CS 252: Fundamentals of Relational Databases: SQL2

Dr. Alexandra I. Cristea
http://www.dcs.warwick.ac.uk/~acristea/
Interim Summary

Material covered so far:
• Everything for data stored in one table.
• Creating base tables without constraints.
• Inserting rows and partial rows, NULL.
• Expressing queries using (`SELECT ... FROM ... WHERE`).
• Predicates (`WHERE`) and predicate combination (`AND` and `OR`).
• Commitment and rollback.
• Deleting (`DELETE`) and updating (`UPDATE`) rows.
• Subqueries.
This lecture

Data description language:
  – Creating tables with constraints.
  – Oracle data types.

• Constraints: NOT NULL, candidate keys, primary keys, foreign keys, CHECK predicates.

Single and multiple table joins using SELECT.

Deleting entire tables with DROP TABLE or their entire contents with TRUNCATE TABLE.

ALTER TABLE for modifying and adding to existing tables.
Problems with One Table

- So far: just one example base table.
- In practice: several; usually impractical and undesirable to cram everything into one!
  - E.g., some CDs contain several separately identifiable pieces of music. If we include information about these in extra columns of our Collection table, then we will have to repeat the general information about a CD containing $n$ pieces of music $n$ times!

In the first lecture we looked at just one example base table. In practice an SQL database consists of several base tables. There are many reasons why it is usually impractical and undesirable to cram everything into one base table.

For example, some CDs contain several separately identifiable pieces of music. If we include information about these in extra columns of our Collection table, then we will have to repeat the general information about a CD containing $n$ pieces of music $n$ times!
Consider classical music albums where data includes:

- conductor – “Simon Rattle"
- soloist – “John Todd"
- composer – “Elgar"
- work – “Cello Concerto"
- orchestra – “Berlin Philharmonic"

These define attributes of one piece on one CD. One CD may contain several equally relevant pieces of music.

We could define separate tables for the classical albums in our Collection. These could contain columns for:

- Track ranges (e.g. 3 to 6, composer, work.)
- Performer, composer, orchestra, soloist.
Multiple (Base) Tables

We want to split the information up into separate tables but we need some way of linking the information together.

⇒ Need a unique reference number for each CD - the 12 digit barcode.

To record the information specific to pop albums:

Please note that, whilst adding a reference number (or ID) is the simple way, it is not the only way, nor is it the elegant way. Further on you will learn alternatives to this linking technique.
### Pop albums

<table>
<thead>
<tr>
<th>barcode</th>
<th>artist</th>
<th>album</th>
</tr>
</thead>
<tbody>
<tr>
<td>042282289827</td>
<td>U2</td>
<td>The Unforgettable Fire</td>
</tr>
<tr>
<td>04228429920</td>
<td>U2</td>
<td>Rattle and Hum</td>
</tr>
<tr>
<td>731451034725</td>
<td>U2</td>
<td>Achtung Baby</td>
</tr>
<tr>
<td>026734000524</td>
<td>Underworld</td>
<td>Second Toughest in the Infants</td>
</tr>
<tr>
<td>72438491321</td>
<td>The Verve</td>
<td>Urban Hymns</td>
</tr>
<tr>
<td>724385583223</td>
<td>Foo Fighters</td>
<td>The Colour and the Shape</td>
</tr>
</tbody>
</table>

column barcode format 999999999999;
Example: Multiple base tables

To split information to do with pop albums that was represented in a single base table previously:

CD company | CD year
---|---
barcode | company | barcode | year

042282289827 | Island | 042282289827 | 1984
042284229920 | Island | 042284229920 | 1988
731451034725 | Island | 731451034725 | 1991
026734000524 | Junior | 026734000524 | 1996
724384491321 | Virgin | 724384491321 | 1997
724385583223 | Capital | 724385583223 | 1997

We also need a similar table for *Number of tracks*.

This example illustrates the so-called *sixth normal form* (6NF) that you learn in the theory section of CS252. We could perhaps have a single table with columns barcode, company, year, and tracks but instead we have split it into two separate tables, one associating bar codes with companies, the other associating bar codes with years.

In the theory section of CS252 you learn that splitting (decomposing) into several relvars is needed when certain attributes do not apply in every case. For example, if the single 3-attribute relvar is used (in Rel) but a certain CD does not have a particular year, then we will not be able to record that CD at all. SQL allows you to overcome that problem by recording the year as NULL instead of splitting the table, but NULL is a highly problematical construct that many people prefer to avoid using.

6NF is not *always* recommended, however. For example, recording the years of albums in a separate table would definitely be *contraindicated* if it is in fact impossible for an album to exist that has a company but does not have a year. With separate tables we might need complicated constraints to tell the DBMS to make sure that at all times every row in the CD_year table has a matching row in the CD_company table and every row in CD_company has a matching row in CD_year.
Example: Representing Band Members

We can create a new table to store information about band members:

<table>
<thead>
<tr>
<th>artist</th>
<th>member1</th>
<th>member2</th>
<th>member3</th>
<th>member4</th>
</tr>
</thead>
<tbody>
<tr>
<td>U2</td>
<td>Bono</td>
<td>Edge</td>
<td>Clayton</td>
<td>Mullen Jr.</td>
</tr>
<tr>
<td>Foo Fighters</td>
<td>Grohl</td>
<td>Smear</td>
<td>Mendel</td>
<td>Goldsmith</td>
</tr>
</tbody>
</table>

*NOTE: Very bad design, not recommended! See the Notes for explanation.*

Here we can use the artist attribute in our *Pop_albums* table to lookup information about each artist.

In order to be able to lookup the artist we have to make sure that the artist name is unique. We can add constraints to enforce uniqueness.

Why is this not a good way to represent band members, and what should we do to correct it?

We use this example to explain certain important features of SQL, but note carefully that actually it illustrates very bad practice indeed. For consider:

Unless we accept the use of NULL, the design restricts us to bands consisting of exactly four members.

Even if we do accept the use of NULL, the design restricts us to bands consisting of at most four members.

If we allow NULL to be used for bands with less than four members, do we allow, for example, NULL to appear for member2 in a row where NULL does not appear for member3? Probably not. We probably want to ensure that if NULL appears in the \( j \)-th column of some row, then it also appears in every \( k \)-th column where \( k > j \).

Even when we do have four band members, such as John Lennon, Paul McCartney, George Harrison, and Ringo Starr, how is it decided which of these is assigned to which member column?

As an exercise, you might like to think about a more appropriate design for recording bands and their members. If you have followed the theory part of CS252 this will not be a difficult exercise for you — remember the example of enrolments (of students on courses).
Advanced Table Creation

The definition of a column can include more than just \textit{column name} and \textit{data type}. The definition of a table consists of a set of column definitions and a set of constraint declarations.

Constraints that can be declared include:

- Numerical precision, string length.
- Prohibiting the appearance of \texttt{NULL} in a column.
- Candidate keys - unique data.
- Primary keys.
- Foreign keys.
- Check constraint - predicate.

Constraints must be satisfied at all times and so are conceptually checked \textit{every time the database is updated} (\texttt{UPDATE} and \texttt{INSERT}). The SQL DBMS does the checking (e.g., Oracle does this for you).
Constraints

Two kinds of constraint:

**Column Constraint** Included in a column definition. Applies to each value in that column.

**Table Constraint** Must be satisfied for the entire table. Specified at the end of the table definition after the columns have been defined.

Note: Some table constraints that reference only one column can be included in the column definition instead of being declared at the end.

It is important to be aware that the distinction between column constraints and table constraints is one of syntax only. Every column constraint can be expressed as a table constraint if preferred. For example, consider the column definition `C CHAR(5) NOT NULL`, appearing in the definition of table `T` (slide 15 shows some examples of such constraints). That NOT NULL constraint could be written as the following table constraint (see slide 24), to be included in the definition of `T`:

```
CHECK ( C IS NOT NULL )
```

Furthermore, even this is really a shorthand, for it means “there must never exist in table `T` a row in which NULL appears in the column `C.”

In standard SQL that can be written out explicitly like this:

```
CHECK ( NOT EXISTS ( SELECT * FROM T WHERE C IS NULL ) )
```

The only bit of SQL in this expression that we have not yet covered is `NOT EXISTS`. It is SQL’s counterpart of the `Tutorial D IS_EMPTY` operator that you learn about in the theory section of CS252. Its meaning is self-explanatory. However, you will not be able to write such constraints in Oracle because Oracle, like most (but not all) SQL implementations, does not permit a SELECT expression to appear inside a constraint declaration.
CREATE TABLE Syntax

Syntax of CREATE statement:

CREATE TABLE table_name (  
[column_name data_type [column_constraint],]*  
column_name data_type [column_constraint]  
[,table_constraints] );

The data type of a column, sometimes called its declared type, is a column constraint that is at least partially checked syntactically — at “compile time”, as it is commonly expressed. For example, if column C is of type INTEGER, and is the only column of table T, then the following command is seen to be erroneous without the DBMS having to attempt to execute it:

INSERT INTO T VALUES ( 1.5 ) ;

The expression “1.5” does not denote a value of type INTEGER.

By contrast, a column constraint or table constraint cannot in general be checked this way. For example, consider the following table constraint:

CHECK ( X > Y )

As we have seen, this is really short for

CHECK ( NOT EXISTS ( SELECT * FROM T WHERE NOT ( X > Y ) ) )

Now consider the following two commands:

1. UPDATE T SET X = 1.5 WHERE Y > 5 ;
2. UPDATE T SET X = X + 1 WHERE Y > 5 ;

The DBMS can reject the first command as a syntax error but cannot possibly tell, by mere inspection of the second command, whether some row in T is going to have an X value less than or equal to its Y value if the command is accepted.

Note that the second command could cause a violation of the data type constraint. If the declared type of X is NUMBER(3) (meaning at most 3 decimal digits) and the X value in some row satisfying Y > 5 is 999, then the update must fail at runtime.
### Number Data Type

**NUMBER** Space for 40 digits. Also “+”, “-”, “.” and “E”.

Possible numbers: 123, 123.45, 123E - 3 = 0.123,
43.2E7 = 43.2 x 10^7.

**NUMBER(size)** Same as NUMBER with size digits and no decimal point.

- Maximum value for size is 105.
- For a NUMBER(3), maximum value is “999” and minimum is “-999”.

**NUMBER(size,d)** Specify NUMBER size and number of digits d following decimal point.

- For a NUMBER(5,2) maximum is “999.99”.

**NUMBER is not a standard SQL data type. Standard SQL numeric data types**
**are DECIMAL(n,m), NUMERIC(n,m) meaning the same as DECIMAL(n,m),**
**INTEGER, SMALLINT, BIGINT, FLOAT, REAL, DOUBLE PRECISION.**
Rounding on Insertion

Note that the **INTEGER** data type is the **NUMBER** data type with the constraint that there is no decimal point allowed.

Values are rounded on insertion.

Rounding to an integer:

• $0.0 \div 0.49$ rounded to $0$
• $0.5 \div 0.9$ rounded to $1$
### Examples:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Inserted</th>
<th>Stored</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUMBER(4,1)</td>
<td>123.4</td>
<td>123.4</td>
</tr>
<tr>
<td></td>
<td>123.45</td>
<td>123.5</td>
</tr>
<tr>
<td></td>
<td>1234.5</td>
<td>FAIL</td>
</tr>
<tr>
<td>NUMBER(4)</td>
<td>123.4</td>
<td>123</td>
</tr>
<tr>
<td></td>
<td>1234.5</td>
<td>1235</td>
</tr>
<tr>
<td></td>
<td>12345</td>
<td>FAIL</td>
</tr>
<tr>
<td>NUMBER(4,-1)</td>
<td>123.4</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td>125</td>
<td>130</td>
</tr>
<tr>
<td></td>
<td>1234.5</td>
<td>1230</td>
</tr>
<tr>
<td>NUMBER</td>
<td>123.445</td>
<td>123.445</td>
</tr>
<tr>
<td></td>
<td>1234.5</td>
<td>1234.5</td>
</tr>
</tbody>
</table>
Character and Date Data Types

Data types for representing strings and dates:

**DATE** Represents dates and times. Range from January 1, 4712 B.C. to December 31, 4712 A.D.

**CHAR(size)** A string of characters of length size. Choose carefully to avoid truncation! Each row entry is of exactly this size. Maximum number of characters is 2000.

**VARCHAR(size)** A string of characters up to the maximum length of size. Storage space varies along with what is stored. Maximum number of characters is 4000.

*The range of dates supported by Oracle is nonstandard. In standard SQL the years range from 9999BC to 9999AD.*

*The maximum lengths for CHAR and VARCHAR are also Oracle-specific and not standard.*

*For VARCHAR, the minimum length should be zero. Oracle has a minimum of 1, however (check this!).*
Not Null Columns

Simplest constraint - NOT NULL - in a column definition, specifies that NULL is prohibited from appearing anywhere in that column. No INSERT or UPDATE that sets this column to NULL will be allowed.

Creating a table to store quantities of a CD in a collection:

```sql
CREATE TABLE Quantity (
    barcode NUMBER(12) NOT NULL,
    quantity NUMBER(3) NOT NULL
);
```

This specifies that bar codes are integers of no more than twelve decimal digits and quantities are integers of no more than three digits (maximum 999).

It is possible to have a negative quantity!
(Candidate) Keys

A (Candidate) Key is a combination of one or more columns, the values of which uniquely identify each row of the table.

Column constraint - use keyword UNIQUE:

```
CREATE TABLE Quantity (  
  barcode NUMBER(12) NOT NULL UNIQUE,  
  quantity NUMBER(3) NOT NULL  
);
```

The following SQL is successful:

```
INSERT INTO Quantity
VALUES (123456789012, 2);
```

1 row created.

In the theory section of CS252 you call a candidate key just ‘key’, following Date. Silberschatz calls it ‘candidate key’. In the theory you learn that a key $K$ is a set of attributes of a relvar such that (a) no two tuples appearing in any relation assigned to that relvar agree in value for each of those attributes, and (b) no proper subset of $K$ has that same “uniqueness” property. You also learn that a key constraint tells the DBMS that the uniqueness property must be adhered to at all times, but the DBMS has no way of checking the second property, known as irreducibility.

The key word UNIQUE in the example on this slide specifies that the single column barcode constitutes something very like a key for the table Quantity. The additional constraint NOT NULL completes that specification — without it, the set \{ barcode \} would remain just “something very like a key” (because the possible appearance of NULL in a key column makes nonsense of the key concept, and in some SQL implementations, contrary to the ISO SQL standard, UNIQUE does imply NOT NULL).
However, this insertion will then fail:

```
INSERT INTO Quantity
VALUES (123456789012, 1);
...
Error at line 1:
ORA-00001: unique constraint (XXX) violated.
```
More than one (Candidate) Key

For the Pop_albums table, there are two (candidate) keys:
- The barcode
- The artist with the album combined.

Can express these in a combination of column and table constraints.

```sql
CREATE TABLE Pop_album (  
  barcode NUMBER(12) NOT NULL UNIQUE,
  artist VARCHAR(100) NOT NULL,
  album VARCHAR(150) NOT NULL,
  UNIQUE (artist, album)
);
```

Now you can see that the UNIQUE on barcode is really a table constraint disguised as a column constraint. Consider the column constraint NOT NULL. The DBMS can see if this is satisfied just by looking at a value that is being assigned to the column in question. That is not the case with UNIQUE, where the DBMS conceptually has to look at the entire table to see if some row with that bar code already exists. (In practice there are ways of making the necessary “look-up” run very fast, as you may well be aware.)
Primary Key

A Primary Key is a candidate key, except there can be only one per table. In all tables with a barcode column, the primary key has been chosen to be this column.

```
CREATE TABLE Quantity (  
  barcode NUMBER(12) PRIMARY KEY,  
  quantity NUMBER(3) NOT NULL  
);
```

It is possible to use both candidate keys and primary keys – imagine table Pop albums with one of the "UNIQUE" keywords replaced by "PRIMARY KEY". – what is the difference?

A primary key is a candidate key, but the reverse is not true. A candidate key that is not the primary key is called an alternate key.

In SQL, a table can have any number of UNIQUE constraints as well as at most one primary key.
PRIMARY KEY vs UNIQUE

There are a couple of differences between a candidate key and a primary key in SQL:

- PRIMARY KEY implies NOT NULL for each column of the specified key.
- A primary key has an 'index' of that primary key that is automatically added to the table.

So we can dispense with the 'PRIMARY KEY' and replace it with a column that is UNIQUE and NOT NULL and has an index on it. Indexes are used to improve performance of certain queries. We will cover indexes later on in the course.

Indexes are an implementation issue only, having no effect on semantics, and that is why they are covered later in the course rather than now.

We will see another advantage of PRIMARY KEY when we look at foreign keys in the next two slides …

An SQL foreign key declaration can specify just the referenced table name when the intended referenced columns are those of the primary key of the referenced table. Even if the only key is specified using UNIQUE, then referencing foreign keys must specify the column names — and in the right order!
### Foreign Key

Imagine a table that stored information about artists:

**Band_members:**

<table>
<thead>
<tr>
<th>artist</th>
<th>member1</th>
<th>member2</th>
<th>member3</th>
<th>member4</th>
</tr>
</thead>
<tbody>
<tr>
<td>U2</td>
<td>Bono</td>
<td>Edge</td>
<td>Clayton</td>
<td>Mullen Jr.</td>
</tr>
<tr>
<td>Foo Fighters</td>
<td>Grohl</td>
<td>Smear</td>
<td>Mendel</td>
<td>Goldsmith</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

The primary key is the artist column. In the *Pop_albums* table, artist is a **Foreign Key**.

A foreign key is a primary key (or unique) from another table. Use `REFERENCES` to establish a foreign key constraint (also called a referential constraint).

On this slide we again see a table constraint masquerading as a column constraint. `REFERENCES Band_members (artist)` is shorthand for the following table constraint:

```sql
FOREIGN KEY (artist) REFERENCES Band_members;
```

Here *Pop_albums* is called the **referencing table** and *Band_members* is the **referenced table**.

We can optionally omit the qualification ``(artist)'' on Band_members because by default the columns of the primary key of the referenced table are implied.

_Beware: This slide and the next may give the misleading impression that a foreign key is always a single column, which is NOT the case._

_The longhand notation that is needed in general is:  
FOREIGN KEY ( <column name comma list> ) REFERENCES <table name> [(column name comma list)]._
CREATE TABLE Pop_albums (  barcode NUMBER(12) PRIMARY KEY,  artist VARCHAR(100) NOT NULL  REFERENCES Band_members(artist),  album VARCHAR(150) NOT NULL,  UNIQUE (artist, album) );

Albums can only be inserted into the Pop albums table if the artist already exists in the Band members table.

“artist” should better be replaced by “Band name” (see slide explaining the bad design).
Check Constraints

Check constraints are logical conditions to check prior to insertion or update. Use keyword CHECK followed by a predicate:

```
CREATE TABLE Quantity (
  barcode NUMBER(12) PRIMARY KEY,
  quantity NUMBER(3) NOT NULL CHECK (quantity >= 0)
);
```

This ensures that the quantity column is never negative. Cannot use subqueries or some functions.
Table Constraints

*Table Constraints* specify constraints at the end of the table definition.

```sql
CREATE TABLE Quantity (
  barcode NUMBER(12),
  quantity NUMBER(3),
  CHECK (barcode IS NOT NULL),
  CHECK (quantity >= 0),
  PRIMARY KEY (barcode)
);
```

Notice that writing `CHECK ( quantity > 0 )` as a column constraint is not really a shorthand for the table constraint. As you can see, the only thing that is saved by writing it thus is a single comma!
Specifying a Particular Table Within a Query

Often tables have columns with the same name. To distinguish between these attributes, use the dot notation:

```
table.column
```

```
SELECT Collection.artist
FROM Collection;
```

Actually, the table name `Collection` here is not being used for the purpose of naming or referencing a table — the name does not here stand for the current value of the base table `Collection`. Rather, it is an example of what E.F. Codd called a `range variable` (and that is the term used in the SQL standard). It is variable that denotes a row — each row in turn of the table `Collection`. It is said to “range over” the rows of `Collection`.

In a pure algebraic relational language such as `Tutorial D` range variables are not used. The reason why they are needed in SQL becomes apparent on the next slide …
Selecting From Multiple Tables

In relational algebra, this is a join. Consider joining the tables Pop albums and CD year:

```sql
SELECT artist, album, year
FROM Pop_albums, CD_year
WHERE Pop_albums.barcode = CD_year.barcode;
```

<table>
<thead>
<tr>
<th>ARTIST</th>
<th>ALBUM</th>
<th>YEAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>U2</td>
<td>The Unforgettable Fire</td>
<td>1984</td>
</tr>
<tr>
<td>U2</td>
<td>Rattle and Hum</td>
<td>1988</td>
</tr>
<tr>
<td>U2</td>
<td>Achtung Baby</td>
<td>1991</td>
</tr>
<tr>
<td>Underworld</td>
<td>Second Toughest in the Infants</td>
<td>1996</td>
</tr>
<tr>
<td>The Verve</td>
<td>Urban Hymns</td>
<td>1997</td>
</tr>
<tr>
<td>Foo Fighters</td>
<td>The Colour and the Shape</td>
<td>1997</td>
</tr>
<tr>
<td>CS252: Fundamentals of Relational Databases</td>
<td>1997</td>
<td></td>
</tr>
</tbody>
</table>

A Tutorial D counterpart of this SELECT expression is

```sql
(Pop_albums JOIN CD_Year) {artist, album, year}
```

but note that really SELECT DISTINCT is needed in the SQL expression to give true counterparts. Because the SELECT clause includes all the columns except the common one, barcode, the above expression is in turn equivalent to

`Pop_albums COMPOSE CD_Year`

The FROM clause yields the Cartesian product of its two operands: each row in the second operand is joined to each row in the first operand. It is very similar to the relational JOIN operation when applied to relations with no common attributes. Recall that this special case of JOIN is sometimes known as TIMES. But SQL’s FROM differs from TIMES in the following respects:

The order of the operands is significant, because the order of columns in an SQL table is significant. In fact, if the first operand has `n` columns, then those columns remain the first `n` columns of the result, the first column of the second operand becomes the `n+1`-th column, and so on. TIMES, by contrast, is commutative, being just a special case of JOIN, which is also commutative.

If some column name is the name of a column in each operand, then the result has two columns of that name. (TIMES requires the headings of its operands to be disjoint.) In the SELECT and WHERE clauses of the same SELECT expression, those columns can be distinguished by the use of range variable names, as shown in the example.

When the FROM clause specifies three or more tables, the result of FROM applied to the first two is joined in similar fashion with the third operand, the result of that is joined similarly with the fourth, and so on.

Note that to achieve the counterpart of a relational JOIN over one or more common attributes, you have to apply an explicitly declared restriction condition in the WHERE clause. It is a common cause of errors in SQL, when joining several tables, to accidentally omit something from the WHERE clause and thus give rise to huge result tables. Recall that if the cardinalities of relations `r1` and `r2` are `j` and `k`, then the cardinality of `r1` TIMES `r2` is `j*k`. The next slide illustrates the kind of WHERE condition that is commonly needed.
Multiple Joins

To get back the table from the first seminar, use:

```sql
SELECT artist, album, tracks, company, year
FROM Pop_albums, CD_tracks, CD_year, CD_company
WHERE Pop_albums.barcode = CD_year.barcode
    AND CD_year.barcode = CD_company.barcode
    AND CD_company.barcode = CD_tracks.barcode;
```
Table Management

So far there has been no mention of how to manage (delete or modify) table structures. This is a complicated issue.

To delete a table - when you don't need it:

```sql
DROP TABLE table_name;
```

You can rollback such a change.

To empty a table of all its values while keeping its structure, use:

```sql
TRUNCATE TABLE table_name;
```

Rollback is not possible - use with extreme care.

The commands we learn in the remaining slides of this lecture have no counterparts in Tutorial D. They illustrate some of the kinds of things that are needed in commercial products but have no special theoretical significance.

The strangely named TRUNCATE TABLE is a proprietary extension in Oracle; it is not defined in the ISO SQL standard.

Note that ALTER TABLE and TRUNCATE TABLE apply exclusively to base tables. You cannot use them on views (special kinds of tables that we teach later in CS252).
Altering Tables

Adding Columns

Use the ALTER TABLE and ADD table elements (i.e., column definitions and constraints):

```
ALTER TABLE table name ADD ( ... );
```

Body of ALTER TABLE same as body of CREATE TABLE.

It is not possible to ADD a NOT NULL column.

---

The SQL standard does allow NOT NULL to be included, but only if an explicit DEFAULT clause is also included. E.g.: ADD ( newcol INTEGER NOT NULL DEFAULT 0 ).
Altering Tables

Modifying Columns

Use the **ALTER TABLE** and **MODIFY** statements:

```
ALTER TABLE table name MODIFY ( ... );
```

Imagine there is an International move to standardise barcodes so that they have 14 digits instead of 12.

```
ALTER TABLE Quantity MODIFY (
  barcode NUMBER(14) );
```

Note: **MODIFY** is a proprietary extension in Oracle, not defined in the ISO SQL standard.
Altering Tables

Renaming Columns

Use the `ALTER TABLE` and `RENAME COLUMN` statements:

```
ALTER TABLE table_name RENAME COLUMN column1 TO column2;
```

```
ALTER TABLE Quantity RENAME COLUMN barcode TO serial_no;
```

RENAME COLUMN is a proprietary extension in Oracle, not defined in the ISO SQL standard.

Standard SQL includes several other options in `ALTER TABLE`, beyond the scope of CS252. For example, there is a `DROP COLUMN` option for removing a column from a base table, and there are also the `ADD CONSTRAINT` and `DROP CONSTRAINT` options.
Rules for Table Alteration

These are the rules for adding a column to a table:
• You may add a column at any time if NOT NULL is not specified.
• You may add a NOT NULL column in three steps:
  1. Add the column without NOT NULL.
  2. Fill every row of the column with data.
  3. Modify the column to be NOT NULL.

These are the rules for modifying a column:
• You can increase a CHAR column's width or a NUMBER column's precision at any time.
• You can add or decrease the number of decimal places in a NUMBER column at any time.

The restriction discussed on this slide is peculiar to Oracle. The ISO SQL standard does allow NOT NULL to be specified on an ADDed column, so long as the definition includes a DEFAULT clause. DEFAULT clauses are permitted on all column definitions, whether in CREATE TABLE or ALTER TABLE. The key word is followed by an expression—usually a simple literal—specifying the value that is to appear in the column (on INSERT) when no values is explicitly given. The specified value must of course be of the data type declared for the column in question.

For example, if base table T does not have a column named C5, the following ALTER TABLE statement is supported by the SQL standard, but not by Oracle:
ALTER TABLE T ADD ( C5 INTEGER NOT NULL DEFAULT 0 );
In addition, if a column is **NULL** for every row of a table, you can make any of these changes:

- Change the data type.
- Decrease a **CHAR** column's width or a **NUMBER** column's precision.
Interim Summary

Data description language:
  – Creating tables with constraints.
  – Oracle data types (apart from DATE).

- Constraints: NOT NULL, candidate keys, primary keys, foreign keys, CHECK predicates.

Single and multiple table joins using SELECT.
Deleting entire tables with DROP TABLE or their entire contents with TRUNCATE TABLE.
ALTER TABLE for modifying and adding to existing tables.