Ad Hoc Constraints

Objectives of the Lecture:

- To consider Ad Hoc Constraints in principle;
- To consider Ad Hoc Constraints in SQL;
- To consider other aspects of integrity constraints in SQL.
**Why Ad Hoc Constraints?**

- These are other constraints on values held in a relation.
- They need to be applied to ensure the database holds accurate data, but don’t fit into any of the previous categories.
- They usually fall into two classes:
  1. *To ensure physical reality applies*.  
     *Example*: weights are always positive.
  2. *To ensure Business Rules are kept* - this reflects the way the organisation works.  
     *Example*: it is company policy to only buy cars with an engine capacity of less than 2 litres.

In practice, there are all sorts of additional constraints that it is useful to apply in order to minimise the possibility of erroneous data getting into the DB. Thus while they do not have such a profound theoretical basis as candidate and foreign keys, and vary very widely in their nature, they are nevertheless very important in practice.

In fact it is being suggested that “business rules” might form a better basis for DB design than E-R modelling, OO modelling, and other currently established DB design methods.
In general, an integrity constraint is a limitation on the permissible values that a DB relation can hold.

It could be expressed as:

\[
\text{IF a tuple appears in a DB relation} \quad \text{THEN that tuple satisfies a certain condition.}
\]

The condition must always be a binary condition, i.e. one that evaluates to \textit{true} or \textit{false}.

Attribute type, candidate key, and referential integrity constraints are special cases of integrity constraints.

An \textit{ad hoc} constraint is one where the binary condition is directly provided. A tuple can be accepted in a relation only if the binary condition is applied to the tuple and the result evaluates to \textit{true}.

In fact, all integrity constraints must be met before a tuple is acceptable in a relation.

Note that in the \textit{implementation} of the relational model, there are normally more efficient ways of checking that the integrity constraints are met than literally putting a tuple in a relation, checking the integrity constraints, and then removing the tuple if it does not meet all the constraints.

To meet all the integrity constraints means that a logical \textit{AND} of all of them must evaluate to \textit{true}.
Ad Hoc Constraints in SQL

- They are applied using the **Check** option.
- The format is **Check ( condition )**
- A whole variety of **conditions** can be inserted.
- Just as with candidate and foreign key constraints, *ad hoc* constraints (**Check** constraints) can be part of an attribute sub-statement
  - **OR**
  - in a separate sub-statement at the end of the **Create Table** statement.
- Likewise *ad hoc* constraints (**Check** constraints) have integrity constraint names, that are either supplied by the user with the prefix **Constraint**, or by the DBMS as a default.

In practice the SQL **Check** option can often be considered as “SQL speak” for an *ad hoc* integrity constraint option.

The condition used in the **Check** must be a binary condition.

The conditions are typically comparisons of attribute values. However there is no limit to what may be written in a condition. Conditions may include calculations. They may also consist of several sub-conditions linked together by logical **ANDs** and **ORs**; and logical **NOT** may also be used. Queries on the DB that evaluate to **true or false** may also be used as a (part of a) condition; however DB queries are not considered here.
### Example Check Conditions (1)

- Constraining the values in a relation about products.

**Create Table PRODUCT**

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prodno</td>
<td>Char(5)</td>
<td>Primary Key, Check (Weight &gt; 0), Check (Price &gt; 0), Check (Quantity * Price &lt; 1000000)</td>
</tr>
<tr>
<td>Weight</td>
<td>Number</td>
<td></td>
</tr>
<tr>
<td>Price</td>
<td>Number</td>
<td></td>
</tr>
<tr>
<td>Quantity</td>
<td>Number</td>
<td></td>
</tr>
</tbody>
</table>

**Ensuring Physical Reality.**

Applying a Business Rule! Applying a Business Rule about the maximum permissible stock.

In the above example, the check that a price is not zero or a negative may be considered common sense (or a check to avoid corruption?) rather than explicitly a business rule or physical reality check. Not every check required necessarily falls into one of these two classes. The classes are there for convenience and support, rather than forming an obligatory logical framework.

The constraint called *Value* is assumed to be a business policy which says that no more than £1,000,000 may be tied up in the stock of one particular kind of product.

Note that the *Value* constraint uses 2 attributes: therefore it cannot appear in the sub-statement of either attribute but must appear in its own sub-statement at the end.
In the revised example above, an upper limit has also been put on weights. Price is now checked to see if it falls into one of three price ranges. SQL has a variety of facilities that can be used to check the values occurring in attributes. They are normally used in queries and will be discussed there; however they can also be used in `Check` conditions.
**SQL Assertions**

- An SQL **Assertion** is essentially an SQL **Check** constraint that can be specified independently of any **Create Table** or **Alter Table** statement.

**Format:**

```
Create Assertion  ASSERTION_NAME  Check ( condition )
```

(There is also a corresponding **Drop Assertion** **ASSERTION_NAME** statement).

- Although an assertion can apply to one relation, its main purpose is to apply an integrity constraint that applies jointly to 2, 3 or more relations, a sort of “referential integrity ad hoc constraint”. It is good practice to limit its use to constraints applicable jointly to 2 or more relations.

- Typically the **condition** uses an SQL query (that must evaluate to **true** or **false**).

Assertions are most often used with **Check** conditions that are DB queries which join 2 (or more) relations into one relation and then check what is in the ‘joint’ table. This is very useful as a referential integrity equivalent of an **ad hoc** integrity constraint.

As queries have not yet been covered, examples of assertions are not given here.
Multiple Integrity Constraints

- Although not illustrated yet, it is possible and sometimes desirable in SQL to assign more than one integrity constraint in an attribute sub-statement.

**Example**:
Consider a different version of the `PRODUCT` relation.

```sql
Create Table PRODUCT (
    Prodno Char(5) Primary Key,
    Weight Number
    Constraint WT_NOT_NULL Not Null
    Constraint POS_WT Check (Weight > 0 )
    Constraint WT_REF References LIST_WEIGHTS (Wts),
    Price Number,
    Quantity Number,
);```

In the above example, the `Weight` attribute is assigned 4 integrity constraints in one sub-statement:
1. its data type,
2. that it cannot be null,
3. that it must have a positive weight,
4. and its value must occur in the `Wts` attribute of relation `LIST_WEIGHTS` (presumably in effect a list of permissible weights).

For completeness, all the constraints have been given names, apart from the data type constraint which cannot receive a name.

For simplicity, other possible integrity constraints have been ignored.
Limitations of Ad Hoc Constraints

- Suppose the relation EMPLOYEE were to contain an attribute that holds the ages of employees, and there are age limits on who can be employed (to meet legal requirements).
- To reflect this, the following ad hoc constraint could be added to the Age attribute:
  \[
  \text{Check}\ (\ Age >= 16 \ \text{And} \ Age <= 65 )
  \]
- However it may be legal and normal to employ school children for some part-time jobs, and retired people for some part-time consultancy that utilises their experience.
  
  Now both comparisons in the ad hoc constraint are inappropriate. What should replace them?
- The problem arises because the boundary conditions of legal age limits are fuzzy. There is no simple solution to this. Fuzzy limits can be difficult to handle and judgement must be used.

Limitations of Integrity Constraints

- Suppose the following tuple were to appear in the EMPLOYEE relation:
  \[
  (\text{‘E2’, ‘Fenwick’, ‘M’, 40000})
  \]
  Assume this tuple satisfies all the integrity constraints, which are properly specified.
- If the company has no employee called “Fenwick”, then the DB still has false data in it, despite all the integrity constraints.
- In this example, no integrity constraint can possibly check which of all the names that are permissible in the DB actually correspond to current employees (& hence are valid).

Conclusion: the integrity constraints form a limited approximation to the real world constraints that would ideally apply. Additional validation is often required, typically provided by human intervention or some specially designed input system.

Note: the Closed World Assumption applies to DBs. Any data in the DB must be true; any not in it must be false.
Ad Hoc Constraints & Specific Data Types

The following example was used earlier to illustrate the derivation of specific data types from underlying types:

```sql
Create Table EMPLOYEE (    ENo Char(2),    EName Varchar2(30),    M-S Char(1)    Check( M-S In (‘S’, ‘M’, ‘W’, ‘D’)),    Sal Number    Check( Sal >999 And Sal < 100000 )
);
```

Previously **CHECK** was used to derive specific data types. But these 2 **CHECK** conditions could equally well be regarded as applying **ad hoc** constraints.

SQL Check Constraints

- Conceptually, specific data types and **ad hoc** constraints are different.

- However the **CHECK** option is used for both in SQL.
  
  If the parameters to a data type - e.g. `CHAR(1)` - don’t narrow the underlying type down sufficiently to achieve the specific data type, then using **CHECK** may be appropriate.

- Some integrity constraints can validly be considered as either specific data types or **ad hoc** constraints.

  **Example** : ensuring “Weight” in the `PRODUCT` relation is a positive value can be considered from either viewpoint.
**SQL Default Values**

- It is possible to specify that an attribute has a *default value*. This means that:
  
  **IF** a tuple is to be put into a relation  
  **AND IF** no value is specified for that attribute in the tuple  
  **THEN** the attribute is given the *default value* in that tuple.

- Default values are *not* integrity constraints.  
  They are mentioned here because they are useful in helping to make sure that attribute values do satisfy the integrity constraints.

- SQL syntax for defaults is:  
  ```sql
  Default default_value
  ```

**Example:**

```sql
Create Table EMPLOYEE (  
  EName Varchar2(30),  
  M-S Char(1)  Check( M-S In (‘S’, ‘M’, ‘W’, ‘D’ ) )  
  Default ‘S’ ) ;
```

Self evidently, the default value must be chosen so that it does satisfy all the integrity constraints!

They are useful in the following circumstances:

- If an attribute frequently has a particular value, then the default can be used by the DBMS to enter that value to save the user from having to enter it. For example, if most employees are male, then the *Sex* attribute value may have a default of ‘male’ so that the value ‘female’ only has to be given in a small percentage of cases.

- If an attribute value is not known, and so no value is entered into the DB, the DBMS enters a default value which is chosen to be helpful or to be a ‘special value’ denoting the fact that there is no known value for the attribute.

In SQL, if no default value is provided for an attribute and the DBMS has to use a default value, then null is used.

In the above example, if the *M-S* (i.e. marital status) attribute is not given a value, then it defaults to ‘S’ (for ‘single’); this is presumably chosen to minimise problems with tax allowances, pension contributions, etc, until the person’s marital status can be established.
**SQL Alter Table**

- It can be useful, or even necessary, to alter relations some time after they have been created.

- SQL provides the **Alter Table** command for this.

- **Alter Table** provides the following ways of changing a relation:
  
  - **Add or drop an attribute**;
    
    an additional attribute may also have integrity constraints assigned to it, using the same syntax as for **Create Table**.
  
  - **Add or drop an existing integrity constraint**;
    
    again adding integrity constraints uses the same syntax as for **Create Table**; dropping an existing integrity constraint requires referring to it by name.
  
  - **Set or drop a default value** for an existing attribute.

In order to drop an attribute or an integrity constraint, it must be named in the statement. Therefore, particularly in the latter case, it may be necessary to first look up the names in the DB’s data dictionary or data catalogue so as to be able to enter the statements correctly. It is easier to get the right constraint name if the constraints have sensible user-given names rather than DBMS default names.

When dropping an attribute or constraint, there are two options in the statement, **Restrict** and **Cascade**. If **Restrict** is chosen - it is also the default option of the two - then it will not be dropped if it is referenced by anything else; thus an alternate key attribute will not be dropped if it is referenced by a foreign key. If **Cascade** is chosen, then anything referencing the dropped item will be dropped as well.

If a new attribute is added to a relation already containing data, then a suitable default value should be provided, since the attribute will initially be filled with this value in every tuple. The default should adhere to the integrity constraints; in some cases, the new attribute may need to have (some of) its values appropriately amended before the relation can be used again - see the example on the next slide.
The default value for the Sex attribute is just a convenient device to minimise the work in adding the extra attribute with the correct values, assuming most employees are male. It is not permanently required. There will have to be manual alterations to the EMPLOYEE relation afterwards to ensure that the female employees are correctly recorded.

If a constraint is added to an existing attribute, then it will be accepted only if it already applies to every value in that attribute. Thus if salaries in the Sal attribute do not already lie in the range given, the constraint will be rejected.