The SQL Query Language

CSc 460 Database Systems

Fall 2013

Acknowledgements

• These slides were written by Richard T. Snodgrass (University of Arizona) and Christian S. Jensen (Aalborg University).

• Kristian Torp (Aalborg University) converted the slides from Island Presents to Powerpoint.

• Curtis E. Dyreson (Washington State University) modified the slides.

Outline

• Overview
  • History
  • Data model

• DDL

• DML

• Data Creation and Destruction

• Other Commonly Used Features

• Summary
Course Concepts

History of SQL
• SEQUEL: Structured English Query Language
  • 23-page research report in 1974 from IBM San Jose Research Labs
  • System R prototype
• SQL: Structured Query Language
  • Oracle in 1979
  • IBM's DB2 in 1983
  • SQL/DS in 1991

Standardization of SQL
• SQL-86
  • ANSI X3.135-1986
  • ISO 9075-1987, “Database Language SQL”
  • A somewhat cleaned up version of DB2 SQL
• SQL-89
  • ANSI X3.168-1989
  • Added embedded SQL
  • 150 pages
Standardization, cont.

• SQL-92
  - ANSI X3.135-1992
  - ISO/IEC 9075:1992
  - Many new DDL and DML features
  - 500 pages

• SQL:1999
  - Object-oriented features
  - Ten parts, eight already accepted by ISO as standards
  - >2000 pages

Standardization, cont.

• SQL:2003
  - Introduced XML-related features

• SQL:2006
  - Defined ways of importing and storing XML data in an SQL database, manipulating that data, and publishing both XML and conventional SQL-date in XML form
  - Allowed XQuery to concurrently access ordinary SQL data and XML documents

Standardization, cont.

• SQL:2008
  - Added INSTEAD OF triggers and the TRUNCATE statement

• SQL:2011
  - Adds temporal support (!)
CSc 460: The SQL Query Language

SQL:2011

• Released December 15, 2011
• Part 1 SQL/Framework, 90 pages (82 in SQL:2008)
• Part 2 SQL/Foundation, 1472 pages (1316 in SQL:2008)
• Part 3 SQL/CLI (Call-Level Interface), 404 pages (389)
• Part 4 SQL/PSM (Persistent Stored Modules), 200 pages (188)
• Part 5, SQL/OLAP dropped from SQL:2008
• Part 6, SQL Transaction, cancelled
• Part 7, SQL/Temporal, (partially) merged into Part 2 (109 pages)
• Part 8, SQL/Objects, merged into Part 2

SQL:2011, cont

• Part 9, SQL/MED (Management of External Data), 486 pages (484)
• Part 10, SQL/OLB (Object Language Bindings), 416 pages (396)
• Part 11, SQL/Replication, in progress
• Part 11, SQL/Schemata (Information and Definition Schemas), 316 pages (286)
• Part 13, SQL/JRT (Java Routines and Types), 210 pages (198)
• Part 14, SQL/XML, 458 pages (438)

• Total: 4052 pages (3776 in SQL:2008)

Relationship with Products

• Most relational DBMSs support most of the SQL-92 syntax and semantics, with some differences.

• No vendor has yet announced SQL:1999 compliance, much less SQL:2011 compliance (but IBM, Oracle, and Teradata have temporal support!).

• IBM and Oracle have significant, though differing, object-oriented features.

• Much development is now focused on XML support.
Chocolate Covered SQL?

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

SQL's Data Model

- Each row of a table has the same columns, over the same domains.
- Duplicate rows are allowed (unless explicitly disallowed via `UNIQUE`). (Tables are multi-sets or bags of rows).

<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pat</td>
<td>1</td>
</tr>
<tr>
<td>Fred</td>
<td>2</td>
</tr>
<tr>
<td>Pam</td>
<td>4</td>
</tr>
<tr>
<td>Fred</td>
<td>2</td>
</tr>
</tbody>
</table>

This is a table in SQL, even though a row is duplicated.

- Order of stored rows cannot be specified (but order of query result can be!)
Outline

• Overview

• DDL
  • Creating/altering schema
  • Data types
  • Constraints
  • Referential integrity and other assertions

• DML

• Data Creation and Destruction

• Other Commonly Used Features

• Summary

Data Definition in SQL

• Three statements are used to define the schema in SQL.
  • CREATE
  • DROP
  • ALTER

• These statements apply to
  • Tables
  • Views
  • Domains
  • Assertions
  • Character sets
  • Translations (between character sets)

CREATE TABLE

• Specifies a new base table.
  
  CREATE TABLE <table name>
  
  (<column name> <data type> [(<size>)] <column constraint>,
  ...
  <table constraints>);

• Name
• Columns with
  • Name
  • Data type
  • Column constraints
  • Default value
• Table constraints
Data Types

- SQL has all the familiar data types
- Some variation from product to product

Basic Data Types

- Numbers
  - Exact
    - INTEGER, SMALLINT, NUMERIC\((p)\), NUMERIC\((p, s)\)
      - \(p\) is the precision (total number of digits) is implementation-defined.
      - \(s\) is the scale (number of digits of the fractional component); defaults to 0.
  - Inexact
    - REAL, FLOAT\((p)\)
- Character strings
  - CHARACTER\((n)\) (CHAR \((n)\))
    - If \(n\) is omitted, it defaults to 1.
    - Space for \(n\) characters is reserved.
  - CHARACTER VARYING\((n)\) (VARCHAR \((n)\))
    - Exactly the number of characters in the data (up to \(n\)) is stored.

More Data Types

- 16-bit character strings
  - LONG CHARACTER \((p)\)
- Dates and times
  - DATE, TIME, TIMESTAMP \((p)\)
  - TIME WITH TIME ZONE
  - INTERVAL of DATE, TIME, or TIMESTAMP \((p)\)
- Bit strings
  - BIT \((p)\)
- Images, sound, video, text documents
  - BLOB (binary large object)
Outline

- Overview
- DDL
  - Creating/altering schema
  - Data types
  - Constraints
  - Referential integrity and other assertions
- DML
- Data Creation and Destruction
- Other Commonly Used Features
- Summary

Column Defaults

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

```sql
CREATE TABLE Movie {
  ...
  SpokenLanguage VARCHAR (15) DEFAULT 'English',
  ...
}
```

Column Constraints

- NOT NULL
  ```sql
  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
  ```sql
  CREATE TABLE Customer {...,
  CustomerID NUMERIC (5) UNIQUE, ...)
  ```
- Ensures that there are no duplicate values
  - Can have one value that is NULL, unless NOT NULL is specified.
Column Constraints, cont.

• **PRIMARY KEY**
  ```sql
  CREATE TABLE Film (
    FilmID NUMERIC (5) PRIMARY KEY,
    ...
  )
  • Implies NOT NULL, UNIQUE (entity integrity).

• **CHECK** (predicate)
  ```sql
  CREATE TABLE Film ( ...
    Kind CHAR(1) CHECK (Kind IN ('F','M','E'))),
  ...
  )
  • Any predicate that may occur in a SELECT’s WHERE clause is allowed.
  • The predicate is assumed to range over the table in which the constraint appears.

Table Constraints

• **UNIQUE** (column names)
  ```sql
  CREATE TABLE Reserved (...,
    UNIQUE (CustomerID, FilmID, ResDate)
  )
  • PRIMARY KEY
  ```sql
  CREATE TABLE Reserved (...,
    PRIMARY KEY (CustomerID, FilmID, ResDate)
  )
  • Requires that all columns of the primary key be NOT NULL.

• **FOREIGN KEY** (column(s))
  ```sql
  REFERENCES table [(column(s))]
  ```sql
  CREATE TABLE Rents (...,
    FOREIGN KEY (FilmID,DVDNum) REFERENCES DVD
  )

Table Constraints: CHECK

• **CHECK** (predicate)
  ```sql
  CREATE TABLE Film (...,
    CONSTRAINT CHECK {
      NOT (SpokenLanguage = SubtitleLanguage)
    }
  )
  • The predicate may include nested select statements, mentioning this table or other tables.
  ```sql
  CREATE TABLE CheckedOut (...,
    CHECK ('OK' IN
      (SELECT Status
        FROM DVD
        WHERE FilmID = DVD.FilmID
        AND DiscNumber = DVD.DiscNumber)),...
  )
Constraints, cont.

- Column and table constraints can be named by the user (otherwise they are named by the DBMS).

```sql
CREATE TABLE Reserved {
    FilmID NUMERIC(5) CONSTRAINT Film_Ref
                  REFERENCES Film,
    ... }

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

Domain Constraints

- No type constructors exist in SQL89 or SQL92. SQL:1999 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
  - Collation

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

Domain Constraints on Customer

- ID is an integer (code is CUSTOMERID)
- State is a two character string.
  - State is limited a list of possible abbreviations (e.g., WA, ID, etc.)
- Zipcode is a number of precision 9 (at most 9 digits).
  - Between 10000 and 999999999
  - Defaults to 99163
- Name is a varying-length character string of max size 50.
  - Defaults to ‘Unknown’
- Street is a varying-length character string of max size 50.
  - Defaults to ‘No address given’
### Other Domain Constraints

- **Preferred Customer**
  - Discount level (an integer - wastes space) is between 0 and 50 (percent), defaults to 0.

- **Film**
  - Year (a four digit number) defaults to 2001 and must be between 1900 and 2010.
  - Title (a varying character string of 100) defaults to ‘Unknown’.
  - FilmID is an integer.

- **Reserves**
  - When is a Date field.

---

### SQL for Oracle

```sql
create table CUSTOMER (
    CUSTOMERID INTEGER       not null,
    NAME VARCHAR2(50) default 'Unknown' not null,
    STATE CHAR(2) default 'WA' null

constraint CKC_STATE_CUSTOMER check (
    STATE is null or
    (STATE in ('WA','ID','OR','CO','CA','WY','MT'))),

STREET VARCHAR2(50) default 'No street address given' null

constraint CKC_ZIPCODE_CUSTOMER check (
    ZIPCODE is null or
    (ZIPCODE between 10000 and 999999999)),

EMPLOYEE_ID INTEGER        null,

DISCOUNT_LEVEL INTEGER        default 0 null

constraint CKC_DISCOUNT_LEVEL_CUSTOMER check (
    DISCOUNT_LEVEL is null or
    (DISCOUNT_LEVEL between 0 and 50)),

constraint PK_CUSTOMER primary key (CUSTOMERID)
)
```

---

### SQL for Informix 7.1

```sql
create table FILM
(
    FILMID INTEGER         not null,
    TITLE VARCHAR(20) default "No film title given",
    YEAR SMALLINT default 2001

check (YEAR is null or (YEAR between 1900 and 2020))

constraint CKC_YEAR_FILM,

primary key (FILMID) constraint PK_FILM
)
```

---
SQL for Sybase 4.x

```
create table FILM(
    FILMID int not null,
    TITLE varchar(20) null,
    YEAR smallint null
)
go
drop rule R_YEAR_FILM

gocreate rule R_YEAR_FILM
as @YEAR between 1900 and 2020
go
execute sp_bindrule R_YEAR_FILM, "FILM.YEAR"
go
drop default D_TITLE_FILM

eexecute sp_binddefault D_TITLE_FILM, "FILM.TITLE"
go
drop default D_YEAR_FILM

create default D_YEAR_FILM
as 2001
go
execute sp_bindefault D_YEAR_FILM, "FILM.YEAR"
go
execute sp_primarykey FILM, FILMID
```

SQL for Sybase 4.x (cont.)

```
create default D_TITLE_FILM
as 'No film title given'
go
execute sp_bindefault D_TITLE_FILM, "FILM.TITLE"
go
drop default D_YEAR_FILM

create default D_YEAR_FILM
as 2001
go
execute sp_bindefault D_YEAR_FILM, "FILM.YEAR"
go
execute sp_primarykey FILM, FILMID
```

Outline

* Overview

* DDL
  * Creating/altering schema
  * Data types
  * Constraints
  * Referential integrity and other assertions

* DML

* Data Creation and Destruction

* Other Commonly Used Features

* Summary
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
  ```sql
  CREATE TABLE Reserves (...,
  CONSTRAINT ReservesToCustomerFK
  REFERENCES Customer(ID),
  ...)"
- Can be specified as a table constraint
  ```sql
  CREATE TABLE Reserves (...,
  CONSTRAINT ReservesToCustomerFK
  FOREIGN KEY (CustomerID) REFERENCES Customer(ID)
  ... )
  ```

Referential Integrity Violation Remedies

- Can specify ON UPDATE and ON DELETE options
- Example
  ```sql
  CREATE TABLE Reserves (...,
  CONSTRAINT ReservesToCustomerFK
  FOREIGN KEY (CustomerID) REFERENCES Customer(ID)
  ON DELETE CASCADE ON UPDATE SET NULL
  ... )
  ```
- Options (next slide)
  - Note: Child table - has the foreign key, references key in parent table
  - Example: Customer is parent, Reserves is child.
  - Note: UPDATE or DELETE is parent table.

Deferring Constraints

- Constraints can optionally be deferred.
  - NOT DEFERRABLE
    - This is the default.
    - The constraint is checked at the end of each SQL statement.
    - If the constraint is violated, the enclosing transaction is aborted.
  - DEFERRABLE
    - The constraint can be checked at the beginning of each transaction:
      - DEFERRABLE INITIALLY IMMEDIATE
    - Or it can be checked at the user's request:
      - DEFERRABLE INITIALLY DEFERRED
      - In that case, the user specifies checking with
        - SET CONSTRAINT name IMMEDIATE
Assertions

• An assertion is a standalone constraint that is normally used to specify a restriction that affects more than one table.
• No one can reserve more than 3 films.

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

• Like constraints, assertions can be specified as DEFERRABLE or NOT DEFERRABLE.

DROP TABLE

• Used to remove a base table and its definition
• The table can no longer be used in queries, updates, or any other commands since its description no longer exists.
• Example:

```
DROP TABLE Customer
```

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 NULLs 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.
Physical Design

- Many database management systems have extensions to specify physical aspects.

  INDEX TABLE Film ON FilmID ASCENDING

- This creates a B+-tree index on the Film table using the FilmID column as the search key.

Summary of Table Definition

- CREATE TABLE
  - Name
  - Columns
    - Name
    - Data type
    - Column constraints
    - Table constraints
  - Table constraints

- DROP TABLE

- ALTER TABLE

Outline

- Overview
- DDL
- DML
  - Single table
  - Multiple tables
  - Optional clauses
  - Additional joins
- Data Creation and Destruction
- Other Commonly Used Features
- Summary
Retrieval Queries in SQL: SELECT

- 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
```

- Approximately (except for duplicates!) means
  \[ \pi_{\text{column list}} (\sigma_{\text{condition}} (\text{table}_1 \times \ldots \times \text{table}_n)) \]

**SELECT: Projecting a Column**

- Find the names of all Customers.
  ```
  SELECT Name
  FROM Customer
  ```
  - This query is termed a *projection query*.
  - Duplicates are eliminated by specifying DISTINCT.
  - In which cities do customers live, each city listed once?
    ```
    SELECT DISTINCT City
    FROM Customer
    ```
  - Relational algebra meaning
    \[ \pi_{\text{city}} (\text{Customer}) \]

**SELECT: Projecting All Columns**

- All the columns can be specified with *.
  ```
  SELECT *
  FROM Customer
  ```
  - Make a copy of the Customer table.
  ```
  SELECT DISTINCT *
  FROM Customer
  ```
  - Can eliminate duplicate rows.
SELECT: Renaming Columns and Tables

- A column can be renamed in generalized projection.
  - Project the City column as Towns
    ```sql
    SELECT DISTINCT City AS Town
    FROM Customer
    ```

- A table can be given a different name within the body of the SELECT statement using a *correlation variable*.
  - Refer to the Customer table using the name C.
    ```sql
    SELECT DISTINCT C.Name
    FROM Customer C
    ```
  - Note: Oracle does not support “AS”.

SELECT: Selecting Rows

- *WHERE* clause is used to choose only rows that meet some condition.
  - Condition can be simple comparison of a column value to
    - a constant
    - a column value
    - result of a SELECT
  - Which customers live in Tucson?
    ```sql
    SELECT DISTINCT Name
    FROM Customer
    WHERE City = 'Tucson'
    ```
  - Relational algebra meaning
    $$\pi_{\text{Name}}(\sigma_{\text{City} = 'Tucson'}(\text{Customer}))$$

SELECT: Using Logical Operators

- **AND**, **OR**, **NOT** may be used on result of comparison
- List outrageously priced films (over $4 or under $1).
  ```sql
  SELECT *
  FROM Film
  WHERE RentalPrice > 4 OR RentalPrice < 1
  ```
- List films that are *not* outrageously priced.
  ```sql
  SELECT DISTINCT *
  FROM Film
  WHERE NOT(RentalPrice > 4 OR RentalPrice < 1)
  ```
  - Relational algebra meaning
    $$\sigma_{\neg (\text{RentalPrice} > 4 \lor \text{RentalPrice} < 1)}(\text{Film})$$
### Outline - The SELECT statement

- Single table
  - Projection
  - Selection
- Multiple tables
  - Cartesian product and join
  - Set operations
  - Subqueries
- Optional clauses
  - Ordering results
  - Computing aggregates on groups
- Additional joins

---

### SELECT: Two or more tables

- **FROM** clause can specify a list of tables, evaluates as Cartesian product of all the tables specified
- List the titles of the reserved films.

```sql
SELECT DISTINCT Title
FROM Film, Reserved
WHERE Reserved.FilmID = Film.FilmID
```

- Relational algebra meaning

\[
\pi_{\text{Title}}(\sigma_{\text{Reserved.FilmID} = \text{Film.FilmID}}(\text{Film} \times \text{Reserved}))
\]

or

\[
\pi_{\text{Title}}(\text{Film} \bowtie_{\text{Reserved}})
\]

---

### SELECT: Queries Over Several Tables

- List the customers who have reserved an expensive film.

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

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

```sql
SELECT Street
FROM Customer, Film, Reserved
WHERE Reserved.CustomerID = Customer.CustomerID
AND Reserved.FilmID = Film.FilmID
AND Kind = 'F'
```
**SELECT: Self Joins**

- Tables can be referenced several times, using different correlation names.
- Which customers have reserved a film that customer 123456 has reserved?
  ```sql
  SELECT A.CustomerID
  FROM Reserved A, Reserved B
  WHERE A.CustomerID = 123456 AND A.FilmID = B.FilmID
  AND A.CustomerID <> B.CustomerID
  ```
- Which films have a higher rental price than some foreign film?
  ```sql
  SELECT DISTINCT A.Title
  FROM Film A, Film B
  WHERE A.RentalPrice > B.RentalPrice
  AND B.Kind = 'F'
  ```

**Union**

- The algebraic operators of union (\(\cup\)), intersection (\(\cap\)) and difference (-) are available, as UNION, INTERSECT, and EXCEPT.
- Must be union-compatible
- List the outrageously priced films (over $4 or under $1).
  ```sql
  (SELECT Title
   FROM Film
   WHERE RentalPrice > 4)
  UNION
  (SELECT Title
   FROM Film
   WHERE RentalPrice < 1)
  ```

**Variants of Union**

- **UNION** eliminates duplicate rows.
- **UNION ALL** preserves duplicate rows. (This is inconsistent with SELECT DISTINCT.)
- **UNION CORRESPONDING** allows one to specify the common columns, when the tables are not quite union-compatible.
  ```sql
  UNION CORRESPONDING (Title)
  ```
  This first performs a projection on Title to produce interesting columns.
- These variants are also available for INTERSECT and EXCEPT.
Outline - The SELECT statement

- Single table
- Multiple tables
  - Cartesian product and join
  - Set operations
  - Subqueries
- Optional clauses
  - Ordering results
  - Computing aggregates on groups
- Additional joins

SELECT: Subqueries

- A SELECT may be nested
  ```sql
  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 mention correlation variables in the enclosed scope.

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?

  ```sql
  SELECT CustomerID
  FROM Customer
  WHERE name = (SELECT C.name
  FROM Customer C
  WHERE C.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:
  \[ V \text{ IN } r \iff \exists t \in r (t = V) \]
- \((246800 \text{ IN } 123456 246800 369121)\) is true.
- \((333333 \text{ IN } 123456 246800 369121)\) is false.
- \((333333 \text{ NOT IN } 123456 246800 369121)\) is true.

---

IN Query

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

```sql
SELECT FilmID
FROM Film
WHERE RentalPrice > 4
AND FilmID IN (SELECT F.FilmID
FROM Reserved F)
```
IN, cont.

• List the ID numbers of the expensive films that have not been reserved.

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

ANY

• ANY is also equivalent to exists:
  
  \[ V \text{ comp } \text{ANY } r \iff \exists t \in r (V \text{ comp } t) \]

• (246800 \(<\) \text{ANY}) is true.

• (369121 \(<\) \text{ANY}) is false.

ANY, cont.

• (246800 \(=\) \text{ANY}) is true.

• (246800 \(<>\) \text{ANY}) is true.

• Comparison with IN
  
  • \(=\) \text{ANY} \iff \text{IN}
  
  • \(<>\) \text{ANY} \iff \text{NOT IN}
ANY Query

- 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:
  \[ V \text{ comp} \ \text{ALL} \ r \iff \forall t \in r \ (V \text{ comp} t) \]

- \((246800 < \text{ALL})\) is false.
- \((100000 < \text{ALL})\) is true.

ALL, cont.

- \((246800 = \text{ALL})\) is false.
- \((246800 <> \text{ALL})\) is false.

- Comparison with **IN**
  - \((<> \text{ALL}) \iff \text{NOT IN}\)
  - \((\neg \text{ALL}) \iff \text{IN}\)
  - \((\neg \text{ALL}) \iff \text{IN}\)
ALL Queries

• Which films rent for more than all foreign films?

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

ALL Queries, cont.

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

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

Correlated Subqueries

• A correlation variable, also called a table variable, is used.
• Find the films with a duplicate title.

```sql
SELECT FilmID, Title
FROM Film F
WHERE Title IN (SELECT F2.Title
                 FROM Film F2
                 WHERE F.FilmID <> F2.FilmID)
```

• Subquery must be re-evaluated (conceptually) for each tuple in outer SELECT.
Binding

- SQL follows binding rules from tuple relational calculus.
- Previous query can be expressed without correlation name F2 (using a default correlation name of Film).

```sql
SELECT FilmID, Title
FROM Film F
WHERE Title IN (SELECT Title
                FROM Film
                WHERE F.FilmID <> FilmID)
```

Select in the From Clause

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

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

- A correlation name is required in such cases.

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**

\[ \text{EXISTS } T \iff T \neq \emptyset \]

- List the customers who live in Tucson and have a film reserved. (Equivalently: there exists a reservation by that customer.)

```sql
SELECT Name
FROM Customer
WHERE City = 'Tucson'
AND EXISTS (SELECT *
              FROM Reserved
              WHERE Reserved.CustomerID = Customer.CustomerID)
```

**EXISTS, cont.**

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

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

```sql
SELECT Name
FROM Customer
WHERE City = 'Tucson'
AND CustomerID IN (SELECT CustomerID
                    FROM Reserved)
```

```sql
SELECT Name
FROM Customer, Reserved
WHERE City = 'Tucson'
AND Customer.CustomerID = Reserved.CustomerID
```

**NOT EXISTS**

\[ \text{NOT EXISTS } T \iff T = \emptyset \]

- List the customers who live in Tucson but have no films reserved. (Equivalently: for which there does not exist a reservation by that customer.)

```sql
SELECT Name
FROM Customer
WHERE City = 'Tucson'
AND NOT EXISTS (SELECT *
                FROM Reserved
                WHERE Reserved.CustomerID = Customer.CustomerID)
```
NOT EXISTS, cont.

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

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

  ```sql
  SELECT Name
  FROM Customer
  WHERE City = 'Tucson'
  AND CustomerID NOT IN (SELECT CustomerID
                           FROM Reserved)
  ```

For All Using NOT EXISTS

- Who has reserved all the foreign films?
- One approach: \( \forall x(P(x)) \iff \neg \exists x(\neg P(x)) \)
- Restated: There does not exist a foreign film that has not been reserved by that customer.

  ```sql
  SELECT Name
  FROM Customer C
  WHERE NOT EXISTS (SELECT FilmID
                      FROM Film
                      WHERE Kind = 'F'
                      EXCEPT
                      SELECT FilmID
                      FROM Film F, Reserved R
                      WHERE Kind = 'F'
                      AND R.FilmID = F.FilmID
                      AND R.CustomerID = C.CustomerID)
  ```

For All, cont.

  ```sql
  SELECT Name
  FROM Customer C
  WHERE NOT EXISTS (SELECT FilmID
                      FROM Film F
                      WHERE Kind = 'F'
                      EXCEPT
                      SELECT FilmID
                      FROM Reserved R
                      WHERE R.CustomerID = C.CustomerID)
  ```

  ```sql
  SELECT Name
  FROM Customer C
  WHERE NOT EXISTS (SELECT FilmID
                      FROM Film
                      WHERE Kind = 'F'
                      AND FilmID NOT IN (SELECT FilmID
                                          FROM Reserved R
                                          WHERE R.CustomerID = C.CustomerID))
  ```
For All, cont.

We’ll see yet another approach soon: using aggregates.

Another alternative is to map the divide query into basic operators, then into SQL.

\[
\pi_{\text{Name}} (\text{Customer} \bowtie (\pi_{\text{CustomerID}, \text{FilmID}} (\text{Reserved}) \\
+ \pi_{\text{FilmID}} (\sigma_{\text{Kind} = 'F'} (\text{Film}))))
\]

\[
 r = \pi_{\text{CustomerID}, \text{FilmID}} (\text{Reserved})
\]

\[
 s = \pi_{\text{FilmID}} (\sigma_{\text{Kind} = 'F'} (\text{Film}))
\]

\[
 p_{\text{Name}} (\text{Customer} \bowtie (r \times s))
\]

\[
 r + s = \pi_{R - S} (r) - \pi_{R - S} ((\pi_{R - S} (r) \times s) - r)
\]

\[
 p_{\text{Name}} (\text{Customer} \bowtie (r \times s) - \pi_{\text{CustomerID}} (\text{Reserved}))
\]

\[
 r - \pi_{R - S} ((\pi_{R - S} (r) \times s) - r)
\]

For All, cont.

SELECT Name
FROM Customer
WHERE NOT EXISTS:
SELECT FilmID
FROM Film
WHERE Kind = 'F'
AND NOT EXISTS:
SELECT FilmID
FROM Reserved
WHERE R.CustomerID = C.CustomerID

We’ll see yet another approach soon: using aggregates.

Another alternative is to map the divide query into basic operators, then into SQL.

\[
\pi_{\text{Name}} (\text{Customer} \bowtie (\pi_{\text{CustomerID}, \text{FilmID}} (\text{Reserved}) \\
+ \pi_{\text{FilmID}} (\sigma_{\text{Kind} = 'F'} (\text{Film}))))
\]

\[
 r = \pi_{\text{CustomerID}, \text{FilmID}} (\text{Reserved})
\]

\[
 s = \pi_{\text{FilmID}} (\sigma_{\text{Kind} = 'F'} (\text{Film}))
\]

\[
 p_{\text{Name}} (\text{Customer} \bowtie (r \times s))
\]

\[
 r + s = \pi_{R - S} (r) - \pi_{R - S} ((\pi_{R - S} (r) \times s) - r)
\]

\[
 p_{\text{Name}} (\text{Customer} \bowtie (r \times s) - \pi_{\text{CustomerID}} (\text{Reserved}))
\]

\[
 r - \pi_{R - S} ((\pi_{R - S} (r) \times s) - r)
\]
Outline - The SELECT statement

• Single table

• Multiple tables

• Optional clauses
  • Ordering results
  • Computing aggregates on groups

• Additional joins

ORDER BY

• Can sort the result of a select, using ORDER BY

• Who has reserved a film?
  SELECT Name
  FROM Customer, Reserved
  WHERE Customer.CustomerID = Reserved.CustomerID
  ORDER BY Name

• Can also sort in descending order, via DESC (ASC is the default)
  SELECT Name
  FROM Customer, Reserved
  WHERE Customer.CustomerID = Reserved.CustomerID
  ORDER BY Name DESC

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

Scalar Aggregates

• Aggregates operate on the set of values of a column of a table, and return a single value.
  • SUM: sum of values
  • AVG: average value
  • MAX: maximum value
  • MIN: minimum value
  • COUNT: number of values

• How many reservations are there?
  SELECT COUNT (*)
  FROM Reserved
Scalar Aggregates, cont.

- What is the total rental price of all films?
  
  ```sql
  SELECT SUM(RentalPrice)
  FROM Film
  ```

- Eliminate duplicates before computing aggregate with `DISTINCT`

- In how many cities do customers reside?
  
  ```sql
  SELECT COUNT(DISTINCT City)
  FROM Customer
  ```

Scalar Aggregates, cont.

- What is the average rental price of reserved films?
  
  ```sql
  SELECT AVG(RentalPrice)
  FROM Film
  WHERE FilmID IN (SELECT FilmID FROM Reserved)
  ```

- Find the film(s) with the highest rental price.
  
  ```sql
  SELECT Title
  FROM Film
  WHERE RentalPrice IN (SELECT MAX(RentalPrice)
                         FROM Film)
  ```

GROUP BY (for Aggregate Functions)

- The table can be partitioned into several groups specifying
  
  ```sql
  GROUP BY <list of columns>
  ```

- Each group has the same attribute values for the indicated columns.

<table>
<thead>
<tr>
<th>Title</th>
<th>RP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lion King</td>
<td>2</td>
</tr>
<tr>
<td>Ocean 11</td>
<td>1</td>
</tr>
<tr>
<td>Joe</td>
<td>2</td>
</tr>
<tr>
<td>Titanic</td>
<td>1</td>
</tr>
</tbody>
</table>

GROUP BY RentalPrice

<table>
<thead>
<tr>
<th>Title</th>
<th>RP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ocean 11</td>
<td>1</td>
</tr>
<tr>
<td>Joe</td>
<td>2</td>
</tr>
<tr>
<td>Lion King</td>
<td>3</td>
</tr>
<tr>
<td>Titanic</td>
<td>3</td>
</tr>
</tbody>
</table>
GROUP BY, cont.

- Aggregate is applied to each group.
- How many films of each rental price are there?

```sql
SELECT DISTINCT RentalPrice, COUNT(RentalPrice)
FROM Film
GROUP BY RentalPrice
```

```
<table>
<thead>
<tr>
<th>Title</th>
<th>RP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lion King</td>
<td>3</td>
</tr>
<tr>
<td>Ocean 11</td>
<td>1</td>
</tr>
<tr>
<td>Joe</td>
<td>2</td>
</tr>
<tr>
<td>Titanic</td>
<td>1</td>
</tr>
</tbody>
</table>
```

GROUP BY, cont.

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

```sql
SELECT Kind, AVG(RentalPrice)
FROM Reserved, Film
WHERE Reserved.FilmID = Film.FilmID
GROUP BY Kind
```

```
<table>
<thead>
<tr>
<th>Kind</th>
<th>AVG(RentalPrice)</th>
</tr>
</thead>
</table>
```

GROUP BY column(s) must appear in the select list.

HAVING

- Individual groups can be eliminated by using HAVING.
- List the average rental price of reserved films, by kind, as long as the average is more than $2.

```sql
SELECT Kind, AVG(RentalPrice)
FROM Reserved, Film
WHERE Reserved.FilmID = Film.FilmID
GROUP BY Kind
HAVING AVG(RentalPrice) > 2
```

```sql
<table>
<thead>
<tr>
<th>Kind</th>
<th>AVG(RentalPrice)</th>
</tr>
</thead>
</table>
```
HAVING, cont.

- Columns in **HAVING** clause must appear in the **GROUP BY** (or be contained within an aggregate).
- List the customers whose average rental price for their reserved films is greater than $3.

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

- Here, **RentalPrice** appears within an aggregate.

---

HAVING, cont.

- What is the ID of the most popular reserved film?

```sql
SELECT FilmID
FROM Reserved
GROUP BY FilmID
HAVING COUNT(*) = (
    SELECT DISTINCT MAX(C.RCount)
    FROM (SELECT COUNT(*) AS RCount, FilmID
            FROM Reserved
            GROUP BY FilmID) AS C
)
```

---

For All Using Count and Having

- Who has reserved all the foreign films?
- Restated: The number of foreign films that have been reserved by that customer equals the number of foreign films.

```sql
SELECT Name
FROM Customer C, Film F, Reserved R
WHERE Kind = 'F'
     AND R.FilmID = F.FilmID
     AND R.CustomerID = C.CustomerID
GROUP BY Name
HAVING count(DISTINCT F.FilmID) =
    (SELECT count(DISTINCT FilmID)
     FROM Film
     WHERE Kind = 'F')
```
Summary

- A query in SQL can consist of up to six clauses, but only the first two, `SELECT` and `FROM`, are mandatory.

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

Summary, cont.

- The query is evaluated in a different order.
  * The table(s) in the from clause are combined using Cartesian product(s).
  * The where predicate is then applied.
  * The resulting tuples are grouped according to the group by clause.
  * The having predicate is applied to each group, possibly eliminating some groups.
  * The aggregate(s) are applied to each remaining group.
  * The select clause is then performed, to generate the result tuples.
  * Finally, the result is optionally sorted.

- The mapping of subqueries is understandably more complex.

Order of Clause Evaluation

1. `FROM`
2. `WHERE` - optional
3. `GROUP BY` - optional, defaults to 1 group
4. `HAVING` – optional
5. `SELECT` - (generalized) projection
6. `ORDER BY` - optional
Summary, cont.

• These steps have correspondences in the relational algebra.

\[ t_1 = \text{table}_1 \times \ldots \times \text{table}_n \]
\[ t_2 = \sigma_{\text{predicate}}(t_2) \]
\[ t_3 = \text{group list} \ \downarrow_{\text{agg}}(t_2) \]
\[ t_4 = \text{having}(t_3) \]
\[ t_5 = \pi_{\text{select list}}(t_4) \]
\[ \text{result} = \text{sort}_{\text{order by}}(t_5) \]

Example

• List the average rental price of reserved films, by kind, as long as the average is more than $2.

```
SELECT Kind, AVG(RentalPrice)
FROM Reserved, Film
WHERE Reserved.FilmID = Film.FilmID
GROUP BY Kind
HAVING AVG(RentalPrice) > 2
ORDER BY Kind
```

```
sort_{\text{Kind}}(\pi_{\text{Kind}, \text{avg}(\sigma_{\text{avg} > 2}(\text{Kind} \ \downarrow_{\text{avg}}(\text{RentalPrice})(
\text{Reserved} \bowtie \text{Film})))))
```

Outline - The SELECT statement

• Single table

• Multiple tables

• Optional clauses
  • Ordering results
  • Computing aggregates on groups

• Additional Joins
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.

```sql
SELECT * 
FROM Customer CROSS JOIN Reserved
```

- 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.

```sql
SELECT Customer.*, Reserved.* 
FROM Customer, Reserved
```

Natural Join

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

```sql
SELECT * 
FROM Customer NATURAL JOIN Reserved
```

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

```sql
SELECT DISTINCT Customer.*, FilmID, ResDate 
FROM Customer, Reserved 
WHERE Customer.CustomerID = Reserved.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 *Employee* tables both include address information. Say we wish to join only on the *CustomerID* column, which is also present in both.

  ```sql
  SELECT * FROM Customer JOIN Employee USING (CustomerID)
  ```

**Outer Join**

- In the FROM list: LEFT OUTER JOIN ON *predicate*

- Also RIGHT OUTER JOIN ON *predicate* and FULL OUTER JOIN ON *predicate*

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

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

**Union Join**

- Similar to FULL OUTER JOIN, except no predicate is applied.

- Null values are guaranteed, even when there is a match.

- The schema of the resulting relation is $R \cup S$.

  $R$: $A$ | $B$ | $C$
  --- | --- | ---
  1 | $y$ | $\emptyset$
  3 | $x$ | $\gamma$
  2 | $y$ | $\emptyset$

  $S$: $D$ | $A$
  --- | ---
  "Tom" | 1
  "Eric" | 2

  $r \cup UNION JOIN s$:

  $A$ | $B$ | $C$ | $D$ | $A$
  --- | --- | --- | --- | ---
  1 | $y$ | $\emptyset$ | Null | Null
  3 | $x$ | $\gamma$ | Null | Null
  2 | $y$ | $\emptyset$ | Null | Null
  Null | Null | Null | "Tom" | 1
  Null | Null | Null | "Eric" | 2
Expressive Power

- SQL is more expressive than either the relational algebra or the relational calculus.

  Reasoning:
  - SQL is relationally complete.
    - Hence, SQL ≥ RA and SQL ≥ RC.
  - SQL can sort tables, and SQL tables can contain duplicates.
    - Hence, SQL > RA and SQL > RC.

Outline

- Overview
- DDL
- DML
  - Data Creation and Destruction
    - Inserting into a table
    - Deleting from a table
    - Modifying values in a table
  - Other Commonly Used Features
- Summary

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.

  ```sql
  INSERT INTO Reserved
  VALUES (123456, 332244, CURRENT_DATE)
  ```
INSERT, cont.

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

```sql
INSERT INTO Reserved
SELECT C2.CustomerID, FilmID, CURRENT_DATE
FROM Reserved, Customer C, Customer C2
WHERE C.Name = 'Eric'
  AND C.CustomerID = Reserved.CustomerID
  AND C2.Name = 'Melanie'
```

DELETE

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

```sql
DELETE FROM Reserved
WHERE CustomerID = 123456
```

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

```sql
DELETE FROM Reserved
WHERE CustomerID IN (SELECT CustomerID
  FROM Customer
  WHERE Name = 'Eric')
```

DELETE, cont.

- Delete the films with a rental price below the average.

```sql
DELETE FROM Film
WHERE RentalPrice < (SELECT AVG(DISTINCT RentalPrice)
  FROM Film)
```

- Implementation difficulty: As we delete rows from Film, the average rental price changes!
  - Not a problem
  - Inner query, first computes the average balance.
  - Outer "loop" deletes tuples without recomputing AVG.
**UPDATE**

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

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

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

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

**UPDATE, cont.**

- Increase the rental price of films with a current price of under $3 by 20%, and all others by 10%.

```sql
UPDATE Film
SET RentalPrice = RentalPrice * 1.10
WHERE RentalPrice >= 3.00
```

```sql
UPDATE Film
SET RentalPrice = RentalPrice * 1.20
WHERE RentalPrice < 3.00
```

- Careful: order is important!

**Outline**

- Overview
- DDL
- DML
- Data Creation and Destruction
- Other Commonly Used Features
  - Views
  - Embedded SQL and cursors
  - Transactions and triggers
- Summary
Views

- Views provide a mechanism to create a virtual table.
- To create a view we use the command:

  ```sql
  CREATE VIEW name AS query expression
  ```
- Define a view of all customers in Tucson.

  ```sql
  CREATE VIEW Tucson_Customers AS
  SELECT *
  FROM Customer
  WHERE City = 'Tucson'
  ```

Views, cont.

- Define a view of all Tucson customers holding reservations and the films they have reserved.

  ```sql
  CREATE VIEW Reservations AS
  SELECT Name, Title
  FROM Tucson_Customers AS TC, Reserved, Film
  WHERE TC.CustomerID = Reserved.CustomerID
  AND Reserved.FilmID = Film.FilmID
  ```

Implementing Views

- View Materialization
  - Compute the view and store it as a separate relation on disk.
  - Advantage: queries on views can be faster.
  - Disadvantages:
    - Requires disk space
    - May require revision when underlying table(s) are modified
- Query Modification
  - Rewrite the query on the view as a query on the base table(s).
Query Modification

- In query modification, a query on a view is transformed into a query on the base tables.
- Query on a view: What films has Tucson resident Melanie reserved?
  
  ```sql
  SELECT Title
  FROM Reservations
  WHERE Name = 'Melanie'
  ```

- Query on the base tables
  
  ```sql
  SELECT Title
  FROM Customer, Reserved, Film
  WHERE Customer.CustomerID = Reserved.CustomerID
  AND Reserved.FilmID = Film.FilmID
  AND Name = 'Melanie' AND City = 'Tucson'
  ```

Embedded SQL

- SQL can be embedded in programming languages such as Ada, C, COBOL, Java, and Pascal.
- Rows in tables are manipulated through cursors.
- A cursor is something like a pointer that traverses a collection of rows.
  
  ```sql
  DECLARE title_cursor CURSOR FOR
  SELECT Title
  FROM Reservations
  WHERE Name = 'Melanie'
  ```

- A subsequent OPEN cursor command evaluates the query and sets the cursor to a position before the first row in the result of the query; this becomes the current row for the cursor.
  
  ```sql
  OPEN title_cursor
  ```

Embedded SQL, cont.

- Subsequent FETCH commands in the program advance the cursor to the next row and copy its column values into C program variables specified in the FETCH command.
  
  ```sql
  FETCH title_cursor INTO :title
  ```

- Here the C variable title was specified, preceded with a colon.
- An implicit C variable SQLCODE communicates to the program the status of SQL embedded commands.
- An SQLCODE of 0 indicates successful execution.
- Different codes are returned to indicate exceptions and errors.
Embedded SQL, cont.

• A **CLOSE** cursor command is issued to indicate that we are done with the result of the query.

  ```sql
  CLOSE title_cursor
  ```

• At this point, the results of the query execution are no longer accessible.

Cursors, cont.

• When a cursor is defined for rows that are to be updated, the clause **FOR UPDATE OF** must be in the cursor declaration, and a list of the names of any columns that will be updated follows.

  ```sql
  DECLARE Customer_cursor CURSOR FOR
  SELECT Name, Street, City, State
  FROM Customer
  FOR UPDATE OF Street, City, State
  ```

• Then once a cursor has been opened, it can be updated.

  ```sql
  UPDATE Customer
  SET City = :newcity
  WHERE CURRENT OF Customer_cursor
  ```

Scrollable Cursors

• With scrollable cursors one can skip rows.
  - **DECLARE name SCROLL CURSOR AS ...**

• Scrollable cursors are not updatable.

• The fetch command has many options for scrollable cursors.
  - **FETCH NEXT**
    - This is the default.
  - **FETCH PRIOR**
  - **FETCH FIRST**
  - **FETCH LAST**
  - **FETCH ABSOLUTE** value
  - **FETCH RELATIVE** value
Transactions

- Transactions are initiated with any SQL statement that modifies the database.
- Transactions can be committed (all the changes are made to the database) or aborted (all the changes are discarded).

**COMMIT WORK**

**ROLLBACK WORK**

- Transactions are also relevant to recovery and to concurrency control.

Triggers (on tables)

- Maintain data integrity
- Associated with a table
- Event-condition-action (ECA)
  - Wait for a table event
  - On event, evaluate condition
  - If condition is true, execute action

Level of Trigger

- **Table-level Trigger**
  - Works with entire table
  - Action evaluated with respect to before image or after image, prior to commit

- **Row-level Trigger**
  - Works with entire table and a buffer of changes
  - New - tuples that were inserted
  - Old - tuples that were deleted
  - Update is insertion + deletion
  - Trigger iterates through buffer of changes
  - Action evaluated with respect to before image or after image, prior to the work being committed
Trigger Examples

• Table-level Trigger
  ▪ Inventory of popular titles should never fall below a stated threshold.
  ▪ Send an alert message if demand for a video doubles in less than three days.

• Row-level Trigger
  ▪ The zipcode should be appropriate for the state.
  ▪ Foreign videos should have a price in a stated range.

Outline

• Overview
• DDL
• DML
• Data Creation and Destruction
• Other Commonly Used Features
• Summary

Summary

• Schema definition
  ▪ CREATE TABLE
  ▪ CREATE DOMAIN
  ▪ CREATE ASSERTION
  ▪ CREATE TRIGGER
  ▪ ALTER
  ▪ DROP

• Queries
  ▪ SELECT
### Summary, cont.

- **Modifications**
  - `INSERT`
  - `DELETE`
  - `UPDATE`

- **Views**
  - `CREATE VIEW`

- **Transaction Management**
  - `COMMIT`
  - `ROLLBACK`