Relational Algebra and SQL

Chapter 5

Relational Query Languages

• Languages for describing queries on a relational database
• *Structured Query Language* (SQL)
  – Predominant application-level query language
  – Declarative
• *Relational Algebra*
  – Intermediate language used within DBMS
  – Procedural

What is an Algebra?

• A language based on operators and a domain of values
• Operators map values taken from the domain into other domain values
• Hence, an expression involving operators and arguments produces a value in the domain
• When the domain is a set of all relations (and the operators are as described later), we get the *relational algebra*
• We refer to the expression as a *query* and the value produced as the *query result*

Relational Algebra

• *Domain*: set of relations
• *Basic operators*: select, project, union, set difference, Cartesian product
• *Derived operators*: set intersection, division, join
• *Procedural*: Relational expression specifies query by describing an algorithm (the sequence in which operators are applied) for determining the result of an expression

The Role of Relational Algebra in a DBMS

Select Operator

• Produce table containing subset of rows of argument table satisfying condition
  
  \[ \sigma_{\text{condition}}(\text{relation}) \]

  • Example:
  
  Person

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>Address</th>
<th>Hobby</th>
</tr>
</thead>
<tbody>
<tr>
<td>1123</td>
<td>John</td>
<td>123 Main</td>
<td>stamps</td>
</tr>
<tr>
<td>1123</td>
<td>John</td>
<td>123 Main</td>
<td>coins</td>
</tr>
<tr>
<td>5556</td>
<td>Mary</td>
<td>7 Lake Dr</td>
<td>hiking</td>
</tr>
<tr>
<td>9876</td>
<td>Bart</td>
<td>5 Pine St</td>
<td>stamps</td>
</tr>
</tbody>
</table>

  \[
  \sigma_{\text{Hobby}=\text{stamps}}(\text{Person})
  \]

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>Address</th>
<th>Hobby</th>
</tr>
</thead>
<tbody>
<tr>
<td>1123</td>
<td>John</td>
<td>123 Main</td>
<td>stamps</td>
</tr>
<tr>
<td>1123</td>
<td>John</td>
<td>123 Main</td>
<td>coins</td>
</tr>
<tr>
<td>9876</td>
<td>Bart</td>
<td>5 Pine St</td>
<td>stamps</td>
</tr>
</tbody>
</table>
Selection Condition

- Operators: <, ≤, ≥, >, =, ≠
- Simple selection condition:
  - <attribute> operator <constant>
  - <attribute> operator <attribute>
- <condition> AND <condition>
- <condition> OR <condition>
- NOT <condition>

Selection Condition - Examples

- \( \sigma \text{Id} > 3000 \text{ OR } \text{Hobby} = \text{hiking} \) (Person)
- \( \sigma \text{Id} > 3000 \text{ AND } \text{Id} < 3999 \) (Person)
- \( \sigma \neg (\text{Hobby} = \text{hiking}) \) (Person)
- \( \sigma \text{Hobby} = \text{hiking} \) (Person)

Project Operator

- Produces table containing subset of columns of argument table
  - \( \pi_{\text{attribute list}}(\text{relation}) \)
- Example:
  - Person
  - \( \pi_{\text{Name,Address}}(\text{Person}) \)

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>Address</th>
<th>Hobby</th>
</tr>
</thead>
<tbody>
<tr>
<td>1123</td>
<td>John</td>
<td>123 Main</td>
<td>stamps</td>
</tr>
<tr>
<td>1123</td>
<td>John</td>
<td>123 Main</td>
<td>coins</td>
</tr>
<tr>
<td>5556</td>
<td>Mary</td>
<td>7 Lake Dr</td>
<td>hiking</td>
</tr>
<tr>
<td>9876</td>
<td>Bart</td>
<td>5 Pine St</td>
<td>stamps</td>
</tr>
</tbody>
</table>

Result is a table (no duplicates); can have fewer tuples than the original

Project Operator

- Example:
  - Person
  - \( \pi_{\text{Name,Address}}(\text{Person}) \)

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>Address</th>
<th>Hobby</th>
</tr>
</thead>
<tbody>
<tr>
<td>1123</td>
<td>John</td>
<td>123 Main</td>
<td>stamps</td>
</tr>
<tr>
<td>1123</td>
<td>John</td>
<td>123 Main</td>
<td>coins</td>
</tr>
<tr>
<td>5556</td>
<td>Mary</td>
<td>7 Lake Dr</td>
<td>hiking</td>
</tr>
<tr>
<td>9876</td>
<td>Bart</td>
<td>5 Pine St</td>
<td>stamps</td>
</tr>
</tbody>
</table>

Expressions

\( \pi_{\text{Id,Name}} (\sigma \text{Hobby} = \text{"stamps" OR Hobby = "coins"}) (\text{Person}) \)

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>Address</th>
<th>Hobby</th>
</tr>
</thead>
<tbody>
<tr>
<td>1123</td>
<td>John</td>
<td>123 Main</td>
<td>stamps</td>
</tr>
<tr>
<td>1123</td>
<td>John</td>
<td>123 Main</td>
<td>coins</td>
</tr>
<tr>
<td>5556</td>
<td>Mary</td>
<td>7 Lake Dr</td>
<td>hiking</td>
</tr>
<tr>
<td>9876</td>
<td>Bart</td>
<td>5 Pine St</td>
<td>stamps</td>
</tr>
</tbody>
</table>

Set Operators

- Relation is a set of tuples, so set operations should apply: \( \cap, \cup, \neg \) (set difference)
- Result of combining two relations with a set operator is a relation \( \Rightarrow \) all its elements must be tuples having same structure
- Hence, scope of set operations limited to union compatible relations
Union Compatible Relations

- Two relations are **union compatible** if
  - Both have same number of columns
  - Names of attributes are the same in both
  - Attributes with the same name in both relations have the same domain
- Union compatible relations can be combined using **union**, **intersection**, and **set difference**

### Example

Tables:
- **Person** (SSN, Name, Address, Hobby)
- **Professor** (Id, Name, Office, Phone)

are not union compatible.

But
- $\pi_{\text{Name}}(\text{Person})$ and $\pi_{\text{Name}}(\text{Professor})$
  - are union compatible so
  - $\pi_{\text{Name}}(\text{Person}) \setminus \pi_{\text{Name}}(\text{Professor})$
    - makes sense.

---

Cartesian Product

- If $R$ and $S$ are two relations, $R \times S$ is the set of all concatenated tuples $(x, y)$, where $x$ is a tuple in $R$ and $y$ is a tuple in $S$
- $R$ and $S$ need not be union compatible
- $R \times S$ is expensive to compute:
  - Factor of two in the size of each row
  - Quadratic in the number of rows

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>x2</td>
<td>y1</td>
<td>y2</td>
<td>x1</td>
<td>x2</td>
<td>y1</td>
<td>y2</td>
</tr>
<tr>
<td>x3</td>
<td>x4</td>
<td>y3</td>
<td>y4</td>
<td>x3</td>
<td>x4</td>
<td>y1</td>
<td>y2</td>
</tr>
</tbody>
</table>

$R \times S$

### Renaming

- Result of expression evaluation is a relation
- Attributes of relation must have distinct names.
  - This is not guaranteed with Cartesian product
  - e.g., suppose in previous example $a$ and $c$ have the same name
- Renaming operator tidies this up. To assign the names $A_1$, $A_2$, ..., $A_n$ to the attributes of the $n$ column relation produced by expression expr use $\text{expr} [A_1, A_2, ..., A_n]$

### Example

Transcript (StudId, CrsCode, Semester, Grade)

Teaching (ProfId, CrsCode, Semester)

\[
\pi_{\text{StudId}, \text{CrsCode}}(\text{Transcript}) \times \pi_{\text{ProfId}, \text{CrsCode}}(\text{Teaching})
\]

This is a relation with 4 attributes:

*StudId*, *CrsCode1*, *ProfId*, *CrsCode2*

### Derived Operation: Join

A **general or theta** join of $R$ and $S$ is the expression

$R \bowtie_{\text{join-condition}} S$

where **join-condition** is a conjunction of terms:

$A_i \text{ oper } B_i$

in which $A_i$ is an attribute of $R$; $B_i$ is an attribute of $S$; and oper is one of $\equiv, <, \geq, \neq, \leq$.

The meaning is:

$\sigma_{\text{join-condition}}(R \times S)$

where **join-condition** and **join-condition** are the same, except for possible renamings of attributes (next)
Join and Renaming

- **Problem**: $R$ and $S$ might have attributes with the same name – in which case the Cartesian product is not defined.
- **Solutions**:
  1. Rename attributes prior to forming the product and use new names in `join-condition`.
  2. Qualify common attribute names with relation names (thereby disambiguating the names). For instance: `Transcript.CrsCode` or `Teaching.CrsCode`.

This solution is nice, but doesn’t always work: consider $R$ join condition in $R$. How do we know which $R$ is meant?

Theta Join – Example

Employee(`Name`, `Id`, `MngrId`, `Salary`)  
Manager(`Name`, `Id`, `Salary`)  

Output the names of all employees that earn more than their managers.

$\pi_{\text{Employee}.\text{Name}} (\pi_{\text{MngrId}} (\pi_{\text{Id}} (\text{Employee} \bowtie \text{Manager})))$

The join yields a table with attributes: `Employee.Name`, `Employee.Id`, `Employee.Salary`, `MngrId`, `Manager.Name`, `Manager.Id`, `Manager.Salary`.

Equijoin Join - Example

*Equijoin*: Join condition is a conjunction of equalities.

$\pi_{\text{Name}, \text{CrsCode}} (\text{Student} \bowtie \text{Transcript} (\text{StudId} = \text{Id}))$

### Student

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>Addr</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>John</td>
<td>......</td>
<td>......</td>
</tr>
<tr>
<td>222</td>
<td>Mary</td>
<td>......</td>
<td>......</td>
</tr>
<tr>
<td>333</td>
<td>Bill</td>
<td>......</td>
<td>......</td>
</tr>
<tr>
<td>444</td>
<td>Joe</td>
<td>......</td>
<td>......</td>
</tr>
</tbody>
</table>

### Transcript

<table>
<thead>
<tr>
<th>StudId</th>
<th>CrsCode</th>
<th>Sem</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>CSE305</td>
<td>S00</td>
<td>B</td>
</tr>
<tr>
<td>222</td>
<td>CSE306</td>
<td>S99</td>
<td>A</td>
</tr>
<tr>
<td>333</td>
<td>CSE304</td>
<td>F99</td>
<td>A</td>
</tr>
</tbody>
</table>

Mary    CSE306  
Bill    CSE304

The equijoin is used very frequently since it combines related data in different relations.

Natural Join

- Special case of equijoin:
  - `join-condition` equates all and only those attributes with the same name (condition doesn’t have to be explicitly stated)
  - duplicate columns eliminated from the result

### Transcript

$[\text{StudId}, \text{CrsCode}, \text{Sem}, \text{Grade}]$

### Teaching

$[\text{ProfId}, \text{CrsCode}, \text{Sem}]$

$\pi_{\text{StudId}, \text{CrsCode}, \text{Sem}, \text{Grade}} (\text{Transcript} \bowtie \text{Teaching})$

Natural Join (cont’d)

- More generally:
  
  $R \bowtie S = \pi_{\text{attr-list}} (\sigma_{\text{join-cond}} (R \times S))$
  
  where
  
  attr-list = attributes ($R$) $\cup$ attributes ($S$)
  
  (duplicates are eliminated) and `join-cond` has the form:
  
  $A_1 = A_1 \text{ AND } \ldots \text{ AND } A_n = A_n$
  
  where
  
  $\{A_1, \ldots, A_n\} = \text{attributes}(R) \cap \text{attributes}(S)$

Natural Join Example

- List all Ids of students who took at least two different courses:

  $\pi_{\text{StudId}} (\sigma_{\text{CrsCode} \neq \text{CrsCode2}} (\langle$

  Transcript $\bowtie \text{Transcript} [\text{StudId}, \text{CrsCode2}, \text{Sem2}, \text{Grade2}]\rangle))$

  We don’t want to join on `CrsCode`, `Sem`, and `Grade` attributes, hence renaming!
Division

- Goal: Produce the tuples in one relation, \( r \), that match \( all \) tuples in another relation, \( s \)
  - \( r(\text{A}_1, \ldots, \text{A}_n, \text{B}_1, \ldots, \text{B}_m) \)
  - \( s(\text{B}_1, \ldots, \text{B}_m) \)
  - \( r/s \), with attributes \( \text{A}_1, \ldots, \text{A}_n \) is the set of all tuples \( \langle a \rangle \) such that for every tuple \( \langle b \rangle \) in \( s \), \( \langle a, b \rangle \) is in \( r \)
- Can be expressed in terms of projection, set difference, and cross-product

Division (cont’d)

Division - Example

- List the Ids of students who have passed \( all \) courses that were taught in spring 2000
- **Numerator:**
  - \( \text{StudId} \) and \( \text{CrsCode} \) for every course passed by every student:
    \[ \pi_{\text{StudId}, \text{CrsCode}}(\sigma_{\text{Grade} = 'F'}(\text{Transcript})) \]
- **Denominator:**
  - \( \text{CrsCode} \) of all courses taught in spring 2000
    \[ \pi_{\text{CrsCode}}(\sigma_{\text{Semester} = 'S2000'}(\text{Teaching})) \]
- Result is numerator/denominator

Schema for Student Registration System

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

Query Sublanguages of SQL

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

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

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 “\( C.CrsCode=T.CrsCode \)”
  - relates facts to each other
- Selection condition “\( T.Semester='S2000' \)”
  - eliminates irrelevant rows
- Equivalent (using natural join) to:
  \[ \pi_{\text{CrsName}}(\sigma_{\text{T.Semester} = 'S2000'}(\text{Tearching})) \]
  \[ \pi_{\text{CrsName}}(\sigma_{\text{T.Semester} = 'S2000'}(\text{Course} \bowtie Tearching)) \]
Correspondence Between SQL and Relational Algebra

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

Also equivalent to:

\[ \pi_{\text{CrsName}}(\sigma_{\text{CrsCode}=T.\text{CrsCode}\text{ AND} T.\text{Semester}=\text{S2000}}(\text{Course}\{\text{C.\text{CrsCode}}, \text{DeptId}, \text{CrsName}, \text{Desc}\}\times \text{Teaching}\{\text{ProfId}, T.\text{CrsCode}, \text{Semester}\}))}\]

- This is the simplest 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 courses in the same semester:

```
SELECT T1.ProfId
FROM Teaching T1, Teaching T2
WHERE T1.ProfId = T2.ProfId
AND T1.Semester = T2.Semester
AND T1.CrsCode !== T2.CrsCode
```

Equivalent to:

\[ \pi_{\text{ProfId}}(\sigma_{\text{T1.CrsCode} \neq T2.\text{CrsCode}}(\text{Teaching}\{\text{ProfId}, T1.\text{CrsCode}, \text{Semester}\}\times \text{Teaching}\{\text{ProfId}, T2.\text{CrsCode}, \text{Semester}\}))}\]

Tuple variables are essential in this query!

Duplicates

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

```
SELECT DISTINCT ..... FROM ..... 
```

Use of Expressions

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

WHERE S.Name < 'P'

Concatenate operator applies to strings

WHERE S.Name || '--' || S.Address = ....

Expressions can also be used in SELECT clause:

```
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

```
(SELECT P.Name
FROM Professor P, Teaching T
WHERE P.Ids=T.ProfId AND T.CrsCode LIKE 'CS%')
UNION
(SELECT P.Name
FROM Professor P
WHERE P.DeptId = 'CS')
```

Nested Queries

List all courses that were not taught in S2000

```
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 <prof, dept> if prof has taught a course in dept.

```
SELECT P.Name, D.Name         -- outer query
FROM Professor P, Department D
WHERE P.Id IN
    -- set of all ProfId's who have taught a course in D.DeptId
    [SELECT T.ProfId         -- subquery
     FROM Teaching T, Course C
     WHERE T.CrsCode=C.CrsCode AND
     C.DeptId=D.DeptId         -- correlation
     ]
)                      -- correlation
```

Correlated Nested Queries (cont)

- 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 in SQL

- **Query type:** Find the subset of items in one set that are related to all items in another set
- **Example:** Find professors who taught courses in all departments
  - Why does this involve division?

Contains row <p,d> if professor p taught a course in department d

<table>
<thead>
<tr>
<th>ProfId</th>
<th>DeptId</th>
<th>DeptId</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProfId</td>
<td>DeptId</td>
<td>All department Ids</td>
</tr>
</tbody>
</table>

\[ \pi_{\text{ProfId}, \text{DeptId}}(\text{Teaching} \rightarrow \text{Course}) / \pi_{\text{DeptId}}(\text{Department}) \]

Division in SQL

- **Strategy for implementing division in SQL:**
  - Find set, A, of all departments in which a particular professor, p, has taught a course
  - Find set, B, of all departments
  - 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
     FROM Department D
     EXCEPT
     SELECT C.DeptId
     -- set A of dept Ids of depts in which P taught a course
     FROM Teaching T, Course C
     WHERE T.ProfId=P.Id 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 (but not hard to add)

```
SELECT COUNT(*)
FROM Professor P

SELECT MAX(Salary)
FROM Employee E
```
Aggregates (cont’d)

Count the number of courses taught in S2000

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

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

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

Grouping

- But how do we compute the number of courses taught in S2000 per professor?
  - Strategy 1: Fire off a separate query for each professor:
    ```
    SELECT COUNT(T.CrsCode)
    FROM Teaching T
    WHERE T.Semester = 'S2000' AND T.ProfId = 123456789
    ```
  - Cumbersome
  - What if the number of professors changes? Add another query?
  - Strategy 2: Define a special grouping operator:
    ```
    SELECT T.ProfId, COUNT(T.CrsCode)
    FROM Teaching T
    WHERE T.Semester = 'S2000'
    GROUP BY T.ProfId
    ```

GROUP BY

```
GROUP BY
```

```
GROUP BY - Example
```

```
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, but works on groups instead of individual tuples)
- HAVING condition is constructed from attributes of GROUP BY list and aggregates on attributes not in that 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
```

Evaluation of GroupBy with Having
Example

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

```sql
SELECT S.Id, S.Name
FROM Student S, Transcript T
WHERE S.Id = T.StudId AND S.Status = 'senior'
GROUP BY S.Id
HAVING AVG(T.Grade) > 3.5 AND SUM(T.Credit) > 90
```

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
```

ORDER BY Clause

- Causes rows to be output in a specified order

```sql
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 with GROUP BY, HAVING, ORDER BY

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.
  - The contents of a view is computed 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
  - As SELECT statement nested in FROM clause

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

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

View - Example
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)

View Benefits (cont’d)

- **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 reused in multiple queries

**Nulls**

- **Conditions**: `x op y` (where `op` is `<`, `>`, `<>`, `=`, etc.) has value `unknown (U)` when either `x` or `y` is null
  - WHERE `T.cost > T.price`
- **Arithmetic expression**: `x op y` (where `op` is `+`, `–`, `*`, etc.) has value `NULL` if `x` or `y` is `NULL`
  - WHERE `(T.price/T.cost) > 2`
- **Aggregates**: `COUNT` counts `NULL`s like any other value; other aggregates ignore `NULLs`
  ```sql
  SELECT COUNT(T.CrsCode), AVG(T.Grade)
  FROM Transcript T
  WHERE T.StudId = ‘1234’
  ```

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

- Insert the rows output by a SELECT

```sql
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 (only 1 table name)
  - Rows satisfying WHERE clause (general form, including subqueries, allowed) are deleted instead of output

```sql
DELETE FROM Transcript T  
WHERE T.Grade IS NULL AND T.Semester <> 'S2000'
```

**Modifying Data - Update**

```sql
UPDATE Employee E  
SET E.Salary = E.Salary * 1.05  
WHERE E.Department = 'R&D'
```

- 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

```sql
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**

```sql
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**

```sql
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 a view might not uniquely specify the change to the base table(s) that results in the desired modification of the view (ambiguity)

```
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 (cont’d)

- 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, but how would the computer know?)

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
```