Chapter 4: SQL

Basic Structure
- SQL is based on set and relational operations with certain modifications and enhancements.
- A typical SQL query has the form:

\[
\text{select } A_1, A_2, ..., A_n \text{ from } r_1, r_2, ..., r_m \text{ where } P
\]

- \(A_i\)s represent attributes.
- \(r_i\)s represent relations.
- \(P\) is a predicate.

This query is equivalent to the relational algebra expression.

\[
\prod_{A_1, A_2, ..., A_n}(\sigma_P(r_1 \times r_2 \times ... \times r_m))
\]

The result of an SQL query is a relation.

The select Clause
- The select clause corresponds to the projection operation of the relational algebra. It is used to list the attributes desired in the result of a query.

\[
\text{select } A_1, A_2, ..., A_n \text{ from } r_i
\]

- In the "pure" relational algebra syntax, the query would be:

\[
\prod_{A_1, A_2, ..., A_n}(\sigma_P(r_i))
\]

- An asterisk in the select clause denotes "all attributes".

\[
\text{select * from } r_i
\]

- NOTE: SQL does not permit the `\(-\)` character in names, so you would use, for example, `branch_name` instead of `branch-name` in a real implementation. We use `-` since it looks nicer!

- NOTE: SQL names are case insensitive, meaning you can use upper case or lower case.

The select Clause (Cont.)
- SQL allows duplicates in relations as well as in query results.

To force the elimination of duplicates, insert the keyword distinct after select.

\[
\text{select distinct } A_1, A_2, ..., A_n \text{ from } r_i
\]

Find the names of all branches in the loan relation, and remove duplicates.

\[
\text{select all } A_1, A_2, ..., A_n \text{ from } r_i
\]

The keyword all specifies that duplicates not be removed.

The select Clause (Cont.)
- The select clause can contain arithmetic expressions involving the operation, +, -, *, and /, and operating on constants or attributes of tuples.

\[
\text{select } \text{loan-number}, \text{branch-name}, \text{amount} \times 100 \text{ from } \text{loan}
\]

Would return a relation which is the same as the loan relations, except that the attribute amount is multiplied by 100.
The where Clause

- The where clause corresponds to the selection predicate of the relational algebra. If consists of a predicate involving attributes of the relations that appear in the from clause.
- The find all loan number for loans made at the Perryridge branch with loan amounts greater than $1200.
- Comparison results can be combined using the logical connectives and, or, and not.
- Comparisons can be applied to results of arithmetic expressions.

The where Clause (Cont.)

- SQL includes a between comparison operator in order to simplify where clauses that specify that a value be less than or equal to some value and greater than or equal to some other value.
- Find the loan number of those loans with loan amounts between $90,000 and $100,000 (that is, $90,000 and $100,000).

The from Clause

- The from clause corresponds to the Cartesian product operation of the relational algebra. It lists the relations to be scanned in the evaluation of the expression.
- Find the Cartesian product borrower x loan

The Rename Operation

- The SQL allows renaming relations and attributes using the as clause:
- Find the name, loan number and loan amount of all customers having a loan at the Perryridge branch.

Tuple Variables

- Tuple variables are defined in the from clause via the use of the as clause.
- Find the customer names and their loan numbers for all customers having a loan at some branch.
- Find the names of all branches that have greater assets than some branch located in Brooklyn.

String Operations

- SQL includes a string-matching operator for comparisons on character strings. Patterns are described using two special characters:
- SQL supports a variety of string operations such as concatenation (using "||"), converting from upper to lower case (and vice versa), finding string length, extracting substrings, etc.
Find all customers who have an account but no loan.

Find all customers who have both a loan and an account.

Find all customers who have a loan, an account, or both:

We may specify

List in alphabetic order the names of all customers having a loan except

Set operations

Duplicates

Set Operations

Aggregate Functions
Aggregate Functions (Cont.)
- Find the average account balance at the Perryridge branch.
  \[ \text{select } \text{avg}(\text{balance}) \text{ from account where branch-name = 'Perryridge'} \]
- Find the number of tuples in the customer relation.
  \[ \text{select } \text{count}(*) \text{ from customer} \]
- Find the number of depositors in the bank.
  \[ \text{select } \text{count}((\text{distinct customer-name}) \text{ from depositor} \]

Aggregate Functions – Group By
- Find the number of depositors for each branch.
  \[ \text{select branch-name, } \text{count}((\text{distinct customer-name}) \text{ from depositor, account where depositor.account-number = account.account-number group by branch-name} \]
Note: Attributes in select clause outside of aggregate functions must appear in group by list.

Aggregate Functions – Having Clause
- Find the names of all branches where the average account balance is more than $1,200.
  \[ \text{select branch-name, } \text{avg}(\text{balance}) \text{ from account group by branch-name having } \text{avg}(\text{balance}) > 1200 \]
Note: predicates in the having clause are applied after the formation of groups whereas predicates in the where clause are applied before forming groups.

Null Values
- It is possible for tuples to have a null value, denoted by null, for some of their attributes.
- null signifies an unknown value or that a value does not exist.
- The predicate is null can be used to check for null values.
  \[ \text{E.g. } \text{Find all loan number which appear in the loan relation with null values for amount.} \]
  \[ \text{select loan-number from loan where amount is null} \]
- The result of any arithmetic expression involving null is null
  \[ \text{E.g. } 5 + \text{null} \text{ returns null} \]
- However, aggregate functions simply ignore nulls
  \[ \text{more on this shortly} \]

Null Values and Three Valued Logic
- Any comparison with null returns unknown
  \[ \text{E.g. } 5 < \text{null} \text{ or null <> null} \text{ or null = null} \]
- Three-valued logic using the truth value unknown:
  \[ \text{OR: (unknown or true) = true, (unknown or false) = unknown (unknown or unknown) = unknown} \]
  \[ \text{AND: (true and unknown) = unknown, (false and unknown) = false, (unknown and unknown) = unknown} \]
  \[ \text{NOT: (not unknown) = unknown} \]
  \[ \text{"P is unknown" evaluates to true if predicate P evaluates to unknown} \]
  \[ \text{Result of where clause predicate is treated as false if it evaluates to unknown} \]

Null Values and Aggregates
- Total all loan amounts
  \[ \text{select sum(amount) from loan} \]
  \[ \text{Above statement ignores null amounts} \]
  \[ \text{result is null if there is no non-null amount, that is the} \]
  \[ \text{All aggregate operations except count(*) ignore tuples with null values on the aggregated attributes.} \]
Nested Subqueries

- SQL provides a mechanism for the nesting of subqueries.
- A subquery is a select-from-where expression that is nested within another query.
- A common use of subqueries is to perform tests for set membership, set comparisons, and set cardinality.

Example Query

- Find all customers who have both an account and a loan at the bank.
  ```
  select distinct customer-name
  from borrower
  where customer-name in (select customer-name
                          from depositor)
  ```

- Find all customers who have a loan at the bank but do not have an account at the bank.
  ```
  select distinct customer-name
  from borrower
  where customer-name not in (select customer-name
                                  from depositor)
  ```

Example Query

- Find all customers who have both an account and a loan at the Perryridge branch.
  ```
  select distinct customer-name
  from borrower, loan
  where borrower.loan-number = loan.loan-number
  and branch-name = 'Perryridge'
  and (branch-name, customer-name)
  in 
  (select branch-name, customer-name
   from depositor, account
   where depositor.account-number = account.account-number)
  ```

- Note: Above query can be written in a much simpler manner. The formulation above is simply to illustrate SQL features.

Set Comparison

- Find all branches that have greater assets than some branch located in Brooklyn.
  ```
  select distinct T.branch-name
  from branch as T, branch as S
  where T.assets > S.assets
  and S.branch-city = 'Brooklyn'
  ```

- Same query using > some clause
  ```
  select branch-name
  from branch
  where assets > some
  (select assets
   from branch
   where branch-city = 'Brooklyn')
  ```

Definition of Some Clause

- F <comp> some r ⇔ ∃ t ∈ r s.t. (F <comp> t)
- Where <comp> can be: <, ≤, >, =, ≠

<table>
<thead>
<tr>
<th>F</th>
<th>some</th>
<th>r</th>
<th>t</th>
<th>s.t. (F &lt;comp&gt; t)</th>
</tr>
</thead>
<tbody>
<tr>
<td>≠</td>
<td>some</td>
<td>5</td>
<td>5</td>
<td>true (since 0 ≠ 5)</td>
</tr>
<tr>
<td>=</td>
<td>some</td>
<td>5</td>
<td>5</td>
<td>true</td>
</tr>
<tr>
<td>&lt;</td>
<td>some</td>
<td>5</td>
<td>5</td>
<td>false</td>
</tr>
<tr>
<td>≤</td>
<td>some</td>
<td>5</td>
<td>5</td>
<td>true (read: 5 ≤ some tuple in the relation)</td>
</tr>
</tbody>
</table>

Definition of All Clause

- F <comp> all r ⇔ ∀ t ∈ r (F <comp> t)

<table>
<thead>
<tr>
<th>F</th>
<th>all</th>
<th>r</th>
<th>t</th>
<th>(F &lt;comp&gt; t)</th>
</tr>
</thead>
<tbody>
<tr>
<td>≠</td>
<td>all</td>
<td>5</td>
<td>5</td>
<td>false</td>
</tr>
<tr>
<td>=</td>
<td>all</td>
<td>5</td>
<td>5</td>
<td>true (since 5 = 5)</td>
</tr>
<tr>
<td>≤</td>
<td>all</td>
<td>5</td>
<td>5</td>
<td>true</td>
</tr>
<tr>
<td>&lt;</td>
<td>all</td>
<td>5</td>
<td>5</td>
<td>false (since 5 &lt; 4 and 5 &lt; 6)</td>
</tr>
<tr>
<td>(≠ all)</td>
<td>=</td>
<td>not in</td>
<td></td>
<td></td>
</tr>
<tr>
<td>However, (≠ all)</td>
<td>≠</td>
<td>in</td>
<td></td>
<td></td>
</tr>
<tr>
<td>However, (≠ all)</td>
<td>≠</td>
<td>in</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example Query

- Find the names of all branches that have greater assets than all branches located in Brooklyn.

\[
\text{select branch-name from branch where assets > (select assets from branch where branch-city = 'Brooklyn')}\]

Test for Empty Relations

- The `exists` construct returns the value true if the argument subquery is nonempty.
- `exists r \iff r \neq \emptyset`
- `not exists r \iff r = \emptyset`

Example Query

- Find all customers who have an account at all branches located in Brooklyn.

\[
\text{select distinct S.customer-name from depositor as S where not exists (select branch-name from branch where branch-city = 'Brooklyn') except (select R.branch-name from depositor as T, account as R where T.customer-name = T.customer-name and R.account-number = account.account-number and account.branch-name = 'Perryridge'))}
\]

- (Schema used in this example)
- Note that \( X - Y = \emptyset \iff X \subseteq Y \)
- Note: Cannot write this query using `all` and its variants

Example Query

- Find all customers who have at least two accounts at the Perryridge branch.

\[
\text{select distinct T.customer-name from depositor T where not unique (select R.customer-name from account, depositor as R where T.customer-name = R.customer-name and R.account-number = account.account-number and account.branch-name = 'Perryridge'))}
\]

- (Schema used in this example)

Test for Absence of Duplicate Tuples

- The `unique` construct tests whether a subquery has any duplicate tuples in its result.
- Find all customers who have at most one account at the Perryridge branch.

\[
\text{select T.customer-name from depositor as T where unique (select R.customer-name from account, depositor as R where T.customer-name = R.customer-name and R.account-number = account.account-number and account.branch-name = 'Perryridge'))}
\]

- (Schema used in this example)

Example Query

- Find all customers who have at least two accounts at the Perryridge branch.

\[
\text{select distinct T.customer-name from depositor T where not unique (select R.customer-name from account, depositor as R where T.customer-name = R.customer-name and R.account-number = account.account-number and account.branch-name = 'Perryridge'))}
\]

- (Schema used in this example)

Views

- Provide a mechanism to hide certain data from the view of certain users. To create a view we use the command:

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

where:
- `<query expression>` is any legal expression
- The view name is represented by `v`
Example Queries

- A view consisting of branches and their customers
  ```sql
  create view all-customer as
  (select branch-name, customer-name
  from depositor, account
  where depositor.account-number = account.account-number)
  union
  (select branch-name, customer-name
  from borrower, loan
  where borrower.loan-number = loan.loan-number)
  
  - Find all customers of the Perryridge branch
    ```sql
    select * from all-customer
    where branch-name = 'Perryridge'
    ```

Derived Relations

- Find the average account balance of those branches where the average account balance is greater than $1200.
  ```sql
  select branch-name, avg-balance
  from (select branch-name, avg(balance)
  from account
  group by branch-name)
  as result
  where avg-balance > 1200
  
  Note that we do not need to use the having clause, since we compute the temporary (view) relation result in the from clause, and the attributes of result can be used directly in the where clause.

With Clause

- With clause allows views to be defined locally to a query, rather than globally. Analogous to procedures in a programming language.
- Find all accounts with the maximum balance
  ```sql
  with max-balance(value)
  as
  select max(balance)
  from account
  
  select account-number
  from account, max-balance
  where account.balance = max-balance.value
  ```

Complex Query using With Clause

- Find all branches where the total account deposit is greater than the average of the total account deposits at all branches
  ```sql
  with branch-total(branch-name, value)
  as
  select branch-name, sum(balance)
  from account
  group by branch-name
  
  with branch-total-avg(value)
  as
  select avg(value)
  from branch-total
  
  select branch-name
  from branch-total, branch-total-avg
  where branch-total.value >= branch-total-avg.value
  ```

Modification of the Database – Deletion

- Delete all account records at the Perryridge branch
  ```sql
  delete from account
  where branch-name = 'Perryridge'
  ```
- Delete all accounts at every branch located in Needham city.
  ```sql
  delete from account
  where branch-name in (select branch-name
  from branch
  where branch-city = 'Needham')
  
  delete from depositor
  where account-number in
  (select account-number
  from branch, account
  where branch-city = 'Needham'
  and branch.branch-name = account.branch-name)
  ```

Example Query

- Delete the record of all accounts with balances below the average at the bank.
  ```sql
  delete from account
  where balance < (select avg (balance)
  from account)
  ```

- Problem: as we delete tuples from deposit, the average balance changes
- Solution used in SQL:
  1. First, compute avg balance and find all tuples to delete
  2. Next, delete all tuples found above (without recomputing avg or retesting the tuples)
Modification of the Database – Insertion

- Add a new tuple to account
  ```sql
  insert into account
  values ('A-9732', 'Perryridge', 1200)
  or equivalently
  insert into account (branch-name, balance, account-number)
  values ('Perryridge', 1200, 'A-9732')
  ```
- Add a new tuple to account with balance set to null
  ```sql
  insert into account
  values ('A-777', 'Perryridge', null)
  ```

Modification of the Database – Updates

- Increase all accounts with balances over $10,000 by 6%, all other accounts receive 5%.
  ```sql
  update account
  set balance = balance * 1.06
  where balance > 10000
  ```

Case Statement for Conditional Updates

- Same query as before: Increase all accounts with balances over $10,000 by 6%, all other accounts receive 5%.
  ```sql
  update account
  set balance =
  case when balance <= 10000 then balance * 1.05
  else balance * 1.06
  end
  ```

Update of a View

- Create a view of all loan data in loan relation, hiding the amount attribute
  ```sql
  create view branch-loan as
  select branch-name, loan-number
  from loan
  ```

Transactions

- A transaction is a sequence of queries and update statements executed as a single unit
  ```sql
  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
  ```sql
  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

Joined Relations

- Join operations take two relations and return as a result another relation.
  - These additional operations are typically used as subquery expressions in the from clause
  - Join condition – defines which tuples in the two relations match, and what attributes are present in the result of the join.
  - Join type – defines how tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated.

Join Conditions

- natural
- on <predicate>
- using (A₁, A₂,..., An)

Join Types

- inner join
- left outer join
- right outer join
- full outer join

Joined Relations – Datasets for Examples

Relation loan

<table>
<thead>
<tr>
<th>loan-number</th>
<th>branch-name</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-170</td>
<td>Downtown</td>
<td>3000</td>
</tr>
<tr>
<td>L-230</td>
<td>Redwood</td>
<td>4000</td>
</tr>
<tr>
<td>L-260</td>
<td>Perryridge</td>
<td>1700</td>
</tr>
</tbody>
</table>

Relation borrower

<table>
<thead>
<tr>
<th>customer-name</th>
<th>loan-number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jones</td>
<td>L-170</td>
</tr>
<tr>
<td>Smith</td>
<td>L-230</td>
</tr>
<tr>
<td>Hayes</td>
<td>L-155</td>
</tr>
</tbody>
</table>

Note: borrower information missing for L-260 and loan information missing for L-155

Joined Relations – Examples

- loan inner join borrower on loan.loan-number = borrower.loan-number

<table>
<thead>
<tr>
<th>loan-number</th>
<th>branch-name</th>
<th>amount</th>
<th>customer-name</th>
<th>loan-number</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-170</td>
<td>Downtown</td>
<td>3000</td>
<td>Jones</td>
<td>L-170</td>
</tr>
<tr>
<td>L-230</td>
<td>Redwood</td>
<td>4000</td>
<td>Smith</td>
<td>L-230</td>
</tr>
</tbody>
</table>

- loan left inner join borrower on loan.loan-number = borrower.loan-number

<table>
<thead>
<tr>
<th>loan-number</th>
<th>branch-name</th>
<th>amount</th>
<th>customer-name</th>
<th>loan-number</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-170</td>
<td>Downtown</td>
<td>3000</td>
<td>Jones</td>
<td>L-170</td>
</tr>
<tr>
<td>L-230</td>
<td>Redwood</td>
<td>4000</td>
<td>Smith</td>
<td>L-230</td>
</tr>
<tr>
<td>L-260</td>
<td>Perryridge</td>
<td>1700</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>

- loan natural inner join borrower

<table>
<thead>
<tr>
<th>loan-number</th>
<th>branch-name</th>
<th>amount</th>
<th>customer-name</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-170</td>
<td>Downtown</td>
<td>3000</td>
<td>Jones</td>
</tr>
<tr>
<td>L-230</td>
<td>Redwood</td>
<td>4000</td>
<td>Smith</td>
</tr>
</tbody>
</table>

- loan natural right outer join borrower

<table>
<thead>
<tr>
<th>loan-number</th>
<th>branch-name</th>
<th>amount</th>
<th>customer-name</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-170</td>
<td>Downtown</td>
<td>3000</td>
<td>Jones</td>
</tr>
<tr>
<td>L-230</td>
<td>Redwood</td>
<td>4000</td>
<td>Smith</td>
</tr>
</tbody>
</table>

- loan full outer join borrower using (loan-number)

<table>
<thead>
<tr>
<th>loan-number</th>
<th>branch-name</th>
<th>amount</th>
<th>customer-name</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-170</td>
<td>Downtown</td>
<td>3000</td>
<td>Jones</td>
</tr>
<tr>
<td>L-230</td>
<td>Redwood</td>
<td>4000</td>
<td>Smith</td>
</tr>
<tr>
<td>L-260</td>
<td>Perryridge</td>
<td>1700</td>
<td>null</td>
</tr>
<tr>
<td>L-155</td>
<td>null</td>
<td>null</td>
<td>Hayes</td>
</tr>
</tbody>
</table>

Find all customers who have either an account or a loan (but not both) at the bank. select customer-name from depositor natural full outer join borrower where account-number is null or loan-number is null
Data Definition Language (DDL)

Allows the specification of not only a set of relations but also information about each relation, including:
- The schema for each relation.
- The domain of values associated with each attribute.
- Integrity constraints.
- The set of indices to be maintained for each relation.
- Security and authorization information for each relation.
- The physical storage structure of each relation on disk.

Date/Time Types in SQL (Cont.)

- **date.** Dates, containing a (4 digit) year, month and date
  - E.g. date '2001-7-27'
- **time.** Time of day, in hours, minutes and seconds.
  - E.g. time '09:00:30'
- **timestamp.** Date plus time of day
  - E.g. '2001-7-27 09:00:30.75'
- **Interval.** Period of time
  - E.g. Interval '1' day
- **cast.** String types to date/time/timestamp
  - E.g. cast <string-valued-expression> as date

Create Table Construct

- An SQL relation is defined using the create table command:
  ```sql
  create table r (A1 D1, A2 D2, ... , An Dn, ... (integrity-constraintk)), ...
  (integrity-constraint1), ...
  (integrity-constraint1), ...
  r is the name of the relation
  each Ai is an attribute name in the schema of relation r
  Di is the data type of values in the domain of attribute Ai
  ```

Example:
```
create table branch
  (branch-name char(15) not null, branch-city char(30), assets integer)
```

Integrity Constraints in Create Table

- **not null**
- **primary key (A1, ..., An)**
- **check (P),** where P is a predicate

Example: Declare branch-name as the primary key for branch and ensure that the values of assets are non-negative.
```
create table branch
  (branch-name char(15), branch-city char(30), assets integer, 
  primary key (branch-name), check (assets >= 0))
```

Drop and Alter Table Constructs

- The drop table command deletes all information about the dropped relation from the database.
- The alter table command is used to add attributes to an existing relation. All tuples in the relation are assigned null as the value for the new attribute. The form of the alter table command is
  ```sql
  alter table r add A D
  ```
  where A is the name of the attribute to be added to relation r and D is the domain of A.
- The alter table command can also be used to drop attributes of a relation
  ```sql
  alter table r drop A
  ```
  where A is the name of an attribute of relation r

Dropping of attributes not supported by many databases

Null values are allowed in all the domain types. Declaring an attribute to be not null prohibits null values for that attribute.
```
create domain person-name char(30) not null
```
Other SQL Features

- SQL sessions
  - client connects to an SQL server, establishing a session
  - executes a series of statements
  - disconnects the session
  - can commit or rollback the work carried out in the session
- An SQL environment contains several components, including a user identifier, and a schema, which identifies which of several schemas a session is using.

Schemas, Catalogs, and Environments

- Three-level hierarchy for naming relations.
  - Database contains multiple catalogs
  - each catalog can contain multiple schemas
  - SQL objects such as relations and views are contained within a schema
  - e.g. catalog5.bank-schema.account
  - Each user has a default catalog and schema, and the combination is unique to the user.
  - Default catalog and schema are set up for a connection
  - Catalog and schema can be omitted, defaults are assumed
  - Multiple versions of an application (e.g. production and test) can run under separate schemas

Procedural Extensions and Stored Procedures

- SQL provides a module language
  - permits definition of procedures in SQL, with if-then-else statements, for and while loops, etc.
  - more in Chapter 9
- Stored Procedures
  - Can store procedures in the database
  - then execute them using the call statement
  - permit external applications to operate on the database without knowing about internal details
- These features are covered in Chapter 9 (Object Relational Databases)