Chapter 4: Advanced SQL
Chapter 4: Advanced SQL

- SQL Data Types and Schemas
- Integrity Constraints
- Authorization
- Embedded SQL
- Dynamic SQL
- Functions and Procedural Constructs**
- Recursive Queries**
- Advanced SQL Features**
**Built-in Data Types in SQL**

- **date**: Dates, containing a (4 digit) year, month and date
  - Example: `date '2005-7-27'`

- **time**: Time of day, in hours, minutes and seconds.
  - Example: `time '09:00:30'`  `time '09:00:30.75'`

- **timestamp**: date plus time of day
  - Example: `timestamp '2005-7-27 09:00:30.75'`

- **interval**: period of time
  - Example: `interval '1' day`
  - Subtracting a date/time/timestamp value from another gives an interval value
  - Interval values can be added to date/time/timestamp values
Build-in Data Types in SQL (Cont.)

- Can extract values of individual fields from date/time/timestamp
  - Example: `extract (year from r.starttime)`

- Can cast string types to date/time/timestamp
  - Example: `cast <string-valued-expression> as date`
  - Example: `cast <string-valued-expression> as time`
**User-Defined Types**

- **create type** construct in SQL creates user-defined type

  ```sql
  create type Dollars as numeric (12,2) final
  ```

- **create domain** construct in SQL-92 creates user-defined domain types

  ```sql
  create domain person_name char(20) not null
  ```

- Types and domains are similar. Domains can have constraints, such as **not null**, specified on them.
Domain Constraints

- **Domain constraints** are the most elementary form of integrity constraint. They test values inserted in the database, and test queries to ensure that the comparisons make sense.

- New domains can be created from existing data types
  - Example: `create domain Dollars numeric(12, 2)
create domain Pounds numeric(12,2)`

- We cannot assign or compare a value of type Dollars to a value of type Pounds.
  - However, we can convert type as below
    - `(cast r.A as Pounds)`
    - (Should also multiply by the dollar-to-pound conversion-rate)
Large-object types

- Large objects (photos, videos, CAD files, etc.) are stored as a large object:
  - **blob**: binary large object -- object is a large collection of uninterpreted binary data (whose interpretation is left to an application outside of the database system)
  - **clob**: character large object -- object is a large collection of character data
  - When a query returns a large object, a pointer is returned rather than the large object itself.
Integrity Constraints

- Integrity constraints guard against accidental damage to the database, by ensuring that authorized changes to the database do not result in a loss of data consistency.
- A checking account must have a balance greater than $10,000.00
- A salary of a bank employee must be at least $4.00 an hour
- A customer must have a (non-null) phone number
Constraints on a Single Relation

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

- Declare `branch_name` for `branch` is not null
  
  `branch_name` char(15) not null

- Declare the domain `Dollars` to be not null

  `create domain Dollars numeric(12,2) not null`
The Unique Constraint

- **unique** (A₁, A₂, …, Aₘ)
- The unique specification states that the attributes A₁, A₂, …, Aₘ form a candidate key.
- Candidate keys are permitted to be null (in contrast to primary keys).
The check clause

- **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.

```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.)

- The check clause in SQL-92 permits domains to be restricted:
  - 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 domain has a constraint that ensures that the hourly_wage is greater than 4.00

- The clause constraint value_test is optional; useful to indicate which constraint an update violated.
Referential Integrity

- Ensures that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.
  - Example: If “Perryridge” is a branch name appearing in one of the tuples in the account relation, then there exists a tuple in the branch relation for branch “Perryridge”.

- Primary and candidate keys and foreign keys can be specified as part of the SQL create table statement:
  - The primary key clause lists attributes that comprise the primary key.
  - The unique key clause lists attributes that comprise a candidate key.
  - The foreign key clause lists the attributes that comprise the foreign key and the name of the relation referenced by the foreign key. By default, a foreign key references the primary key attributes of the referenced table.
Referential Integrity in SQL – Example

create table customer
    (customer_name char(20),
     customer_street char(30),
     customer_city  char(30),
     primary key (customer_name ))

create table branch
    (branch_name char(15),
     branch_city  char(30),
     assets       numeric(12,2),
     primary key (branch_name ))
Referential Integrity in SQL – Example (Cont.)

create table account
    (account_number char(10),
     branch_name    char(15),
     balance        integer,
     primary key    (account_number),
     foreign key    (branch_name) references branch )

create table depositor
    (customer_name  char(20),
     account_number char(10),
     primary key    (customer_name, account_number),
     foreign key    (account_number ) references account,
     foreign key    (customer_name ) references customer )
**Assertions**

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

- An assertion in SQL takes the form

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

- When an assertion is made, the system tests it for validity, and tests it again on every update that may violate the assertion
  - This testing may introduce a significant amount of overhead; hence assertions should be used with great care.

- Asserting
  - for all \( X, P(X) \)
  - is achieved in a round-about fashion using
  - not exists \( X \) such that not \( P(X) \)
Assertion Example

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)))
```
Assertion Example

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

```sql
create assertion sum_constraint check
  (not exists (select *
    from branch
    where (select sum(amount )
      from loan
      where loan.branch_name =
        branch.branch_name )
    >= (select sum (amount )
      from account
      where loan.branch_name =
        branch.branch_name )))
```
Authorization

Forms of authorization on parts of the database:

- **Read** - allows reading, but not modification of data.
- **Insert** - allows insertion of new data, but not modification of existing data.
- **Update** - allows modification, but not deletion of data.
- **Delete** - allows deletion of data.

Forms of authorization to modify the database schema (covered in Chapter 8):

- **Index** - allows creation and deletion of indices.
- **Resources** - allows creation of new relations.
- **Alteration** - allows addition or deletion of attributes in a relation.
- **Drop** - allows deletion of relations.
Authorization Specification in SQL

- The **grant** statement is used to confer authorization
  
  ```sql
  grant <privilege list>
  on <relation name or view name> to <user list>
  ```

- **<user list>** is:
  - a user-id
  - **public**, which allows all valid users the privilege granted
  - A role (more on this in Chapter 8)

- Granting a privilege on a view does not imply granting any privileges on the underlying relations.

- The grantor of the privilege must already hold the privilege on the specified item (or be the database administrator).
Privileges in SQL

- **select**: allows read access to relation, or the ability to query using the view
  - Example: grant users $U_1$, $U_2$, and $U_3$ **select** authorization on the *branch* relation:
    
    ```
    grant select on branch to $U_1$, $U_2$, $U_3$
    ```

- **insert**: the ability to insert tuples
- **update**: the ability to update using the SQL update statement
- **delete**: the ability to delete tuples.
- **all privileges**: used as a short form for all the allowable privileges
- more in Chapter 8
Revoking Authorization in SQL

- The `revoke` statement is used to revoke authorization.
  
  ```
  revoke <privilege list> 
  on <relation name or view name> from <user list>
  ```

- Example:
  
  ```
  revoke select on branch from U1, U2, U3
  ```

- `<privilege-list>` may be `all` to revoke all privileges the revokee may hold.

- If `<revokee-list>` includes `public`, all users lose the privilege except those granted it explicitly.

- If the same privilege was granted twice to the same user by different grantees, the user may retain the privilege after the revocation.

- All privileges that depend on the privilege being revoked are also revoked.
Embedded SQL

The SQL standard defines embeddings of SQL in a variety of programming languages such as C, Java, and Cobol.

A language to which SQL queries are embedded is referred to as a **host language**, and the SQL structures permitted in the host language comprise **embedded SQL**.

The basic form of these languages follows that of the System R embedding of SQL into PL/I.

**EXEC SQL** statement is used to identify embedded SQL request to the preprocessor

```
EXEC SQL <embedded SQL statement > END_EXEC
```

Note: this varies by language (for example, the Java embedding uses

```
# SQL { .... }; 
```
From within a host language, find the names and cities of customers with more than the variable amount dollars in some account.

Specify the query in SQL and declare a cursor for it

```
EXEC SQL
  declare c cursor for
  select depositor.customer_name, customer_city
  from depositor, customer, account
  where depositor.customer_name = customer.customer_name
    and depositor.account_number = account.account_number
    and account.balance > :amount
END_EXEC
```
The **open** statement causes the query to be evaluated

```
EXEC SQL open c END_EXEC
```

The **fetch** statement causes the values of one tuple in the query result to be placed on host language variables.

```
EXEC SQL fetch c into :cn, :cc END_EXEC
```
Repeated calls to **fetch** get successive tuples in the query result.

A variable called SQLSTATE in the SQL communication area (SQLCA) gets set to ‘02000’ to indicate no more data is available.

The **close** statement causes the database system to delete the temporary relation that holds the result of the query.

```
EXEC SQL close c END_EXEC
```

Note: above details vary with language. For example, the Java embedding defines Java iterators to step through result tuples.
Updates Through Cursors

- Can update tuples fetched by cursor by declaring that the cursor is for update

  ```sql
  declare c cursor for
  select *
  from account
  where branch_name = 'Perryridge'
  for update
  ```

- To update tuple at the current location of cursor `c`

  ```sql
  update account
  set balance = balance + 100
  where current of c
  ```
Dynamic SQL

- Allows programs to construct and submit SQL queries at run time.
- Example of the use of dynamic SQL from within a C program.

```c
char * sqlprog = "update account
set balance = balance * 1.05
where account_number = ?"
EXEC SQL prepare dynprog from :sqlprog;
EXEC SQL execute dynprog using :account;
```

- The dynamic SQL program contains a ?, which is a place holder for a value that is provided when the SQL program is executed.
ODBC and JDBC

- API (application-program interface) for a program to interact with a database server
- Application makes calls to
  - Connect with the database server
  - Send SQL commands to the database server
  - Fetch tuples of result one-by-one into program variables
- ODBC (Open Database Connectivity) works with C, C++, C#, and Visual Basic
- JDBC (Java Database Connectivity) works with Java
ODBC

- Open DataBase Connectivity (ODBC) standard
  - standard for application program to communicate with a database server.
  - application program interface (API) to
    - open a connection with a database,
    - send queries and updates,
    - get back results.
- Applications such as GUI, spreadsheets, etc. can use ODBC
ODBC (Cont.)

- Each database system supporting ODBC provides a "driver" library that must be linked with the client program.
- When client program makes an ODBC API call, the code in the library communicates with the server to carry out the requested action, and fetch results.
- ODBC program first allocates an SQL environment, then a database connection handle.
- Opens database connection using SQLConnect(). Parameters for SQLConnect:
  - connection handle,
  - the server to which to connect
  - the user identifier,
  - password
- Must also specify types of arguments:
  - SQL_NTS denotes previous argument is a null-terminated string.
ODBC Code

```
int ODBCexample()
{
    RETCODE error;
    HENV env;  /* environment */
    HDBC conn;  /* database connection */
    SQLAllocEnv(&env);
    SQLAllocConnect(env, &conn);
    SQLConnect(conn, "aura.bell-labs.com", SQL_NTS, "avi", SQL_NTS, "avipasswd", SQL_NTS);
    { .... Do actual work ... }
    SQLDisconnect(conn);
    SQLFreeConnect(conn);
    SQLFreeEnv(env);
}
```
ODBC Code (Cont.)

- Program sends SQL commands to the database by using SQLExecDirect
- Result tuples are fetched using SQLFetch()
- SQLBindCol() binds C language variables to attributes of the query result
  - When a tuple is fetched, its attribute values are automatically stored in corresponding C variables.
  - Arguments to SQLBindCol()
    - ODBC stmt variable, attribute position in query result
    - The type conversion from SQL to C.
    - The address of the variable.
    - For variable-length types like character arrays,
      - The maximum length of the variable
      - Location to store actual length when a tuple is fetched.
      - Note: A negative value returned for the length field indicates null value
- Good programming requires checking results of every function call for errors; we have omitted most checks for brevity.
Main body of program

```c
char branchname[80];
float balance;
int lenOut1, lenOut2;
HSTMT stmt;

SQLAllocStmt(conn, &stmt);
char * sqlquery = "select branch_name, sum (balance)
from account
  group by branch_name";

error = SQLExecDirect(stmt, sqlquery, SQL_NTS);
if (error == SQL_SUCCESS) {
  SQLBindCol(stmt, 1, SQL_C_CHAR, branchname, 80, &lenOut1);
  SQLBindCol(stmt, 2, SQL_C_FLOAT, &balance, 0, &lenOut2);

  while (SQLFetch(stmt) >= SQL_SUCCESS) {
    printf ("%s %g\n", branchname, balance);
  }
}

SQLFreeStmt(stmt, SQL_DROP);
```
More ODBC Features

- **Prepared Statement**
  - SQL statement prepared: compiled at the database
  - Can have placeholders: E.g. `insert into account values(?,?,?,?)`
  - Repeatedly executed with actual values for the placeholders

- **Metadata features**
  - Finding all the relations in the database and
  - Finding the names and types of columns of a query result or a relation in the database.

- By default, each SQL statement is treated as a separate transaction that is committed automatically.
  - Can turn off automatic commit on a connection
    - `SQLSetConnectOption(conn, SQL_AUTOCOMMIT, 0)`
  - Transactions must then be committed or rolled back explicitly by
    - `SQLTransact(conn, SQL_COMMIT)` or
    - `SQLTransact(conn, SQL_ROLLBACK)`
ODBC Conformance Levels

- Conformance levels specify subsets of the functionality defined by the standard.
  - Core
  - Level 1 requires support for metadata querying
  - Level 2 requires ability to send and retrieve arrays of parameter values and more detailed catalog information.

- SQL Call Level Interface (CLI) standard similar to ODBC interface, but with some minor differences.
JDBC is a Java API for communicating with database systems supporting SQL.

JDBC supports a variety of features for querying and updating data, and for retrieving query results.

JDBC also supports metadata retrieval, such as querying about relations present in the database and the names and types of relation attributes.

Model for communicating with the database:

- Open a connection
- Create a “statement” object
- Execute queries using the Statement object to send queries and fetch results
- Exception mechanism to handle errors
public static void JDBCexample(String dbid, String userid, String passwd)
{
    try {
        Class.forName("oracle.jdbc.driver.OracleDriver");
        Connection conn = DriverManager.getConnection("jdbc:oracle:thin:@aura.bell-labs.com:2000:bankdb", userid, passwd);
        Statement stmt = conn.createStatement();
        ... Do Actual Work ....
        stmt.close();
        conn.close();
    }
    catch (SQLException sqle) {
        System.out.println("SQLException : "+ sqle);  
    }
}
JDBC Code (Cont.)

- Update to database

```java
try {
    stmt.executeUpdate("insert into account values ('A-9732', 'Perryridge', 1200")");
} catch (SQLException sqle) {
    System.out.println("Could not insert tuple. " + sqle);
}
```

- Execute query and fetch and print results

```java
ResultSet rset = stmt.executeQuery("select branch_name,
    avg(balance)
    from account
    group by branch_name");

while (rset.next()) {
    System.out.println(rset.getString("branch_name") + " " + rset.getFloat(2));
}
```
JDBC Code Details

- Getting result fields:
  - `rs.getString("branchname")` and `rs.getString(1)` equivalent if branchname is the first argument of select result.

- Dealing with Null values
  ```java
  int a = rs.getInt("a");
  if (rs.wasNull()) Systems.out.println("Got null value");
  ```
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)
Functions and Procedures

- SQL:1999 supports functions and procedures
  - Functions/procedures can be written in SQL itself, or in an external programming language
  - Functions are particularly useful with specialized data types such as images and geometric objects
    - Example: functions to check if polygons overlap, or to compare images for similarity
  - Some database systems support table-valued functions, which can return a relation as a result
- SQL:1999 also supports a rich set of imperative constructs, including
  - Loops, if-then-else, assignment
- Many databases have proprietary procedural extensions to SQL that differ from SQL:1999
SQL Functions

- Define a function that, given the name of a customer, returns the count of the number of accounts owned by the customer.

```sql
create function account_count (customer_name varchar(20)) returns integer begin
    declare a_count integer;
    select count (*) into a_count from depositor
    where depositor.customer_name = customer_name
    return a_count;
end
```

- Find the name and address of each customer that has more than one account.

```sql
select customer_name, customer_street, customer_city from customer
where account_count (customer_name) > 1
```
Table Functions

- SQL:2003 added functions that return a relation as a result
- Example: Return all accounts owned by a given customer

```sql
create function accounts_of (customer_name char(20))
returns table ( account_number char(10),
branch_name char(15),
balance numeric(12,2))

return table
(select account_number, branch_name, balance
from account A
where exists ( select *
from depositor D
where D.customer_name = accounts_of.customer_name
and D.account_number = A.account_number ))
```
Table Functions (cont’d)

- Usage

```sql
select *
from table (accounts_of ('Smith'))
```
The *author_count* function could instead be written as procedure:

```sql
create procedure account_count_proc (in title varchar(20),
                                      out a_count integer)
begin
    select count(author) into a_count
    from depositor
    where depositor.customer_name = account_count_proc.customer_name
end
```

Procedures can be invoked either from an SQL procedure or from embedded SQL, using the `call` statement.

```sql
declare a_count integer;
call account_count_proc( 'Smith', a_count);
```

Procedures and functions can be invoked also from dynamic SQL.

SQL:1999 allows more than one function/procedure of the same name (called name **overloading**), as long as the number of arguments differ, or at least the types of the arguments differ.
Procedural Constructs

- Compound statement: `begin ... end`,
  - May contain multiple SQL statements between `begin` and `end`.
  - Local variables can be declared within a compound statement.

- `While` and `repeat` statements:
  ```
  declare n integer default 0;
  while n < 10 do
    set n = n + 1
  end while

  repeat
    set n = n - 1
  until n = 0
  end repeat
  ```
Procedural Constructs (Cont.)

- **For** loop
  - Permits iteration over all results of a query
  - Example: find total of all balances at the Perryridge branch

```sql
declare n integer default 0;
for r as
  select balance from account
  where branch_name = 'Perryridge'
do
  set n = n + r.balance
end for
```
Procedural Constructs (cont.)

- Conditional statements (if-then-else)
  E.g. To find sum of balances for each of three categories of accounts (with balance <1000, >=1000 and <5000, >= 5000)

  ```
  if r.balance < 1000
    then set l = l + r.balance
  elseif r.balance < 5000
    then set m = m + r.balance
  else set h = h + r.balance
  end if
  ```

- SQL:1999 also supports a case statement similar to C case statement

- Signaling of exception conditions, and declaring handlers for exceptions

  ```
  declare out_of_stock condition
  declare exit handler for out_of_stock
  begin
    ...
    signal out-of-stock
  end
  ```

  - The handler here is exit -- causes enclosing begin..end to be exited
  - Other actions possible on exception
SQL:1999 permits the use of functions and procedures written in other languages such as C or C++

Declaring external language procedures and functions

```
create procedure account_count_proc(
in customer_name varchar(20),
    out count integer)
language C
external name ‘/usr/avi/bin/account_count_proc’
```

```
create function account_count(customer_name varchar(20))
returns integer
language C
external name ‘/usr/avi/bin/author_count’
```
External Language Routines (Cont.)

- Benefits of external language functions/procedures:
  - more efficient for many operations, and more expressive power

- Drawbacks
  - Code to implement function may need to be loaded into database system and executed in the database system’s address space
    - risk of accidental corruption of database structures
    - security risk, allowing users access to unauthorized data
  - There are alternatives, which give good security at the cost of potentially worse performance
  - Direct execution in the database system’s space is used when efficiency is more important than security
Security with External Language Routines

- To deal with security problems
  - Use *sandbox* techniques
    - that is use a safe language like Java, which cannot be used to access/damage other parts of the database code
  - Or, run external language functions/procedures in a separate process, with no access to the database process’ memory
    - Parameters and results communicated via inter-process communication
- Both have performance overheads
- Many database systems support both above approaches as well as direct executing in database system address space
Recursion in SQL

- SQL:1999 permits recursive view definition
- Example: find all employee-manager pairs, where the employee reports to the manager directly or indirectly (that is manager’s manager, manager’s manager’s manager, etc.)

```
with recursive empl (employee_name, manager_name) as (  
  select employee_name, manager_name  
  from manager  
  union  
  select manager.employee_name, empl.manager_name  
  from manager, empl  
  where manager.manager_name = empl.employe_name)  
select *  
from empl
```

This example view, empl, is called the transitive closure of the manager relation.
The Power of Recursion

- Recursive views make it possible to write queries, such as transitive closure queries, that cannot be written without recursion or iteration.
  - Intuition: Without recursion, a non-recursive non-iterative program can perform only a fixed number of joins of manager with itself
    - This can give only a fixed number of levels of managers
    - Given a program we can construct a database with a greater number of levels of managers on which the program will not work

- Computing transitive closure
  - The next slide shows a manager relation
  - Each step of the iterative process constructs an extended version of empl from its recursive definition.
  - The final result is called the fixed point of the recursive view definition.

- Recursive views are required to be monotonic. That is, if we add tuples to manager the view contains all of the tuples it contained before, plus possibly more
**Example of Fixed-Point Computation**

<table>
<thead>
<tr>
<th>employee_name</th>
<th>manager_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alon</td>
<td>Barinsky</td>
</tr>
<tr>
<td>Barinsky</td>
<td>Estovar</td>
</tr>
<tr>
<td>Corbin</td>
<td>Duarte</td>
</tr>
<tr>
<td>Duarte</td>
<td>Jones</td>
</tr>
<tr>
<td>Estovar</td>
<td>Jones</td>
</tr>
<tr>
<td>Jones</td>
<td>Klinger</td>
</tr>
<tr>
<td>Rensal</td>
<td>Klinger</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Iteration number</th>
<th>Tuples in empl</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>(Duarte), (Estovar)</td>
</tr>
<tr>
<td>2</td>
<td>(Duarte), (Estovar), (Barinsky), (Corbin)</td>
</tr>
<tr>
<td>3</td>
<td>(Duarte), (Estovar), (Barinsky), (Corbin), (Alon)</td>
</tr>
<tr>
<td>4</td>
<td>(Duarte), (Estovar), (Barinsky), (Corbin), (Alon)</td>
</tr>
</tbody>
</table>
Create a table with the same schema as an existing table:

```sql
create table temp_account like account
```

SQL:2003 allows subqueries to occur anywhere a value is required provided the subquery returns only one value. This applies to updates as well.

SQL:2003 allows subqueries in the `from` clause to access attributes of other relations in the `from` clause using the `lateral` construct:

```sql
select C.customer_name, num_accounts
from customer C,
     lateral (select count(*)
                           from account A
                           where A.customer_name = C.customer_name )
     as this_customer (num_accounts )
```
Advanced SQL Features (cont’d)

- Merge construct allows batch processing of updates.
- Example: relation `funds_received (account_number, amount)` has batch of deposits to be added to the proper account in the `account` relation

```sql
merge into account as A
    using (select *
                  from funds_received as F)
    on (A.account_number = F.account_number)
    when matched then
        update set balance = balance + F.amount
```
End of Chapter