Chapter 6: Integrity Constraints

- Functional Dependencies
- Foreign Keys
- Assertions
- Referential Integrity
- Domain Constraints
ensure that the comparisons make sense.

They test values inserted in the database, and test queries to

constrain.

Domain constraints are the most elementary form of integrity.

Do not result in a loss of data consistency.

By ensuring that authorized changes to the database

Integrity constraints guard against accidental damage to the

Domain Constraints
indicate which constraint an update violated.

- The clause constraint value-test is optional; useful to
  hourly-wage is greater than 4.00.
- The domain has a constraint that ensures that the
  with 5 digits, 2 of which are after the decimal point
  - The domain hourly-wage is declared to be a decimal number
    constraint value-test check(value >= 4.00)
    create domain hourly-wage numeric(5,2)
    allows only values greater than a specified value.
- Use check clause to ensure that an hourly-wage domain
  - The check clause in SQL-92 permits domains to be restricted:
Referential Integrity Constraint:

\[ \Pi_{Y_1}^{(1)} \subseteq \Pi_{Y_2}^{(2)} \]

\( t \) such that \( t \in \Pi_{Y_1}^{(1)} \)

Relation \( R_1 \), if for every \( t \) in \( R_2 \) there must be a tuple \( t \) in relation \( R_1 \) if \( R_2 \) is a foreign key referencing \( Y_1 \) in \( R_1 \) and \( Y_2 \) respectively.

Let \( t \in \Pi_{Y_1}^{(1)} \) and \( t \in \Pi_{Y_1}^{(2)} \) be relations with primary keys \( Y_1 \) and \( Y_2 \) respectively.

**Formal Definition**

Tuple in the branch relation for branch "Periyride".

A tuple in the branch relation for branch "Periyride" is a branch name appearing in one another relation.

Example: if "Periyride" is a branch name appearing in one set of attributes also appears for a certain set of attributes in another set, that a value that appears in one relation for a given

Referential Integrity
include the primary key of the entity set on which it depends.

Weak entity sets are also a source of referential integrity.

For $E_1$ and $E_2$ respectively.

Then $K_1$ and $K_2$ form foreign keys on the relational schemas $E_1$ and $E_2$.

Consider relationship set $R$ between entity sets $E_1$ and $E_2$.
deleted (cascading deletions are possible).

If this set is not empty, either the delete command is reflected

\[(z, t) [Y] t = a_0\]

compute the set of tuples in \(t_2\) that reference \(t_1\):

- **Delete.** If a tuple \(t_1\) is deleted from \(t_1\), the system must

\[(t_1) \subseteq [a] t_2\]

That is that there is a tuple \(t_2\) in \(t_1\) such that \(t_1 = [Y] t_2 = [a] t_2\). That is

\[(t_1) [Y] \subseteq (z, t) a\]

Following referential integrity constraint:

- The following tests must be made in order to preserve the

DataBase Modification
tuples in the set, or the tuples in the set may be deleted.

repeated as an error, or the update may be cascaded to the
appropriate. If this set is not empty, the update may be

using the old value of \( t_1 \) (the value before the update is

\[
(t_2) \left[ Y \right]_{t_1} = 0
\]

delete case is made. The system must compute

values for the primary key \( Y \); then a test similar to the

values in relation \( t_1 \) and the update modifies

\[
(t_1) Y \quad \exists [a], t_2
\]

tuple \( t_2 \). The system must ensure that

the insert case is made. Let \( t_2 \)' denote the new value of

modifies values for the foreign key \( a \); then a test similar to

- If a tuple \( t_2 \) is updated in relation \( t_2 \) and the update

- Update. There are two cases:

**Database Modification (Cont.)**
Referential Integrity in SQL

- The **FOREIGN KEY** clause of the `CREATE TABLE` statement...

- The **UNIQUE KEY** clause of the `CREATE TABLE` statement...

- The **PRIMARY KEY** clause of the `CREATE TABLE` statement...

Foreign key...
(primary_key (branch-name))
  integer,
  branch-name (30),
  branch-name (15) not null,
  create table branch

(primary_key (customer-name))
  customer-name (30),
  customer-name (30),
  customer-name (20) not null,
  create table customer
create table deposit

<table>
<thead>
<tr>
<th>foreign key (customer-name, reference.branch)</th>
</tr>
</thead>
<tbody>
<tr>
<td>primary key (account-number)</td>
</tr>
<tr>
<td>balance</td>
</tr>
<tr>
<td>account-number</td>
</tr>
<tr>
<td>not null(10)</td>
</tr>
<tr>
<td>branch-name(15)</td>
</tr>
<tr>
<td>create table account</td>
</tr>
</tbody>
</table>

create table customer

<table>
<thead>
<tr>
<th>foreign key (customer-name, reference.branch)</th>
</tr>
</thead>
<tbody>
<tr>
<td>primary key (account-number)</td>
</tr>
<tr>
<td>account-name</td>
</tr>
<tr>
<td>not null(20)</td>
</tr>
<tr>
<td>not null(10)</td>
</tr>
<tr>
<td>char(15)</td>
</tr>
<tr>
<td>create table account</td>
</tr>
</tbody>
</table>
Assertion should be used with great care. Hence assertions may introduce a significant amount of overhead.

This testing may introduce a significant amount of overhead.

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

\texttt{create asserti}on \texttt{check name} \texttt{predicate} \texttt{asserti}on

An assertion in SQL-92 takes the form

\texttt{An asserti}on is a predicate expressing a condition that we wish

\begin{center}
\textbf{Asser}tions
\end{center}
where loan.branch-name = branch.branch-name

where (select sum(amount) from account
    where loan.branch-name = branch.branch-name)

where (select sum(amount) from loan
    where exists (select * from branch)
    create assertion sum-constaint check

the sum of all account balances at the branch.
The sum of all loan amounts for each branch must be less than •
(\(\text{account.balance} \leq 1000\))

and \(\text{depositor.account-number} = \text{account.account-number}\)

and \(\text{borrower.customer-name} = \text{depositor.customer-name}\)

where \(\text{loan.loan-number} = \text{borrower.loan-number}\)

from \(\text{borrower, depositor, account}\)

* where not exists ( select

* where not exists ( select * from loan

create assertion balance-constraint check

with a minimum balance of $1000.00.

Every loan has at least one borrower who maintains an account

assertion example
Implementations support triggers.
The SQL-92 standard does not include triggers, but many
– Specify the conditions under which the trigger is to be executed.
– To design a trigger mechanism, we must:
  • A trigger is a statement that is executed automatically by the database.
account relation that results in a negative balance value.

The condition for executing the trigger is an update to the

number of the overdraft account

- Giving this loan a loan number identical to the account

- Creating a loan in the amount of the overdraft

- Setting the account balance to zero

Suppose that instead of allowing negative account balances, the
The keyword `new` used before `T.balance` indicates that the value of `T.balance` after the update should be used, if it is omitted, the value before the update is used.

```sql
where J.account-number = J.account-number
set J.balance = 0
update account

where J.account-number = depositor.account-number
from depositor

select customer-name, account-number
from depositor

insert into borrower
("
branch-name", J.account-number, "new J.balance"

then (insert into loan values
0 > "new J.balance"

define trigger on update of account

TRIGGER Example (Cont.)
```
A functional dependency is a generalization of the notion of a key.

- Uniquely the value for another set of attributes.
- Require that the value for a certain set of attributes determines constraints on the set of legal relations.
\[ R \leftarrow \alpha \subseteq R, \quad \text{for no } \alpha \subseteq R, \quad \text{and}\]

\[ R \leftarrow \alpha \subseteq R, \quad \text{if and only if}\]

- \( \alpha \) is a candidate key for \( R \)
- \( \alpha \) is a superkey for relation schema \( R \)

\[ [\varnothing]_{\alpha}^{2} = [\varnothing]_{\alpha}^{1} \Leftrightarrow [\alpha]_{\alpha}^{2} = [\alpha]_{\alpha}^{1} \]

also agree on the attributes \( \varnothing \). That is,

any two tuples \( t_1 \) and \( t_2 \) of \( \varnothing \) agree on the attributes \( \alpha \), they hold on \( R \) if and only if for any legal relations \( t(R) \), whenever

\[ \varnothing \leftarrow \alpha \]

The functional dependency

\[ \alpha \subseteq R \]

Let \( R \) be a relation schema
loan-number → customer-name

but would not expect the following to hold:

loan-number → branch-name
loan-number → amount

We expect this set of functional dependencies to hold:

(customer-name, amount) → loan

loan-info-schema = (branch-name, loan-number, etc)

cannot be expressed using superkeys. Consider the schema:

Functional dependencies allow us to express constraints that

Functional Dependencies (cont.)
customer-name.

of loan-scheme may, by chance, satisfy loan-number —

not hold on all legal instances. For example, a specific instance
functional dependency even if the functional dependency does
a specific instance of a relation scheme may satisfy a

. Note: A specific instance of a relation scheme may satisfy a

functional dependence $P$.

$P$ holds on $R$ if all legal relations on $R$ satisfy the set of
specific constraints on the set of legal relations, we say that
— specifically constraints on the set of legal relations. If a relation $r$ satisfies $P$,
functional dependence $P$. If a relation $r$ is legal under a set
— test relations to see if they are legal under a given set of

$P$.

Use of Functional Dependencies
These rules are sound and complete:

(Transitivity) $\exists \iota \therefore \iota' \leftarrow \iota$ 

(Reflexivity) $\iota' \leftarrow \iota$, $\iota \leftarrow \iota'$

We can find all of $P^+$ by applying Armstrong's Axioms:

- We denote the closure of $P$ by $P^+$. 
- The closure of $P$ is the set of all functional dependencies logically implied by $P$.
- Given a set $\xi$, set of functional dependencies, there are certain

---

**Closure of a Set of Functional Dependencies**
The above rules can be inferred from Armstrong's axioms.

(\textit{pseudotransitivities})
\begin{align*}
\forall \varphi & \\iff \varphi \wedge \varphi' \iff \varphi' & \iff (\text{decomposition}) \\
\forall \varphi & \\iff \varphi' \iff \varphi & \iff (\text{union}) \\
\forall \varphi & \\iff \varphi' \iff \varphi' \wedge \varphi & \iff (\text{intersection}) \\
\end{align*}

Following additional rules.

We can further simplify computation of \( F^+ \) by using the closure (\textit{cont.})
\[
\begin{align*}
I H & \leftarrow \mathcal{O} - \\
I & \leftarrow \mathcal{A} - \\
H & \leftarrow \mathcal{A} - \\
+ & \text{ some members of } \mathcal{I} \\
\{ H \leftarrow B \\
I & \leftarrow \mathcal{O} \\
H & \leftarrow \mathcal{O} \\
\mathcal{O} & \leftarrow \mathcal{A} \\
B & \leftarrow \mathcal{A} \} = \mathcal{A} \\
\text{(I, B, C, H, I) = } \mathcal{R} \\
\end{align*}
\]
begin
for each $\gamma$ in $\mathcal{F}$ do
while (changes to result)
do
result := $\alpha$;
end
$\forall \gamma \in \mathcal{F}$ result $\subset \gamma$
if $\gamma \cap \text{result} : = \text{result}$

Algorithm to compute $\gamma^+$, the closure of $\gamma$ under $\mathcal{F}$

$\gamma^+ \supset \gamma \Leftrightarrow \gamma$ is in $\mathcal{F}$ if $\gamma \leftarrow \alpha$

Attributes that are functionally determined by $\alpha$ under $\mathcal{F}$:

Define the closure of $\alpha$ under $\mathcal{F}$ (denoted by $\alpha^+$) as the set of

Closure of Attribute Sets
3. does $C+$?
2. does $A+$?
1. $AC \leftarrow R$

Is $AC$ a candidate key?

$(cc \leftarrow I \text{ and } cc \subseteq AGBCH) \quad \text{result} = ABCGI$

$(cc \leftarrow H \text{ and } cc \subseteq AGBCH) \quad \text{result} = ABCGH$

$(\text{ and } cc \subseteq AGB) \quad \text{result} = ABC$

$(\text{ and } cc \subseteq AGB) \quad \text{result} = AC$

Example
Each left side of a functional dependency in $\mathcal{F}$ is unique.

- No functional dependency in $\mathcal{F}$ contains an extraneous attribute.
- All dependencies in $\mathcal{F}$, and further logically implies all dependencies in $\mathcal{F}$, and $\mathcal{F}$.

A canonical cover $\mathcal{F}$ for $\mathcal{F}$ is a set of dependencies such that

\begin{equation}
\mathcal{F} = \{ (\forall \: - \: g) \leftarrow \: a \} \\
\cap \: (\{g \leftarrow a\} \quad - \mathcal{F})
\end{equation}

functional dependencies.

- Attribute $\mathcal{A}$ is extraneous in $\mathcal{F}$ if $\mathcal{A} \in \mathcal{F}$ and the set of
  \begin{equation}
  \{g \leftarrow (\forall \: - a)\} \cap \: (\{g \leftarrow a\} \quad - \mathcal{F})
  \end{equation}
  implies $\mathcal{A}$ is extraneous in $\mathcal{F}$ and $\mathcal{A}$ logically implies $\mathcal{A}$.

- Consider a set $\mathcal{F}$ of functional dependencies and the functional dependencies $\mathcal{F}$.

\textbf{Canonical Cover}
until $F$ does not change

$\beta \leftarrow a$ if an extraneous attribute is found, delete it from $F$

$\beta \leftarrow a$ or $b$ if an extraneous attribute either in $a$ or in $b$

Find a functional dependency $a \rightarrow b$ with an $a_1$ and $a_2$ with $a_1 \rightarrow b_1$ and $a_2 \rightarrow b_2$.

Use the union rule to replace any dependences in $F$. Repeat

Compute a canonical cover for $F$.  }

Canonical Cover (cont.)
\( C \leftarrow B \)
\( B \leftarrow A \)

The canonical cover is:

- By \( A \leftarrow B \) and \( B \leftarrow C \).
- \( A \) is extraneous in \( A \) since \( A \) is logically implied.
- \( C \) is extraneous in \( A \) since \( A \) is logically implied.
- \( A \leftarrow B \) because \( C \leftarrow B \) implies \( A \leftarrow B \).
- Combine \( A \) and \( B \) into \( A \) and \( B \).

\[ \{A \leftarrow AB, B \leftarrow A, C \leftarrow B, BC \leftarrow A\} = \emptyset \]
\[ (A, B, C) = \emptyset \]

Example of computing a canonical cover.