SQL – The Relational Database Standard

SQL

• Data Definition in SQL
• Retrieval Queries in SQL
• Specifying Updates in SQL
• Relational Views in SQL
Overview

• Background of SQL
• Data Definition Language (DDL) in SQL
  – Domain Types
  – Schema Definition
  – Integrity Constraints

Background

• IBM Sequel language, the original version of SQL. (In the early 1970s)
• Renamed to Structured Query Language (SQL)
• ANSI and ISO standard SQL:
  – SQL-86 (SQL1)
  – SQL-89
  – SQL-92 (SQL2)
  – SQL:1999 (SQL3)
  – SQL:2003 (2003 SQL3)
• Our description is based on
  – SQL-92 (SQL2)
• Not all examples here may work on your particular system.
Data Definition Language (DDL)

- **Domain Types**
  - Basic Domain Types
  - Additional Domain Types
- Schema Definition
- Integrity Constraints

### Basic Domain Types

- **char**\((n)\): a fixed length character string
- **varchar**\((n)\): a variable length character string
- **int/integer**: an integer
- **smallint**: a small integer
- **numeric**\((p,d)\): a fixed point number with user-specified precision
- **real, double precision**: floating point and double-precision floating point numbers
- **float**\((n)\): a floating point number
Additional Data Types in SQL2

- **DATE**: Made up of year-month-day in the format yyyymm-dd
- **TIME**: Made up of hour:minute:second in the format hh:mm:ss
- **TIME(i)**: Made up of hour:minute:second plus i additional digits specifying fractions of a second; format is hh:mm:ss:ii...i
- **TIMESTAMP**: Has both **DATE** and **TIME** components (‘2002-09-27 09:12:47’)
- **INTERVAL**: Specifies a relative value rather than an absolute value
  - Can be DAY/TIME intervals or YEAR/MONTH intervals
  - When added to or subtracted from an absolute value, the result is an absolute value

Data Definition Language (DDL)

- **Domain Types**
- **Schema Definition**
  - Create Table
    - Insert
    - Delete
  - Drop Table
  - Alter Table
- **Integrity Constraints**
Data Definition in SQL

• Used to CREATE, DROP, and ALTER the descriptions of the tables (relations) of a database

• CREATE TABLE:
  – Specifies a new base relation by giving it a name, and specifying each of its attributes and their data types (INTEGER, FLOAT, DECIMAL(i,j), CHAR(n), VARCHAR(n))
  – A constraint NOT NULL may be specified on an attribute

Create Table

• 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),
   branch_city char(30),
   assets integer)
  ```
CREATE table example

CREATE TABLE DEPARTMENT
( DNAME VARCHAR(10) NOT NULL,
  DNUMBER INTEGER NOT NULL,
  MGRSSN CHAR(9),
  MGRSTARTDATE CHAR(9) );

Key attributes can be specified via the PRIMARY KEY and UNIQUE phrases

CREATE TABLE DEPT
( DNAME VARCHAR(10) NOT NULL,
  DNUMBER INTEGER NOT NULL,
  MGRSSN CHAR(9),
  MGRSTARTDATE CHAR(9),
  PRIMARY KEY (DNUMBER),
  UNIQUE (DNAME),
  FOREIGN KEY (MGRSSN) REFERENCES EMP(SSN) );

Insert

• The insert command inserts a tuple into a relation
  – Example:
    
    insert into branch
    values (‘Perryridge’, ‘Dallas’, null)

  – Attribute list can be omitted if it is the same as in CREATE TABLE
  – NULL and DEFAULT values can be specified
Delete

• The **delete** command deletes tuples from a relation
  – Simple form: delete all tuples from a relation
    • Example: **delete from branch**

  – Other forms: allow specific tuples to be deleted
    (covered later)

DROP TABLE

• Used to remove a relation (base table) *and its definition*
• The relation can no longer be used in queries, updates, or any other commands since its description no longer exists
• **E.g.**
  – **DROP TABLE** DEPENDENT
  – **DROP TABLE** Branch
Alter Table

- The `alter table` command is used to add attributes to an existing relation:
  - Example:
    ```sql
    alter table branch add phone_number char(10)
    ```
  - 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:
  - Example:
    ```sql
    alter table branch drop branch_city
    ```
  - Dropping of attributes is not supported by many databases

ALTER TABLE

- Used to add/drop an attribute, change a column definition, add/drop table constrains
- The new attribute will have NULLs in all the tuples of the relation right after the command is executed; hence, the NOT NULL constraint is *not allowed* for such an attribute
- e.g.,
  - `ALTER TABLE EMPLOYEE ADD JOB VARCHAR(12);`
  - `Alter table branch add phone_number char(10);`
  - `Alter table branch drop branch_city;`
Data Definition Language (DDL)

- Domain Types
- Schema Definition
- **Integrity Constraints**
  - Constraints on a Single Relation
  - Referential Integrity
  - Assertions

Constraints on a Single Relation

- **not null**
- **primary key**
- **default**
- **unique**
- **check** \((P)\), where \(P\) is a predicate
Not Null Constraint

• Not null constraint: null value is not permitted.
• Example:
  – Declare \textit{branch\_name} for \textit{branch} to be \textbf{not null}
    \begin{verbatim}
    branch\_name char(15) not null
    \end{verbatim}
  
  – Declare the domain \textit{Dollars} to be \textbf{not null}
    \begin{verbatim}
    create domain Dollars numeric(12,2) not null
    \end{verbatim}

Primary Key

• \textbf{primary key} \((A_1, \ldots, A_n)\)
• Example: Declare \textit{branch\_name} as the primary key for \textit{branch}
  \begin{verbatim}
  create table branch
  (branch\_name char(15),
   branch\_city char(30),
   assets integer,
   primary key (branch\_name))
  \end{verbatim}

• \textbf{primary key} declaration implies \textbf{not null} and \textbf{unique}. 
Default Constraint

• Specify a default value for the attribute with the `default` clause
• Example:

```sql
create table account
  (account_number char(10),
   branch_name char(15),
   balance integer default 0,
   primary key (branch_name))
```

Unique Constraint

• `unique (A_1, A_2, ..., A_m)`
• The unique specification states that the attributes \((A_1, A_2, \ldots A_m)\) form a candidate key.
The check Clause

- **check** \((P)\), where \(P\) is a predicate
- The **check** clause can be applied to relation declarations.
- Example 1: Declare \textit{branch\_name} as the primary key for \textit{branch} and ensure that the values of \textit{assets} are non-negative.

```sql
create table branch
    (branch\_name char(15),
     branch\_city char(30),
     assets integer,
     primary key (branch\_name),
     check (assets >= 0))
```

The check Clause (Cont.)

- Example 2:

```sql
create table student
    (name char(15) not null,
     student\_id char(10),
     degree\_level char(15),
     primary key (student\_id),
     check (degree\_level in ('Bachelors', 'Masters', 'Doctorate')))```
The check Clause (Cont.)

• The **check** clause in SQL-92 permits domains to be restricted.
• Example 3: Use **check** clause to ensure that an hourly_wage domain allows only values greater than a specified value.

```sql
create domain hourly_wage numeric(5,2)
constraint value_test check(value > = 4.00)
```

The check Clause (Cont.)

• Example 4:

```sql
create domain AccountType varchar(10)
constraint type_test
    check (value in (‘Checking’, ‘Saving’))
```
Referential Integrity

- Enforce referential integrity by `foreign key` clause
- Example:

  ```sql
  create table branch
  (branch_name  char(15),
   ...)

  create table account
  (...)
  branch_name  char(15),
  foreign key (branch_name) references branch
  (branch_name),
  ...
  ```

- A foreign key only references the primary key attributes or the candidate key attributes of the referenced relation.

Referential Integrity (Cont.)

- When a referential integrity constraint is violated
  - Default procedure: reject the action
  - Other procedures: change the tuple in the referencing relation to restore the constraint.
    - Set to null: on delete/update set null
    - Set to default value: on delete/update set default
    - Propagate delete/update: on delete/update cascade
  - Example:

    ```sql
    create table account
    (...)
    foreign key (branch_name) references branch
    on delete cascade
    on update cascade,
    ...
    ```
Referential Integrity (Cont.)

- Attributes of foreign keys are allowed to be null.
- What if any attribute of a foreign key is null?
- The tuple is defined to satisfy the foreign key constraint.
- Can we alter integrity constraints to an existing relation?
  - `alter table r add constraint`
  - `alter table r drop constraint constraint name`

Referential Integrity (Cont.)

- Transactions consist of several steps.
- What if intermediate steps violate referential integrity, but later steps remove the violation?
- **Initially deferred**: the constraint will be checked at the end of a transaction.
- **Deferrable**: checked immediately by default, but can be deferred when desired.
  - `set constraints constraint-list deferred`
Assertions

• An **assertion** is a predicate expressing a condition that we wish the database always to satisfy.

  ```sql
  create assertion <assertion-name> check <predicate>
  ```

• Assertion testing may introduce a significant amount of overhead.

• Asserting for all $X$, $P(X)$ by using

  ```sql
  not exists $X$ such that not $P(X)$
  ```

 Assertions (Example 1)

• Every loan has at least one borrower who maintains an account with a minimum balance or $1000.00

  ```sql
  create assertion balance_constraint check (not exists (select * from loan
  where not exists (select *
    from borrower, depositor, account
    where loan.loan_number = borrower.loan_number
    and borrower.customer_name = depositor.customer_name
    and depositor.account_number = account.account_number
    and account.balance >= 1000)))
  ```
Assertions (Example 2)

- The sum of all loan amounts for each branch must be less than the sum of all account balances at the branch.

```sql
CREATE TABLE DEPARTMENT
(
    DNAME VARCHAR(10) NOT NULL,
    DNUMBER INTEGER NOT NULL,
    MGRSSN CHAR(9),
    MGRSTARTDATE CHAR(9)
);
```

Key attributes can be specified via the **PRIMARY KEY** and **UNIQUE** phrases.

```sql
CREATE TABLE DEPT
(
    DNAME VARCHAR(10) NOT NULL,
    DNUMBER INTEGER NOT NULL,
    MGRSSN CHAR(9),
    MGRSTARTDATE CHAR(9),
    PRIMARY KEY (DNUMBER),
    UNIQUE (DNAME),
    FOREIGN KEY (MGRSSN) REFERENCES EMP(SSN)
);
```
Features Added in SQL2

• CREATE SCHEMA
  – Specifies a new database schema by giving it a name
  – CREATE SCHEMA COMPANY ;

• REFERENTIAL INTEGRITY OPTIONS
  – specify CASCADE or SET NULL or SET DEFAULT on referential integrity constraints (foreign keys)

CREATE TABLE DEPT
  ( DNAME VARCHAR(10) NOT NULL,
    DNUMBER INTEGER NOT NULL,
    MGRSSN CHAR(9),
    MGRSTARTDATE CHAR(9),
    PRIMARY KEY (DNUMBER),
    UNIQUE (DNAME),
    FOREIGN KEY (MGRSSN) REFERENCES EMP
      ON DELETE SET DEFAULT ON UPDATE CASCADE );

CREATE TABLE EMP
  ( ENAME VARCHAR(30) NOT NULL,
    ESSN CHAR(9),
    BDATE DATE,
    DNO INTEGER DEFAULT 1,
    SUPERSSN CHAR(9),
    PRIMARY KEY (ESSN),
    FOREIGN KEY (DNO) REFERENCES DEPT
      ON DELETE SET DEFAULT ON UPDATE CASCADE,
    FOREIGN KEY (SUPERSSN) REFERENCES EMP
      ON DELETE SET NULL ON UPDATE CASCADE );
Retrieval Queries in SQL

- Basic SQL queries correspond to using the **SELECT**, **PROJECT**, and **JOIN** operations of the relational algebra.

- Basic form of the SQL SELECT statement is called a *mapping* or a **SELECT-FROM-WHERE block**

```
SELECT <attribute list>
FROM <table list>
WHERE <condition>
```

- `<attribute list>` is a list of attribute names whose values are to be retrieved by the query.
- `<table list>` is a list of the relation names required to process the query.
- `<condition>` is a conditional (Boolean) expression that identifies the tuples to be retrieved by the query.

- the SELECT basic statement for retrieving information is **not the same as** the SELECT operation of the relational algebra.

- Distinction between SQL and the formal relational model:
  - SQL allows a table (relation) to have two or more tuples that are identical in all their attribute values.
  - an SQL relation (table) is a *multi-set* of tuples; it is not a set of tuples.
  - SQL relations can be constrained to be sets by specifying PRIMARY KEY or UNIQUE attributes, or by using the DISTINCT option in a query.
Schema for Student Registration System

Student (Id, Name, Addr, Status)
Professor (Id, Name, DeptId)
Course (DeptId, CrsCode, CrsName, Descr)
Transcript (StudId, CrsCode, Semester, Grade)
Teaching (ProfId, CrsCode, Semester)
Department (DeptId, Name)

Query Example

• Select all the course names offered by Department of computer science.
  – Assume: DeptID of Computer Science = ‘CS’.
Query Sublanguage of SQL

SELECT C.CrsName
FROM Course C
WHERE C.DeptId = 'CS'

• Tuple variable C ranges over rows of Course.
• Evaluation strategy:
  – FROM clause produces Cartesian product of listed tables
  – WHERE clause assigns rows to C in sequence and produces table containing only rows satisfying condition
  – SELECT clause retains listed columns
• Equivalent to: \( \pi_{\text{CrsName}} \sigma_{\text{DeptId}='CS'}(\text{Course}) \)

Join Queries

• List all courses taught in S2000 semester.
Join Queries

Rename relation tables
Course C, Teaching T

• List all courses taught in S2000
• Selection condition “ T.Semester='S2000' ”
  – eliminates irrelevant rows
• Join condition “C.CrsCode=T.CrsCode”
  – eliminates garbage
• Equivalent (using natural join) to relational algebra:
  \[ \pi_{\text{CrsName}}(\text{Course} \quad \ast \quad \sigma_{\text{Sem}='S2000'}(\text{Teaching}) ) \]

Join Queries

SELECT C.CrsName
FROM Course C, Teaching T
WHERE C.CrsCode=T.CrsCode AND T.Semester='S2000'
• List CS courses taught in S2000
• Tuple variables clarify meaning.
• Join condition “C.CrsCode=T.CrsCode”
  – eliminates garbage
• Selection condition “ T.Sem='S2000' ”
  – eliminates irrelevant rows
• Equivalent (using natural join) to:
  \[ \pi_{\text{CrsName}}(\text{Course} \quad \ast \quad \sigma_{\text{Sem}='S2000'}(\text{Teaching}) ) \]
Join Queries

SELECT C.CrsName
FROM Course C, Teaching T
WHERE C.CrsCode=T.CrsCode AND T.Sem='S2000'
• List CS courses taught in S2000
• Tuple variables clarify meaning.
• Join condition “C.CrsCode=T.CrsCode”
  – eliminates garbage
• Selection condition “ T.Sem='S2000' ”
  – eliminates irrelevant rows
• Equivalent (using natural join) to:
  \[ \pi_{\text{CrsName}} (\sigma_{\text{Sem}='S2000'}(\text{Teaching})) \]
  \[ \pi_{\text{CrsName}} (\sigma_{\text{Sem}='S2000'}(\text{Course} \bowtie \text{Teaching})) \]

Correspondence Between SQL and Relational Algebra

SELECT C.CrsName
FROM Course C, Teaching T
WHERE C.CrsCode=T.CrsCode AND T.Sem='S2000'

Also equivalent to:
\[ \pi_{\text{CrsName}} \sigma_{\text{C.CrsCode}=T.CrsCode \text{ AND Sem}='S2000'} (\text{Course \{C.CrsCode, DeptId, CrsName, Desc\}} \bowtie \text{Teaching \{ProfId, T.CrsCode, Sem\}}) \]

This is the simple evaluation algorithm for SELECT. Relational algebra expressions are procedural. Which of the two equivalent expressions is more easily evaluated?
Self-join Queries

Find Ids of all professors who taught at least two different courses in the same semester:

```sql
SELECT T1.ProfId
FROM Teaching T1, Teaching T2
WHERE T1.ProfId = T2.ProfId
AND T1.Semester = T2.Semester
AND T1.CrsCode <> T2.CrsCode
```

(Tuple variables now essential)

Equivalent to:

```
π_{ProfId} (σ_{T1.CrsCode≠T2.CrsCode}(Teaching[ProfId, T1.CrsCode, Sem] * Teaching[ProfId, T2.CrsCode, Sem]))
```
Duplicates

- Duplicate rows not allowed in a relation
- However, duplicate elimination from query result is costly and not automatically done; it must be explicitly requested:

```sql
SELECT DISTINCT ..... FROM .....;
```

Use of Expressions

Equality and comparison operators apply to strings (based on lexical ordering)

```sql
WHERE S.Name < 'P'
```

Concatenate operator applies to strings

```sql
WHERE S.Name || -- || S.Address = ....
```

Expressions can also be used in SELECT clause:

```sql
SELECT  S.Name || -- || S.Address AS NmAdd FROM  Student S
```
Set Operators

- SQL provides UNION, EXCEPT (set difference), and INTERSECT for union compatible tables
- Example: Find all professors in the CS Department and all professors that have taught CS courses

\[
\begin{align*}
\text{SELECT} & \quad \text{P.Name} \\
\text{FROM} & \quad \text{Professor P, Teaching T} \\
\text{WHERE} & \quad \text{P.Id=} T.\text{ProfId AND T.CrsCode LIKE ‘CS%’} \\
\text{UNION} & \quad \text{(SELECT} \quad \text{P.Name} \\
\text{FROM} & \quad \text{Professor P} \\
\text{WHERE} & \quad \text{P.DeptId = ‘CS’)}
\end{align*}
\]
Nested Queries

List all courses that were not taught in S2000

Evaluation strategy: subquery evaluated once to produces set of courses taught in S2000. Each row (as Course) tested against this set.

```
SELECT C.CrsName
FROM Course C
WHERE C.CrsCode NOT IN
    (SELECT T.CrsCode      --subquery
     FROM Teaching T
     WHERE T.Sem = 'S2000')
```

Evaluation strategy: subquery evaluated once to produces set of courses taught in S2000. Each row (as C) tested against this set.
Correlated Nested Queries

Output a row \(<\text{prof}, \text{dept}>\) if \text{prof} has taught a course in \text{dept}.

\[
\begin{align*}
\text{SELECT} & \quad \text{P.Name, D.Name} & \text{--outer query} \\
\text{FROM} & \quad \text{Professor P, Department D} & \\
\text{WHERE} & \quad \text{P.Id IN (set of Id’s of all profs who have taught a course in D.DeptId)}
\end{align*}
\]

\[
\begin{align*}
\text{SELECT} & \quad \text{T.ProfId} & \text{--subquery} \\
\text{FROM} & \quad \text{Teaching T, Course C} & \\
\text{WHERE} & \quad \text{T.CrsCode=C.CrsCode AND C.DeptId=D.DeptId} & \text{--correlation}
\end{align*}
\]

Correlated Nested Queries (con’t)

- Tuple variables T and C are local to subquery
- Tuple variables P and D are global to subquery
- Correlation: subquery uses a global variable, D
- The value of D.DeptId parameterizes an evaluation of the subquery
- Subquery must (at least) be re-evaluated for each distinct value of D.DeptId
- Correlated queries can be expensive to evaluate
Division

• **Query type:** Find the subset of items in one set that are related to *all* items in another set

• **Example:** Find professors who have taught courses in *all* departments
  – Why does this involve division?

![Diagram showing Division]

Contains row
\(<p,d>\) if professor
\(p\) has taught a
course in
department \(d\)

Division

• **Strategy for implementing division in SQL:**
  – Find set of all departments in which a particular professor, \(p\), has taught a course - A
  – Find set of all departments - B
  – Output \(p\) if \(A \supseteq B\), or equivalently if \(B-A\) is empty
Division – SQL Solution

SELECT P.Id
FROM Professor P
WHERE NOT EXISTS
  (SELECT D.DeptId -- B: set of all dept Ids
   FROM Department D
   EXCEPT
   SELECT C.DeptId -- A: set of dept Ids of depts in
   -- which P has taught a course
   FROM Teaching T, Course C
   WHERE T.ProfId=P.Id --global variable
     AND T.CrsCode=C.CrsCode)

Aggregates

- Functions that operate on sets:
  - COUNT, SUM, AVG, MAX, MIN
- Produce numbers (not tables)
- Not part of relational algebra

SELECT COUNT(*)
FROM Professor P

SELECT MAX(Salary)
FROM Employee E
Aggregates

Count the number of courses taught in S2000

```sql
SELECT COUNT (T.CrsCode)
FROM Teaching T
WHERE T.Semester = 'S2000'
```

But if multiple sections of same course are taught, use:

```sql
SELECT COUNT (DISTINCT T.CrsCode)
FROM Teaching T
WHERE T.Semester = 'S2000'
```

Aggregates: Proper and Improper Usage

```sql
SELECT COUNT (T.CrsCode), T.ProfId
```

…… --makes no sense (in the absence of GROUP BY clause)

```sql
SELECT COUNT (*), AVG (T.Grade)
```

…… --but this is OK

```sql
WHERE T.Grade > COUNT (SELECT ....
    --aggregate cannot be applied to result of SELECT statement
```
GROUP BY

Table output by WHERE clause:
- Divide rows into groups based on subset of attributes;
- All members of a group agree on those attributes

Each group can be described by a single row in a table with attributes limited to:
- Attributes all group members share (listed in GROUP BY clause)
- Aggregates over group

GROUP BY - Example

Transcript

Attributes:
- student’s Id
- avg grade
- number of courses

SELECT T.StudId, AVG(T.Grade), COUNT (*)
FROM Transcript T
GROUP BY T.StudId
HAVING Clause

- Eliminates unwanted groups (analogous to WHERE clause)
- HAVING condition constructed from attributes of GROUP BY list and aggregates of attributes not in list

```
SELECT T.StudId, AVG(T.Grade) AS CumGpa, COUNT(*) AS NumCrs
FROM Transcript T
WHERE T.CrsCode LIKE 'CS%'
GROUP BY T.StudId
HAVING AVG(T.Grade) > 3.5
```

Example

- Output the name and address of all seniors on the Dean’s List

```
SELECT S.Name, S.Address
FROM Student S, Transcript T
WHERE S.StudId = T.StudId AND S.Status = 'senior'
GROUP BY S.StudId -- wrong
             S.Name, S.Address -- right
HAVING AVG(T.Grade) > 3.5 AND SUM(T.Credit) > 90
```
ORDER BY Clause

• Causes rows to be output in a specified order

SELECT T.StudId, COUNT (*) AS NumCrs, AVG(T.Grade) AS CumGpa
FROM Transcript T
WHERE T.CrsCode LIKE ‘CS%’
GROUP BY T.StudId
HAVING AVG(T.Grade) > 3.5
ORDER BY DESC CumGpa, ASC StudId

Query Evaluation Strategy

1 Evaluate FROM: produces Cartesian product, A, of tables in FROM list
2 Evaluate WHERE: produces table, B, consisting of rows of A that satisfy WHERE condition
3 Evaluate GROUP BY: partitions B into groups that agree on attribute values in GROUP BY list
4 Evaluate HAVING: eliminates groups in B that do not satisfy HAVING condition
5 Evaluate SELECT: produces table C containing a row for each group. Attributes in SELECT list limited to those in GROUP BY list and aggregates over group
6 Evaluate ORDER BY: orders rows of C
Views

• Used as a relation, but rows are not physically stored.
  – A view is *materialized* when it is used within an SQL statement

• View is the result of a SELECT statement over other views and base relations

• When used in an SQL statement, the view definition is substituted for the view name in the statement
  – SELECT statement can be nested in FROM clause

View - Example

CREATE VIEW CumGpa (StudId, Cum) AS
  SELECT T.StudId, AVG (T.Grade)
  FROM Transcript T
  GROUP BY T.StudId

SELECT S.Name, C.Cum
FROM CumGpa C, Student S
WHERE C.StudId = S.StudId AND C.Cum > 3.5
View Benefits

- Access Control: Users not granted access to base tables. Instead they are granted access to the view of the database appropriate to their needs.
  - External schema is composed of views.
  - View allows owner to provide SELECT access to a subset of columns (analogous to providing UPDATE and INSERT access to a subset of columns)

Views - Limiting Visibility

```sql
CREATE VIEW PartOfTranscript (StudId, CrsCode, Semester) AS
  SELECT T.StudId, T.CrsCode, T.Semester -- limit columns
  FROM Transcript T
  WHERE T.Semester = 'S2000' -- limit rows

GRANT SELECT ON PartOfTranscript TO joe

This is analogous to:
GRANT UPDATE (Grade) ON Transcript TO joe
```
View Benefits (con’t)

- Customization: Users need not see full complexity of database. View creates the illusion of a simpler database customized to the needs of a particular category of users
- A view is similar in many ways to a subroutine in standard programming
  - Can be used in multiple queries

Nulls

- Conditions: $x \text{ op } y$ (where $\text{op}$ is $<$, $>$, $\leq$, $\geq$, $=$, etc.) has value *unknown* ($U$) when either $x$ or $y$ is null
  - WHERE $T.\text{cost} > T.\text{price}$
- Arithmetic expression: $x \text{ op } y$ (where $\text{op}$ is $+$, $-$, $\ast$, etc.) has value NULL if $x$ or $y$ is null
  - WHERE $(T.\text{price}/T.\text{cost}) > 2$
- Aggregates: $\text{COUNT}$ counts nulls like any other value; other aggregates ignore nulls

```
SELECT  COUNT(T.CrsCode),  AVG(T.Grade)
FROM Transcript T
WHERE T.StudId = '1234'
```
Nulls (con’t)

- WHERE clause uses a three-valued logic to filter rows. Portion of truth table:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>C1 AND C2</th>
<th>C1 OR C2</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>U</td>
<td>U</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>U</td>
<td>F</td>
<td>U</td>
</tr>
<tr>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
</tbody>
</table>

- Rows are discarded if WHERE condition is false or unknown
- Ex: WHERE T.CrsCode = ‘CS305’ AND T.Grade > 2.5

Modifying Tables - Insert

- Inserting a single row into a table
  - Attribute list can be omitted if it is the same as in CREATE TABLE (but do not omit it)
  - NULL and DEFAULT values can be specified

```
INSERT INTO Transcript (StudId, CrsCode, Semester, Grade)
VALUES (12345, ‘CSE305’, ‘S2000’, NULL)
```
Bulk Insertion

- Insert the rows output by a SELECT

CREATE TABLE DeansList (  
    StudId INTEGER,  
    Credits INTEGER,  
    CumGpa FLOAT,  
    PRIMARY KEY StudId)

INSERT INTO DeansList (StudId, Credits, CumGpa)  
SELECT T.StudId, 3 * COUNT (*), AVG(T.Grade)  
FROM Transcript T  
GROUP BY T.StudId  
HAVING AVG (T.Grade) > 3.5 AND COUNT(*) > 30

Modifying Tables - Delete

- Similar to SELECT except:
  - No project list in DELETE clause
  - No Cartesian product in FROM clause
  - Rows satisfying WHERE clause (general form, including subqueries, allowed) are deleted instead of output

DELETE FROM Transcript T  
WHERE T.Grade IS NULL AND  
T.Semester <> ‘S2000’
Modifying Data - Update

UPDATE Employee E
SET E.Salary = E.Salary * 1.05
WHERE E.Department = 'research'

• Updates rows in a single table
• All rows satisfying WHERE clause (general form, including subqueries, allowed) are updated

Updating Views

• Question: Since views look like tables to users, can they be updated?
• Answer: Yes – a view update changes the underlying base table to produce the requested change to the view

CREATE VIEW CsReg (StudId, CrsCode, Semester) AS
SELECT T.StudId, T.CrsCode, T.Semester
FROM Transcript T
WHERE T.CrsCode LIKE 'CS%' AND T.Semester='S2000'
Updating Views - Problem 1

```
INSERT INTO CsReg (StudId, CrsCode, Semester)
VALUES (1111, 'CSE305', 'S2000')
```

- **Question**: What value should be placed in attributes of underlying table that have been projected out (e.g., Grade)?
- **Answer**: NULL (assuming null allowed in the missing attribute) or DEFAULT

Updating Views - Problem 2

```
INSERT INTO CsReg (StudId, CrsCode, Semester)
VALUES (1111, 'ECO105', 'S2000')
```

- **Problem**: New tuple not in view
- **Solution**: Allow insertion (assuming the ‘WITH CHECK OPTION’ clause has not been appended to the CREATE VIEW statement)
Updating Views - Problem 3

• Update to the view might not *uniquely* specify
  the change to the base table(s) that results in
  the desired modification of the view

CREATE VIEW ProfDept (PrName, DeName) AS
SELECT  P.Name, D.Name
FROM    Professor P, Department D
WHERE   P.DeptId = D.DeptId

Updating Views - Problem 3
(con’t)

• Tuple <Smith, CS> can be deleted from
  ProfDept by:
  – Deleting row for Smith from Professor (but this
    is inappropriate if he is still at the University)
  – Deleting row for CS from Department (not
    what is intended)
  – Updating row for Smith in Professor by setting
    DeptId to null (seems like a good idea)
Updating Views - Restrictions

- Updatable views are restricted to those in which
  - No Cartesian product in FROM clause
  - no aggregates, GROUP BY, HAVING
  - ...

For example, if we allowed:
  CREATE VIEW AvgSalary (DeptId, Avg_Sal) AS
  SELECT E.DeptId, AVG(E.Salary)
  FROM Employee E
  GROUP BY E.DeptId
then how do we handle:
  UPDATE AvgSalary
  SET Avg_Sal = 1.1 * Avg_Sal