Unit 3: Structured Query Language

SQL:

Structured Query Language is a special-purpose programming language designed for managing data held in a RDBMS. Originally based upon relational algebra and relational-calculus constructs, SQL consists of a data definition language and data manipulation language.
The SQL language has several parts:

- **Data-definition language (DDL).** The SQL DDL provides commands for defining relation schemas, deleting relations, and modifying relation schemas.
- **Interactive data-manipulation language (DML).** The SQL DML includes a query language based on both the relational algebra and the tuple relational calculus. It includes also commands to insert tuples into, delete tuples from, and modify tuples in the database.
- **View definition.** The SQL DDL includes commands for defining views.
- **Transaction control.** SQL includes commands for specifying the beginning and ending of transactions.
- **Embedded SQL and dynamic SQL.** Embedded and dynamic SQL define how SQL statements can be embedded within general-purpose programming languages, such as C, C++, Java, PL/I, Cobol, Pascal, and Fortran.
- **Integrity.** The SQL DDL includes commands for specifying integrity constraints that the data stored in the database must satisfy. Updates that violate integrity constraints are disallowed.
- **Authorization.** The SQL DDL includes commands for specifying access rights to relations and views.

In this chapter, we cover the DML and the basic DDL features of SQL. The enterprise that we use in the examples in this chapter, is a banking enterprise with the following relation schemas:

Branch-schema = (*branch-name, branch-city, assets*)
Customer-schema = (*customer-name, customer-street, customer-city*)
Loan-schema = (*loan-number, branch-name, amount*)
Borrower-schema = (*customer-name, loan-number*)
Account-schema = (*account-number, branch-name, balance*)
Depositor-schema = (*customer-name, account-number*)

Basic Structure of SQL

The basic structure of an SQL expression consists of three clauses: **select**, **from**, and **where**.

- 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.
- 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.
- The **where** clause corresponds to the selection predicate of the relational algebra. It consists of a predicate involving attributes of the relations that appear in the **from** clause.

A typical SQL query has the form

select *A*1*, A*2*, . . .,An*
from *r*1*, r*2*, . . . , rm*
where *P*

Each *Ai* represents an attribute, and each *ri* a relation. *P* is a predicate. The query is equivalent to the relational-algebra expression

$\Pi A1, A2,...,An(\sigma P\ (r1 \times r2 \times \cdots \times rm))$

If the **where** clause is omitted, the predicate *P* is **true**. However, unlike the result of a relational-algebra expression, the result of the SQL query may containmultiple copies of some tuples.

SQL forms the Cartesian product of the relations named in the **from** clause, performs a relational-algebra selection using the **where** clause predicate, and then projects the result onto the attributes of the **select** clause.

The select Clause:

Let us consider a simple query using our banking example, "Find the names of all branches in the *loan* relation":

select *branch-name*
from *loan*

The result is a relation consisting of a single attribute with the heading *branch-name*. It display all branch-name with duplicates also if there any.

In those cases where we want to force the elimination of duplicates, we insert the keyword **distinct** after **select**. We can rewrite the preceding query as

select distinct *branch-name*
from *loan*

if we want duplicates removed.

SQL allows us to use the keyword **all** to specify explicitly that duplicates are not removed:

select all *branch-name*
from *loan*

Since duplicate retention is the default, we will not use **all** in the examples. The asterisk symbol **" * "** can be used to denote "all attributes." A select clause of the form **select** * indicates that all attributes of all relations appearing in the **from** clause are selected.

The **select** clause may also contain arithmetic expressions involving the operators +, −, *, and / operating on constants or attributes of tuples. For example, the query

select *loan-number*, *branch-name*, *amount* * 100
from *loan*

will return a relation that is the same as the *loan* relation, except that the attribute *amount* is multiplied by 100.

The where Clause:

Consider the query "Find all loan numbers for loans made at the Perryridge branch with loan amounts greater that $1200." This query can be written in SQL as:

select *loan-number*
from *loan*
where *branch-name* = 'Perryridge' **and** *amount* > 1200

SQL uses the logical connectives **and**, **or**, and **not**—rather than the mathematical symbols ∧, ∨, and ¬—in the **where** clause. The operands of the logical connectives can be expressions involving the comparison operators <, <=, >, >=, =, and <>.

SQL includes a **between** comparison operator 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. If we wish to find the loan number of those loans with loan amounts between $90,000 and $100,000, we can use the **between** comparison to write

select *loan-number*
from *loan*
where *amount* **between** 90000 **and** 100000

Similarly, we can use the **not between** comparison operator.

The from Clause:

The **from** clause by itself defines a Cartesian product of the relations in the clause. Since the natural join is defined in terms of a Cartesian product, a selection, and a projection, it is a relatively simple matter to write an SQL expression for the natural join.
We write the relational-algebra expression

Π*customer-name, loan-number, amount* (*borrower* ⨝ *loan*)

for the query "For all customers who have a loan from the bank, find their names,loan numbers and loan amount." In SQL, this query can be written as

select *customer-name, borrower.loan-number, amount*
from *borrower, loan*
where *borrower.loan-number* = *loan.loan-number*

Notice that SQL uses the notation *relation-name.attribute-name*, as does the relational algebra, to avoid ambiguity in cases where an attribute appears in the schema of more than one relation.

Another e.g

"Find the customer names, loan numbers, and loan amounts for all loans at the Perryridge branch." To write this query, we need to state two constraints in the where clause, connected by the logical connective and:

select *customer-name, borrower.loan-number, amount*
from *borrower, loan*
where *borrower.loan-number* = *loan.loan-number* and
branch-name = 'Perryridge'

The Rename Operation:

SQL provides a mechanism for renaming both relations and attributes. It uses the **as** clause, taking the form:

old-name **as** *new-name*

The **as** clause can appear in both the **select** and **from** clauses.
Consider again the query that we used earlier:

select *customer-name, borrower.loan-number, amount*
from *borrower, loan*
where *borrower.loan-number* = *loan.loan-number*

The result of this query is a relation with the following attributes:

customer-name, loan-number, amount.

For example, if we want the attribute name *loan-number* to be replaced with the name *loan-id*,we can rewrite the preceding query as

select *customer-name, borrower.loan-number* **as** *loan-id, amount*
from *borrower, loan*
where *borrower.loan-number* = *loan.loan-number*

Tuple Variables:

The **as** clause is particularly useful in defining the notion of tuple variables, as is done in the tuple relational calculus. A tuple variable in SQL must be associated with a particular relation. Tuple variables are defined in the **from** clause by way of the **as** clause. To illustrate, we rewrite the query "For all customers who have a loan from the bank, find their names, loan numbers, and loan amount" as

select *customer-name, T.loan-number, S.amount*
from *borrower* **as** *T*, *loan* **as** *S*
where *T.loan-number = S.loan-number*

Note that we define a tuple variable in the **from** clause by placing it after the name of the relation with which it is associated, with the keyword **as** in between (the keyword **as** is optional).

Tuple variables are most useful for comparing two tuples in the same relation. Suppose that we want the query "Find the names of all branches that have assets greater than at least one branch located in Brooklyn."We can write the SQL expression

select distinct *T.branch-name*
from *branch* **as** *T*, *branch* **as** *S*
where *T.assets > S.assets* **and** *S.branch-city* = 'Brooklyn'

Observe that we could not use the notation *branch.asset*, since it would not be clear which reference to *branch* is intended.

String Operations:

SQL specifies strings by enclosing them in single quotes, for example, 'Perryridge'. The most commonly used operation on strings is pattern matching using the operator **like**. We describe patterns by using two special characters:

- Percent (%): The % character matches any substring.
- Underscore ( ): The character matches any character.

Patterns are case sensitive; that is, uppercase characters do not match lowercase characters, or vice versa. To illustrate pattern matching, we consider the following examples:

- 'Perry%' matches any string beginning with "Perry".
- '%idge%' matches any string containing "idge" as a substring, for example, 'Perryridge', 'Rock Ridge', 'Mianus Bridge', and 'Ridgeway'.
- '_ _ _' matches any string of exactly three characters.
- '_ _ _ %' matches any string of at least three characters.

SQL expresses patterns by using the **like** comparison operator. Consider the query "Find the names of all customers whose street address includes the substring 'Main'." This query can be written as

select *customer-name*
from *customer*
where *customer-street* **like** '%Main%'

Set Operations:

The SQL operations union, intersect, and except operate on relations and correspond to the relational-algebra operations $\cup$, $\cap$, and $-$.

The Union Operation:

To find all customers having a loan, an account, or both at the bank, we write

(**select** *customer-name*
from *depositor*)
union
(**select** *customer-name*
from *borrower*)

The **union** operation automatically eliminates duplicates, unlike the **select** clause. If we want to retain all duplicates, we must write **union all** in place of **union**:

(**select** *customer-name*
from *depositor*)
union all
(**select** *customer-name*
from *borrower*)

The number of duplicate tuples in the result is equal to the total number of duplicates that appear in both *d* and *b*. Thus, if Jones has three accounts and two loans at the bank, then there will be five tuples with the name Jones in the result.

The Intersect Operation:

To find all customers who have both a loan and an account at the bank, we write

(**select distinct** *customer-name*
from *depositor*)
intersect
(**select distinct** *customer-name*
from *borrower*)

The **intersect** operation automatically eliminates duplicates. If we want to retain all duplicates, we must write **intersect all** in place of **intersect**:

(**select** *customer-name*
from *depositor*)
intersect all
(**select** *customer-name*
from *borrower*)

The number of duplicate tuples that appear in the result is equal to the minimum number of duplicates in both *d* and *b*. Thus, if Jones has three accounts and two loans at the bank, then there will be two tuples with the name Jones in the result.

The Except Operation:

To find all customers who have an account but no loan at the bank, we write

(**select distinct** *customer-name*
from *depositor*)
except
(**select** *customer-name*
from *borrower*)

The **except** operation automatically eliminates duplicates. If we want to retain all duplicates, we must write **except all** in place of **except**:

(**select** *customer-name*
from *depositor*)
except all
(**select** *customer-name*
from *borrower*)

The number of duplicate copies of a tuple in the result is equal to the number of duplicate copies of the tuple in *d* minus the number of duplicate copies of the tuple in *b*, provided that the difference is positive. Thus, if Jones has three accounts and one loan at the bank, then there will be two tuples with the name Jones in the result. If, instead, this customer has two accounts and three loans at the bank, there will be no tuple with the name Jones in the result.

Aggregate Functions:

Aggregate functions are functions that take a collection (a set or multiset) of values as input and return a single value. SQL offers five built-in aggregate functions:

- Average: **avg**
- Minimum: **min**
- Maximum: **max**
- Total: **sum**
- Count: **count**

The input to **sum** and **avg** must be a collection of numbers, but the other operators can operate on collections of nonnumeric data types, such as strings, as well. As an illustration, consider the query "Find the average account balance at the Perryridge branch."We write this query as follows:

select avg (*balance*)
from *account*
where *branch-name* = 'Perryridge'

There are circumstances where we would like to apply the aggregate function not only to a single set of tuples, but also to a group of sets of tuples; we specify this wish in SQL using the **group by** clause. The attribute or attributes given in the **group by** clause are used to form groups. Tuples with the same value on all attributes in the **group by** clause are placed in one group.

As an illustration, consider the query "Find the average account balance at each branch."We write this query as follows:

select *branch-name*, **avg** (*balance*)
from *account*
group by *branch-name*

Retaining duplicates is important in computing an average. There are cases where we must eliminate duplicates before computing an aggregate function. If we do want to eliminate duplicates, we use the keyword **distinct** in the aggregate expression. An example arises in the query "Find the number of depositors for each branch." In this case, a depositor counts only once, regardless of the number of accounts that depositor may have .We write this query as follows:

select *branch-name*, **count** (**distinct** *customer-name*)
from *depositor, account*
where *depositor.account-number* = *account.account-number*
group by *branch-name*

At times, it is useful to state a condition that applies to groups rather than to tuples. For example, we might be interested in only those branches where the average account balance is more than $1200. This condition does not apply to a single tuple; rather, it applies to each group constructed by the **group by** clause. To express such a query, we use the **having** clause of SQL.

select *branch-name*, **avg** (*balance*)
from *account*
group by *branch-name*
having avg (*balance*) > 1200

If a **where** clause and a **having** clause appear in the same query, SQL applies the predicate in the **where** clause first. Tuples satisfying the **where** predicate are then placed into groups by the **group by** clause. SQL then applies the **having** clause, if it is present, to each group; it removes the groups that do not satisfy the **having** clause predicate. The **select** clause uses the remaining groups to generate tuples of the result of the query.

To illustrate the use of both a **having** clause and a **where** clause in the same query, we consider the query "Find the average balance for each customer who lives in Harrison and has at least three accounts."

select *depositor.customer-name*, **avg** (*balance*)
from *depositor, account, customer*
where *depositor.account-number* = *account.account-number* **and**
depositor.customer-name = *customer.customer-name* **and**
customer-city = 'Harrison'
group by *depositor.customer-name*
having count (**distinct** *depositor.account-number*) >= 3

Nested Subqueries:

SQL provides a mechanism for nesting 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, make set comparisons, and determine set cardinality.

Set Membership: SQL draws on the relational calculus for operations that allow testing tuples for membership in a relation. The **in** connective tests for set membership, where the set is a collection of values produced by a select clause. The **not in** connective tests for the absence of set membership.
As an illustration, reconsider the query "Find all customers who have both a loan and an account at the bank." We begin by finding all account holders, and we write the subquery

(**select** *customer-name*
from *depositor*)

We then need to find those customers who are borrowers from the bank and who appear in the list of account holders obtained in the subquery. We do so by nesting the subquery in an outer **select**. The resulting query is now

select distinct *customer-name*
from *borrower*
where *customer-name* **in** (**select** *customer-name*
from *depositor*)

In the preceding example, we tested membership in a one-attribute relation. It is also possible to test for membership in an arbitrary relation in SQL. We can thus write the query "Find all customers who have both an account and a loan at the Perryridge branch" in yet another way:

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*)

We use the **not in** construct in a similar way. For example, to find all customers who do have a loan at the bank, but do not have an account at the bank, we can write

select distinct *customer-name*
from *borrower*
where *customer-name* **not in** (**select** *customer-name*
from *depositor*)

Set Comparison: As an example of the ability of a nested subquery to compare sets, consider the query "Find the names of all branches that have assets greater than those of at least one branch located in Brooklyn." Previously(**in tuple variables example**) we wrote this query as follows:

select distinct *T.branch-name*
from *branch* as *T*, *branch* **as** *S*
where *T.assets* > *S.assets* **and** *S.branch-city* = 'Brooklyn'

SQL does, however, offer an alternative style for writing the preceding query. The phrase "greater than at least one" is represented in SQL by > **some**. This construct allows us to rewrite the query in a form that resembles closely our formulation of the query in English.

select *branch-name*
from *branch*
where *assets* > **some** (**select** *assets* **from** *branch* **where** *branch-city* = 'Brooklyn')

SQL also allows < **some**, <= **some**, >= **some**, = **some**, and<> **some** comparisons.
Now we modify our query slightly. Let us find the names of all branches that have an asset value greater than that of each branch in Brooklyn. The construct > **all** corresponds to the phrase "greater than all." Using this construct, we write the query as follows:

```
select branch-name
from branch
where assets > all (select assets
                    from branch
                    where branch-city = 'Brooklyn')
```

As it does for **some**, SQL also allows < **all**, <= **all**, >= **all**, = **all**, and <> **all** comparisons

Test for Empty Relations:

SQL includes a feature for testing whether a subquery has any tuples in its result. The **exists** construct returns the value **true** if the argument subquery is nonempty. Using the **exists** construct, we can write the query "Find all customers who have both an account and a loan at the bank" in still another way:

```
select customer-name
from borrower
where exists (select * from depositor
              where depositor.customer-name = borrower.customer-name)
```

We can test for the nonexistence of tuples in a subquery by using the **not exists** construct. To illustrate the **not exists** operator, consider again the query "Find all customers who have an account at all the branches located in Brooklyn." For each customer, we need to see whether the set of all branches at which that customer has an account contains the set of all branches in Brooklyn. Using the **except** construct, we can write the query as follows:

```
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.account-number = R.account-number and S.customer-name =
                        T.customer- name))
```

Views:

We define a view in SQL by using the **create view** command. To define a view, we must give the view a name and must state the query that computes the view. The form of the **create view** command is

```
create view v as <query expression>
```

where <query expression> is any legal query expression. The view name is represented by *v.*
As an example, consider the view consisting of branch names and the names of customers who have either an account or a loan at that branch. Assume that we want this view to be called *all-customer*.We define this view as follows:

```
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*)

Using the view *all-customer*, we can find all customers of the Perryridge branch by writing

select *customer-name*
from *all-customer*
where *branch-name* = 'Perryridge'

Modification of the Database:

Now, we discuss how to add, remove, or change information with SQL.

Deletion

A delete request is expressed in much the same way as a query. We can delete only whole tuples; we cannot delete values on only particular attributes. SQL expresses a deletion by

delete from *r*
where *P*

where *P* represents a predicate and *r* represents a relation. A **delete** command operates on only one relation.

Here are examples of SQL delete requests:

- Delete all account tuples in the Perryridge branch.

  delete from *account*
  where *branch-name* = 'Perryridge'

- Delete all loans with loan amounts between $1300 and $1500.

  delete from *loan*
  where *amount* **between** 1300 **and** 1500

- Delete all account tuples at every branch located in Needham.

  delete from *account*
  where *branch-name* **in** (**select** *branch-name*
  from *branch*
  where *branch-city* = 'Needham')

This **delete** request first finds all branches in Needham, and then deletes all *account* tuples pertaining to those branches.

Note that, although we may delete tuples from only one relation at a time, we may reference any number of relations in a **select-from-where** nested in the **where** clause of a **delete**. The **delete** request can contain a nested **select** that references the relation from which tuples are to be deleted. For example, suppose that we want to delete the records of all accounts with balances below the average at the bank.We could write

delete from *account*
where *balance* < (**select avg** (*balance*)
from *account*)

Insertion

To insert data into a relation, we either specify a tuple to be inserted or write a query whose result is a set of tuples to be inserted. The attribute values for inserted tuples must be members of the attribute's domain.

The simplest **insert** statement is a request to insert one tuple. Suppose that we wish to insert the fact that there is an account A-9732 at the Perryridge branch and that is has a balance of $1200.We write

insert into *account*
values ('A-9732', 'Perryridge', 1200)

or we can also write as

insert into *account* (*account-number, branch-name, balance*)

values ('A-9732', 'Perryridge', 1200) **OR**
insert into *account* (*branch-name, account-number, balance*)
values ('Perryridge', 'A-9732', 1200)

Updates

In certain situations, we may wish to change a value in a tuple without changing *all* values in the tuple. For this purpose, the **update** statement can be used.
Suppose that annual interest payments are being made, and all balances are to be increased by 5 percent.We write

update *account*
set *balance* = *balance* * 1.05

If interest is to be paid only to accounts with a balance of $1000 or more, we can write

update *account*
set *balance* = *balance* * 1.05
where *balance* >= 1000

Another example: "Pay 5 percent interest on accounts whose balance is greater than average"

update *account*
set *balance* = *balance* * 1.05
where *balance* > **select avg** (*balance*)
from *account*

Joined Relations:

SQL provides not only the basic Cartesian-product mechanism for joining tuples of relations , but, SQL also provides various other mechanisms for joining relations, including condition joins and natural joins, as well as various forms of outer joins. These additional operations are typically used as subquery expressions in the **from** clause.

loan

loan-number	*branch-name*	*amount*
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

borrower

customer-name	*loan-number*
Jones	L-170
Smith	L-230
Hayes	L-155

Figure 4.1 The *loan* and *borrower* relations.

Examples:

We illustrate the various join operations by using the relations *loan* and *borrower* in Figure 4.1. We start with a simple example of inner joins. Figure 4.2 shows the result of the expression

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

The expression computes the theta join of the *loan* and the *borrower* relations, with the join condition being *loan.loan-number* = *borrower.loan-number*. The attributes of the result consist of the attributes of the left-hand-side relation followed by the attributes of the right-hand-side relation. Note that the attribute *loan-number* appears twice in the figure—the first occurrence is from *loan*, and the second is from *borrower*. The SQL standard does not require attribute names in such results to be unique. An **as** clause should be used to assign unique names to attributes in query and subquery results. We rename the result relation of a join and the attributes of the result relation by using an **as** clause, as illustrated here:

loan **inner join** *borrower* **on** *loan.loan-number* = *borrower.loan-number*
as *lb(loan-number, branch, amount, cust, cust-loan-num)*

We rename the second occurrence of *loan-number* to *cust-loan-num*. The ordering of the attributes in the result of the join is important for the renaming.

loan-number	*branch-name*	*amount*	*customer-name*	*loan-number*
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230

Figure 4.2 The result of *loan* **inner join** *borrower* **on** *loan.loan-number = borrower.loan-number*.

Next, we consider an example of the **left outer join** operation:

loan **left outer join** *borrower* **on** *loan.loan-number* = *borrower.loan-number*

loan-number	*branch-name*	*amount*	*customer-name*	*loan-number*
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230
L-260	Perryridge	1700	*null*	*null*

Figure 4.3 The result of *loan* **left outer join** *borrower* **on** *loan.loan-number = borrower.loan-number*.

Now, we consider an example of the **natural join** operation:

loan **natural inner join** *borrower*

This expression computes the natural join of the two relations. The only attribute name common to *loan* and *borrower* is *loan-number*. Figure 4.4 shows the result of the expression.

loan-number	*branch-name*	*amount*	*customer-name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith

Figure 4.4 The result of *loan* **natural inner join** *borrower*.

Each of the variants of the join operations in SQL consists of a *join type* and a *join condition*. The join condition defines which tuples in the two relations match and what attributes are present in the result of the join. The 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. Figure 4.5 shows some of the allowed join types and join conditions.

Join types
inner join
left outer join
right outer join
full outer join

Join Conditions
natural
on <predicate>
using $(A_1, A_1, \ldots, A_n)$

Figure 4.5 Join types and join conditions.

The meaning of the join condition **natural**, in terms of which tuples from the two relations match, is straightforward. The ordering of the attributes in the result of a natural join is as follows. The join attributes (that is, the attributes common to both relations) appear first, in the order in which they appear in the left-hand-side relation. Next come all nonjoin attributes of the left-hand-side relation, and finally all nonjoin attributes of the right-hand-side relation.

The **right outer join** is symmetric to the **left outer join**. Tuples from the right-hand-side relation that do not match any tuple in the left-hand-side relation are padded with nulls and are added to the result of the right outer join.

Here is an example of combining the natural join condition with the right outer join type:

loan **natural right outer join** *borrower*

Figure 4.6 shows the result of this expression.

The **full outer join** is a combination of the left and right outer-join types. After the operation computes the result of the inner join, it extends with nulls tuples from the left-hand-side relation that did not match with any from the right-hand-side, and adds them to the result. Similarly, it extends with nulls tuples from the right-handside relation that did not match with any tuples from the left-hand-side relation and adds them to the result.

For example, Figure 4.7 shows the result of the expression

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

loan-number	*branch-name*	*amount*	*customer-name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-155	*null*	*null*	Hayes

Figure 4.6 The result of *loan* **natural right outer join** *borrower*.

loan-number	*branch-name*	*amount*	*customer-name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	*null*
L-155	*null*	*null*	Hayes

Figure 4.7 The result of *loan* **full outer join** *borrower* **using**(*loan-number*).

Stored Procedure:

A stored procedure in SQL is a type of pre-written code that can be stored for later execution and then used many times hence, saving time. It is a group of SQL statements that performs the task. The stored procedure can be invoked explicitly whenever required. It may accept some inputs in the form of parameters, these may be one parameter or multiple parameters. A procedure doesn't return values. To create a procedure, we use the CREATE PROCEDURE command.

To demonstrate how to create and call a procedure, we will create a procedure named myProcedure() that helps us select the name column from the book table. Here is the procedure:

Eg. Suppose we have following table: book(id, name)

```
DELIMITER $$
    CREATE PROCEDURE myProcedure()
        BEGIN
            SELECT name FROM book;
        END;
;
$$
```

```
MariaDB [Demo]>     CREATE PROCEDURE myProcedure()
    ->          BEGIN
    ->              SELECT name FROM book;
    ->          END;
    -> ;
    -> $$
Query OK, 0 rows affected (0.125 sec)

MariaDB [Demo]>
```

The procedure has been created. We have simply enclosed the SELECT statement within the BEGIN and END clauses of the procedure.
Now, we can call the procedure by its name as shown below:

```
CALL myProcedure();
```

```
MariaDB [Demo]> CALL myProcedure();
    -> $$
+---------------+
| name          |
+---------------+
| MariaDB Book1 |
| MariaDB Book2 |
| MariaDB Book3 |
| MariaDB Book4 |
| MariaDB Book5 |
+---------------+
5 rows in set (0.065 sec)

Query OK, 0 rows affected (0.077 sec)

MariaDB [Demo]>
```

The procedure returns the name column of the book table when called.
We can create a procedure that takes in a parameter. For example, we need to select the name of the book and filter using the book id. We can create the following procedure for this:

```
DELIMITER $$
  CREATE PROCEDURE myProcedure2(book_id int)
     BEGIN
        SELECT name FROM book WHERE id=book_id;
     END;
  ;
  $$
```

```
MariaDB [Demo]>     CREATE PROCEDURE myProcedure2(book_id int)
    ->          BEGIN
    ->              SELECT name FROM book WHERE id = book_id;
    ->          END;
    -> ;
    -> $$
Query OK, 0 rows affected (0.101 sec)

MariaDB [Demo]>
```

Above, we have created a procedure named myProcedure2(). This procedure takes one integer parameter named book_id which is the id of the book whose name we need to see. To see the name of the book with an id of 3, we can call the procedure as follows:

```
CALL myProcedure2(3);
```

```
MariaDB [Demo]> CALL myProcedure2(3);
    -> $$
+----------------+
| name           |
+----------------+
| MariaDB Book3  |
+----------------+
1 row in set (0.001 sec)

Query OK, 0 rows affected (0.005 sec)

MariaDB [Demo]>
```