Chapter 6

Integrity Constraints

1. Integrity constraints provide a way of ensuring that changes made to the database by authorized users do not result in a loss of data consistency.
2. We saw a form of integrity constraint with E-R models:
   - **key declarations:** stipulation that certain attributes form a candidate key for the entity set.
   - **form of a relationship:** mapping cardinalities 1-1, 1-many and many-many.
3. An integrity constraint can be any arbitrary predicate applied to the database.
4. They may be costly to evaluate, so we will only consider integrity constraints that can be tested with minimal overhead.

6.1 Domain Constraints

1. A domain of possible values should be associated with every attribute. These domain constraints are the most basic form of integrity constraint.

   They are easy to test for when data is entered.
2. Domain types
   (a) Attributes may have the same domain, e.g. *cname* and *employee-name*.
   (b) It is not as clear whether *bname* and *cname* domains ought to be distinct.
   (c) At the implementation level, they are both character strings.
   (d) At the conceptual level, we do not expect customers to have the same names as branches, in general.
   (e) Strong typing of domains allows us to test for values inserted, and whether queries make sense. Newer systems, particularly object-oriented database systems, offer a rich set of domain types that can be extended easily.
3. The **check** clause in SQL-92 permits domains to be restricted in powerful ways that most programming language type systems do not permit.
   (a) The **check** clause permits schema designer to specify a predicate that must be satisfied by any value assigned to a variable whose type is the domain.

(b) Examples:

create domain *hourly-wage* **numeric**(5,2)
  constraint *wage-value-test* **check**(**value** >= 4.00)

Note that "**constraint** *wage-value-test*" is optional (to give a name to the test to signal which constraint is violated).

create domain *account-number* **char**(10)
  constraint *account-number-null-test* **check**(**value not null**)

create domain *account-type* **char**(10)
  constraint *account-type-test* **check**(**value in** ("Checking", "Saving"))

6.2 Referential Integrity

Often we wish to ensure that a value appearing in a relation for a given set of attributes also appears for another set of attributes in another relation. This is called **referential integrity**.

6.2.1 Basic Concepts

1. Dangling tuples.
   - Consider a pair of relations $r(R)$ and $s(S)$, and the natural join $r \bowtie s$.
   - There may be a tuple t_r in r that does not join with any tuple in s.
   - That is, there is no tuple t_s in s such that $t_r[R \cap S] = t_s[R \cap S]$.
   - We call this a **dangling tuple**.
   - Dangling tuples may or may not be acceptable.
2. Suppose there is a tuple t_1 in the *account* relation with the value $t_1[bname] =$"Lunartown", but no matching tuple in the *branch* relation for the Lunartown branch.

   This is undesirable, as t_1 should refer to a branch that exists.

   Now suppose there is a tuple t_2 in the *branch* relation with $t_2[bname] =$"Mokan", but no matching tuple in the *account* relation for the Mokan branch.

   This means that a branch exists for which no accounts exist. This is possible, for example, when a branch is being opened. We want to allow this situation.
3. Note the distinction between these two situations: *bname* is the primary key of *branch*, while it is not for *account*.

   In account, *bname* is a **foreign key**, being the primary key of another relation.
   - Let $r_1(R_1)$ and $r_2(R_2)$ be two relations with primary keys K_1 and K_2 respectively.
   - We say that a subset α of R_2 is a *foreign key* referencing K_1 in relation r_1 if it is required that for every tuple t_2 in r_2 there must be a tuple t_1 in r_1 such that $t_1[K_1] = t_2[\alpha]$
   - We call these requirements **referential integrity constraints**.
   - Also known as **subset dependencies**, as we require

$$\Pi_\alpha(r_2) \subseteq \Pi_{K_1}(r_1)$$

6.2.2 Referential Integrity in the E-R Model

1. These constraints arise frequently. Every relation arising from a relationship set has referential integrity constraints.
   - Figure 6.1 shows an n-ary relationship set R relating entity sets $E_1, E_2, \ldots E_n$.

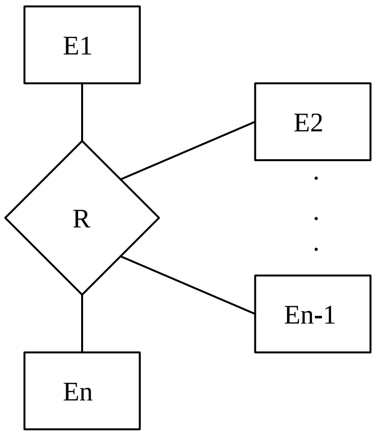

Figure 6.1: An n-ary relationship set

- Let K_i denote the primary key of E_i.
- The attributes of the relation scheme for relationship set R include $K_1 \cup K_2 \cup \ldots \cup K_n$.
- Each K_i in the scheme for R is a foreign key that leads to a referential integrity constraint.

2. Relation schemes for weak entity sets must include the primary key of the strong entity set on which they are existence dependent. This is a foreign key, which leads to another referential integrity constraint.

6.2.3 Database Modification

1. Database modifications can cause violations of referential integrity. To preserve the referential integrity constraint

$$\Pi_\alpha(r_2) \subseteq \Pi_{K_1}(r_1)$$

in the following operations:

- **Insert:** if a tuple t_2 is inserted into r_2, the system must ensure that there is a tuple t_1 in r_1 such that $t_1[K] = t_2[\alpha]$, i.e.

$$t_2[\alpha] \in \Pi_K(r_1)$$

- **Delete:** if a tuple t_1 is deleted from r_1, the system must compute the set of tuples in r_2 that reference t_1:

$$\sigma_{\alpha = t_1[K]}(r_2)$$

  If this set is not empty, either reject delete command, or delete also the tuples that reference t_1.

- **Update:** two cases
  - updates to referencing relation: test similar to insert case must be made, ensuring if t_2' is new value of tuple,

$$t_2'[\alpha] \in \Pi_K(r_1)$$

  - updates to referenced relation: test similar to delete if update modifies values for primary key, must compute

$$\sigma_{\alpha = t_1[K]}(r_2)$$

    to ensure that we are not removing a value referenced by tuples in r_2.

6.2.4 Referential Integrity in SQL

1. An addition to the original standard allows specification of primary and candidate keys and foreign keys as part of the **create table** command:
   - **primary key** clause includes a list of attributes forming the primary key.
   - **unique key** clause includes a list of attributes forming a candidate key.
   - **foreign key** clause includes a list of attributes forming the foreign key, and the name of the relation referenced by the foreign key.
2. An example illustrates several features mentioned so far:

```
create table customer
  (cname char(20) not null,
  street char(30),
  city char(30),
  primary key (cname))

create table branch
  (bname char(15) not null,
  bcity char(30),
  assets integer,
  primary key (bname)
  check (assets >= 0))

create table account
  (account# char(10) not null,
  (bname char(15),
  balance integer,
  primary key (account#)
  foreign key (bname) references branch,
  check (balance >= 0))

create table depositor
  (cname char(20) not null,
  account# char(10) not null,
  primary key (cname, account#)
  foreign key (cname) references customer,
  foreign key (account#) references account)
```

3. Notes on foreign keys:
   - A short form to declare a single column is a foreign key.

     ```
     bname char(15) references branch
     ```
   - When a referential integrity constraint is violated, the normal procedure is to reject the action. But a foreign key clause in SQL-92 can specify steps to be taken to change the tuples in the referenced relation to restore the constraint.
   - Example.

     ```
     create table account
       ...
       foreign key (bname) references branch
             on delete cascade
             on insert cascade,
       ...
     ```

     If a delete of a tuple in *branch* results in the preceding referential integrity constraints being violated, the delete is not rejected, but the delete "cascade" to the *account* relation, deleting the tuple that refers to the branch that was deleted. Update will be cascaded to the new value of the branch!

- SQL-92 also allows the **foreign key** clause to specify actions other than cascade, such as setting the refencing field to null, or to a default value, if the constraint is violated.
- If there is a chain of foreign key dependencies across multiple relations, a deletion or update at one end of the chain can propagate across the entire chain.

  If a cascading update or delete causes a constraint violation that cannot be handled by a further cascading operation, the system aborts the transaction and all the changes caused by the transaction and its cascading actions are undone.

4. Given and complexity and arbitrary nature of the way constraints in SQL behave with null values, it is the best to ensure that all columns of **unique** and **foreign** key specifications are declared to be nonnull.

6.3 Assertions

1. An **assertion** is a predicate expressing a condition we wish the database to always satisfy.
2. Domain constraints, functional dependency and referential integrity are special forms of assertion.
3. Where a constraint cannot be expressed in these forms, we use an assertion, e.g.
   - Ensuring the sum of loan amounts for each branch is less than the sum of all account balances at the branch.
   - Ensuring every loan customer keeps a minimum of $1000 in an account.
4. An assertion in DQL-92 takes the form,

   create assertion ⟨assertion-name⟩ **check** ⟨predicate⟩

5. Two assertions mentioned above can be written as follows.

   Ensuring the sum of loan amounts for each branch is less than the sum of all account balances at the branch.

```
create assertion sum-constraint check
(not exists (select * from branch
  where (select sum)amount) from loan
    where (loan.bname = branch.bname >=
      (select sum)amount) from account
      where (account.bname = branch.bname)))
```

6. Ensuring every loan customer keeps a minimum of $1000 in an account.

```
create assertion balance-constraint check
(not exists (select * from loan L
  (where not exists (select *
  from borrower B, depositor D, account A
  where L.loan# = B.loan# and B.cname = D.cname
      and D.account# = A.account# and A.balance >= 1000 )))
```

7. When an assertion is created, the system tests it for validity.

   If the assertion is valid, any further modification to the database is allowed only if it does not cause that assertion to be violated.

   This testing may result in significant overhead if the assertions are complex. Because of this, the **assert** should be used with great care.

8. Some system developer omits support for general assertions or provides specialized form of assertions that are easier to test.

6.4 Triggers

1. Another feature not present in the SQL standard is the **trigger**.

   Several existing systems have their own non-standard trigger features.

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

3. We need to

   - Specify the conditions under which the trigger is executed.
   - Specify the actions to be taken by the trigger.

4. For example, suppose that an overdraft is intended to result in the account balance being set to zero, and a loan being created for the overdraft amount.

   The trigger actions for tuple t with a negative balance are then

   - Insert a new tuple s in the *borrow* relation with

     $s[bname] = t[bname]$
     $s[loan\#] = t[account\#]$
     $s[amount] = -t[balance]$
     $s[cname] = t[cname]$
   - We need to negate balance to get amount, as balance is negative.
   - Set $t[balance]$ to 0.

   Note that this is not a good example. What would happen if the customer already had a loan?

5. SQL-92 does not include triggers. To write this trigger in terms of the original System R trigger:

```
define trigger overdraft
on update of account T
  (if new T.balance < 0
  then (insert into loan values
          (T.bname, T.account#, − new T.balance)
      insert into borrower
          (select cname, account#
          from depositor
          where T.coount# = depositor.account#)
    update account S
    set S.balance = 0
    where S.account# = T.account# ))
```

6.5 Functional Dependencies

6.5.1 Basic Concepts

1. Functional dependencies.

   - Functional dependencies are a constraint on the set of legal relations in a database.
   - They allow us to express facts about the real world we are modeling.
   - The notion generalizes the idea of a superkey.
   - Let $\alpha \subseteq R$ and $\beta \subseteq R$.
   - Then the functional dependency $\alpha \rightarrow \beta$ holds on R if in any legal relation $r(R)$, for all pairs of tuples t_1 and t_2 in r such that $t_1[\alpha] = t_2[\alpha]$, it is also the case that $t_1[\beta] = t_2[\beta]$.

A	B	C	D
a_1	b_1	c_1	d_1
a_1	b_2	c_1	d_2
a_2	b_2	c_2	d_2
a_2	b_3	c_2	d_3
a_3	b_3	c_2	d_4

Figure 6.2: Sample relation r.

- Using this notation, we say K is a superkey of R if $K \rightarrow R$.
- In other words, K is a superkey of R if, whenever $t_1[K] = t_2[K]$, then $t_1[R] = t_2[R]$ (and thus $t_1 = t_2$).

2. Functional dependencies allow us to express constraints that cannot be expressed with superkeys.

3. Consider the scheme

   Loan-info-schema = (bname, loan#, cname, amount)

   if a loan may be made jointly to several people (e.g. husband and wife) then we would not expect *loan#* to be a superkey. That is, there is no such dependency

   loan# $\rightarrow$ *cname*

   We do however expect the functional dependency

   loan# $\rightarrow$ *amount*
   loan# $\rightarrow$ *bname*

   to hold, as a loan number can only be associated with one amount and one branch.

4. A set F of functional dependencies can be used in two ways:

   - To specify constraints on the set of legal relations. (Does F hold on R?)
   - To test relations to see if they are legal under a given set of functional dependencies. (Does r satisfy F?)

5. Figure 6.2 shows a relation r that we can examine.

6. We can see that $A \rightarrow C$ is satisfied (in this particular relation), but $C \rightarrow A$ is not. $AB \rightarrow C$ is also satisfied.

7. Functional dependencies are called **trivial** if they are satisfied by all relations.

8. In general, a functional dependency $\alpha \rightarrow \beta$ is trivial if $\beta \subseteq \alpha$.

9. In the *customer* relation of figure 5.4, we see that *street* $\rightarrow$ *ccity* is satisfied by this relation. However, as in the real world two cities can have streets with the same names (e.g. Main, Broadway, etc.), we would not include this functional dependency in our list meant to hold on *Customer-scheme.*

10. The list of functional dependencies for the example database is:

    - On *Branch-scheme:*
      bname $\rightarrow$ *bcity*
      bname $\rightarrow$ *assets*
    - On *Customer-scheme:*
      cname $\rightarrow$ *ccity*
      cname $\rightarrow$ *street*
    - On *Loan-scheme:*
      loan# $\rightarrow$ *amount*
      loan# $\rightarrow$ *bname*

- On *Account-scheme:*
  $account\# \rightarrow balance$
  $account\# \rightarrow bname$

  There are no functional dependencies for *Borrower-schema*, nor for *Depositor-schema*.

6.5.2 Closure of a Set of Functional Dependencies

1. We need to consider *all* functional dependencies that hold. Given a set F of functional dependencies, we can prove that certain other ones also hold. We say these ones are **logically implied** by F.
2. Suppose we are given a relation scheme $R = (A, B, C, G, H, I)$, and the set of functional dependencies:
   $A \rightarrow B$
   $A \rightarrow C$
   $CG \rightarrow H$
   $CG \rightarrow I$
   $B \rightarrow H$

   Then the functional dependency $A \rightarrow H$ is logically implied.
3. To see why, let t_1 and t_2 be tuples such that
   $t_1[A] = t_2[A]$
   As we are given $A \rightarrow B$, it follows that we must also have
   $t_1[B] = t_2[B]$
   Further, since we also have $B \rightarrow H$, we must also have
   $t_1[H] = t_2[H]$
   Thus, whenever two tuples have the same value on A, they must also have the same value on H, and we can say that $A \rightarrow H$.
4. The **closure** of a set F of functional dependencies is the set of all functional dependencies logically implied by F.
5. We denote the closure of F by F^+.
6. To compute F^+, we can use some rules of inference called **Armstrong's Axioms**:
   - **Reflexivity rule:** if α is a set of attributes and $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ holds.
   - **Augmentation rule:** if $\alpha \rightarrow \beta$ holds, and γ is a set of attributes, then $\gamma\alpha \rightarrow \gamma\beta$ holds.
   - **Transitivity rule:** if $\alpha \rightarrow \beta$ holds, and $\beta \rightarrow \gamma$ holds, then $\alpha \rightarrow \gamma$ holds.
7. These rules are **sound** because they do not generate any incorrect functional dependencies. They are also **complete** as they generate all of F^+.
8. To make life easier we can use some additional rules, derivable from Armstrong's Axioms:
   - **Union rule:** if $\alpha \rightarrow \beta$ and $\alpha \rightarrow \gamma$, then $\alpha \rightarrow \beta\gamma$ holds.
   - **Decomposition rule:** if $\alpha \rightarrow \beta\gamma$ holds, then $\alpha \rightarrow \beta$ and $\alpha \rightarrow \gamma$ both hold.
   - **Pseudotransitivity rule:** if $\alpha \rightarrow \beta$ holds, and $\gamma\beta \rightarrow \delta$ holds, then $\alpha\gamma \rightarrow \delta$ holds.
9. Applying these rules to the scheme and set F mentioned above, we can derive the following:
   - $A \rightarrow H$, as we saw by the transitivity rule.
   - $CG \rightarrow HI$ by the union rule.
   - $AG \rightarrow I$ by several steps:
     - Note that $A \rightarrow C$ holds.
     - Then $AG \rightarrow CG$, by the augmentation rule.
     - Now by transitivity, $AG \rightarrow I$.

   (You might notice that this is actually pseudotransivity if done in one step.)

6.5.3 Closure of Attribute Sets

1. To test whether a set of attributes α is a superkey, we need to find the set of attributes functionally determined by α.

2. Let α be a set of attributes. We call the set of attributes determined by α under a set F of functional dependencies the **closure** of α under F, denoted α^+.

3. The following algorithm computes α^+:

```
result := α
while (changes to result) do
  for each functional dependency β → γ
        in F do
    begin
      if β ⊆ result
      then result := result ∪γ;
    end
```

4. If we use this algorithm on our example to calculate (AG^+) then we find:

   - We start with *result* = AG.
   - $A \rightarrow B$ causes us to include B in *result*.
   - $A \rightarrow C$ causes *result* to become ABCG.
   - $CG \rightarrow H$ causes *result* to become ABCGH.
   - $CG \rightarrow I$ causes *result* to become ABCGHI.
   - The next time we execute the while loop, no new attributes are added, and the algorithm terminates.

5. This algorithm has worst case behavior quadratic in the size of F. There is a linear algorithm that is more complicated.

6.5.4 Canonical Cover

1. To minimize the number of functional dependencies that need to be tested in case of an update we may restrict F to a **canonical cover** F_c.

2. A canonical cover for F is a set of dependencies such that F logically implies all dependencies in F_c, and vice versa.

3. F_c must also have the following properties:

   - Every functional dependency $\alpha \rightarrow \beta$ in F_c contains no **extraneous** attributes in α (ones that can be removed from α without changing F_c^+). So A is extraneous in α if $A \in \alpha$ and

     $$(F_c - \{\alpha \rightarrow \beta\}) \cup \{\alpha - A \rightarrow \beta\}$$

     logically implies F_c.

   - Every functional dependency $\alpha \rightarrow \beta$ in F_c contains no **extraneous** attributes in β (ones that can be removed from β without changing F_c^+). So A is extraneous in β if $A \in \beta$ and

     $$(F_c - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow \beta - A\}$$

     logically implies F_c.

- Each left side of a functional dependency in F_c is unique. That is there are no two dependencies $\alpha_1 \rightarrow \beta_1$ and $\alpha_2 \rightarrow \beta_2$ in F_c such that $\alpha_1 = \alpha_2$.

4. To compute a canonical cover F_c for F,

```
repeat
   Use the union rule to replace dependencies of the form
      $\alpha_1 \rightarrow \beta_1$ and $\alpha_1 \rightarrow \beta_2$ with $\alpha_1 \rightarrow \beta_1\beta_2$.
   Find a functional dependency $\alpha \rightarrow \beta$ with an
      extraneous attribute in $\alpha$ or in $\beta$.
   If an extraneous attribute is found, delete it from $\alpha \rightarrow \beta$
until $F$ does not change.
```

5. An example: for the relational scheme $R = (A, B, C)$, and the set F of functional dependencies

$$\begin{aligned} &A \rightarrow BC \\ &B \rightarrow C \\ &A \rightarrow B \\ &AB \rightarrow C \end{aligned}$$

we will compute F_c.

- We have two dependencies with the same left hand side:

$$\begin{aligned} &A \rightarrow BC \\ &A \rightarrow B \end{aligned}$$

  We can replace these two with just $A \rightarrow BC$.

- A is extraneous in $AB \rightarrow C$ because $B \rightarrow C$ logically implies $AB \rightarrow C$.

- Then our set is

$$\begin{aligned} &A \rightarrow BC \\ &B \rightarrow C \end{aligned}$$

- We still have an extraneous attribute on the right-hand side of the first dependency. C is extraneous in $A \rightarrow BC$ because $A \rightarrow B$ and $B \rightarrow C$ logically imply that $A \rightarrow BC$.

- So we end up with

$$\begin{aligned} &A \rightarrow B \\ &B \rightarrow C \end{aligned}$$