Careful study of these notes is best left until most of the lectures on CS252 have been given. Many of these notes relate SQL to the theory section of CS252. In some cases they refer to things you will encounter in the theory section after this lecture has been given.
Interim Summary

Material covered so far:
Data description language:
- Creating tables with constraints.
- Oracle data types (apart from \texttt{DATE}).

- Constraints: \texttt{NOT NULL}, (candidate) keys, primary keys, foreign keys, \texttt{CHECK} predicates.

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

- Mathematical functions, for single values and groups of values.
- `GROUP BY` and `HAVING` for producing summaries.
- Storing dates and times in the database.
- Creating views with “`CREATE VIEW ... AS`”.
- Creating tables from tables with ”`CREATE TABLE ... AS`”.
Formulas in SQL

So far, only the exact values have been selected from database tables. It is possible to use formulas where simple literals or column references can appear (SELECT, WHERE, SET, VALUES used in INSERT, HAVING):

SELECT function (column name) FROM ...

Function can be an infix operator:

<table>
<thead>
<tr>
<th>Function</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>value1 + value2</td>
<td>Addition</td>
</tr>
<tr>
<td>value1 - value2</td>
<td>Subtraction</td>
</tr>
<tr>
<td>value1 * value2</td>
<td>Multiplication</td>
</tr>
<tr>
<td>value1 / value2</td>
<td>Division</td>
</tr>
</tbody>
</table>

In fact, the elements of a SELECT clause (i.e., the SELECT clause of a SELECT expression) are in general expressions of arbitrary complexity.

But it’s not just the SELECT clause where such expressions are permitted. It’s also, for example, the WHERE clause, the SET clause of UPDATE, the VALUES expression (though column references are not applicable here), and the HAVING clause.

As is normal in computer languages, wherever a simple literal or column reference is permitted to appear, an expression of arbitrary complexity is also allowed to appear. Such expressions typically invoke functions, and the arguments to such invocations can reference columns by name. The following examples illustrate the use of the usual infix operators for doing arithmetic.

Note: By the way, the example illustrates the bad practice of “selecting” a column that has no name.
Imagine for insurance purposes, the value of CDs from table Quantity is calculated:

```sql
SELECT quantity, quantity * 10.99 AS "value"
FROM Quantity;
```

Or:

```sql
SELECT quantity, quantity * 10.99 AS value
FROM Quantity;
```

Please note: SQL allows double quotes around identifiers but requires them only when the identifier includes special characters or spaces that might otherwise lead to ambiguity.

Here we illustrate the use of expressions in the SELECT clause. The result of this query is a table with two columns, named quantity and value. The first column inherits its name from the expression defining it for the obvious reason that the expression is just a simple column reference. The second column is given the name value by the AS clause immediately following the expression that defines it.

Note the use of double-quotes around the column name value. SQL permits these but does not always require them, and in fact they could safely be omitted here, just as they are omitted around quantity. Double-quotes are required for names in general (not just column names) whenever the name includes spaces or special characters such as punctuation or arithmetic operators.

Because DISTINCT is omitted there is no Tutorial D counterpart of this example (the result is not in general a relation), but if DISTINCT is added we would have

```sql
(EXTEND Quantity ADD ( quantity * 10.99 AS value ))
{ quantity, value }
```

We first use EXTEND to produce a relation with the required additional attribute, then we project the result over the attributes we require in the final result.

It might help you to consider that the first element of the SELECT clause is really short for quantity AS quantity

which you can read as meaning “take the values of the quantity column in the input table (in this case the result of FROM Quantity) and name the resulting column quantity.”

SQL does not require every column in the result of a SELECT expression to be named. For example, we could have reduced the example on this slide to just

```sql
SELECT quantity, quantity * 10.99
FROM Quantity;
```

In that case the second column would have no name at all. While this presents little or no problem in ad hoc queries, it is not good practice in general and you are encouraged to get into the habit of always using AS for expressions other than simple column references in a SELECT clause.
Subqueries in the FROM Clause

Notice the repetition here:

```sql
SELECT quantity, quantity * 10.99 AS value
FROM Quantity
WHERE (quantity * 10.99) > 50;
```

Tempting, but **wrong**, is:

```sql
SELECT quantity, quantity * 10.99 AS value
FROM Quantity
WHERE value > 50;
```

because the column value exists in the table resulting from the
`
SELECT` clause, not the `FROM` clause. Correct is:

```sql
SELECT *
FROM ( SELECT quantity, quantity*10.99 AS value 
          FROM Quantity )
WHERE value > 50;
```

You have learned subqueries before in the WHERE clause. It might not surprise you much to find out that you can also use subqueries in the FROM clause. However, there’s a reason for teaching you this separately:

Original SQL did not allow subqueries to be written in the FROM clause. As a consequence SQL was relationally incomplete — certain queries that are expressible in relational theory could not be expressed in SQL. This was a very serious defect that IBM’s System/R team in the 1970s had failed to notice, though they eventually rectified matters in the 1980s and their correction found its way into IBM’s commercial product (DB2) and, in 1992, the ISO SQL standard.
### Function Definition

<table>
<thead>
<tr>
<th>Function</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS(value)</td>
<td>Absolute value</td>
</tr>
<tr>
<td>CEIL(value)</td>
<td>Smallest larger integer</td>
</tr>
<tr>
<td>COS(value)</td>
<td>cos value</td>
</tr>
<tr>
<td>EXP(value)</td>
<td>$e^{value}$</td>
</tr>
<tr>
<td>FLOOR(value)</td>
<td>Largest smaller integer</td>
</tr>
<tr>
<td>LN(value)</td>
<td>ln value</td>
</tr>
<tr>
<td>LOG(value)</td>
<td>$\log_{10} value$</td>
</tr>
<tr>
<td>POWER(x,y)</td>
<td>$x^y$</td>
</tr>
<tr>
<td>ROUND(v,p)</td>
<td>Round v precision p</td>
</tr>
<tr>
<td>SIN(value)</td>
<td>sin value</td>
</tr>
<tr>
<td>SQRT(value)</td>
<td>square root value</td>
</tr>
<tr>
<td>TAN(value)</td>
<td>tan value</td>
</tr>
<tr>
<td>TRUNC(v,p)</td>
<td>Truncate v precision p</td>
</tr>
</tbody>
</table>

**Some Additional Numeric Operators:**

The operators shown on this slide are all defined in the ISO SQL standard, except (curiously) for COS, SIN, TAN, ROUND, TRUNC, and LOG. Those nonstandard operators are found in many SQL implementations as well as Oracle. As you would expect, SQRT(value) returns the nonnegative root of value.
Formulas and NULLs

Any NULL in a formula will cause a NULL result, not a value of 0.

**DBS_marks**

<table>
<thead>
<tr>
<th>student</th>
<th>mark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paul Smith</td>
<td>43</td>
</tr>
<tr>
<td>Rachel Sewell</td>
<td>57</td>
</tr>
<tr>
<td>Helen Treacy</td>
<td>72</td>
</tr>
<tr>
<td>Pinder Surree</td>
<td>NULL</td>
</tr>
</tbody>
</table>

Table shows marked scripts for the DBS assignment (fictional!). Pinder has an extension and mark is therefore NULL.

In principle, whenever an operand is NULL, the result of the invocation is also NULL.

There are some exceptions to the “NULL-in, NULL-out” rule, but they mainly arise in aggregation, which we study later. This is where SQL lets you, for example, add up all the values in a single column. In such an operation, SQL just ignores the NULLs and adds up the remaining values, so the result is NULL only when all the inputs are NULL. However, the expression $x + y$ yields NULL whenever either $x$ is NULL or $y$ is NULL.

The example on this slide shows an appearance of NULL in a particular column, mark, of a particular row in a table. Note in passing that here NULL does not really mean that Pinder Surree’s mark is unknown, even though that was the purpose for which SQL’s NULL was devised. The fact that NULL can have all sorts of different meanings and yet is always treated the same way in SQL operations is among the many problems with NULL that its critics like to point out.

See also: [http://www.oracle.com/technology/oramag/oracle/05-jul/o45sql.html](http://www.oracle.com/technology/oramag/oracle/05-jul/o45sql.html)
Select statement and Oracle output:

```sql
SELECT student, mark, mark/3 AS "mark/3" FROM DBS_marks;
```

<table>
<thead>
<tr>
<th>STUDENT</th>
<th>MARK</th>
<th>MARK/3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paul Smith</td>
<td>43</td>
<td>14.33333</td>
</tr>
<tr>
<td>Rachel Sewell</td>
<td>57</td>
<td>19</td>
</tr>
<tr>
<td>Helen Treacy</td>
<td>72</td>
<td>24</td>
</tr>
<tr>
<td>Pinder Surree</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Giving Pinder an average mark to complete the results using NVL (or COALESCE):

```sql
SELECT student, mark, NVL(mark,50)/3 FROM DBS_marks;
```

<table>
<thead>
<tr>
<th>STUDENT</th>
<th>MARK</th>
<th>NVL(MARK,50)/3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paul Smith</td>
<td>43</td>
<td>14.3333333</td>
</tr>
<tr>
<td>Rachel Sewell</td>
<td>57</td>
<td>19</td>
</tr>
<tr>
<td>Helen Treacy</td>
<td>72</td>
<td>24</td>
</tr>
<tr>
<td>Pinder Surree</td>
<td></td>
<td>16.6666667</td>
</tr>
</tbody>
</table>

Please note: NVL is Oracle-proprietary and supports only two arguments; COALESCE is in the SQL standard and supports an arbitrarily long list of arguments.

In Oracle/PLSQL, the NVL function lets you substitute a value when a null value is encountered.

The syntax for the NVL function is:

```
NVL( string1, replace_with )
```

string1 is the string to test for a null value.
replace_with is the value returned if string1 is null.

See also COALESCE, the standard equivalent

In the first example, Pinder Surree’s mark and mark divided by 3 both appear as NULL (which sqlplus, perhaps unwisely, converts to white space in the displayed result). In the second example we use the SQL COALESCE operator to “convert” the NULL to something other than NULL—in this case 50.

COALESCE takes a list of two or more expressions and returns the result of the first of those expressions, in the order written, that does not evaluate to NULL. If they all evaluate to NULL, then NULL is the result.
Rounding the marks using ROUND:

```sql
SELECT student, mark,
    ROUND(COALESCE(mark,50)/3) AS "credit"
FROM DBS_marks;
```

Examples of rounding and truncation:

- \( \text{ROUND}(55.5) = 56 \)
- \( \text{ROUND}(23.453,2) = 23.45 \)
- \( \text{TRUNC}(55.5) = 55 \)
- \( \text{TRUNC}(23.453,2) = 23.45 \)
- \( \text{CEIL}(55.5) = 56 \)
- \( \text{TRUNC}(23.453,2) = 23.45 \)
- \( \text{FLOOR}(55.5) = 55 \)
- \( \text{TRUNC}(23.453,2) = 23.4 \)
Aggregate Functions

Operate on a group of values or a column.
To select the average mark from DBS_marks:

```
SELECT AVG(mark) AS Avg_mark FROM DBS_marks;
```

```
Avg_mark
--------
57.3333333
1 row selected.
```

The operand of an aggregate function is any expression (of the required data type) that could appear as part of the condition of a WHERE clause. Such expressions usually reference at least one column, as in the simple example shown on the slide. The operator aggregates over all the rows of its input table, which in this case is all the rows of DBS_marks. The operand expression is evaluated for each row and the result, if not NULL, is included in the aggregation (in this case an average).
### Function Definition

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVG(value)</td>
<td>Average of column</td>
</tr>
<tr>
<td>SUM(value)</td>
<td>Sum P column</td>
</tr>
<tr>
<td>COUNT(value)</td>
<td>Count rows of column</td>
</tr>
<tr>
<td>MAX(value)</td>
<td>Maximum out of column</td>
</tr>
<tr>
<td>MIN(value)</td>
<td>Minimum out of column</td>
</tr>
<tr>
<td>STDDEV(value)</td>
<td>Standard deviation</td>
</tr>
<tr>
<td>VARIANCE(value)</td>
<td>Variance</td>
</tr>
</tbody>
</table>

NULLs are ignored.

The treatment of NULL in these aggregate functions needs careful consideration. In the absence of NULL, the expressions $\text{SUM}(x) + \text{SUM}(y)$ and $\text{SUM}(x+y)$ would be equivalent. Remembering that $x+y$ evaluates to NULL if *either* of $x$ and $y$ is NULL, you can easily see why this equivalence does not obtain in SQL.
GROUP BY

Suppose we wish to obtain, from the following table, a table showing the total marks for each student:

**CS_marks**

<table>
<thead>
<tr>
<th>student</th>
<th>course</th>
<th>mark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paul Smith</td>
<td>DBS</td>
<td>43</td>
</tr>
<tr>
<td>Rachel Sewell</td>
<td>ISE</td>
<td>57</td>
</tr>
<tr>
<td>Helen Treacy</td>
<td>DBS</td>
<td>72</td>
</tr>
<tr>
<td>Paul Smith</td>
<td>ISE</td>
<td>65</td>
</tr>
<tr>
<td>Rachel Sewell</td>
<td>DBS</td>
<td>42</td>
</tr>
</tbody>
</table>
SELECT student, SUM(mark) AS Total_marks
FROM CS_marks GROUP BY student;

<table>
<thead>
<tr>
<th>STUDENT</th>
<th>Total_marks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Helen Treacy</td>
<td>72</td>
</tr>
<tr>
<td>Paul Smith</td>
<td>108</td>
</tr>
<tr>
<td>Rachel Sewell</td>
<td>99</td>
</tr>
</tbody>
</table>

The GROUP BY clause effectively partitions the input table into separate tables according to the values of the columns (in the present example just one column) specified in the GROUP BY clause. These separate tables are referred to as groups. Invocations of aggregate functions in the SELECT clause now operate on each group. We can now see that our earlier example (Slide 9) is a degenerate case in which the whole input table is taken as a single group.

It is possible to specify any number of columns in the GROUP BY clause. Thus SQL queries using GROUP BY and aggregate functions are SQL’s counterpart of Tutorial D’s SUMMARIZE operator. The example here could be expressed in Tutorial D thus:

```
SUMMARIZE CS_marks BY { Student }
    ADD ( SUM(mark) AS Total_marks )
```

The braces used for the BY operand indicate that the order of elements, when there is more than one, is insignificant. The order is also insignificant in the SQL GROUP BY clause.

SUMMARIZE requires a BY (or PER) operand, so the example on Slide 9 would be

```
SUMMARIZE DBS_marks BY {}
    ADD ( Avg(mark) AS Avg_mark )
```

As it happens, standard SQL also allows “grouping by no columns” to be written explicitly, as GROUP BY (). In this single case the parentheses around the GROUP BY operand are not optional.
Having

Used to eliminate groups we don’t want:

```sql
SELECT student, SUM(mark)
FROM CS_marks
GROUP BY student
HAVING SUM(mark) > 100;
```

<table>
<thead>
<tr>
<th>STUDENT</th>
<th>SUM(MARK)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paul Smith</td>
<td>108</td>
</tr>
</tbody>
</table>

Elimination occurs after the grouping has taken place.
Different to WHERE - it works on the groups formed with GROUP BY instead of rows.

You might be wondering why HAVING is needed at all in SQL. For example, why can’t we write WHERE SUM(mark) > 100 in the example on the slide? The answer is in two parts. First, recall that the GROUP BY clause is optional. We can leave it out when we want to aggregate over the whole of the input table. But the input table might be the output of a WHERE clause, as here:

```sql
SELECT SUM(mark) AS Total_mark
FROM CS_Marks
WHERE course IN ('DBS', 'CBS');
```

WHERE operates on the result of a FROM clause, whereas HAVING operates on the result of a GROUP BY clause, possibly the implied GROUP BY ()

In that case, you might ask, why not write this, in place of the example shown on the slide:

```sql
SELECT *
FROM ( SELECT student, SUM(mark) AS Total_Marks
      FROM CS_marks
      GROUP BY student )
WHERE Total_marks > 100;
```

And the answer to that is that when HAVING was invented (by IBM’s System/R team back in the 1970s) SQL did not allow subqueries to be written in the FROM clause!
Dates and Times

Oracle supports DATE arithmetic:
- Day, month, year.
- Hour, minute, second.

Default date format is “DD-MON-YY”. Use DATE as the data type in CREATE TABLE statement.

To record the current date and time:

```
INSERT INTO Table VALUES ( SYSDATE );
```

Adding 1 to a date results in the next day, so SYSDATE + 1 is tomorrow.

SYSDATE is Oracle’s nonstandard name for a certain built-in system variable. The ISO SQL standard’s name for this variable is CURRENT_DATE. The SQL standard defines several system variables. Another is CURRENT_USER, which holds the userid (officially called an authorization identifier) of the current session (i.e., your own userid when you are working with sqlplus).
Some date functions:

<table>
<thead>
<tr>
<th>Function Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD_MONTHS(date, n)</td>
</tr>
<tr>
<td>LAST_DAY(date)</td>
</tr>
<tr>
<td>NEXT_DAY(date, day)</td>
</tr>
</tbody>
</table>

(day is 'Monday', etc.)

Converting to/from Dates

Need to convert to Oracle date format on insertion and from the internal format to the date format on selection.

TO_CHAR(date, format) Covert an internal date to a string.

TO_DATE(string, format) Covert a string into an internal date.

TO_DATE and TO_CHAR are non-standard proprietary extensions to SQL in Oracle. Standard SQL provides a highly generalized function, CAST, which supports conversion of a value of any data type to a corresponding value of another type. For example, CAST (‘1’ AS INTEGER) returns the integer 1 and CAST (‘x’ AS INTEGER) results in a run-time exception because there is no integer that is denoted by the character string ‘x’.

The standard CAST is a very complicated operator whose details are not really relevant to the basic understanding of SQL we try to teach on CS252, so we stick with Oracle’s easier, non-standard TO_DATE and TO_CHAR.
CREATE TABLE Birthdays (name VARCHAR(50), birthday DATE);

INSERT INTO Birthdays VALUES ('Helen Treacy',
    TO_DATE('10 September 1976','DD MONTH YYYY'));

SELECT name, TO_CHAR(birthday, 'DD/MM/YY') FROM Birthdays;

<table>
<thead>
<tr>
<th>NAME</th>
<th>TO_CHAR(</th>
</tr>
</thead>
<tbody>
<tr>
<td>Helen Treacy</td>
<td>10/09/76</td>
</tr>
<tr>
<td>Code</td>
<td>Definition</td>
</tr>
<tr>
<td>------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>MM</td>
<td>Number in month</td>
</tr>
<tr>
<td>MON</td>
<td>3 letter abbrev. month</td>
</tr>
<tr>
<td>MONTH</td>
<td>Fully specified month</td>
</tr>
<tr>
<td>DD</td>
<td>Days through month</td>
</tr>
<tr>
<td>DDD</td>
<td>Days in year since Jan. 1</td>
</tr>
<tr>
<td>D</td>
<td>Day through week</td>
</tr>
<tr>
<td>DY</td>
<td>3 letter abbrev. day</td>
</tr>
<tr>
<td>DAY</td>
<td>Fully specified day</td>
</tr>
<tr>
<td>YYYY</td>
<td>4 digit year</td>
</tr>
<tr>
<td>YY</td>
<td>Last 2 digits of year</td>
</tr>
<tr>
<td>AD</td>
<td>Displays AD or BC</td>
</tr>
<tr>
<td>HH12</td>
<td>Hours of day (1-12)</td>
</tr>
<tr>
<td>HH24</td>
<td>Hours of day, 24hr</td>
</tr>
<tr>
<td>MI</td>
<td>Minute of hour</td>
</tr>
<tr>
<td>SS</td>
<td>Second of minute</td>
</tr>
<tr>
<td>AM</td>
<td>Displays AM or PM</td>
</tr>
</tbody>
</table>
Dates Comparisons

You sometimes want to do comparisons on dates, in WHERE clauses and other places. Some examples:

- Testing if two dates are different:
  
  \[
  \text{... WHERE date1 - date2 != 0}
  \]

- Testing for all dates prior to a particular date:
  
  \[
  \text{... WHERE date1 <= date2}
  \]

- Testing for all dates between:
  
  \[
  \text{... WHERE date1 BETWEEN '01-JAN-98' AND '02-NOV-98'}
  \]

Note that the default date format can be used in some comparisons.

For more complex queries, use TO_DATE.

The first example could easily be written as date1 != date2, but we take the opportunity to show you Oracle’s support for date arithmetic. The subtraction of one date from another yields an integer denoting a number of days. This is nonstandard SQL. The ISO SQL standard defines a set of INTERVAL data types of various precisions and scales, and date subtraction yields a value of type INTERVAL DAY. For example subtraction of yesterday’s date from today’s yields the value denoted by the literal INTERVAL 1 DAY. And CAST (INTERVAL 1 DAY TO INTEGER) yields the integer 1.

“Interval” is not the word normally used in mathematics for this SQL construct. In mathematics an interval is something that has a lower bound and an upper bound, such as the interval [1:10] (also written as [1..10], denoting the integers from 1 to 10 inclusive. Standard SQL is using the key word INTERVAL for what the mathematician would call the duration of an interval. The duration of [1:10] is 10, as is the duration of [11:20].
Views

Views: naming the result of a query.

up-to-date table

virtual tables, no additional data storage space

interacted with as tables, including DELETE, UPDATE and INSERT! - subject to restrictions.

make normalised data more intelligible to users. A DBA will often assign a particular view of data to individual users. E.g., a join that is expected to be commonly used in queries.

Views are a way of naming the result of a query. Furthermore, a view continues to exist as an up-to-date table consistent with the specified query.

Views are virtual tables that take up no additional data storage space to the (base) tables from which they are constructed. They can be interacted with as if they were tables in their own right, including DELETE, UPDATE and INSERT! - subject to restrictions - there are limits on what can be done via a view.

Views make normalised data more intelligible to users. A DBA will often assign a particular view of data to individual users. A view, for example, can be defined on a join that is expected to be commonly used in queries.

The view is SQL’s counterpart of Tutorial D’s construct called the virtual relvar.
Creating Views

A view is created with the "CREATE VIEW ... AS" statement:

```
CREATE VIEW Pop_years
AS SELECT artist, album, year
FROM Pop_albums, CD_year
WHERE Pop_albums.barcode = CD_year.barcode;
```

View Pop_years appears to be a table in its own right.

```
SQL> SELECT * FROM Pop_years WHERE year > 1996;

+-------------+---------+--------+
| ARTIST      | ALBUM   | YEAR   |
|-------------+---------+--------|
| The Verve   | Urban Hymns | 1997   |
| Foo Fighters| The Colour and the Shape | 1997   |
```

The following UPDATE statement will change table Pop albums through the view Pop_years:

```
UPDATE Pop_years
SET album = 'Urban Hymns'
WHERE artist = 'The Verve';
```
Rules for Views

Restrictions on working with views include:

1. It is not possible to **INSERT** into a view if any of the underlying base tables have **NOT NULL** columns that do not appear in the view.

2. It is not possible to **INSERT or UPDATE** a view if one of its columns is defined, directly or indirectly, by an expression that is other than a simple column reference.

3. It is not possible to **INSERT or UPDATE** a view if **SELECT DISTINCT** is used to define one of its columns.

4. A **FROM** clause in a view definition can reference another view.

Note that a view can contain functions and calculations — e.g., to produce a report.

Views can be dropped, like tables, with the **DROP VIEW** statement. No data is discarded, only the definition of the view.

Note that a view can be defined on **any** **SELECT** expression — useful for defining the input to a regularly produced report, for example.

Rules for updating views are not defined in **Tutorial D**. The topic is widely discussed in relational theory and SQL’s rules are criticized as being **ad hoc** and too restrictive. Because the topic is complicated and controversial we regard it as beyond the scope of a foundation module such as CS252.
Creating Tables from Tables

It is possible to create new tables using queries with the statement
“CREATE TABLE ... AS”.
New tables are created as a snapshot of the specified query.

```
CREATE TABLE Pop_company
AS SELECT artist, album, company
FROM Pop_albums, CD_company
WHERE Pop_albums.barcode = CD_company.barcode;
```

The new table `Pop_company` inherits the names and data types of its columns from the existing tables.

The **Tutorial D** counterpart of this CREATE TABLE statement would be

```tutorial_d
VAR Pop_company BASE INIT ( ( Pop_albums JOIN CD_company )
{ artist, album, year } );
```

(but again the SELECT DISTINCT observation — see the notes for Slide 24 — applies). In **Tutorial D** every variable declaration can specify an initial value for that variable, using INIT. The declared type of the variable is implicitly that of the expression used to denote the initial value.
The following `UPDATE` statement will not then affect the `Pop_albums` table, only the table `Pop_company`.

```sql
UPDATE Pop_company
SET album = 'Urban Hims'
WHERE artist = 'The Verve';
```

This happens because creating new (base) tables based on other (base) tables means copying the data across and working independently with it; unlike with views, which basically represent pointers to the information in the base tables, and where the updating rules allow it, work directly on the base tables.
• http://www.dcs.warwick.ac.uk/~acristea/
• Select: courses
• Select CS252
Interim Summary

Topics covered:
• Mathematical functions, for single values and groups of values.
• `GROUP BY` and `HAVING` for producing summaries.
• Storing dates and times in the database.
• Creating views with “CREATE VIEW ... AS”.
• Creating tables from tables with ”CREATE TABLE ... AS”.