Topic 03: SQL

ICT285 Databases

Dr Danny Toohey

About this topic

In this topic, we give an overview of the industry standard language for relational databases, SQL. Most of the labs will involve SQL, and we will introduce its main features throughout the practical work.

Topic Learning Outcomes

After completing this topic you should be able to:

- Describe some of the major features of SQL and their usage
- Use SQL for querying and creating tables
  (NB: much of the practicalities of SQL are addressed in the labs)
- Know where in the unit we will cover the different aspects of SQL in more detail

Resources for this topic

READING

- Kroenke & Auer, Chapter 2 'Introduction to Structured Query Language' (13th and 14th editions)

OTHER RESOURCES

- Oracle database SQL Language Reference https://docs.oracle.com/database/121/SQLRF/toc.htm
- SQLCourse.com has some useful tutorials and an interactive SQL interpreter for additional practice: http://www.sqlcourse.com/intro.html

Kroenke, D.M., and Auer, D.J., 2016, Database Processing: Fundamentals, Design and Implementation, 14th Edition, Pearson, Boston.

Lab 03: More on the SELECT statement

This lab continues our treatment of the SQL SELECT statement by introducing aggregate and grouping queries, and queries based on set operations. We will also cover ways of handing duplicates and nulls in queries.

Topic Outline

1. Introduction
2. SQL: data manipulation
3. SQL: data definition
4. SQL: other features

Topic 03: Part 01

Introduction

Functions of a DBMS

Codd (1982) listed 8 services that should be provided by any full-scale DBMS:

- Data storage, retrieval, and update
- User accessible catalogue
- Transaction support
- Concurrency control services
- Recovery services
- Authorisation services
- Support for data communications
- Integrity services

Database languages

- A database *language* allows the user to interact with the database to provide this functionality
- This involves:
  - **Data definition** – creating tables and other objects
  - **Data manipulation** – retrieving and adding/deleting/ updating the data records
  - **Data control** – control access to the database objects
  - **Transaction support**
- (Often the term *database query language* is used to cover all of these features, though strictly a query language is only used for retrieval)

Database languages

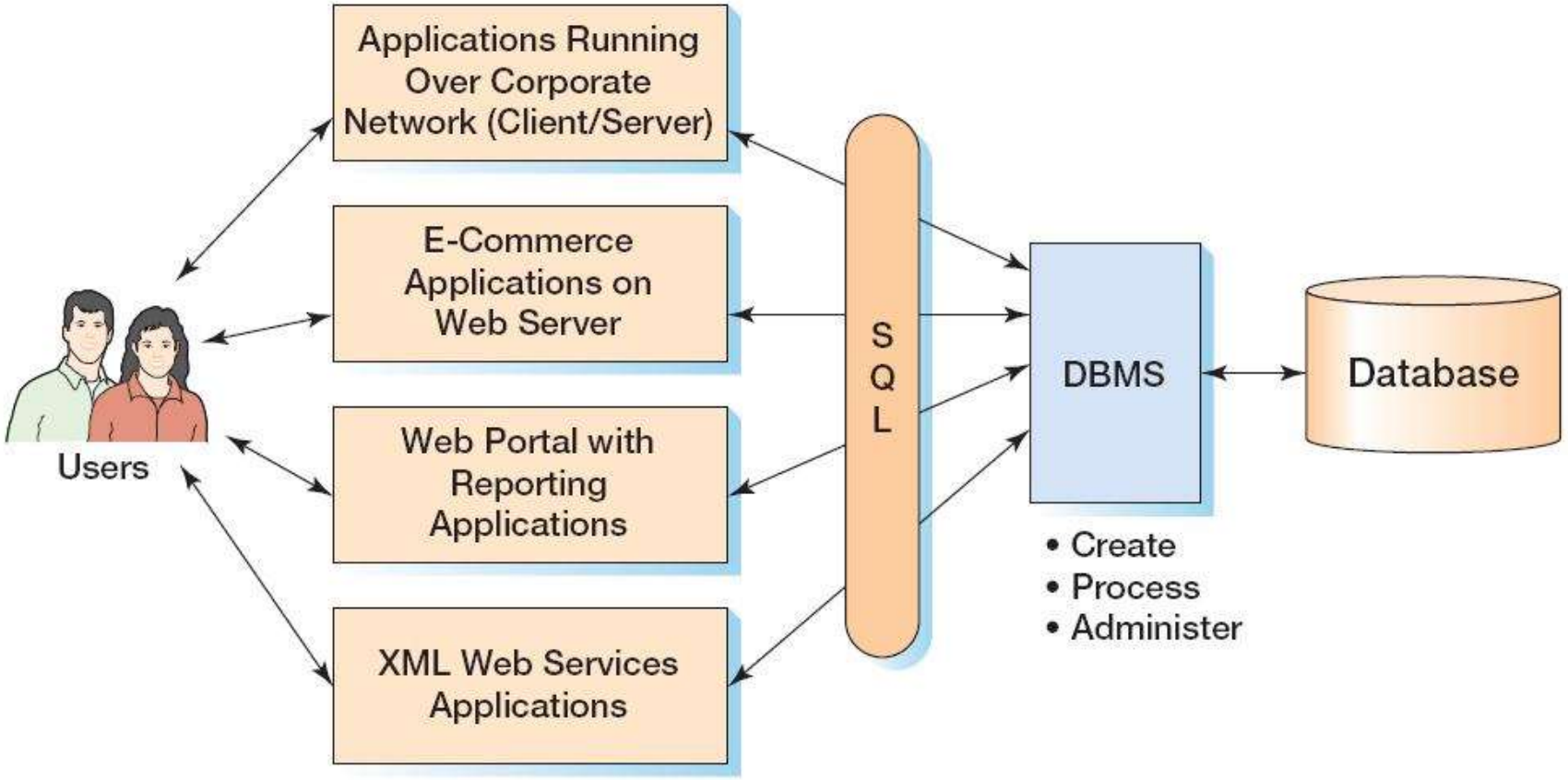
Why not use the programming languages with which we are already familiar?

- … after all, we can model and manipulate data in the more "traditional" programming languages using structures such as structs (in C) or objects (in Java)

- Using a language *specific to databases* provides:
  - Ease of programming
  - Data independence
  - Ability of the DBMS to produce highly optimised code
  - Portability

In practice, this language is usually SQL

An example database system:



Kroenke & Auer, Fig 1-15

Relational algebra and SQL

Both of these are relational database languages

- **Relational algebra** is part of the theoretical relational data model, and defines the operations that are possible on it
- **SQL** is an implementation language that provides both data manipulation and data definition as well as other features

SQL

- SQL is a standard
- It is a **non-procedural** (declarative) language
- It uses **table, row, column** in place of relation, tuple and attribute
- It does not include ‘control-flow’ commands (e.g., if…then…else etc)
- Usage:
  - Standalone statements (as you will mostly be doing in the labs)
  - Embedded in code

A Brief History of SQL

1972: System R (IBM)
1974: SQUARE
1974: SEQUEL
1976: SEQUEL 2
1977: Name change to SQL
1978: First commercial implementation (Oracle Corp)
1986: SQL86 Standard approved (150 pages)
1992: SQL92 Standard (600 pages)
1999: SQL:1999 (> 2100 pages)
2003: SQL:2003 - introduced XML-related features
2006: SQL:2006 standard released
2008: SQL:2008 standard released
2011: SQL:2011 standard released
2016: SQL:2016 standard released

SQL standards

- Most vendors support MOST of the standard
- The level of support differs between products
- And names for features often differ (e.g. T-SQL for Microsoft; PL/SQL for Oracle, SQL PL for DB2) for embedded SQL
- Wikipedia has a useful page comparing many RDBMSs, including SQL features: https://en.wikipedia.org/wiki/Comparison_of_relational_database_management_systems

Some SQL features

Data Definition (DDL)

- CREATE TABLE, ALTER TABLE, DROP TABLE
- CREATE VIEW, DROP VIEW
- CREATE INDEX...

Data Manipulation (DML)

- INSERT, DELETE, UPDATE
- SELECT...

Data Control (DCL)

- GRANT, REVOKE

Transaction Control (TCL)

- COMMIT, ROLLBACK

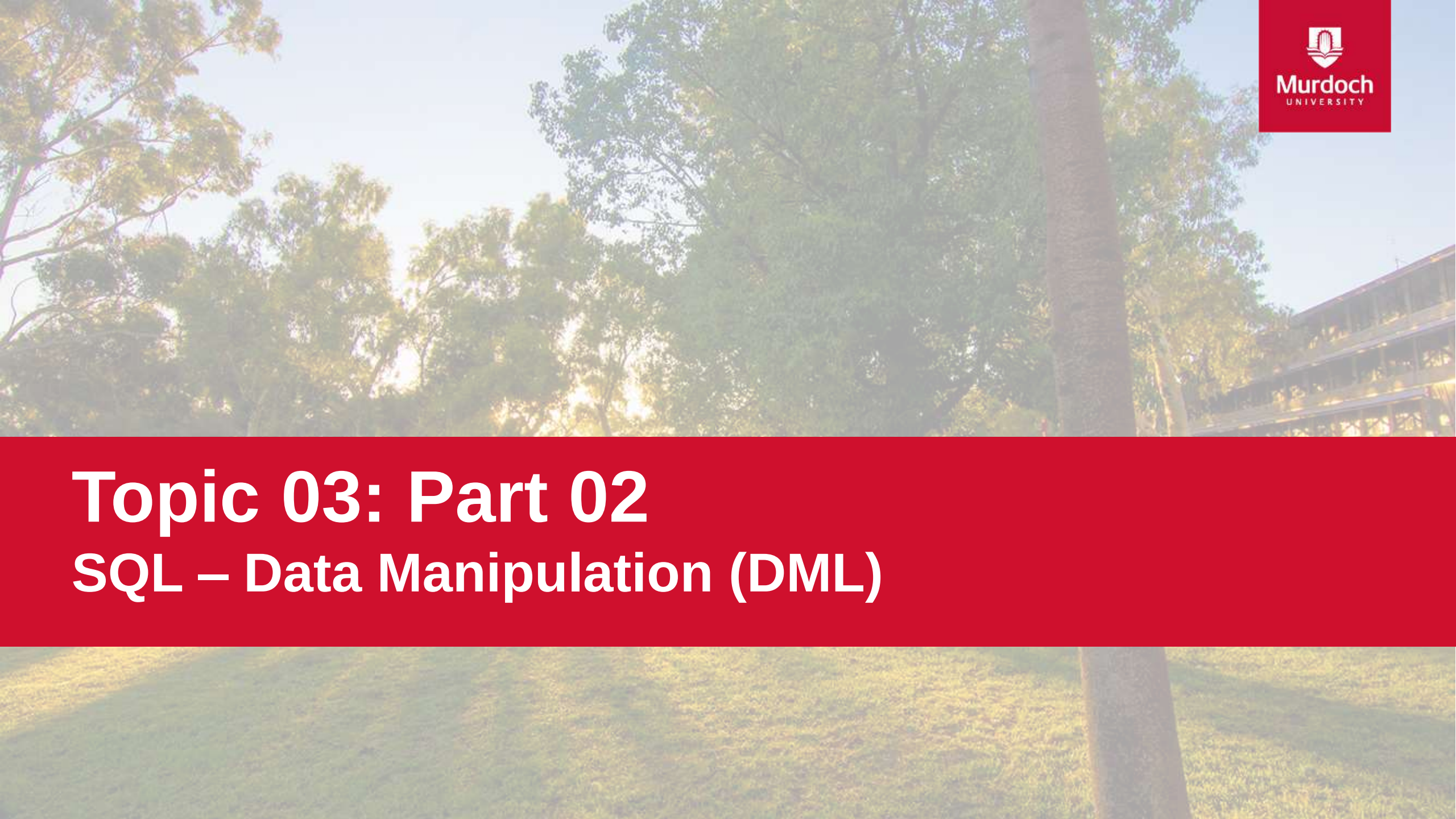
SQL/Persistent stored modules (SQL/PSM)

- Extensions of SQL to allow for procedural programming concepts, e.g., functions, triggers, procedures

… and much, much more …

SQL in labs

SQL for:	Lab
Data definition	
CREATE, ALTER, DROP …	
…TABLE	4, 6
…VIEW	7
…CONSTRAINT	4
…INDEX	7,8
Data manipulation	
SELECT	1,2,3,11
INSERT, DELETE, UPDATE	4
Data control	
GRANT, REVOKE	7
Transaction control	
COMMIT, ROLLBACK	9
Procedural language (PL/SQL	
Triggers, Stored procedures	10



Topic 03: Part 02

SQL – Data Manipulation (DML)

SQL for data manipulation: SELECT

single tables
joins
grouping and aggregation
set operations

subqueries

Basic SQL SELECT

SELECT queries retrieve data from one or more tables

Simplified syntax:

```
SELECT <list of column expressions>
  FROM <list of tables and join operations>
  WHERE <list of logical expressions for rows>
  GROUP BY <list of grouping columns>
  HAVING <list of logical expressions for groups>
  ORDER BY <list of sorting specifications>
```

SELECT …

The SELECT line allows us to retrieve only specified **columns** from a table (similar to the relational algebra *project* operator)

"List the full names of all students"

```
SELECT StdFirstName, StdLastName
FROM STUDENT;
```

WHERE

The WHERE clause allows us to retrieve only specified **rows** from a table (similar to the relational algebra *restrict* operator)

"List the names of students with a StdGPA >= 3.0"

```
SELECT StdFirstName, StdLastName, StdGPA
FROM STUDENT
WHERE StdGPA >= 3.0;
```

STUDENT					
StudentNo	StdFirstName	StdLastName	StdCity	StdMajor	StdGPA

Comparison Operators

Standard Comparison Operators:

=	equal to
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
<> or !=	not equal

The WHERE clause can also contain:

- Arithmetic operators (+, -, *, /, **)
- Logical operators (AND, NOT, OR)
- Range search (BETWEEN/AND)
- LIKE – used for inexact matching
- ANY and ALL (used with subqueries)

Logical Operators AND, OR

Can be used to define WHERE criteria more closely

"List the names of students with a GPA >= 3.0 and who are in the Games Tech major"

```
SELECT StdFirstName, StdLastName, StdGPA, StdMajor
FROM STUDENT
WHERE StdGPA >= 3.0
AND StdMajor = 'Games Tech';
```

STUDENT					
StudentNo	StdFirstName	StdLastName	StdCity	StdMajor	StdGPA

IS NULL, IS NOT NULL

Checks for existence of an attribute value

"List the name of any student that does not have a GPA"

```
SELECT StdFirstName, StdLastName
FROM STUDENT
WHERE StdGPA IS NULL;
```

- THIS IS NOT THE SAME AS StdGPA = 0!

STUDENT					
StudentNo	StdFirstName	StdLastName	StdCity	StdMajor	StdGPA

IN, NOT IN

Used with a set of values, or a subquery (later)

"List the names of students who are from Koondoola, Girrawheen, or Balga"

```
SELECT StdFirstName, StdLastName, StdCity
FROM Student
WHERE StdCity IN ('Koondoola', 'Girrawheen','Balga');
```

STUDENT					
StudentNo	StdFirstName	StdLastName	StdCity	StdMajor	StdGPA

Pattern Matching: LIKE

Where the match is partial rather than exact

- Use LIKE in the WHERE clause
- Use meta characters to specify the pattern

Wildcard:

- **%** in Oracle

Single Character:

- _ in Oracle

Pattern Matching: Examples

"List the unit code and title of units with Finance in their description"

```
SELECT UnitCode, UnitTitle
FROM UNIT
WHERE UnitTitle LIKE '%Finance%';
```

Range Searching

"List the names and GPA of students with a GPA between 3 and 4"

```
SELECT StdFirstName, StdLastName, StdGPA
FROM Student
WHERE StdGPA BETWEEN 3 AND 4;
```

Note that BETWEEN/AND uses >= and <=

STUDENT					
StudentNo	StdFirstName	StdLastName	StdCity	StdMajor	StdGPA

SQL Joins

There are various ways to implement *joins* in SQL:

- Using the JOIN operator
- Matching values (usually PK, FK) in the WHERE clause
- A useful coverage of the different types of joins in Oracle can be found at: https://docs.oracle.com/database/121/SQLRF/queries006.htm#SQLRF30046

Joins using the JOIN operator

```
SELECT <list of column expressions>
 FROM Table1 INNER JOIN Table2
 ON Table1.PrimaryKey = Table2.ForeignKey
 WHERE <list of logical expressions for rows>
```

Can create multiple joins by nesting the JOIN Operator, but the join can only ever be between two tables, i.e:

(Table1 Join Table2) Join Table3

JOIN - example

"List the Names of students enrolled in ICT285 in Semester 2, 2020"

```
SELECT StdFirstName, StdLastName
FROM STUDENT S INNER JOIN ENROLMENT E
ON S.StudentNo = E.StudentNo
WHERE UnitCode = 'ICT285'
AND YearSemester = 'S2 2020';
```

ENROLMENT			
StudentNo	**UnitCode**	**YearSemester**	Mark

STUDENT					
StudentNo	StdFirstName	StdLastName	StdCity	StdMajor	StdGPA

JOIN 2+ TABLES

"List the names of students enrolled in Databases in Semester 2, 2020"

```
SELECT StdFirstName, StdLastName
FROM (STUDENT S INNER JOIN ENROLMENT E
ON S.StudentNo = E.StudentNo)
        INNER JOIN UNIT U ON E.UnitCode =
U.UnitCode
WHERE UnitTitle = 'Databases'
AND YearSemester = 'S2 2020';
```

ENROLMENT			
StudentNo	**UnitCode**	**YearSemester**	Mark

UNIT	
UnitCode	UnitTitle

STUDENT					
StudentNo	StdFirstName	StdLastName	StdCity	StdMajor	StdGPA

JOINS based on PK/FK Equality

```
SELECT <list of column expressions>
 FROM Table1, Table2
WHERE Table1.PrimaryKey=Table2.ForeignKey
```

Can create multiple joins by listing extra tables, however, each join MUST include its "join condition"

```
FROM Table1, Table2, Table3
WHERE Table1.PrimaryKey=Table2.ForeignKey
AND Table2.PrimaryKey=Table3.ForeignKey
```

If the "join condition" is not specified, what do you think will happen??

JOIN USING PK/FK

"List the Names of students enrolled in ICT285 in Semester 2, 2020"

```
SELECT StdFirstName, StdLastName
FROM STUDENT S, ENROLMENT E
WHERE S.StudentNo = E.StudentNo
AND UnitCode = 'ICT285'
AND YearSemester = 'S2 2020';
```

ENROLMENT			
StudentNo	**UnitCode**	**YearSemester**	Mark

STUDENT					
StudentNo	StdFirstName	StdLastName	StdCity	StdMajor	StdGPA

JOIN 2+ TABLES USING PK/FK

"List the names of students enrolled in Databases in Semester 2, 2020"

```
SELECT StdFirstName, StdLastName
FROM STUDENT S, ENROLMENT E, UNIT U
WHERE S.StudentNo = E.StudentNo
AND E.UnitCode = U.UnitCode
AND UnitTitle = 'Databases'
AND YearSemester = 'S2 2020';
```

ENROLMENT			
StudentNo	**UnitCode**	**YearSemester**	Mark

UNIT	
UnitCode	UnitTitle

STUDENT					
StudentNo	StdFirstName	StdLastName	StdCity	StdMajor	StdGPA

Outer Joins

Outer joins are specified in the same way as the inner join, but use the word LEFT JOIN (or RIGHT JOIN)

"List every department with the employee number and last name of the manager, including departments without a manager"

```
SELECT DeptNo, DeptName, EmpNo, LastName
FROM DEPARTMENT LEFT JOIN EMPLOYEE
ON MgrNo=EmpNo;
```

Note LEFT OUTER JOIN is also correct

Grouping and aggregating

Aggregate Functions

SQL has several built-in functions that operate on a single column of a table and return a single value:

- COUNT, MIN, MAX, SUM, AVG

COUNT

Using COUNT(*) returns the number of rows retrieved by the query

```
SELECT COUNT(*)
FROM STUDENT;
```

Using COUNT(ColumnName) will return the number of non-null values in the column

STUDENT					
StudentNo	StdFirstName	StdLastName	StdCity	StdMajor	StdGPA

AVG, MIN, MAX, SUM

Use to find the average, minimum, maximum or sum of the values in the named column:

```
SELECT AVG(StdGPA)
FROM STUDENT;
```

The result column can be renamed using AS

```
SELECT MAX(StdGPA) AS TopGPA
FROM STUDENT;
```

STUDENT					
StudentNo	StdFirstName	StdLastName	StdCity	StdMajor	StdGPA

Using aggregate functions in WHERE clause

Note that aggregate functions **can't** be used in the WHERE clause directly:

```
SELECT StdFirstName, StdLastName
FROM STUDENT
WHERE StdGPA > AVG(StdGPA);
```

does **not** work as a solution to: "Give the names of students who have a higher GPA than the average"

Instead, you need to use a subquery (next slide)

STUDENT					
StudentNo	StdFirstName	StdLastName	StdCity	StdMajor	StdGPA

Using aggregate functions in WHERE clause cont'd

```
SELECT StdFirstName, StdLastName
FROM STUDENT
WHERE StdGPA >
    (SELECT AVG(StdGPA)
    FROM STUDENT);
```

STUDENT					
StudentNo	StdFirstName	StdLastName	StdCity	StdMajor	StdGPA

GROUP BY

GROUP BY ‘separates’ the rows of a table into groups that have the same value for a specified attribute

- e.g. Students can be separated into groups depending on their Major:
- CS, BIS, GT

We can then perform aggregate functions on the *group*, rather than the whole table…

GROUP BY

"List the minimum, maximum and average GPA for **each** of the majors"

```
SELECT StdMajor, MIN(StdGPA), MAX(StdGPA), AVG(StdGPA)
FROM STUDENT
GROUP BY StdMajor;
```

In this example, Major is the grouping attribute, so the SELECT applies to each GROUP in the table rather than each ROW

STUDENT					
StudentNo	StdFirstName	StdLastName	StdCity	StdMajor	StdGPA

GROUP BY …

Since the SELECT applies to a group, each expression in the select clause must have a **single value per group**, i.e.,

- Either an aggregate function
- Or a grouping attribute

So, any column named in the SELECT clause has to appear in the GROUP BY (unless it has an aggregate function applied to it)

GROUP BY … HAVING

HAVING is used to select GROUPS in the same way that WHERE is used to select ROWS

- "List the majors and the average GPA of those majors whose GPA is greater than 3.3"

```
SELECT StdMajor, AVG(StdGPA)
FROM STUDENT
GROUP BY StdMajor
HAVING AVG(StdGPA) > 3.3;
```

STUDENT					
StudentNo	StdFirstName	StdLastName	StdCity	StdMajor	StdGPA

Subqueries

Subqueries

A query that appears in the WHERE or HAVING clause of another query

Can be used with comparison operators, IN/NOT IN, EXISTS/NOT EXISTS, ANY/ALL

Type 1 subquery

- The query executes ONCE and produces a result
- The inner query does not reference the outer query

Type 2 subquery

- The inner query references a table in the outer query
- The inner query is evaluated for **every** row of the outer query

Type 1 Subquery Example

"List the numbers and names of students who are currently studying ICT285"

```
SELECT StudentNo, FirstName, LastName
FROM STUDENT
WHERE StudentNo IN
    (SELECT StudentNo
    FROM ENROLMENT
    WHERE UnitCode = 'ICT285'
        and YearSemester = 'S2 2020');
```

STUDENT					
StudentNo	StdFirstName	StdLastName	StdCity	StdMajor	StdGPA

ENROLMENT			
StudentNo	**UnitCode**	**YearSemester**	Mark

Type 1 Subquery Example cont'd …

We can think of the previous example as evaluating the inner query:

```
(SELECT StudentNo
  FROM Enrolment
  WHERE UnitCode = 'ICT285'
and YearSemester = 'S2 2020');
```

to produce the result set that includes the student numbers of all students studying ICT285 in S2 2020

- and using that as the input for the WHERE clause of the outer query

ENROLMENT			
StudentNo	**UnitCode**	**YearSemester**	Mark

Another Type 1 Example

NOT IN

- "List the numbers and names of students that are NOT enrolled in any units in S2 2020"

```
SELECT StudentNo, StdFirstName, StdLastName
FROM STUDENT
WHERE StudentNo NOT IN
    (SELECT StudentNo
    FROM ENROLMENT
    AND YearSemester = 'S2 2020');
```

STUDENT					
StudentNo	StdFirstName	StdLastName	StdCity	StdMajor	StdGPA

ENROLMENT			
StudentNo	**UnitCode**	**YearSemester**	Mark

NOT IN is not the same as <>

The previous query is NOT THE SAME AS:

```
SELECT StudentNo, FirstName, LastName
FROM Student, Enrolment
WHERE Student.StudentNo <> Enrolment.StudentNo;
```

This would join the rows from the two tables where the primary and foreign keys DIDN'T match – which makes no sense at all!

STUDENT					
StudentNo	StdFirstName	StdLastName	StdCity	StdMajor	StdGPA

ENROLMENT			
StudentNo	**UnitCode**	**YearSemester**	Mark

Multiple-level Subqueries

The query will be evaluated from the innermost first:

"List the names of those students that have been enrolled in Databases"

Student numbers used here – evaluated last

```
SELECT StdFirstName, StdLastName
FROM Student
WHERE StudentNo IN
    (SELECT StudentNo
    FROM Enrolment
    WHERE UnitCode IN
        (SELECT UnitCode
        FROM Unit
        WHERE UnitTitle = "Databases"));
```

This part evaluated 2nd. Returns StudentNo for students in ICT285

This part evaluated first: returns "ICT285"

ENROLMENT			
StudentNo	**UnitCode**	**YearSemester**	Mark

UNIT	
UnitCode	UnitTitle

STUDENT					
StudentNo	StdFirstName	StdLastName	StdCity	StdMajor	StdGPA

Subqueries and Joins

You will have noted that sub-queries can be used to express a join

- Sub-queries can be used to express JOINS **as long as the columns in the SELECT statement are from a SINGLE table**
- The following CANNOT be expressed as a Type 1 subquery:

```
SELECT S.StdLastName, E.Mark
FROM STUDENT S, ENROLMENT E
WHERE S.StudentNo = E.StudentNo;
```

STUDENT					
StudentNo	StdFirstName	StdLastName	StdCity	StdMajor	StdGPA

ENROLMENT			
StudentNo	**UnitCode**	**YearSemester**	Mark

Type 2 Subqueries

In a Type 2 Subquery:

- The *inner* query references a table used in the *outer* query
- The inner query is evaluated for *every row* in the outer query

Also known as **Correlated Subqueries**

Example Type 2 Subquery

"List the titles of units being offered in Semester 2 2020"

```
SELECT UnitTitle
FROM UNIT U
WHERE UnitCode IN
    (SELECT UnitCode
    FROM OFFERING O
    WHERE O.UnitCode = U.UnitCode
    AND YearSemester = 'S2 2020';
```

UNIT	
UnitCode	UnitTitle

OFFERING	
UnitCode	**YearSemester**

EXISTS and NOT EXISTS

Used with a Type 2 subquery

The subquery returns only True (if there is at least one row in the result returned by the subquery) or False (if no rows are returned)

"Find all staff who work at a London branch"

```
SELECT StaffNo, StaffName
FROM STAFF S
WHERE EXISTS
        (SELECT * FROM BRANCH B
         WHERE S.BranchNo=B.BranchNo
         AND City = 'London');
```

EXISTS and NOT EXISTS example

"Retrieve the StaffNumber, the name, school , and salary of staff members who are NOT students." *[NB: this assumes that if a member of staff is also a student, then their student number and staff number will be identical]*

```
SELECT StaffNo, LastName, SchoolName
 FROM STAFF
 WHERE NOT EXISTS
  ( SELECT * FROM STUDENT
      WHERE STUDENT.StudentNo= STAFF.StaffNo);
```

Set operations

Set Operators

UNION, INTERSECTION, MINUS

- As with relational algebra, the two queries must be **UNION COMPATIBLE**
- In this context, this means that both SELECT statements must return EQUIVALENT columns
- Duplicates are eliminated from the result
- The operators can only be used between *complete* SELECT statements, NOT between subqueries within a select

UNION

"Give the student number and names of students who have a mark of > 60 for Systems Analysis OR who have a mark of > 75 for Databases"

We need to construct two queries:

1. Students with a mark of >60 for Systems Analysis
2. Students with a mark of >75 for Databases

The result is the UNION of the first and second queries

UNION

```
SELECT S.StudentNo, StdFirstName, StdLastName
FROM Student S, ClassList C, Unit U
WHERE S.StudentNo = C.StudentNo
AND U.UnitCode = C.UnitCode
AND U.UnitTitle = 'Systems Analysis'
AND C.Mark > 60
UNION
SELECT S.StudentNo, StdFirstName, StdLastName
FROM Student S, ClassList C, Unit U
WHERE S.StudentNo = C.StudentNo
AND U.UnitCode = C.UnitCode
AND U.UnitTitle = 'Databases'
AND C.Mark > 75;
```

STUDENT					
StudentNo	StdFirstName	StdLastName	StdCity	StdMajor	StdGPA

UNIT	
UnitCode	UnitTitle

CLASSLIST		
StudentNo	**UnitCode**	Mark

INTERSECT

Retrieves rows that are in A **AND** B

"Retrieve students who have achieved a grade higher than 65 in **BOTH** Systems Analysis **AND** Databases"

INTERSECT

```
SELECT S.StudentNo, StdFirstName, StdLastName
FROM Student S, ClassList C, Unit U
WHERE S.StudentNo = C.StudentNo
AND U.UnitCode = C.UnitCode
AND U.UnitTitle = 'Systems Analysis'
AND C.Mark > 65
INTERSECT
SELECT S.StudentNo, StdFirstName, StdLastName
FROM Student S, ClassList C, Unit U
WHERE S.StudentNo = C.StudentNo
AND U.UnitCode = C.UnitCode
AND U.UnitTitle ='Databases'
AND C.Mark > 65;
```

STUDENT					
StudentNo	StdFirstName	StdLastName	StdCity	StdMajor	StdGPA

UNIT	
UnitCode	UnitTitle

CLASSLIST		
StudentNo	**UnitCode**	Mark

MINUS (Difference)

Retrieves the rows in A **but not** B

"List the students who have been enrolled in ICT284 but not ICT285"

```
SELECT StudentNo
  FROM ClassList
  WHERE UnitCode = ICT284
MINUS
SELECT StudentNo
  FROM ClassList
  WHERE UnitCode = ICT285;
```

CLASSLIST		
StudentNo	**UnitCode**	Mark

DIVISION

- The relational algebra **division** operator **is not** directly implemented in SQL
- However it can be implemented in several ways in SQL:
  - Using nested NOT EXISTS
  - Using COUNT

Division: Nested NOT EXISTS

- "Find the students that have had enrolments in **all** ICT units"

- Using nested NOT EXISTS, this becomes:

  - "Find the students such that there **does not exist** an ICT unit in which they are **not enrolled**"

Division: Nested NOT EXISTS Example

```
SELECT StudentNo
FROM STUDENT S
WHERE NOT EXISTS
    (SELECT *
    FROM OFFERING O
    WHERE UnitCode LIKE 'ICT%'
    AND NOT EXISTS
        (SELECT *
        FROM CLASSLIST C
        WHERE C.StudentNo = S.StudentNo
                    AND C.UnitCode = O.UnitCode));
```

"Find the students that have enrolments in **all** ICT units"

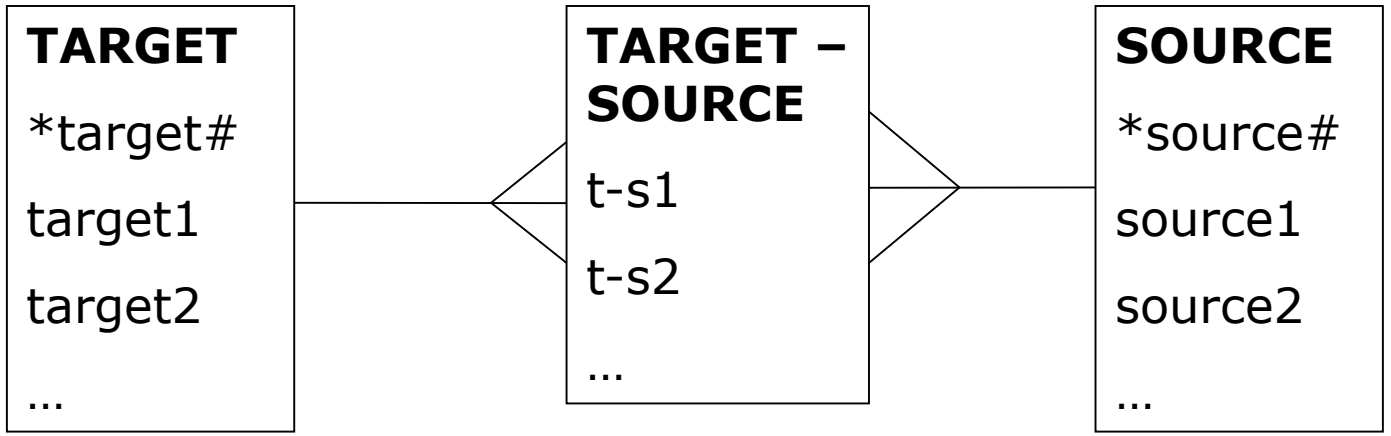
STUDENT					
StudentNo	StdFirstName	StdLastName	StdCity	StdMajor	StdGPA

CLASSLIST		
StudentNo	**UnitCode**	Mark

OFFERING		
UnitCode	**Semester**	**Year**

Division template

Watson (2002) has a division template you may find useful:



"Find the target that has appeared in all sources"

(next slide)

Division template SQL

The SQL would be:

```
SELECT Target1 FROM TARGET
WHERE NOT EXISTS
    (SELECT * FROM SOURCE
    WHERE NOT EXISTS
        (SELECT * FROM TARGET-SOURCE
        WHERE TARGET-SOURCE.Target# =TARGET.Target#
        AND TARGET-SOURCE.Source# =SOURCE.Source#))
```

Division using COUNT

"Find the students that have enrolments in all IT units"

- Using COUNT, this becomes:

Find the students for which the number of **distinct** ICT units in which they have been enrolled is equal to the number of **distinct** ICT units that there are

DIVISION using COUNT: Example

```
SELECT Student.StudentNo
FROM Student, ClassList
WHERE Student.StudentNo = ClassList.StudentNo
AND UnitCode IN
    (SELECT UnitCode
    FROM Unit
    WHERE UnitCode LIKE 'ICT%')
  GROUP BY Student.StudentNo
  HAVING COUNT (Student.StudentNo) =
                (SELECT COUNT(UnitCode)
                FROM Unit
                WHERE UnitCode LIKE 'ICT%');
```

This part counts the number of IT units in which each student is enrolled

This part counts the total IT units

CLASSLIST		
StudentNo	**UnitCode**	Mark

UNIT	
UnitCode	UnitTitle

STUDENT					
StudentNo	StdFirstName	StdLastName	StdCity	StdMajor	StdGPA

SQL for data manipulation: INSERT, DELETE, UPDATE

INSERT

INSERT adds a single row (record) to the table

```
INSERT INTO tablename [(columnlist)]
VALUES (dataValueList);
```

```
INSERT INTO ARTIST
VALUES (1, 'Miro','Joan','Spanish',1893, 1983);
```

INSERT rows from other tables

INSERT may be used to add several rows as the result of a SELECT from another table(s):

- INSERT INTO TableName [(columnList)]
  SELECT ….
  FROM ……
  WHERE ….

UPDATE

UPDATE is used to modify the attribute values in selected rows.

- Rows are selected using WHERE:

```
UPDATE ARTIST
SET DateOfDeath = 1984
WHERE ArtistID = 1;
```

If no where clause specified **all** rows are updated:

```
UPDATE TRANS
SET AskingPrice = 1.1*AskingPrice;
```

DELETE

Rows defined by a **WHERE** clause are deleted:

```
DELETE FROM ARTIST
    WHERE LastName = 'Tobey';
```

- If no WHERE clause is specified all rows are deleted:

```
DELETE FROM ARTIST;
```

(Note that you can't delete *columns*, only rows – deleting a column is altering the structure of a table)

Topic 03: Part 03

SQL – Data Definition (DDL)

SQL for data definition: CREATE, ALTER, DROP

SQL for data definition

This involves creating, altering and dropping objects to do with the table *structure* rather than its contents

- Tables, views
- Constraints, indexes

CREATE TABLE

Basic syntax of CREATE TABLE is…

```
CREATE TABLE TableName
{(colName dataType [NOT NULL] [UNIQUE]
[DEFAULT defaultOption]
[CHECK searchCondition] [,...]}
[PRIMARY KEY (listOfColumns),]
{[UNIQUE (listOfColumns),] […,]}
{[FOREIGN KEY (listOfFKColumns)
  REFERENCES ParentTableName [(listOfPKColumns)],
  [ON UPDATE referentialAction]
  [ON DELETE referentialAction ]] [,…]}
 {[CHECK (searchCondition)] [,…] })
```

CREATE TABLE: Example

```
CREATE TABLE Customer
      (CustNo  CHAR(8) CONSTRAINT CustPK PRIMARY KEY,
      CustFirstName VARCHAR2(20) CONSTRAINT FNameNN NOT
      NULL,
      CustLastName VARCHAR2(30) NOT NULL,
      CustStreet VARCHAR2(50),
      CustCity    VARCHAR2(30),
      CustState  CHAR(2),
      CustZip     CHAR(10),
      CustBal     DECIMAL(12,2) DEFAULT 0);
```

CREATE TABLE AS …

You can create a table from an existing table(s) using a query to define which rows and columns to use:

```
CREATE TABLE CSStudents AS
SELECT *
FROM STUDENT
WHERE StdMajor = 'CS';
```

- The new table inherits the data type and size of the original table's columns, **but not any constraints**.
- The data in the new table is **NOT** updated when the original is

STUDENT					
StudentNo	StdFirstName	StdLastName	StdCity	StdMajor	StdGPA

CREATE VIEW…

Views are 'virtual' tables created as subsets of base table by using SELECT

CREATE VIEW is very similar to the 'CREATE TABLE AS' statement:

```
CREATE VIEW CSStudents AS
SELECT *
FROM STUDENT
WHERE StdMajor = 'CS';
```

Once the view is created, it can be queried and (sometimes) updated the same as a base table

Views are **dynamic** – changes to the base table(s) are reflected in the view

Constraints

- SQL permits several constraints to be implemented:
  - Required (NOT NULL)
  - Unique
  - Entity integrity (primary key)
  - Referential integrity (foreign key)
  - Check constraints (domain)
- Constraints can be specified when the table is created, or afterwards using CREATE CONSTRAINT

Required data constraint

Can be implemented in SQL table definition using **NOT NULL** (i.e., NULL is not allowed in this column):

```
CREATE TABLE Customer
     (CustNo  CHAR(8) CONSTRAINT CustPK PRIMARY KEY ,
     CustFirstName VARCHAR2(20) CONSTRAINT FNameNN
     NOT NULL,
     CustLastName VARCHAR2(30) NOT NULL,
     CustStreet VARCHAR2(50),
     CustCity  VARCHAR2(30),
     CustState  CHAR(2),
     CustZip    CHAR(10),
     CustBal    DECIMAL(12,2) DEFAULT 0);
```

Uniqueness constraint

Can specify a particular column (or combination of columns) will only allow unique values:

```
CREATE TABLE Customer
        (CustNo  CHAR(8) CONSTRAINT CustPK PRIMARY KEY ,
        CustFirstName VARCHAR2(20) CONSTRAINT FNameNN NOT NULL,
        CustLastName VARCHAR2(30) NOT NULL,
        CustEmail VARCHAR2(25) CONSTRAINT
        UniqueEmail UNIQUE,
        CustStreet VARCHAR2(50),
        CustCity  VARCHAR2(30),
        CustState  CHAR(2),
        CustZip     CHAR(10),
        CustBal     DECIMAL(12,2) DEFAULT 0);
```

Entity integrity constraint

The entity integrity constraint says that primary key values are unique and cannot be null. If you define a **primary key constraint** both of these are automatically enforced

```
CREATE TABLE Customer
        (CustNo  CHAR(8) CONSTRAINT CustPK PRIMARY KEY,
        CustFirstName VARCHAR2(20) CONSTRAINT FNameNN NOT
        NULL,
        CustLastName VARCHAR2(30) NOT NULL
....
CREATE TABLE Customer_Artist_Int
        (CustNo CHAR(8),
        ArtistID CHAR(8),
        CONSTRAINT CustArtistPK PRIMARY KEY (CustNo,
        ArtistID));
```

Referential integrity constraint

Implemented by defining a **foreign key:**

```
CREATE TABLE Work
                    (WorkID NUMBER(4),
                    Title VARCHAR2(35) NOT NULL,
                    Copy VARCHAR2(12) NOT NULL,
                    Medium VARCHAR2(35),
                    Description VARCHAR2(1000),
                    ArtistID  NUMBER(4) NOT NULL,
                    CONSTRAINT WorkPK PRIMARY KEY(WorkID),
                    CONSTRAINT WorkAK1 UNIQUE (Title, Copy),
                    CONSTRAINT ArtistFK FOREIGN KEY (ArtistID)
                              REFERENCES ARTIST(ArtistID));
```

Preserving referential integrity - Referential actions

- SQL allows you to specify actions for UPDATE and DELETE in the foreign key constraint definition – i.e. what should happen to the CHILD record (the one with the FK) if the parent is changed
- The decision depends on the meaning of the data – e.g. sometimes it may be appropriate to delete child records (*cascade* the delete); sometimes it is preferable to *prevent* the delete
- We will look in more detail at this when considering logical database design

Preserving referential integrity - Referential actions

- INSERTs that would violate referential integrity are always disallowed
- Options for UPDATE and DELETE are:

  NO ACTION: prevent deletion if referential integrity would be violated
  SET NULL: set foreign key value to null
  SET DEFAULT: set to a default value defined for that field
  CASCADE: cascade the update/delete to the child table

- **NO ACTION** is the default if no other action is specified

Referential Actions in Oracle SQL

```
CREATE TABLE WORK
                    (WorkID NUMBER(4),
                    Title VARCHAR2(35) NOT NULL,
                    Copy VARCHAR2(12) NOT NULL,
                    Medium VARCHAR2(35),
                    Description VARCHAR2(1000),
                    ArtistID  NUMBER(4) NOT NULL,
                    CONSTRAINT WorkPK PRIMARY KEY(WorkID),
                    CONSTRAINT WorkAK1 UNIQUE (Title, Copy),
                    CONSTRAINT ArtistFK FOREIGN KEY (ArtistID)
                              REFERENCES ARTIST(ArtistID)
                              ON DELETE CASCADE);
```

Referential Actions in Oracle SQL

Note that Oracle *defaults* to ON DELETE NO ACTION but does NOT include it in the statement ***– this will give an error***

- Oracle DOES allows you to specify ON DELETE CASCADE

CHECK constraint

Restrict the values to a set of allowable values as defined in an expression

```
ALTER TABLE ARTIST
ADD CONSTRAINT BirthValuesCheck CHECK (DateOfBirth < DateDeceased);
```

(This example adds the constraint after the table was created, using ALTER TABLE, but could also have been done in the CREATE TABLE statement)

DROP TABLE

DROP TABLE TABLENAME [RESTRICT | CASCADE]

e.g. DROP TABLE CUSTOMER;

- Removes named table and all rows within it
- With RESTRICT, if any other objects depend for their existence on continued existence of this table, SQL does not allow request
- With CASCADE, SQL drops all dependent objects (and objects dependent on these objects)

ALTER TABLE

Modifies the table structure after it has been created

- Add/drop columns
- Modify column definitions
- Add/drop constraints

Example

```
CREATE TABLE Customer
      (CustNo  CHAR(8) CONSTRAINT CustPK PRIMARY KEY ,
      CustFirstName VARCHAR2(20) CONSTRAINT FNameNN
      NOT NULL,
      CustLastName VARCHAR2(30) NOT NULL,
      CustStreet VARCHAR2(50),
      CustCity  VARCHAR2(30),
      CustState  CHAR(2),
      CustZip     CHAR(10),
      CustBal     DECIMAL(12,2) DEFAULT 0);
```

e.g. drop column CustBal:

ALTER TABLE CUSTOMER DROP COLUMN CustBal;

e.g. Add a NOT NULL constraint for CustCity:

ALTER TABLE CUSTOMER MODIFY CustCity NOT NULL;

Useful summary of ALTER:
http://www.tutorialspoint.com/sql/sql-alter-command.htm

CREATE, DROP INDEX

CREATE INDEX indexname ON tablename(columnname);

e.g. CREATE INDEX idxCustName ON CUSTOMER(CustName);

DROP INDEX idxCustName;

Topic 03: Part 04

SQL – Other Features

Other SQL features

Data control
Transaction management
Embedding SQL in program code

GRANT, REVOKE

Grant and revoke privileges on tables (or other database objects to users or roles)

GRANT [SELECT, INSERT, UPDATE, DELETE, ALL] ON TABLE TO [PUBLIC | role| userID];

e.g.

GRANT SELECT, UPDATE ON CUSTOMER TO PUBLIC;

GRANT UPDATE ON CUSTOMER TO 12345;

REVOKE UPDATE ON CUSTOMER FROM 12345;

SQL for transactions

- A transaction is a ‘single logical unit of work’ that may consist of several SQL statements. Transactions must be completed in full or not at all
- In Oracle a transaction begins implicitly at the first SQL statement and continues until either **COMMIT** or **ROLLBACK** is encountered.

  e.g. Deleting a student and the details of their enrolments should be a single transaction:

```
DELETE FROM ENROLMENT WHERE StudentID = ‘123’;
DELETE FROM STUDENT WHERE StudentID = ‘123’;
COMMIT;
```

SQL/Persistent Stored Modules

- Each DBMS product has its own variations on SQL, including features that allow it to function similarly to a procedural programming language
- The ANSI/ISO standard calls these SQL/PSM, SQL Persistent Stored Modules
- The Oracle SQL version is **PL/SQL** (Procedural Language/SQL) and permits the creation of:
  - **User-defined functions**
  - **Triggers**
  - **Stored procedures**

CREATE TRIGGER

Create a trigger that will record the username of any user who makes changes to the TRANS table and provide a history of those changes in another table

```
CREATE or REPLACE TRIGGER TransHistoryTrigger
 AFTER UPDATE
  ON Trans
 FOR EACH ROW

DECLARE
 Mod_User_Name Varchar2(20);

BEGIN

 SELECT user into Mod_User_Name
 FROM dual;

 INSERT INTO Trans_History (
  HistoryID,
  TransactionID,
  DATE_ACQUIRED_OLD,
  ACQUISITION_PRICE_OLD,
  DATE_SOLD_OLD,
  ASKING_PRICE_OLD,
  SALES_PRICE_OLD,
  MOD_USER_NAME,
  MOD_DATE)
 VALUES
  (seqTransHistID.NextVal,
  :New.TransactionID,
  :New.DateAcquired,
  :New.AcquisitionPrice,
  :New.DateSold,
  :New.AskingPrice,
  :New.SalesPrice,
  Mod_User_Name,
  sysdate);

END;
/
```

Topic 03: Part 05
Conclusion

Topic Learning Outcomes Revisited

After completing this topic you should be able to:

- Describe some of the major features of SQL and their usage
- Use SQL for querying and creating tables
  (NB: much of the practicalities of SQL are addressed in the labs)
- Know where in the unit we will cover the different aspects of SQL in more detail

What's next?

- In next week's lecture topic we return to the theory of the relational model by looking at normalisation. Normalisation is a process by which relational databases can be put in a well-designed form that provides maximum flexibility, minimum redundancy, and prevents modification anomalies.
- In Lab 4, we continue with SQL by covering creating, dropping and altering tables, inserting, updating and deleting records, and creating simple constraints on the data in a table.