clause 2

- Can have complex conditions in domain check

Examples:
- Make sure account is either checking or savings
  - `create domain AccountType char(10) constraint account-type-test check (value in ('Checking', 'Savings'))`
  - Make sure branch-name is valid one
  - `check (branch-name in (select branch-name from branch))`
Roadmap

- Joined Relations
- SQL – Data Definition Language
  - Domain Types and Definitions
  - Domain Constraints
  - Referential Integrity
  - Schema Changes

Referential Integrity

- Ensures that a value that appears in one relation for a given set of attributes, also appears for a certain set of attributes in another relation.

Examples:
- If "Perryridge" is a branch name appearing in one of the tuples in the account relation, then there exists a tuple in the branch relation for branch "Perryridge".
- If faculty_id "12345678" appears in one of the tuples in the teaches relation, then there exists a tuple in the faculty relation for faculty with id="12345678".

Referential Integrity – Definition

- Formal Definition
  - Let \( r_1(R_1) \) and \( r_2(R_2) \) be relations with primary keys \( K_1 \) and \( K_2 \) respectively.
  - The subset \( \alpha \) of \( R_2 \) is a foreign key referencing \( K_1 \) in relation \( r_1 \), if for every \( t_2 \) in \( r_2 \) there must be a tuple \( t_1 \) in \( r_1 \) such that \( t_1[K_1] = t_2[\alpha] \).
  - Referential integrity constraint also called subset dependency since its can be written as \( \Pi_{\alpha}(r_2) \subseteq \Pi_{K_1}(r_1) \).

RI on Database Modifications

- \( \alpha \) of \( R_2 \) is a foreign key referencing \( K_1 \) in relation \( r_1 \).
- Insert. If a tuple \( t_2 \) is inserted into \( r_2 \), we must ensure that there is a tuple \( t_1 \) in \( r_1 \) such that \( t_1[K] = t_2[\alpha] \).
  - \( t_2[\alpha] \in \Pi_{K}(r_1) \)
- Delete. If a tuple, \( t_1 \) is deleted from \( r_1 \), we must compute the set of tuples in \( r_2 \) that reference \( t_1 \):
  - \( \sigma_{\alpha - t_1[K]}(r_2) \)
  - If this set is not empty
    - either the delete command is rejected as an error, or
    - the tuples that reference \( t_1 \) must themselves be deleted (cascading deletions are possible).
RI on Updates

- α of R₂ is a foreign key referencing K₁ in relation r₁

- If a tuple t₂ is updated in relation r₂ and the update modifies values for foreign key α, then a test similar to the insert case is made:
  - Let t₂' denote the new value of tuple t₂. We must ensure that t₂'[α] ∈ \prod_{K}(r₁)

RI on Updates (2)

- α of R₂ is a foreign key referencing K₁ in relation r₁

- If a tuple t₁ is updated in r₁, and the update modifies values for the primary key (K), then a test similar to the delete case is made:
  - We must compute αₜ = \prod_{K}(r₁)
  - using the old value of t₁ (the value before the update is applied).
  - If this set is not empty
    1. the update may be rejected as an error, or
    2. the update may be cascaded to the tuples in the set, or
    3. the tuples in the set may be deleted.

Referential Integrity in SQL

- Primary, candidate and foreign keys can be specified as part of the SQL create table statement:
  - primary key clause:
    - list attributes that comprise the primary key.
  - unique key clause:
    - list attributes that comprise a candidate key.
  - foreign key clause:
    - list attributes that comprise the foreign key and specify name of the relation referenced by the foreign key.
  - By default, a foreign key references the primary key attributes of the referenced table
  - Short form for specifying a single column as foreign key

RI – SQL Example

```sql
create table customer
(customer-name char(20),
customer-street char(30),
customer-city char(30),
primary key (customer-name))
```

```sql
create table branch
(branch-name char(15),
branch-city char(30),
assets integer,
primary key (branch-name))
```
RI – SQL Example (cont)

`create table account
(account-number char(10),
branch-name char(15),
balance integer,
primary key (account-number),
foreign key (branch-name) references branch)`

`create table depositor
(customer-name char(20),
account-number char(10),
primary key (customer-name, account-number),
foreign key (account-number) references account,
foreign key (customer-name) references customer)`

Cascading Actions in SQL

`create table account
...`  
`foreign key(branch-name) references branch`  
`on delete cascade`  
`on update cascade`  
`...`  

- **on delete cascade**
  - ➔ if a delete of a tuple in `branch` results in referential-integrity constraint violation, the delete “cascades” to the `account` relation, deleting the tuple that refers to the branch that was deleted.

- Cascading updates are similar.

Referential Integrity in SQL (Cont.)

- Alternative to cascading:
  - on delete set null
  - on delete set default

- Null values in foreign key attributes complicate SQL referential integrity semantics, and are best prevented using **not null**
  - if any attribute of a foreign key is null, the tuple is defined to satisfy the foreign key constraint!

Roadmap

- Joined Relations
- **SQL – Data Definition Language**
  - Domain Types and Definitions
  - Domain Constraints
  - Referential Integrity
  - Schema Changes
### Drop Table Construct

- The **drop table** command deletes all information about the dropped relation from the database.
  - E.g., `drop table customer`
  - delete all tuples and schema information

- Note difference from **delete**
  - E.g., `delete from customer`
  - all tuples are deleted, but schema information remains

- Note difference from **drop view**
  - E.g., `drop view all-customer`
  - no tuple is deleted, but view definition is deleted

### Alter Table Construct

- The **alter table** command is used to add attributes to an existing relation.
  - `alter table r add A D`
  where `A` is the name of the attribute to be added to relation `r` and `D` is the domain of `A`.
  - All tuples in the relation are assigned `null` as the value for the new attribute.

- The **alter table** command can also be used to drop attributes of a relation
  - `alter table r drop A`
  where `A` is the name of an attribute of relation `r`
  - Dropping of attributes not supported by many databases

### Integrity Constraints & Security

**Integrity Constraints**
- guard against accidental damage to the database
- we focus on constraints that can be tested with minimal overhead

**Security**
- prevent unauthorized access to data (read or write)
Triggers

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

- To design a trigger mechanism, we must:
  - Specify the conditions under which the trigger is to be executed.
  - Specify the actions to be taken when the trigger executes.

- Event-Condition-Action model for triggers

Trigger Example

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

- The condition for executing the trigger is an update to the **account** relation that results in a negative **balance** value.

Trigger Example in SQL

```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 (n.row.account-number, nrow.branch-name, - nrow.balance);
    update account set balance = 0
    where account.account-number = nrow.account-number
end
```

Triggering Events and Actions in SQL

- Triggering event can be **insert**, **delete** or **update**

- Triggers on update can be restricted to specific attributes
  - `create trigger overdraft-trigger after update of balance on account`

- Values of attributes before and after an update can be referenced
  - `referencing old row as :` for deletes and updates
  - `referencing new row as :` for inserts and updates

- Triggers can be activated before an event, which can serve as extra constraints. E.g. convert blanks to null.
  ```sql
  create trigger setnull-trigger before update on r
  referencing new row as nrow
  for each row
  when nrow.phone-number = '
  update r set nrow.phone-number = null
  ```
Statement Level Triggers

- Instead of executing a separate action for each affected row, a single action can be executed for all rows affected by a transaction.

  - Use `for each statement` instead of `for each row`.
  - Use `referencing old table` or `referencing new table` to refer to temporary tables (called transition tables) containing the affected rows.
  - Can be more efficient when dealing with SQL statements that update a large number of rows.

Triggers for External World Actions

- We sometimes require external world actions to be triggered on a database update.
- E.g., re-ordering an item whose quantity in a warehouse has become small, or turning on an alarm light.
- Triggers cannot be used to directly implement external-world actions, BUT:
- Triggers can be used to record actions-to-be-taken in a separate table.
- Have an external process that repeatedly scans the table, carries out external-world actions and deletes action from table.

External World Actions (cont.)

- E.g., suppose a warehouse has the following tables:
  - `inventory(item, level)`: How much of each item is in the warehouse.
  - `minlevel(item, level)`: What is the minimum desired level of each item.
  - `reorder(item, amount)`: What quantity should we re-order at a time.
  - `orders(item, amount)`: Orders to be placed (read by external process).

  ```sql
  create trigger reorder-trigger 
  after update of amount on inventory 
  referencing old row as orow, new row as nrow 
  for each row 
  when nrow.level <= (select level 
      from minlevel 
      where minlevel.item = orow.item) 
  and orow.level > (select level 
      from minlevel 
      where minlevel.item = orow.item) 
  begin 
      insert into orders 
      (select item, amount 
       from reorder 
       where reorder.item = orow.item) 
  end
  ```

External World Actions (Cont.)
### When Not To Use Triggers

- Triggers were used earlier for tasks such as:
  - maintaining summary data (e.g., total salary of each department)
  - Replicating databases by recording changes to special relations (called change or delta relations) and having a separate process that applies the changes over to a replica.

- There are better ways of doing these now:
  - Databases today provide built-in materialized view facilities to maintain summary data.
  - Databases provide built-in support for replication.
  - Encapsulation facilities can be used instead of triggers in many cases:
    - Define methods to update fields.
    - Carry out actions as part of the update methods instead of through a trigger.

### Roadmap

- **Triggers**
- **Security**
  - Authorization
  - Authorization in SQL

### Security

- **Security**: protection from malicious attempts to steal or modify data.

- Database system level
- Operating system level
- Network level
- Physical level
- Human level

### Security (cont.)

- **Database system level**
  - Authentication and authorization mechanisms to allow specific users access only to required data.
  - We concentrate on authorization in the rest of this chapter.

- **Operating system level**
  - Operating system super-users can do anything they want to the database! Good operating system level security is required.

- **Network level**: must use encryption to prevent eavesdropping (unauthorized reading of messages) and masquerading (pretending to be an authorized user or sending messages supposedly from authorized users).
Security (cont.)

- Physical level
  - Physical access to computers allows destruction of data by intruders; traditional lock-and-key security is needed
  - Computers must also be protected from floods, fire, etc.

- Human level
  - Users must be screened to ensure that an authorized users do not give access to intruders
  - Users should be trained on password selection and secrecy

Roadmap

- Triggers
- Security
- Authorization
- Authorization in SQL

Authorization

Forms of authorization on parts of the database:

- **Read authorization** - allows reading, but not modification of data.
- **Insert authorization** - allows insertion of new data, but not modification of existing data.
- **Update authorization** - allows modification, but not deletion of data.
- **Delete authorization** - allows deletion of data

Authorization (cont.)

Forms of authorization to modify the database schema:

- **Index authorization** - allows creation and deletion of indices.
- **Resources authorization** - allows creation of new relations.
- **Alteration authorization** - allows addition or deletion of attributes in a relation.
- **Drop authorization** - allows deletion of relations.
Authorization and Views

- Users can be given authorization on views, without being given any authorization on the relations used in the view definition.

- Ability of views to hide data serves both to:
  - Simplify usage of the system, and
  - Enhance security by allowing users access only to data they need for their job.

- A combination or relational-level security and view-level security can be used to limit a user’s access to precisely the data that he/she needs.

View Example

- Suppose a bank clerk needs to know the names of the customers of each branch, but is not authorized to see specific loan information.

- Approach: Deny direct access to the loan relation, but grant access to the view cust-loan, which consists only of the names of customers and the branches at which they have a loan.

- The cust-loan view is defined in SQL as follows:

  ```sql
  create view cust-loan as
  select branchname, customer-name
  from borrower, loan
  where borrower.loan-number = loan.loan-number
  ```

- The clerk is authorized to see the result of the query:

  ```sql
  select * from cust-loan
  ```

  When the query processor translates the result into a query on the actual relations in the database, we obtain a query on borrower and loan.

  Authorization must be checked on the clerk’s query before query processing replaces a view by the definition of the view.

Authorization on Views

- Creation of view does not require resources authorization since no real relation is being created.

- Creator of a view gets only those privileges that provide no additional authorization beyond that she already had.

- Example:

  If creator of view cust-loan had only read authorization on borrower and loan, he gets only read authorization on cust-loan.
Granting of Privileges

- The passage of authorization from one user to another may be represented by an authorization graph.
  - The nodes of this graph are the users.
  - The root of the graph is the database administrator.

Consider graph for update authorization on loan.

An edge $U_i \rightarrow U_j$ indicates that user $U_i$ has granted update authorization on loan to $U_j$.

Authorization Grant Graph

- **Requirement**: All edges in an authorization graph must be part of some path originating with the database administrator
  - If DBA revokes grant from $U_1$:
    - Grant must be revoked from $U_4$ since $U_1$ no longer has authorization
    - Grant must not be revoked from $U_5$ since $U_5$ has another authorization path from DBA through $U_2$
  - Must prevent cycles of grants with no path from the root:
    - DBA grants authorization to $U_7$
    - $U_7$ grants authorization to $U_8$
    - $U_8$ grants authorization to $U_7$
    - DBA revokes authorization from $U_7$
    - Must revoke grant $U_7$ to $U_8$ and from $U_8$ to $U_7$ since there is no path from DBA to $U_7$ or to $U_8$ anymore.

Roadmap

- Triggers
- Security
- Authorization
- **Authorization in SQL**

Security Specification in SQL

- The grant statement is used to confer authorization
  - `grant <privilege list>` on `<relation name or view name>` to `<user list>`
  - `<user list>` is:
    - a user-id
    - `public`, which allows all valid users the privilege granted
    - A role (more on this later)
- Granting a privilege on a view does not imply granting any privileges on the underlying relations.
- The grantor of the privilege must already hold the privilege on the specified item (or be the database administrator).
Privileges in SQL

- **select**: allows read access to relation, or the ability to query using the view.
  
  Example: grant users \( U_1, U_2 \), and \( U_3 \) select authorization on the branch relation:
  
  ```sql
  grant select on branch to U_1, U_2, U_3
  ```

- **insert**: the ability to insert tuples.

- **update**: the ability to update using the SQL update statement.

- **delete**: the ability to delete tuples.

- **references**: ability to declare foreign keys when creating relations.

- **all privileges**: used as a short form for all the allowable privileges.

Privilege To Grant Privileges

- **with grant option**: allows a user who is granted a privilege to pass the privilege on to other users.

  Example:
  
  ```sql
  grant select on branch to U_i with grant option
  ```

  gives \( U_i \) the select privileges on branch and allows \( U_i \) to grant this privilege to others.

Roles

- Roles permit common privileges for a class of users can be specified just once by creating a corresponding "role".

- Privileges can be granted to or revoked from roles, just like user.

- Roles can be assigned to users, and even to other roles.

Roles Example

```sql
create role teller
create role manager

grant select on branch to teller
grant update (balance) on account to teller
grant all privileges on account to manager
grant teller to manager
grant teller to alice, bob
grant manager to avi
```
Revoking Authorization in SQL

- The `revoke` statement is used to revoke authorization.
  
  ```sql
  revoke <privilege list>
on <relation name or view name> from <user list>
  [restrict|cascade]
  ```

- Example:
  ```sql
  revoke select on branch from U1, U2, U3 cascade
  ```

- Revocation of a privilege from a user may cause other users also to lose that privilege
  - referred to as cascading of the `revoke`.

- We can prevent cascading by specifying `restrict`:
  ```sql
  revoke select on branch from U1, U2, U3 restrict
  ```

- With `restrict`, the `revoke` command fails if cascading revokes are required.

---

Revoking Authorization in SQL (cont)

- `<privilege-list>` may be all to revoke all privileges the revokee may hold.

- If `<revokee-list>` includes public all users lose the privilege except those granted it explicitly.

- If the same privilege was granted twice to the same user by different grantees, the user may retain the privilege after the revocation.

- All privileges that depend on the privilege being revoked are also revoked.

---

Limitations of SQL Authorization

- SQL does not support authorization at a tuple level
  - E.g. we cannot restrict students to see only (the tuples storing) their own grades
  - With the growth in Web access to databases, database accesses come primarily from application servers.
    - End users don't have database user ids, they are all mapped to the same database user id
  - All end-users of an application (such as a web application) may be mapped to a single database user
  - The task of authorization in above cases falls on the application program, with no support from SQL
    - Benefit: fine grained authorizations, such as to individual tuples, can be implemented by the application.
    - Drawback: Authorization must be done in application code, and may be dispersed all over an application
    - Checking for absence of authorization loopholes becomes very difficult since it requires reading large amounts of application code

---

Audit Trails

- An audit trail is a log of all changes (inserts/deletes/updates) to the database along with information such as which user performed the change, and when the change was performed.

- Used to track erroneous/fraudulent updates.

- Can be implemented using triggers, but many database systems provide direct support.
Encryption

- Data may be *encrypted* when database authorization provisions do not offer sufficient protection.

- Properties of good encryption technique:
  - Relatively simple for authorized users to encrypt and decrypt data.
  - Encryption scheme depends not on the secrecy of the algorithm but on the secrecy of a parameter of the algorithm called the encryption key.
  - Extremely difficult for an intruder to determine the encryption key.