SQL: Outline

- Brief History
- Data Definition Language
- Data Manipulation Language
- Basic SQL Queries
- More Powerful SQL Queries
- Constraints in SQL

History of SQL

- *SEQUEL*: Structured English Query Language
  - 23-page research report in 1974 from IBM San Jose Research Labs
  - System R prototype
- *SEQUEL 2*: A Unified Approach to Data Definition, Manipulation and Control, IBM Systems Journal, 1976
- *SQL*: Structured Query Language (IBM's DB2 in 1983)

Standardization of SQL

- SQL-86
  - ISO 9075-1987, “Database Language SQL”
  - A somewhat cleaned up version of DB2 SQL
- SQL-89
  - Added embedded SQL
  - 150 pages
- SQL-92 (also called SQL2)
  - ISO/IEC 9075:1992
  - Many new DDL and DML features
  - 500 pages
- SQL:1999/SQL:2003 (also called SQL3)
  - Object-oriented features
  - Fifteen parts
  - ~5000 pages

Levels of SQL-92

- Entry SQL
  - SQL-89 with some small changes
- Intermediate SQL
  - Approximately half of the new features of SQL-92
  - Expected that initial implementations will be at this level.
- Full SQL
  - All features in the standard
  - Few DBMS supports this level even today
- Commercial systems offer most, if not all, SQL-92 features, plus varying feature sets from later standards and special proprietary features.
  - Not all examples here may work on your particular system.

SQL: Outline

- Brief History
- Data Definition Language
- Data Manipulation Language
- Basic SQL Queries
- More Powerful SQL Queries
- Constraints in SQL

SQL's Data Model

- Tables are multi-sets (bags) of rows.
- Each row of a table has the same columns, over the same domains.
- Duplicates are allowed
  - unless explicitly disallowed via **`UNIQUE`**.
- Order in which tuples are stored cannot be specified
  - But some DBMS vendors allow this in various ways

Domain Types in SQL

- **char(n).** Fixed length character string, with user-specified length *n*.
- **varchar(n).** Variable length character strings, with user-specified maximum length *n*.
- **int.** Integer (a finite subset of the integers that is machine-dependent).
- **smallint.** Small integer (a machine-dependent subset of the integer domain type).
- **numeric(p,d).** Fixed point number, with user-specified precision of *p* digits, with *n* digits to the right of decimal point.
- **real, double precision.** Floating point and double-precision floating point numbers, with machine-dependent precision.
- **float(n).** Floating point number, with user-specified precision of at least *n* digits.
- More are covered in Chapter 4.

Data Definition in SQL

- Three statements are used to define the schema in SQL.
  - **CREATE**
  - **DROP**
  - **ALTER**
- These statements apply to
  - Tables (and indirectly columns)
  - Views
  - Domains
  - Assertions
  - Character sets
    - Internationalization of software, hard-work and boring
  - Translations (between character sets)

CREATE TABLE

- Specifies a new base table.
- Name
- Columns with
  - Name
  - Data type
  - Column constraints
  - Default value
- Table constraints
  - Primary and unique keys
  - Foreign keys

Video Store Example

```
CREATE TABLE Film (
            FilmID              NUMERIC (5),
            Title               VARCHAR (50),
            PubDate             DATE,
            RentalPrice         NUMERIC (5,2),
            Distributor         VARCHAR (20),
            Kind                CHAR (1),
            RecommededAge       NUMERIC (2),
            SpokenLanguage      VARCHAR (15),
            SubtitleLanguage    VARCHAR (15))

CREATE TABLE Reserves (
            CustomerID          NUMERIC (5),
            FilmID              NUMERIC (5),
            ResDate             DATE)

CREATE TABLE Customer (
            CustomerID          NUMERIC (5),
            Name                VARCHAR (50),
            Street              VARCHAR (50),
            City                VARCHAR (50),
            State               CHAR (2))
```

Column Defaults

- If not specified, the value of the column will be **NULL**.
- A particular value can also be specified.

```
CREATE TABLE Film (
            ...
    SpokenLanguage VARCHAR (15) DEFAULT 'English',
            ...
    )

CREATE TABLE Reserves (
            ...
    ResDate DATE DEFAULT CURRENT_DATE,
            ...
    )
```

Column Constraints

- **NOT NULL**

```
CREATE TABLE Film ( ...
                PubDate DATE NOT NULL, ...)
```

  - If no default is given, then **INSERT** must specify a value, otherwise the constraint will be violated.

- **UNIQUE**

```
CREATE TABLE Customer (...,
        ID NUMERIC (5) UNIQUE, ...)
```

  - Ensures that there are no duplicate tuples, with the same column value.
  - Can have one value that is **NULL**, unless **NOT NULL** is specified.

Column Constraints, cont.

- **PRIMARY KEY**

```
CREATE TABLE Film (
            FilmID NUMERIC (5) PRIMARY KEY,
            ...)
```

  - Implies **NOT NULL**, **UNIQUE** (entity integrity).

- **CHECK** (*predicate*)

```
CREATE TABLE Film ( ...,
    Kind CHAR(1) CHECK (Kind IN ('F','M','E')),
    ...)
```

  - Any predicate that may occur in an SQL where clause is allowed.
  - The predicate is assumed to range only over the row in which the constraint appears.

DROP TABLE

- Used to remove a table and its definition
- Once dropped, the table can no longer be used in queries, updates, or any other commands since its description no longer exists.
- Example: remove the file table created before

```
DROP TABLE Film
```

ALTER TABLE

- Used to
  - Add a column
  - Drop a column
  - Change a column's default
  - Add a constraint
  - Drop a constraint
- Example

```
ALTER TABLE Film ADD COLUMN PurchasePrice
                                NUMERIC (5,2)
```

  - The new attribute will have **NULL**s in all the tuples of the table immediately after the command is executed.
  - Hence, the **NOT NULL** constraint is not allowed for such an attribute, unless a **DEFAULT** is specified.
  - The database users must still enter a value for the new column `PurchasePrice` for each `Film` row.

SQL: Outline

- Brief History
- Data Definition Language
- Data Manipulation Language
- Basic SQL Queries
- More Powerful SQL Queries
- Constraints in SQL

Data Manipulation in SQL

- Basic SQL statements for data manipulation
  - **`SELECT`** - retrieve data from the a database
  - **`INSERT`** - insert data into a table
  - **`UPDATE`** - updates existing data within a table
  - **`DELETE`** - deletes all records from a table, but the space for the records remain
    - Different from DROP table

SQL: Outline

- Brief History
- Data Definition Language
- Data Manipulation Language
- Basic SQL Queries
  - SELECT-FROM-WHERE Statement
  - **`NULL`** Values
  - **`OEDER BY`**
  - **`GROUP BY`** and **`HAVING`**
- More Powerful SQL Queries
- Constraints in SQL

The Basic SQL Statement

- The clauses are specified in the following order.
  - **`SELECT`** *column(s)*
  - **`FROM`** *table list*
  - **`WHERE`** *condition*
  - **`GROUP BY`** *grouping column(s)*
  - **`HAVING`** *group condition*
  - **`ORDER BY`** *sort list*

Retrieval Queries in SQL

- SQL has one basic statement for retrieving information from a database; the **SELECT** statement.
- This is not the same as the select (σ) operation of the relational algebra.
- The basic form of the SQL **SELECT** statement is called a *mapping* or a *select-from-where block*.

```
SELECT column list
FROM   table list
WHERE  condition
```

- Equivalent to the following relational algebra formula.

$$\pi_{column\ list}(\sigma_{condition}(table_1 \times ... \times table_n))$$

Queries Over Several Tables

- List the IDs of the films that are expensive and have been reserved.

```
SELECT Film.FilmID
FROM   Film, Reserves
WHERE  RentalPrice > 4
       AND Film.FilmID = Reserves.FilmID
```

- List the customer names who have reserved a film.

```
SELECT DISTINCT Name
FROM   Customer, Reserves
WHERE  Reserves.CustomerID = Customer.CustomerID
```

Queries Over Several Tables, cont.

- List the customer names who have reserved an expensive film.

```
SELECT DISTINCT Name
FROM   Customer, Film, Reserves
WHERE  Reserves.CustomerID = Customer.CustomerID
       AND Reserves.FilmID = Film.FilmID
       AND RentalPrice > 4
```

- List the streets of customers who have reserved foreign films.

```
SELECT Street
FROM   Customer, Film, Reserves
WHERE  Reserves.CustomerID = Customer.CustomerID
       AND Reserves.FilmID = Film.FilmID
       AND Kind = 'F'
```

NULL Values

- **NULL** values are used if
  - a value is missing
  - a value is unknown
  - a value does not exist
  - ...

  (29 possible interpretations for **NULL** were identified)

- Operations and functions return **NULL** if one of their values in **NULL**.
  - Aggregates is an exception
    - they ignore **NULL** values

NULL Values, Cont.

- In predicates, **NULL** is handled with a three-valued logic approach, i.e., a logic with truth values TRUE, FALSE, and UNKNOWN.

Value of **X**	Condition	Result
NULL	**X IS NULL**	TRUE
NULL	**X IS NOT NULL**	FALSE
10	**X IS NULL**	FALSE
10	**X IS NOT NULL**	TRUE
NULL	**X = NULL**	UNKNOWN
NULL	**X <> NULL**	UNKNOWN
10	**X = NULL**	UNKNOWN
10	**X <> NULL**	UNKNOWN

ORDER BY

- Can sort the result of a select, using **ORDER BY**.
- Who has reserved a film?

```
SELECT Name
FROM   Customer, Reserves
WHERE  Customer.CustomerID = Reserves.CustomerID
ORDER BY Name
```

- Can also sort in descending order, via **DESC**
  - (**ASC** is the default).

```
SELECT Name
FROM   Customer, Reserves
WHERE  Customer.CustomerID = Reserves.CustomerID
ORDER BY Name DESC
```

- Only columns in the select list can be used for the ordering.

SELECT in the **FROM** Clause

- The table in a from clause can itself be a select statement.
- List the customers who have reserved an expensive film.

```
SELECT CustomerID
FROM   Reserves,(SELECT *
                 FROM   Film
                 WHERE  RentalPrice > 4) AS Expensive
WHERE  Reserves.FilmID = Expensive.FilmID
```

- A correlation name is required in such case
  - `Expensive` in this case
  - Use **AS** as the keyword of naming
    - **AS** should be removed on Oracle

Aggregates

- Aggregates operate on the set of values of a column of a table, and return a single value.
- SQL offers 5 built-in aggregate functions:
  - **SUM**: sum of values
  - **AVG**: average value
  - **MAX**: maximum value
  - **MIN**: minimum value
  - **COUNT**: number of values

- What is the total rental price of all films?

```
SELECT SUM (RentalPrice)
FROM   Film
```

Aggregates, cont.

- What is the average rental price of reserved films?

```
SELECT AVG(RentalPrice)
FROM   Reserves, Film
WHERE  Reserves.FilmID = Film.FilmID
```

- How many reservations are there?

```
SELECT COUNT (*)
FROM   Reserves
```

- **DISTINCT** can eliminate duplicates before computing the aggregate.
  - How many cities do customers reside in?

```
SELECT COUNT (DISTINCT City)
FROM   Customer
```

  - **DISTINCT** cannot be used with **COUNT** (*)

GROUP BY

- The aggregate can be applied to several groups by specifying **`GROUP BY`**.

- What is the average rental price of all films, by kind?

```
SELECT DISTINCT Kind, AVG(RentalPrice)
FROM  Film
GROUP BY Kind
```

Title	*Kind*	*RentalPrice*
Lethal Weapon 4	Action	3
Unforgiven	Western	4
Once upon a Time in the West	Western	3
Star Wars	Fiction	3
Rocky V	Action	2

Kind	*AvgRentalPrice*
Action	2.5
Western	3.5
Fiction	3

A Special GROUP BY Example

Films:

Title	*Kind*	*RentalPrice*
Lethal Weapon 4	Action	3
Unforgiven	Western	4
Once upon a Time in the West	Western	3
Star Wars	Fiction	3
Rocky V	Action	2

```
SELECT *
FROM Films
GROUP BY Kind
```

Result:

Title	*Kind*	*RentalPrice*
Lethal Weapon 4	Action	3
Rocky V	Action	2
Unforgiven	Western	4
Once upon a Time in the West	Western	3
Star Wars	Fiction	3

HAVING

- Individual groups can be eliminated by using **HAVING** that follows **GROUP BY**.
- List the customers whose average rental price for reserved films is greater than $3.

```
SELECT Name
FROM   Customer, Reserves, Film
WHERE  Customer.CustomerID = Reserves.CustomerID
       AND Reserves.FilmID = Film.FilmID
GROUP BY Name
HAVING AVG(RentalPrice) > 3
```

HAVING Example

Films:

Title	*Kind*	*RentalPrice*
Lethal Weapon 4	Action	3
Unforgiven	Western	4
Once upon a Time in the West	Western	3
Star Wars	Fiction	3
Rocky V	Action	2

```
FROM Films
GROUP BY Kind
HAVING Kind = 'Western'
OR     Kind = 'Action'
```

Result:

Title	*Kind*	*RentalPrice*
Lethal Weapon 4	Action	3
Rocky V	Action	2
Unforgiven	Western	4
Once upon a Time in the West	Western	3

HAVING Example (cont)

Films:

Title	*Kind*	*RentalPrice*
Lethal Weapon 4	Action	3
Unforgiven	Western	4
Once upon a Time in the West	Western	3
Star Wars	Fiction	3
Rocky V	Action	2

```
FROM Films
GROUP BY Kind
HAVING MAX(RentalPrice) <= 3
```

Result:

Title	*Kind*	*RentalPrice*
Lethal Weapon 4	Action	3
Rocky V	Action	2
Star Wars	Fiction	3

HAVING Example (cont)

Films:

Title	*Kind*	*RentalPrice*
Lethal Weapon 4	Action	3
Unforgiven	Western	4
Once upon a Time in the West	Western	3
Star Wars	Fiction	3
Rocky V	Action	2

SQL: Outline

- Brief History
- Data Definition Language
- Data Manipulation Language
- Basic SQL Queries
- More Powerful SQL Queries
  - Joins
  - Modifications
  - Subqueries
  - IN/ALL/ANY/EXISTS
  - Views
- Constraints in SQL

Joins

- Inner Joins
  - Cartesian product
  - Old style join
    - Specified by including several tables in the from clause.
    - Can also be specified directly: **`INNER JOIN`**.
  - Condition join
    - Specified by including a predicate in the where clause.
  - Cross join
  - Natural join
  - Column name join
- Outer Join
- Union Join

Cross Join

- The *cross join* is equivalent to a Cartesian product.
- List all the customer information, including all reservations.

```
SELECT *
FROM   Customer CROSS JOIN Reserves
```

- Note that the result will include too much information, since all reservations will be paired with all customers.
- This query is equivalent to the following.

```
SELECT Customer.*, Reserves.*
FROM   Customer, Reserves
```

Natural Join

- As with the relational algebra, the natural join in SQL eliminates duplicate columns.
- List all the customer information, and all their reservations.

```
SELECT *
FROM   Customer NATURAL JOIN Reserves
```

- This query retrieves the correct information.
- It is equivalent to the following.

```
SELECT Customer.*, FilmID, ResDate
FROM   Customer, Reserves
WHERE  Customer.CustomerID = Reserves.CustomerID
```

Column Name Join

- The *column name join* uses only some of the columns that a natural join would use in the equality test.
- The `Customer` and `Reserves` tables both include address information. Say we wish to join only on the `CustomerID` column, which is also present in both.

```
SELECT *
FROM   Customer JOIN Reserves USING (CustomerID)
```

Outer Join

- In the **FROM** list
  - **LEFT OUTER JOIN ON** *predicate*
  - **RIGHT OUTER JOIN ON** *predicate*
  - **FULL OUTER JOIN ON** *predicate*

- List the films, along with the customers who reserved each, if applicable.

```
SELECT Title, Name
FROM   Film LEFT OUTER JOIN Reserves
          ON Film.FilmID = Reserves.FilmID, Customer
WHERE  Customer.CustomerID = Reserves.CustomerID
```

Modifications

- There are three modification statements.
  - **INSERT**
  - **UPDATE**
  - **DELETE**
- For insertions, either values can be specified, or a select statement provides the values.
- Enter a reservation for Eric for the film 332244.

```
INSERT INTO Reserves
  VALUES (123456, 332244, CURRENT_DATE)
```

- Let Melanie reserve all the films that Eric has reserved.

```
INSERT INTO Reserves
  SELECT C2.CustomerID, FilmID, CURRENT_DATE
  FROM   Reserves, Customer AS C1, Customer AS C2
  WHERE  C1.Name = 'Eric'
         AND C1.CustomerID = Reserves.CustomerID
         AND C2.Name = 'Melanie'
```

DELETE

- A where clause identifies which rows to remove from the table.
- Delete all the reservations of customer 123456.

```
DELETE FROM Reserves
WHERE CustomerID = 123456
```

- Other tables can be consulted to determine which rows should be removed.
- Delete all of Eric's reservations.

```
DELETE FROM Reserves
WHERE CustomerID = ANY(SELECT CustomerID
                       FROM   Customer
                       WHERE  Name = 'Eric')
```

A set is returned

Someone from the set

UPDATE

- Increase the rental price of all films by 10%.

```
UPDATE Film
      SET RentalPrice = RentalPrice * 1.10
```

- The update statement has an optional where clause.
- Increase the rental price of foreign films by 10%.

```
UPDATE Film
      SET RentalPrice = RentalPrice * 1.10
      WHERE Kind = 'F'
```

SELECT: Subqueries

- A SELECT may be nested

```
SELECT …
FROM …
WHERE <cond> (SELECT …
              FROM …
              WHERE …)
```

- Subqueries may produce
  - A scalar (single value)
  - A single--column table
    - **ANY**, **ALL**, **IN, EXISTS**
  - A multiple-column table
    - **EXISTS**
- Correlated subqueries

Scalar Producing Subquery

- The subquery produces a single value that can be compared
- What are the IDs of customers with the same name as the customer with ID 123456?

```
SELECT CustomerID
 FROM Customer
 WHERE name = (SELECT name
               FROM Customer
               WHERE CustomerID = 123456)
```

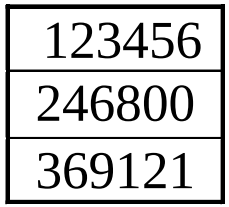
Single Attribute Producing Subquery

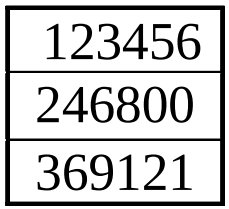
- The subquery produces a table with a single column
- **`IN`**
  - true if value exists in result of subquery
- *comparisonOperator* **`ANY`**
  - true for comparison with at least one tuple in subquery produced table
- *comparisonOperator* **`ALL`**
  - true for comparison with every tuple in subquery produced table

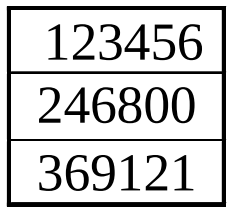
IN

- **`IN`** is equivalent to a restricted form of exists:

$$V \texttt{ IN } r \Leftrightarrow \exists t \in r\ (t = V)$$

- (246800 **`IN`** 
123456
246800
369121
) is true.

- (333333 **`IN`** 
123456
246800
369121
) is false.

- (333333 **`NOT IN`** 
123456
246800
369121
) is true.

IN Query Example

- List the IDs of the films that are expensive and have been reserved.

```
SELECT FilmID
FROM   Film
WHERE  RentalPrice > 4
       AND FilmID IN (SELECT FilmID
                      FROM   Reserves)


SELECT FilmID
FROM   Reserves
WHERE  FilmID IN (SELECT FilmID
                  FROM   Film
                  WHERE RentalPrive > 4)
```

IN, cont.

- List the IDs of the expensive films that have not been reserved.

```
SELECT FilmID
FROM   Film
WHERE  RentalPrice > 4
       AND FilmID NOT IN (SELECT FilmID
                          FROM   Reserves)
```

ANY

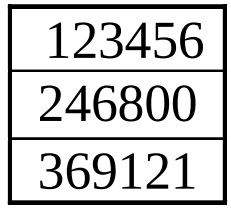
- **ANY** is also equivalent to exists:

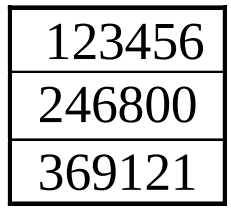
$$V \; comp \;\; \textbf{ANY} \; r \Leftrightarrow \exists t \in r \; (V \; comp \; t)$$

- (246800 < **ANY** | 123456 / 246800 / 369121) is true.

- (369121 < **ANY** | 123456 / 246800 / 369121) is false.

ANY, cont.

- (246800 = **ANY** ) is true.

- (246800 <> **ANY** ) is true.

- Comparison with **IN**
  - (= **ANY**) ⇔ **IN**
  - (<> **ANY**) ⇎ **NOT IN**

ANY Query Example

- Which films rent for more than *some* foreign film?

```
SELECT Title
FROM   Film
WHERE  RentalPrice > ANY(SELECT RentalPrice
                         FROM   Film
                         WHERE  Kind = 'F')
```

ALL

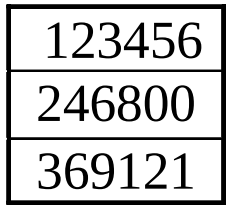
- **`ALL`** is equivalent to for all:

$$V\ comp\ \ \mathbf{ALL}\ r \Leftrightarrow \forall t \in r\ (V\ comp\ t)$$

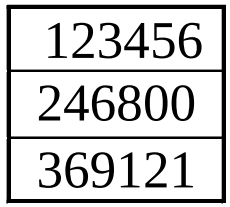
- (`246800 < ALL` | 123456 | 246800 | 369121 |) is false.

- (`100000 < ALL` | 123456 | 246800 | 369121 |) is true.

ALL, cont.

- (246800 = **ALL** 

) is false.

- (246800 <> **ALL** 

) is false.

- Comparison with **IN**
  - (<> **ALL**) ⇔ **NOT IN**
  - (= **ALL**) ⇏ **IN**

ALL Query Example

- Which films rent for more than *all* foreign films?

```
SELECT Title
FROM   Film
WHERE  RentalPrice >  ALL(SELECT RentalPrice
                          FROM   Film
                          WHERE  Kind = 'F')
```

ALL Query Exmaple, cont.

- Find the film(s) with the highest rental price.

```
SELECT Title
FROM   Film
WHERE  RentalPrice >= ALL (SELECT RentalPrice
                           FROM   Film )
```

Correlated Subqueries

- Subquery must be re-evaluated for each tuple in outer **SELECT**
- Table variable is used
- Find the customers who live at more than one address
  - assume just street name needs to be different.

```
SELECT Name
FROM   Customer C
WHERE  CustomerID IN
            (SELECT D.CustomerID
             FROM  Customer D
             WHERE C.Street <> D.Street);
```

Binding

- SQL follows binding rules from tuple relational calculus.
- Previous query can be expressed without correlation name `D` (using a default correlation name of `Customer`).

```
SELECT Name
FROM   Customer C
WHERE  CustomerID IN
            (SELECT CustomerID
             FROM Customer
             WHERE C.Street <> Street);
```

Multiple Attribute Producing Subquery

- The subquery produces a table with several columns
- **`EXISTS`**
  - true if subquery produced table has a tuple
- **`NOT EXISTS`**
  - true if subquery produced table is empty

EXISTS

EXISTS $T \Leftrightarrow T \neq \emptyset$

- List the customers who live in Pullman and have a film reserved.

```
SELECT Name
FROM   Customer
WHERE  City = 'Pullman'
         AND EXISTS( SELECT *
                     FROM   Reserves
                     WHERE  Reserves.CustomerID =
                               Customer.CustomerID)
```

EXISTS, cont.

- Often, **EXISTS** can be replaced with another correlation name.

- List the customers who live in Pullman and have a film reserved.

```
SELECT Name
FROM   Customer
WHERE  City = 'Pullman'
         AND CustomerID IN (SELECT CustomerID
                            FROM   Reserves)

SELECT Name
FROM   Customer, Reserves
WHERE  City = 'Pullman'
         AND Customer.CustomerID = Reserves.CustomerID
```

NOT EXISTS

NOT EXISTS $T \Leftrightarrow T = \emptyset$

- List the customers who live in Pullman but have no films reserved.

```
SELECT Name
FROM   Customer
WHERE  City = 'Pullman'
          AND NOT EXISTS(SELECT *
                         FROM   Reserves
                         WHERE  Reserves.CustomerID =
                                  Customer.CustomerID)
```

NOT EXISTS, cont.

- Often, **NOT EXISTS** can be replaced with **NOT IN**.

- List the customers who live in Pullman but have no films reserved.

```
SELECT Name
FROM   Customer
WHERE  City = 'Pullman'
AND    CustomerID NOT IN (SELECT CustomerID
                          FROM   Reserves)
```

View Definition

- *Views* provide a mechanism to hide certain data from a certain class of users.
- To create a view we use the command:

  `CREATE VIEW` *`name`* **`AS`** *`query expression`*

- Once a view is defined, the view name can be used to refer to the virtual relation that the view generates.
- When a view is created, the query expression is stored in the database; the expression is substituted into queries using the view.

View Example

- Consider a person who needs to know only customer names and films they've reserved.
- Define a view of all customers holding reservations, and the films they have reserved.

```
CREATE VIEW Reservations AS
    SELECT Name, Title
    FROM  Customer, Reserves, Film
    WHERE  Customer.CustomerID =
           Reserves.CustomerID
          AND Reserves.FilmID = Film.FilmID
```

Query on View

- A query on a view is transformed into a query on the base tables.
- Query on view `Reservations`:

  What films has Melanie reserved?

```
SELECT Title
FROM   Reservations
WHERE  Name = 'Melanie'
```

- Query on the base tables

```
SELECT Title
FROM   Customer, Reserves, Film
WHERE  Customer.CustomerID = Reserves.CustomerID
       AND Reserves.FilmID = Film.FilmID
       AND Name = 'Melanie'
```

SQL: Outline

- Brief History
- Data Definition Language
- Data Manipulation Language
- Basic Structure of SQL Queries
- More Powerful SQL Queries
- Constraints in SQL
  - Column Constraints (We've seen them in CREATE TABLE)
  - Table Constraints
  - Referential Integrity
  - Domain Constraints
  - Assertion

Table Constraints

- **UNIQUE** (*column names*)

```
CREATE TABLE Reserves (...,
      UNIQUE ( CustomerID, FilmID, ResDate )
      )
```

- **PRIMARY KEY**

```
CREATE TABLE Reserves (...,
      PRIMARY KEY ( CustomerID, FilmID, ResDate)
      )
```

  - Requires that all columns of the primary key be **NOT NULL**.

- **FOREIGN KEY** (*column(s)*)
  REFERENCES *table* [(*column(s)*)]

```
CREATE TABLE Reserves (...,
      FOREIGN KEY (CustomerID) REFERENCES Customer)
```

Table Constraints: **CONSTRAINT/CHECK**

- **CHECK** (*predicate*)

```
CREATE TABLE Film (...,
    CONSTRAINT (
          NOT (SpokenLanguage = SubtitleLanguage)))
```

- The predicate may include nested select statements, mentioning the current or other tables.

```
CREATE TABLE CheckedOut (...,
  CHECK ('OK' IN
         (SELECT Status
          FROM   VideoTape
          WHERE  FilmID = VideoTape.FilmID)),... )
```

Referential Integrity

- Referential integrity says “pointed to” information must exist.
  - A foreign key points to data in some relation
- Example
  - Customer information must exist for a customer to reserve a film.
  - No `CustomerID` can be in `Reserves` and not in `Customer`.
- Can be specified as a column constraint

```
CREATE TABLE Reserves (...
   CustomerID INTEGER
     CONSTRAINT ReservesToCustomerFK
       REFERENCES Customer(CustomerID),
...)
```

Constraint name

- Can be specified as a table constraint

```
CREATE TABLE Reserves (...,
  CONSTRAINT ReservesToCustomerFK
    FOREIGN KEY (CustomerID) REFERENCES
                Customer(CustomerID) ... )
```

Referential Integrity Violation Remedies

- Can specify **`ON UPDATE`** and **`ON DELETE`** options
- Example

```
CREATE TABLE Reserves (...,
  CONSTRAINT ReservesToCustomerFK
  FOREIGN KEY (CustomerID) REFERENCES
      Customer(CustomerID)
  ON DELETE CASCADE ON UPDATE SET NULL
 ... )
```

- Options (next slides)
  - Note: *Child* table - has the foreign key, references key in *parent* table

    Example: `Customer` is parent, `Reserves` is child.

Remedy Options (1)

- None
  - Update or delete parent value
  - No change to matching child value
- Restrict
  - Cannot update or delete parent value if one or more matching values exist in the child table
  - No change to matching child value

None Remedy Example

- Update `Customer` with `ID` 2, changing `ID` to `5`

Before

`Customer`

Name	ID
Fred	2
Pam	4
Fred	3

`Reserves`

FilmID	CID
7	2
2	4

After

`Customer`

Name	ID
Fred	5
Pam	4
Fred	3

`Reserves`

FilmID	CID
7	2
2	4

- Update allowed, referential integrity is lost. **BAD!**

Restrict Remedy Example

- Update `Customer` with `ID` 2, changing `ID` to `5`

Before

`Customer`

Name	ID
Fred	2
Pam	4
Fred	3

`Reserves`

FilmID	CID
7	2
2	4

After

`Customer`

Name	ID
Fred	2
Pam	4
Fred	3

`Reserves`

FilmID	CID
7	2
2	4

- Cannot update. Must first drop tuple from `Reserves`.

Remedy Options (2)

- Cascade
  - Update or delete parent value
  - Update or delete matching values in child table
- Set Null
  - Update or delete parent value
  - Set matching values in child table to NULL
- Set Default
  - Update or delete parent value
  - Set matching values in child table to default value

Cascade Remedy Example

- Delete `Customer` with `ID` 2

Before

`Customer`

Name	ID
Fred	2
Pam	4
Fred	3

`Reserves`

FilmID	CID
7	2
2	4

After

`Customer`

Name	ID
Pam	4
Fred	3

`Reserves`

FilmID	CID
2	4

Cascade Remedy Example

- Update `Customer` with `ID` 2, changing `ID` to `5`

Before

`Customer`

Name	ID
Fred	2
Pam	4
Fred	3

`Reserves`

FilmID	CID
7	2
2	4

After

`Customer`

Name	ID
Fred	5
Pam	4
Fred	3

`Reserves`

FilmID	CID
7	5
2	4

Set Null Remedy Example

- Update `Customer` with `ID` 2, changing ID to `5`

Before

`Customer`

Name	ID
Fred	2
Pam	4
Fred	3

`Reserves`

FilmID	CID
7	2
2	4

After

`Customer`

Name	ID
Fred	5
Pam	4
Fred	3

`Reserves`

FilmID	CID
7	NULL
2	4

Set Null Remedy Example

- Delete `Customer` with `ID` 2

Before

`Customer`

Name	ID
Fred	2
Pam	4
Fred	3

`Reserves`

FilmID	CID
7	2
2	4

After

`Customer`

Name	ID
Pam	4
Fred	3

`Reserves`

FilmID	CID
7	NULL
2	4

Domain Constraints

- No type constructors exist in SQL-92. SQL3 includes abstract data types.
- SQL-92 allows domains to be defined. These pull together a specific data type, as well as other characteristics of the type.
  - Size
  - Default
  - Constraints
- Example

```
CREATE DOMAIN CustomerDomain INT (6)
        CHECK ( VALUE IS NOT NULL
                AND VALUE > 99999)
```

Assertions

- An assertion is a standalone constraint that we wish the database ALWAYS to satisfy. Such a restriction normally affects more than one table.
  - Once created, it is checked for validity immediately.
  - Any future modification on the database will only allowed if it does not violates the assertion. (Potential high overhead!)
- No one can reserve more than 3 films.

```
CREATE ASSERTION res_limit
CHECK ( NOT EXISTS (
          SELECT Name
          FROM   Customer
          WHERE  3 < (SELECT COUNT (*)
                      FROM   Reserves
                      WHERE  Reserves.CustomerID =
                             Customer.CustomerID)))
```

Summary

- The SQL Query Language has two main parts
  - DDL statements
    - **CREATE**, **DROP**, **ALTER**
    - Constraint definition
  - DML statements
    - **SELECT .. FROM .. WHERE ..**
    - **INSERT**, **DELETE**, **UPDATE**
    - Impacts of constraints

- A query in SQL can consist of up to six clauses, but only the first two, **SELECT** and **FROM**, are mandatory.
- Complex SQL statement can be build using
  - **UNION**/**INTERSECT**/**EXCEPT** (not covered in these slides)
  - **IN**/**ALL**/**ANY**/**EXISTS**