Test for Empty Relations

- The \textbf{exists} construct returns the value \textbf{true} if the argument subquery is nonempty.
- \textbf{exists } \( r \Leftrightarrow r \neq \emptyset \)
- \textbf{not exists } \( r \Leftrightarrow r = \emptyset \)
Correlated Subqueries

• Yet another way of specifying the query “Find all courses taught in both the Fall 2009 semester and in the Spring 2010 semester”

```sql
select course_id
from section F
where semester = 'Fall' and year = 2009 and
exists (select *
        from section S
        where semester = 'Spring' and year = 2010
        and F.course_id = S.course_id);
```

• Correlation name or correlation variable

Not Exists

• Find all students who have taken all courses offered in the Biology department.

```sql
select distinct S.ID, S.name
from student as S
where not exists (select course_id
        from course
        where dept_name = 'Biology')
        minus
        (select T.course_id
        from takes as T
        where S.ID = T.ID);
```

- Note that $X - Y = \emptyset \iff X \subseteq Y$
- Note: Cannot write this query using = all and its variants
**Test for Absence of Duplicate Tuples**

- Find all courses that were offered at most once in 2009:

**WRONG:** unique is used to define constraints at table creation.

```sql
select T.course_id
from course T
where unique (select R.course_id
from section R
where T.course_id= R.course_id and R.year = 2009);
```

**RIGHT:**

```sql
select T.course_id from course T
where 1 = (select count(R.course_id)
from section R
where T.course_id= R.course_id and R.year = 2009);
```

**Derived Relations**

- SQL allows a subquery expression to be used in the `from` clause

- Find the average instructors’ salaries of those departments where the average salary is greater than $42,000.”

```sql
select dept_name, avg_salary
from (select dept_name, avg(salary) as avg_salary
from instructor
group by dept_name)
where avg_salary > 42000;
```

- Note that we aren’t using a `having` clause
Triggers

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

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

**Trigger Example in SQL:1999**

```sql
create trigger overdraft-trigger after update on account
referencing new row as nrow
for each row
when nrow.balance < 0
begin atomic
    actions to be taken
end
```
**Create Trigger Example in SQL:1999**

```sql
create trigger overdraft-trigger after update on account
  referencing new row as nrow
for each row
when nrow.balance < 0
begin atomic
  insert into borrower
  (select customer-name, account-number
    from depositor
    where nrow.account-number = depositor.account-number);
  insert into loan values
  (nrow.account-number, nrow.branch-name, nrow.balance);
  update account set balance = 0
  where account.account-number = nrow.account-number
end
```

---

**Triggers...**

- **External World Actions**
  - How does the DB *order* something if the inventory is low?

- **Syntax**
  - Every system has its own syntax

- **Careful with triggers**
  - Cascading triggers, Infinite Sequences...

- **More Info/Examples:**
  - [https://www.tutorialspoint.com/postgresql/postgresql_triggers.htm](https://www.tutorialspoint.com/postgresql/postgresql_triggers.htm)
  - Google “create trigger postgresql”
Views

- In some cases, it is not desirable for all users to see the entire logical model (that is, all the actual relations stored in the database.)
- Consider a person who needs to know an instructor’s name and department, but not the salary. This person should see a relation described, in SQL, by

  \[
  \text{select } \text{ID}, \text{name}, \text{dept}\_\text{name} \\
  \text{from instructor}
  \]

- A \textit{view} provides a mechanism to hide certain data from the view of certain users.
- Any relation that is not of the conceptual model but is made visible to a user as a “virtual relation” is called a \textit{view}.

View Definition

- A view is defined using the \texttt{create view} statement which has the form

  \[
  \text{create view } v \text{ as } <\text{query expression}>
  \]

  \textit{where }<\text{query expression}>\textit{ is any legal SQL expression. The view name is represented by } v. 
- Once a view is defined, the view name can be used to refer to the virtual relation that the view generates.
- View definition is not the same as creating a new relation by evaluating the query expression
  - Rather, a view definition \textit{causes the saving of an expression}; the expression is substituted into queries using the view.
Example Views

- A view of instructors without their salary

  ```sql
  create view faculty as
  select ID, name, dept_name
  from instructor
  ```

- Find all instructors in the Biology department

  ```sql
  select name
  from faculty
  where dept_name = 'Biology'
  ```

- Create a view of department salary totals

  ```sql
  create view departments_total_salary(dept_name, total_salary) as
  select dept_name, sum(salary)
  from instructor
  group by dept_name;
  ```

Views Defined Using Other Views

- create view physics_fall_2009 as

  ```sql
  select course.course_id, sec_id, building, room_number
  from course, section
  where course.course_id = section.course_id
  and course.dept_name = 'Physics'
  and section.semester = 'Fall'
  and section.year = '2009';
  ```

- create view physics_fall_2009_watson as

  ```sql
  select course_id, room_number
  from physics_fall_2009
  where building = 'Watson';
  ```
View Expansion

- Expand use of a view in a query/another view

```sql
create view physics_fall_2009_watson as
    select course_id, room_number
    from physics_fall_2009
    where building = 'Watson';
```

```sql
create view physics_fall_2009_watson as
    (select course_id, room_number
     from (select course.course_id, building, room_number
            from course, section
            where course.course_id = section.course_id
            and course.dept_name = 'Physics'
            and section.semester = 'Fall'
            and section.year = '2009')
     where building = 'Watson';
```

Views Defined Using Other Views

- One view may be used in the expression defining another view,
- A view relation $v_1$ is said to depend directly on a view relation $v_2$ if $v_2$ is used in the expression defining $v_1$
- A view relation $v_1$ is said to depend on view relation $v_2$ if either $v_1$ depends directly to $v_2$ or there is a path of dependencies from $v_1$ to $v_2$
- A view relation $v$ is said to be recursive if it depends on itself.
View Expansion

- A way to interpret queries w/ views...
  - Let view \( v \) be defined by an expression \( e \) that may itself contain uses of view relations.
  - View expansion of an expression \( e \) repeats the following replacement step:
    
    ```
    repeat
    
    Find any view relation \( v \) in \( e \)
    Replace the view relation \( v \) by expression \( e_i \)
    until no more view relations are present in \( e \)
    ```
  - As long as the view definitions are not recursive, this loop will terminate.

Update of (through) a View

- Add a new tuple to `faculty` view which we defined earlier
  
  ```
  insert into faculty values ('30765', 'Green', 'Music');
  ```
  
  This insertion must be represented by the insertion of the tuple
  
  ```
  ('30765', 'Green', 'Music', null)
  ```

  into the `instructor` relation.
Some Updates cannot be Translated Uniquely

- **create view** instructor_info **as**
  - *select* ID, name, building
  - *from* instructor, department
  - *where* instructor.dept_name = department.dept_name;

- **insert into** instructor info **values** (’69987’, ’White’, ”Taylor’);
  - which department, if multiple departments in Taylor?
  - what if no department is in Taylor?

- Most SQL implementations allow updates only on simple views
  - The *from* clause has only one database relation.
  - The *select* clause contains only attribute names of the relation, and does not have any expressions, aggregates, or distinct specification.
  - Any attribute not listed in the *select* clause can be set to null
  - The query does not have a group by or having clause.

And Some Not at All

- **create view** history_instructors **as**
  - *select* *
  - *from* instructor
  - *where* dept_name = ’History’;

- Insert (’25566’, ’Brown’, ’Biology’, 100000) into history_instructors
What we will cover...

- We will mainly discuss structured data
  - That can be represented in tabular forms (called Relational data)
  - We will spend some time on XML
  - We will also spend some time on MapReduce-like stuff

- Still the biggest and most important business (?)
  - Well-defined problem with really good solutions that work
    - Contrast XQuery for XML vs SQL for relational
  - Solid technological foundations

- Many of the basic techniques however are directly applicable
  - E.g. reliable data storage etc.
  - Cf. Many recent attempts to add SQL-like capabilities, transactions to Mapreduce and related technologies
    - E.g., Spark DataFrames

What we will cover...

- representing information
  - data modeling
  - semantic constraints

- languages and systems for querying data
  - complex queries & query semantics
  - over massive data sets

- concurrency control for data manipulation
  - ensuring transactional semantics

- reliable data storage
  - maintain data semantics even if you pull the plug
  - fault tolerance
What we will cover...

- representing information
  - data modeling: relational models, E/R models, XML/JSON
  - semantic constraints: integrity constraints, triggers
- languages and systems for querying data
  - complex queries & query semantics: SQL
  - over massive data sets: indexes, query processing, optimization, parallelization/cluster processing, streaming
- concurrency control for data manipulation
  - ensuring transactional semantics: ACID properties, distributed consistency
- reliable data storage
  - maintain data semantics even if you pull the plug: durability
  - fault tolerance: RAID

Summary

- Why study databases?
  - Shift from computation/data to information
    - Always true in corporate domains
    - Increasing true for personal and scientific domains
  - Need has exploded in recent years
    - Data is growing at a very fast rate
  - Solving the data management problems is going to be a key
- Database Management Systems provide
  - Data abstraction: Key in evolving systems
  - Guarantees about data integrity
    - In presence of concurrent access, failures...
  - Speed !!
Today’s Plan

- SQL (Chapter 3, 4)
  - Views (4.2)
  - Triggers (5.3)
  - Transactions
  - Integrity Constraints (4.4)
  - Functions and Procedures (5.2), Recursive Queries (5.4), Authorization (4.6), Ranking (5.5)
  - Return to / Finishing the Relational Algebra
  - E/R Diagrams (7)

- Some Complex SQL Examples
Transactions

- A transaction is a sequence of queries and update statements executed as a single unit
  - Transactions are started implicitly and terminated by one of
    - commit work: makes all updates of the transaction permanent in the database
    - rollback work: undoes all updates performed by the transaction.

Motivating example
- Transfer of money from one account to another involves two steps:
  - deduct from one account and credit to another
- If one steps succeeds and the other fails, database is in an inconsistent state
  - Therefore, either both steps should succeed or neither should
- If any step of a transaction fails, all work done by the transaction can be undone by rollback work.
- Rollback of incomplete transactions is done automatically, in case of system failures

Transactions (Cont.)

- In most database systems, each SQL statement that executes successfully is automatically committed.
  - Each transaction would then consist of only a single statement
  - Automatic commit can usually be turned off, allowing multi-statement transactions, but how to do so depends on the database system
  - Another option in SQL:1999: enclose statements within
    begin atomic
    ...
    end
Integrity Constraints

- Predicates on the database
- Must always be true (checked whenever db gets updated)

- There are the following 4 types of IC’s:
  - **Key constraints** (1 table)
    - e.g., 2 accts can’t share the same acct_no
  - **Attribute constraints** (1 table)
    - e.g., accts must have nonnegative balance
  - **Referential Integrity constraints** (2 tables)
    - E.g. bnames associated w/ loans must be names of real branches
  - **Global Constraints** (n tables)
    - E.g., all loans must be carried by at least 1 customer with a savings acct

---

Key Constraints

Idea: specifies that a relation is a set, not a bag

SQL examples:

1. **Primary Key:**
   ```sql
   CREATE TABLE branch(
       bname CHAR(15) PRIMARY KEY,
       bcity CHAR(20),
       assets INT);
   ```
   or
   ```sql
   CREATE TABLE depositor(  
       cname CHAR(15),
       acct_no CHAR(5),
       PRIMARY KEY(cname, acct_no));
   ```

2. **Candidate Keys:**
   ```sql
   CREATE TABLE customer (  
       ssn CHAR(9) PRIMARY KEY,
       cname CHAR(15),
       address CHAR(30),
       city CHAR(10),
       UNIQUE (cname, address, city));
   ```
Key Constraints

Effect of SQL Key declarations
PRIMARY (A1, A2, .., An) or
UNIQUE (A1, A2, ..., An)

Insertions: check if any tuple has same values for A1, A2, .., An as any
inserted tuple. If found, reject insertion

Updates to any of A1, A2, ..., An: treat as insertion of entire tuple

Primary vs Unique (candidate)
1. 1 primary key per table, several unique keys allowed.
2. Only primary key can be referenced by “foreign key” (ref integrity)
3. DBMS may treat primary key differently
   (e.g.: create an index on PK)

How would you implement something like this?

Attribute Constraints

- Idea:
  - Attach constraints to values of attributes
  - Enhances types system (e.g.: >= 0 rather than integer)
- In SQL:
  1. NOT NULL
     e.g.: CREATE TABLE branch(
            bname CHAR(15) NOT NULL,
            ....
       )
     Note: declaring bname as primary key also prevents null values
  2. CHECK
     e.g.: CREATE TABLE depositor(
            ....
            balance int NOT NULL,
            CHECK( balance >= 0),
            ....
       )
     affect insertions, update in affected columns
Attribute Constraints

Domains: can associate constraints with DOMAINS rather than attributes

e.g: instead of:

```sql
CREATE TABLE depositor(
    ....
    balance INT NOT NULL,
    CHECK (balance >= 0)
)
```

One can write:

```sql
CREATE DOMAIN bank-balance INT (
    CONSTRAINT not-overdrawn CHECK (value >= 0),
    CONSTRAINT not-null-value CHECK( value NOT NULL));
```

```sql
CREATE TABLE depositor ( 
    ....
    balance  bank-balance, 
)
```

Advantages?

---

Attribute Constraints

Advantage of associating constraints with domains:

1. can avoid repeating specification of same constraint for multiple columns

2. can name constraints

   e.g.: CREATE DOMAIN bank-balance INT (
         CONSTRAINT not-overdrawn
         CHECK (value >= 0),
         CONSTRAINT not-null-value
         CHECK( value NOT NULL));

allows one to:

1. add or remove:

   ```sql
   ALTER DOMAIN bank-balance
   ADD CONSTRAINT capped
   CHECK( value <= 10000)
   ```

2. report better errors (know which constraint violated)
Referential Integrity Constraints

Idea: prevent “dangling tuples” (e.g.: a loan with a bname, *Kenmore*, when no *Kenmore* tuple in branch)

Referencing Relation (e.g. loan)

```
<table>
<thead>
<tr>
<th>Referencing Relation (e.g. loan)</th>
</tr>
</thead>
<tbody>
<tr>
<td>“foreign key” bname</td>
</tr>
</tbody>
</table>
```

Referenced Relation (e.g. branch)

```
<table>
<thead>
<tr>
<th>Referenced Relation (e.g. branch)</th>
</tr>
</thead>
<tbody>
<tr>
<td>primary key bname</td>
</tr>
</tbody>
</table>
```

Ref Integrity:
- ensure that:
  - foreign key value \( \rightarrow \) primary key value

(note: don’t need to ensure \( \leftarrow \), i.e., not all branches have to have loans)

---

CREATE TABLE A (   ......
FOREIGN KEY c REFERENCES B  action
............ )

Action:
1) left blank (deletion/update rejected)
2) ON DELETE SET NULL/ ON UPDATE SET NULL
   sets \( \text{t}_i[c] = \text{NULL}, \text{t}_j[c] = \text{NULL} \)
3) ON DELETE CASCADE
   deletes \( \text{t}_i, \text{t}_j \)
   ON UPDATE CASCADE
   sets \( \text{t}_i[c], \text{t}_j[c] \) to new key values
Global Constraints

Idea: two kinds

1) single relation (constraints spans multiple columns)
   - E.g.: `CHECK (total = svngs + check)` declared in the CREATE TABLE

SQL examples:
   All Bkln branches must have assets > 5M

   CREATE TABLE branch (  
   ...........  
   bcity CHAR(15),  
   assets INT,  
   CHECK (NOT(bcity = 'Bkln') OR assets > 5M))

   Affects:  
   insertions into branch  
   updates of bcity or assets in branch

Global Constraints

2) Multiple relations: every loan has a borrower with a savings account

   CHECK (NOT EXISTS (  
   SELECT *  
   FROM loan AS L  
   WHERE NOT EXISTS(  
   SELECT *  
   FROM borrower B, depositor D, account A  
   WHERE B.cname = D.cname AND  
   D.acct_no = A.acct_no AND  
   L.lno = B.lno))))

   Problem: Where to put this constraint? At depositor? Loan? ....

   Ans: None of the above:  
   CREATE ASSERTION loan-constraint  
   CHECK( ..... )

Checked with EVERY DB update! 
very expensive.....
## Summary: Integrity Constraints

<table>
<thead>
<tr>
<th>Constraint Type</th>
<th>Where declared</th>
<th>Affects...</th>
<th>Expense</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Constraints</td>
<td>CREATE TABLE (PRIMARY KEY, UNIQUE)</td>
<td>Insertions, Updates</td>
<td>Moderate</td>
</tr>
<tr>
<td>Attribute Constraints</td>
<td>CREATE TABLE CREATE DOMAIN (Not NULL, CHECK)</td>
<td>Insertions, Updates</td>
<td>Cheap</td>
</tr>
</tbody>
</table>
| Referential Integrity| Table Tag (FOREIGN KEY .... REFERENCES ....) | 1. Insertions into referencing rel'n  
2. Updates of referencing rel'n of relevant attrs  
3. Deletions from referenced rel'n  
4. Update of referenced rel'n | 1,2: like key constraints. Another reason to index/sort on the primary keys  
3,4: depends on a. update/delete policy chosen  
b. existence of indexes on foreign key |
| Global Constraints   | Table Tag (CHECK) or outside table (CREATE ASSERTION) | 1. For single rel'n constraint, with insertion, deletion of relevant attrs  
2. For assertions w/ every db modification | 1. cheap  
2. very expensive |