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**:

   ```
   CREATE TABLE branch(
     bname       CHAR(15) PRIMARY KEY,
     bcity       CHAR(20),
     assets      INT);
   ```

   or

   ```
   CREATE TABLE depositor(
     cname       CHAR(15),
     acct_no     CHAR(5),
     PRIMARY KEY(cname, acct_no));
   ```

2. **Candidate Keys**:

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

         One can write:
         CREATE DOMAIN bank-balance INT (
           CONSTRAINT not-overdrawn CHECK (value >= 0),
           CONSTRAINT not-null-value CHECK( value NOT NULL));

         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:
      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)  Referenced Relation (e.g. branch)

“foreign key” bname  primary key bname

Ref Integrity: ensure that:
foreign key value → primary key value

(note: don’t need to ensure ←, 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 ti[c] = NULL, tj[c] = NULL

3) ON DELETE CASCADE
deletes ti, tj
ON UPDATE CASCADE
sets ti[c], tj[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:
   1) single relation: 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 |

---

### Today’s Plan

**SQL (Chapter 3, 4)**

» Views (4.2)

» Triggers (5.3)

» Transactions (4.3)

» Integrity Constraints (4.4)

» Functions and Procedures (5.2), Authorization (4.6), Ranking (5.5)

» Return to / Finishing the Relational Algebra

» E/R Diagrams
**SQL Functions**

Function to count number of instructors in a department

```sql
create function dept_count (dept_name varchar(20))
returns integer
begin
    declare d_count integer;
    select count (*) into d_count
    from instructor
    where instructor.dept_name = dept_name
    return d_count;
end
```

Can use in queries:

```sql
select dept_name, budget
from department
where dept_count (dept_name) > 12
```

**SQL Procedures**

Same function as a procedure:

```sql
create procedure dept_count_proc (in dept_name varchar(20),
                               out d_count integer)
begin
    select count(*) into d_count
    from instructor
    where instructor.dept_name = dept_count_proc.dept_name
end
```

But use differently:

```sql
declare d_count integer;
call dept_count_proc( 'Physics', d_count);
```

**HOWEVER:** Syntax can be wildly different across different systems

- Was put in place by DBMS systems before standardization
- Hard to change once customers are already using it
**Recursion in SQL**

Example: find which courses are a prerequisite, whether directly or indirectly, for a specific course

```sql
with recursive rec_prereq(course_id, prereq_id) as (
    select course_id, prereq_id
    from prereq
    union
    select rec_prereq.course_id, prereq.prereq_id,
    from rec_prereq, prereq
    where rec_prereq.prereq_id = prereq.course_id
)
select *
from rec_prereq;
```

Makes SQL Turing Complete (i.e., you can write any program in SQL)

But: Just because you can, doesn't mean you should

---

**Authorization/Security**

GRANT and REVOKE keywords

- grant select on instructor to U₁, U₂, U₃
- revoke select on branch from U₁, U₂, U₃

Can provide select, insert, update, delete privileges

Can also create “Roles” and do security at the level of roles

Some databases support doing this at the level of individual “tuples”

- PostgreSQL: [https://www.postgresql.org/docs/10/ddl-rowsecurity.html](https://www.postgresql.org/docs/10/ddl-rowsecurity.html)
## Ranking

Ranking is done in conjunction with an order by specification.

Consider: \( student\_grades(ID, \text{GPA}) \)

Find the rank of each student.

\[
\text{select } ID, \text{ rank()} \text{ over (order by GPA desc) as s\_rank}
\]

\[
\text{from student\_grades}
\]

\[
\text{order by s\_rank}
\]

Equivalent to:

\[
\text{select } ID, (1 + (\text{select count(*) from student\_grades B where B.GPA > A.GPA})) \text{ as s\_rank}
\]

\[
\text{from student\_grades A}
\]

\[
\text{order by s\_rank};
\]

## Today’s Plan

SQL (Chapter 3, 4)

» Views (4.2)
» Triggers (5.3)
» Transactions (4.3)
» 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
Relational Algebra, Again

Procedural language

Six basic operators
  » select
  » project
  » union
  » set difference
  » Cartesian product
  » rename

Set semantics

The operators take one or more relations as inputs and give a new relation as a result.

Rename Operation

Allows us to name, and therefore to refer to, the results of relational-algebra expressions.

Allows us to refer to a relation by more than one name.

Example:

\[ \rho_X (E) \]

returns the expression \( E \) under the name \( X \)

If a relational-algebra expression \( E \) has arity \( n \), then

\[ \rho_X [A_1, A_2, \ldots, A_n] (E) \]

returns the result of expression \( E \) under the name \( X \), and with the attributes renamed to \( A_1, A_2, \ldots, A_n \).
Relational Algebra

Those are the basic operations

What about SQL Joins?
  » Compose multiple operators together
    \[ \sigma_{A=C}(r \times s) \]

Additional Operations
  » Set intersection
  » Natural join
  » Division
  » Assignment

Additional Operators

Set intersection (\( \cap \))
  » \( r \cap s = r - (r - s) \)
  » SQL Equivalent: intersect

Assignment (\( \leftarrow \))
  » A convenient way to right complex RA expressions
  » Essentially for creating “temporary” relations
    - \( temp1 \leftarrow \Pi_{R \cdot S}(r) \)
  » SQL Equivalent: “create table as...”
### Additional Operators: Joins

**Natural join (⋈)**
- A Cartesian product with equality condition on common attributes
- Example:
  - if \( r \) has schema \( R(A, B, C, D) \), and if \( s \) has schema \( S(E, B, D) \)
  - Common attributes: \( B \) and \( D \)
  - Then:
    \[
    r \bowtie s = \prod_{r.A, r.B, r.C, r.D, s.E} (\sigma_{r.B = s.B \land r.D = s.D} (r \times s))
    \]

**SQL Equivalent:**
- select \( r.A, r.B, r.C, r.D, s.E \) from \( r, s \) where \( r.B = s.B \) and \( r.D = s.D \), OR
- select * from \( r \) natural join \( s \)

### Additional Operators: Joins

**Equi-join**
- A join that only has equality conditions

**Theta-join (⋈θ)**
- \( r \bowtie_{θ} s = σ_{θ}(r \times s) \) (combines cartesian and select in single statement)

**Left outer join (⟕)**
- Say \( r(A, B), s(B, C) \)
- We need to somehow find the tuples in \( r \) that have no match in \( s \)
- Consider: \( (r - \pi_{A, B}(r \bowtie_{θ} s)) \)
- We are done:
  \[
  (r \bowtie_{θ} s) \cup \rho_{\text{temp}}(A, B, C) \left( (r - \pi_{A, B}(r \bowtie_{θ} s)) \times \{(\text{NULL})\} \right)
  \]
Additional Operators: Join Variations

Tables: \( r(A, B), s(B, C) \)

<table>
<thead>
<tr>
<th>name</th>
<th>Symbol</th>
<th>SQL Equivalent</th>
<th>RA expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>cross product</td>
<td>( \times )</td>
<td><code>select * from r, s;</code></td>
<td>( r \times s )</td>
</tr>
<tr>
<td>natural join</td>
<td>( \bowtie )</td>
<td>natural join</td>
<td>( \pi_{r.A, r.B, s.C} \sigma_{r.B = s.B}(r \times s) )</td>
</tr>
<tr>
<td>equi-join</td>
<td>( \bowtie_\theta )</td>
<td>(theta must be equality)</td>
<td></td>
</tr>
<tr>
<td>theta join</td>
<td>( \bowtie_\theta )</td>
<td>from .. where ( \theta; )</td>
<td>( \sigma_\theta(r \times s) )</td>
</tr>
<tr>
<td>left outer join</td>
<td>( r \bowtie s )</td>
<td>left outer join (with “on”)</td>
<td>(see previous slide)</td>
</tr>
<tr>
<td>full outer join</td>
<td>( r \bowtie s )</td>
<td>full outer join (with “on”)</td>
<td>-</td>
</tr>
<tr>
<td>(left) semijoin</td>
<td>( r \bowtie s )</td>
<td>none</td>
<td>( \pi_{r.A, r.B}(r \bowtie s) )</td>
</tr>
<tr>
<td>(left) antijoin</td>
<td>( r \bowtie s )</td>
<td>none</td>
<td>( r - \pi_{r.A, r.B}(r \bowtie s) )</td>
</tr>
</tbody>
</table>

Additional Operators: Division

Assume \( r(R), s(S) \), for queries where \( S \subseteq R \):

\( r \div s \)

Think of it as “opposite of Cartesian product”

\( r \div s = t \text{ iff } t \times s \subseteq r \)
Relational Algebra Examples

Find all loans of over $1200

\( \sigma_{\text{amount} > 1200} (\text{loan}) \)

Find the loan number for each loan of an amount greater than $1200

\( \Pi_{\text{loan-number}} (\sigma_{\text{amount} > 1200} (\text{loan})) \)

Find the names of all customers who have a loan, an account, or both, from the bank

\( \Pi_{\text{customer-name}} (\text{borrower}) \cup \Pi_{\text{customer-name}} (\text{depositor}) \)

Relational Algebra Examples

Find the names of all customers who have a loan and an account at bank.

\( \Pi_{\text{customer-name}} (\text{borrower}) \cap \Pi_{\text{customer-name}} (\text{depositor}) \)

Find the names of all customers who have a loan at the Perryridge branch.

\( \Pi_{\text{customer-name}} (\sigma_{\text{branch-name} = \text{Perryridge}} (\sigma_{\text{borrower.loan-number} = \text{loan.loan-number}} (\text{borrower x loan}))) \)

Find the largest account balance

1. Rename the account relation a \( d \)
\( \Pi_{\text{balance}(account)} - \Pi_{\text{account.balance}} (\sigma_{\text{account.balance} < \text{d.balance}} (\text{account x p}_d (\text{account}))) \)
Relational Algebra Examples

Find *largest* account balance(balance), assume \{(1), (2), (3)\}

1. Rename the account relation to *d*

\[ \Pi_{balance}(account) - \Pi_{account.balance} (\sigma_{account.balance < d.balance} (account \times \rho_{d} (account))) \]

Generalized Projection

- Extends the projection operation by allowing arithmetic functions to be used in the projection list.

\[ \Pi_{F1, F2, \ldots, Fn} (E) \]

- *E* is any relational-algebra expression
- Each of *F1*, *F2*, ..., *Fn* are are arithmetic expressions involving constants and attributes in the schema of *E*.
- Given relation *instructor*(ID, name, dept_name, salary) where salary is annual salary, get the same information but with monthly salary

\[ \Pi_{ID, name, dept_name, salary/12} (instructor) \]
Aggregate Functions and Operations

- **Aggregation function** takes a collection of values and returns a single value as a result.
  
  - `avg`: average value
  - `min`: minimum value
  - `max`: maximum value
  - `sum`: sum of values
  - `count`: number of values

- **Aggregate operation** in relational algebra

  \[
  G_1, G_2, \ldots, G_n \quad \bigg\vec{\quad} \quad F_1(A_1), F_2(A_2), \ldots, F_n(A_n) \quad (E)
  \]

  - `E` is any relational-algebra expression
  - `G_1, G_2, \ldots, G_n` is a list of attributes on which to group (can be empty)
  - Each `F_i` is an aggregate function
  - Each `A_i` is an attribute name

- Note: Some books/articles use `g` instead of \( \bigg\vec \) (Calligraphic G)

Aggregate Operation – Example

- Relation `r`:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>α</td>
<td>α</td>
<td>7</td>
</tr>
<tr>
<td>α</td>
<td>β</td>
<td>7</td>
</tr>
<tr>
<td>β</td>
<td>β</td>
<td>3</td>
</tr>
<tr>
<td>β</td>
<td>β</td>
<td>10</td>
</tr>
</tbody>
</table>

  \[
  \bigg\vec{\quad} \quad \text{sum}(c) \quad (r)
  \]

  - `sum(c)`: 27
Aggregate Operation – Example

- Find the average salary in each department
  \[ \text{dept\_name} \ G \ \text{avg\_salary} (\text{instructor}) \]

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
<th>dept_name</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>76766</td>
<td>Crick</td>
<td>Biology</td>
<td>72000</td>
</tr>
<tr>
<td>45565</td>
<td>Katz</td>
<td>Comp. Sci.</td>
<td>75000</td>
</tr>
<tr>
<td>10101</td>
<td>Srinivasan</td>
<td>Comp. Sci.</td>
<td>65000</td>
</tr>
<tr>
<td>83821</td>
<td>Brandt</td>
<td>Comp. Sci.</td>
<td>92000</td>
</tr>
<tr>
<td>98345</td>
<td>Kim</td>
<td>Elec. Eng.</td>
<td>80000</td>
</tr>
<tr>
<td>12121</td>
<td>Wu</td>
<td>Finance</td>
<td>90000</td>
</tr>
<tr>
<td>76543</td>
<td>Singh</td>
<td>Finance</td>
<td>80000</td>
</tr>
<tr>
<td>32343</td>
<td>El Said</td>
<td>History</td>
<td>60000</td>
</tr>
<tr>
<td>58583</td>
<td>Califieri</td>
<td>History</td>
<td>62000</td>
</tr>
<tr>
<td>15151</td>
<td>Mozart</td>
<td>Music</td>
<td>40000</td>
</tr>
<tr>
<td>33456</td>
<td>Gold</td>
<td>Physics</td>
<td>87000</td>
</tr>
<tr>
<td>22222</td>
<td>Einstein</td>
<td>Physics</td>
<td>95000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>dept_name</th>
<th>avg_salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Biology</td>
<td>72000</td>
</tr>
<tr>
<td>Comp. Sci.</td>
<td>77333</td>
</tr>
<tr>
<td>Elec. Eng.</td>
<td>80000</td>
</tr>
<tr>
<td>Finance</td>
<td>85000</td>
</tr>
<tr>
<td>History</td>
<td>61000</td>
</tr>
<tr>
<td>Music</td>
<td>40000</td>
</tr>
<tr>
<td>Physics</td>
<td>91000</td>
</tr>
</tbody>
</table>

Aggregate Functions (Cont.)

- Result of aggregation does not have a name
  - Can use rename operation to give it a name
  - For convenience, we permit renaming as part of aggregate operation

  \[ \text{dept\_name} \ G \ \text{avg\_salary} \ as \ \text{avg\_sal} (\text{instructor}) \]
**Modification of the Database**

- The content of the database may be modified using the following operations:
  - Deletion
  - Insertion
  - Updating

- All these operations can be expressed using the assignment operator

\[ temp1 \leftarrow R \times S \]
\[ temp2 \leftarrow \sigma_{r.A_1=s.A_1 \land \ldots \land r.A_n=s.A_n} (temp1) \]
\[ \text{result} = \Pi_{R \cup S} (temp2) \]

The result of R x S potentially has duplicated attributes. For example, \( r(A, B) \times s(B, C) \) results in tuples w/ attributes \{A, B, B, C\}; \( \prod_{R \cup S} \) gets rid of the extra B. Duplicated tuples are an entirely different thing, and are not present in the relational algebra.

---

**Multiset Relational Algebra**

- Pure relational algebra removes all duplicates
  - e.g. after projection

- Multiset relational algebra retains duplicates, to match SQL semantics
  - SQL duplicate retention was initially for efficiency, but is now a feature

- Multiset relational algebra defined as follows
  - selection: has as many duplicates of a tuple as in the input, if the tuple satisfies the selection
  - projection: one tuple per input tuple, even if it is a duplicate
  - cross product: If there are \( m \) copies of \( t1 \) in \( r \), and \( n \) copies of \( t2 \) in \( s \), there are \( m \times n \) copies of \( t1.t2 \) in \( r \times s \)
  - Other operators similarly defined
    - E.g. union: \( m + n \) copies, intersection: \( \min(m, n) \) copies
      - difference: \( \min(0, m - n) \) copies