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

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".
This lecture

• This session: Indexes, Sequences, Relational Algebra and Advanced Joins
Indexes

Indexes are redundant data structures that improve the performance of a database management system. An index makes it faster for a database to look up a row based on a value within that row. An index has a storage overhead. Updates on a table that has an index can take longer than one that hasn’t. It is best to only put indexes on columns that are often used for a search. They are not required for correctness, they do not affect the result returned by a query.

An index is automatically added to any column declared as a Primary Key.

Database implementations have their own specific indexing schemes and in some you can specify exactly what kind of index to use.

Please note what the last sentence means: that your database allows direct access to internal storage structures! This contradicts principles of relational algebra, especially the separation of physical implementation from logical one – but these are not issues to be concern with in this module.

An index makes a physical connection between column values that are stored in a base table and the rows of that table in which they appear. Internally, each stored row is identified by some kind of pointer, generated by the DBMS when that row is stored.

An index for a base table is defined on one or more columns. Take the simple case where it is defined on just one column $c$. Then the index provides a mapping of each value $v$ appearing anywhere in $c$ to the row(s) in which $v$ appears. The index is organized in such a way as to provide for a very fast method of looking up a particular value. For example, the values might be sorted into numerical or alphabetical order, enabling a particular value to be looked up by a "binary search" technique. In practice more advanced techniques are often used, such as hashing or B-trees (which you might be aware of though they are beyond the scope of CS252).
Creating an index

not part of SQL standard but nearly all DBMS support a common syntax.

An index can be created on a single column:

```
CREATE INDEX year_index ON CD_Year (year);
```

This will improve performance on queries that look up 'year'. e.g.:

```
SELECT barcode FROM CD_Year WHERE year = 1994;
```

Creating indexes is not part of the SQL standard but nearly all databases support a common syntax.

The reason why not is explained in the previous slide comments.

If you create an index on a valid table and column name, Oracle responds:

```
Index created.
```

Even more importantly, the index can speed up joins dramatically. For example, in the following query to discover pairs of CDs published in the same year:

```
SELECT CD1.barcode AS bc1, CD2.barcode AS bc2
FROM "CD_year" AS CD1, "CD_year" AS CD2
WHERE CD1.year = CD2.year AND CD1.barcode < CD2.barcode
```
Creating a multiple-key index

An index can also be created on a multiple columns:

CREATE INDEX index ON Pop_albums (artist, album);

This will improve performance on queries that look up both artist and album, e.g.:

SELECT barcode FROM Pop_albums WHERE artist='U2' AND album='Rattle and Hum';
Sequences

Some DBMS support an 'Identity' column type, automatically populated with a unique reference number (URN).

- useful when creating surrogate keys (a key not derived data & acts as a primary key).
- MS Access, MS SQL Server: identity + GUID (Globally Unique IDentifier) column type.
- MySQL: AUTO_INCREMENT column type.
- Oracle has Sequences.

Some databases management systems support an 'Identity' column type that is automatically populated with a unique reference number (URN).

This kind of column is useful when creating surrogate keys (a key that is not derived from any data in the database and whose only purpose is to act as a primary key).

For example MS Access and MS SQL Server both have an Identity data type and a GUID (Globally Unique IDentifier) column type.

MySQL has an AUTO_INCREMENT column type.

Oracle does not have a data type like this, but it has Sequences.

AUTO_INCREMENT values effectively number the rows of a table in the order in which they are added to the table by means of INSERT commands. They start at 1 and go up in 1’s. It is a pity that the numbering always has to be within the entire table. A common requirement that they do not address is for the numbering to be within groups within the table. For example, in a table giving all the details of all orders accepted by a company, each row is identified by a line number "within" order number, such that the line numbers of a particular order identify the lines of that order. We do not know of any SQL implementation that provides for auto-incrementing in such cases, nor is there any provision for them in the international standard for SQL.

Oracle's sequences are criticized by some authorities for being over-elaborate and too difficult to use for the simple cases that are addressed by "identity" and "auto-increment". However, Oracle is so influential in the SQL community that its sequences did recently (2003) find their way into the international standard, and some vendors (IBM for example) have added them to their own products to make it easier to woo users away from Oracle!
Creating a Sequence

To create a sequence that starts at 1 and increments by 1 use:

```sql
CREATE SEQUENCE seq1;
```

To create a sequence that starts at 10 and increments by 2 use:

```sql
CREATE SEQUENCE seq2 INCREMENT BY 2 START WITH 10;
```

To delete a sequence use:

```sql
DROP SEQUENCE seq2;
```

Note that sequence names are unique within database, as table names are. Thus, the same sequence can be used for all sorts of different purposes, though it might be unwise to use one for more than one purpose.

It might be possible to use sequences for the numbering within groups referred to in the notes for Slide 7. One would have to create a new sequence for every new order, by include the order number in the sequence name. Then use the appropriate sequence to assign line numbers sequentially to the lines of the corresponding order.
Using a Sequence

CREATE TABLE test ( urn NUMBER, name VARCHAR(10) );

To insert a row where one of the columns should be an URN:

INSERT INTO test VALUES (seq1.nextval,'Tim');

unique reference number (URN).

Curiously, there is no guarantee that the invocation seq1.nextval will return the integer that is just 1 higher than what was returned by the most recent previous invocation of seq1.nextval. The reason is given as being to do with concurrency problems—guaranteeing a monotonically increasing sequence would apparently incur too much overhead.
Relational Algebra and SQL

SQL manipulates tables. Relational operators are closed over relations, so, if translated into SQL, should be closed over these tables.

6 primitive operators: union, difference, product, projection, selection and renaming.

NB The SQL keyword SELECT is associated with projection, not selection!
Also: derived operators (cf operators in arithmetic, such as square(x) = x * x). Examples include intersection and join.

There is a counterpart for relational algebra, as you have learned it in the theoretical part, in SQL, although there is no full equivalence.

SQL manipulates tables. Relational operators are closed over relations, so, if translated into SQL, should be closed over these tables.

Thus, the result of a relational operator on one or more tables is another table.

There are six primitive operators in relational algebra, according to Codd’s original set, are - union, difference, product, projection, selection; later, renaming was added. This part of the course is based on Codd’s original set. Later theories in the ’70’s, of which Hugh Darwen is one of the early implementers, have defined as basic primitive operators of RA JOIN, projection, RENAME, restriction, extension, SUMMARIZE, UNION and "semidifference" (NOT MATCHING). As an exercise, after this lecture, try and find their SQL counterpart!

NB The SQL keyword SELECT is associated with projection, not selection!
Also have derived operators (cf operators in arithmetic, such as square(x) = x * x). Examples include intersection and join.
Primitive Relational Operators

- The 5 primitive relational operators:
  - union \( A \cup B \) Combines all tuples from \( A \) and \( B \).
  - difference \( A \setminus B \) All tuples from \( A \) with those common to \( B \) (\( A \cap B \)) removed.
  - product \( A \times B \) Cartesian product of \( A \) and \( B \).
  - project \( A[a; b \ldots] \) Select only attributes \( a; b \ldots \) from relation \( A \).
  - select \( A : C \) Select only those tuples satisfying the specified Boolean condition \( C \), where \( C \) is constructed from arithmetic comparisons involving attributes by using propositional connectives.

E.F. Codd, in his seminal 1970 paper, defined a relational algebra consisting of the primitive operators shown on this slide. Although it soon became apparent that that algebra was deficient in several respects, the work by others in the early 1970s that led to the improved version we teach in CS252 has not found its way into all the relevant textbooks and university courses — and nor was it noticed by the IBM researchers who devised the original SQL!

We will find that SQL's UNION is not quite the same as relational UNION, and this is because Codd himself did not give a precise definition of that operator. Like SQL, he didn't pay sufficient attention to attribute names (column names in SQL).

Insufficient attention to attribute names also led Codd to define Cartesian product (TIMES) as primitive instead of JOIN. He allowed \( r1 \times r2 \) to be defined for any two relations \( r1 \) and \( r2 \), even when some attribute name \( a \) is common to both operands, which would lead to a result having two attributes of the same name. He then defined what he called the "natural join", \( r1 \ JOIN \ r2 \), to be equivalent to, in Tutorial D, \( ((r1 \times r2) \ WHERE \ r1.c1 = r2.c1 \ AND \ r1.c2 = r2.c2 \ \ldots \ AND \ r1.cn = r2.cn) \ \{\ \text{ALL BUT} \ r2.c1, r2.c2 \ \ldots, r2.cn \} \), where \( c1, c2 \ldots, cn \) are the common attributes.
Derived Relational Operators

**intersection** $A \cap B$ : tuples common to $A$ and $B$.

**join** $A$($A.a = B.b$) JOIN $B$ Join tables $A$ and $B$ together - for each row match attribute $A.a$ with $B.b$, discarding duplicate columns.

Another example of a derived operator is **divideby** - this attempts to invert product, in so far as this is possible.

*intersection* $A \cap B$ : The result of $A$ INTERSECT $B$ is the relation whose body consists of the "tuples common to both $A$ and $B".

For a more formal definition of these operators, refer to HACD.6.

Note that in some cases, namely where $r1$ and $r2$ have disjoint headings, $r1$ DIVIDEBY $r2$ is actually equivalent to $r1$ TIMES $r2$, so to say that it "attempts to invert product" is not applicable for all cases.
Codd's completeness criterion:

a query language is complete if it can express each of the five primitive relational operators.
Union

Union A ∪ B
Use SQL keyword UNION. Tables must be compatible ... have the same attributes (column headings).

\[ \text{(SELECT artist FROM Pop_albums WHERE artist LIKE 'U%') UNION (SELECT artist FROM Band_members WHERE mbr1 = 'Grohl');} \]

Result is a one column table containing three entries: Foo Fighters, U2 and Underworld.

SQL's UNION differs significantly from the relational UNION you learn in the theory section of CS252. Recall that the attributes of a relation are not considered to be placed in any particular order. By contrast the columns of an SQL table are very definitely ordered, and this gives rise to many complications in the definitions of the operators. In the case of SQL's UNION, the operands must have the same number, \( n \), of columns, and the \( k \)-th column (\( 1 \leq k \leq n \)) of the first operand must be of the same data type as the \( k \)-th column of the second operand. So it is like relational UNION except that columns are paired together by ordinal position, whereas in relational UNION the attributes are paired by name.

Recall that in relational UNION the attributes of the result have the same names as those of both operands. A question for you to investigate is, what are the column names of the result of an SQL UNION? — or at least an Oracle SQL UNION (for SQL implementations do not all agree on this issue, even though the international standard is clear on it). Having answered that question, you can then decide for yourself whether Oracle's UNION is commutative. (Recall that relational UNION is commutative.)
Intersection

Intersection $A \cap B$

Use SQL keyword \texttt{INTERSECT}. Tables must be compatible.

Query selects \textit{U2} and \textit{Foo Fighters}:

\begin{verbatim}
(SELECT artist FROM Pop_albums)
INTERSECT
(SELECT artist FROM Band_members);
\end{verbatim}

Recall that in \textbf{Tutorial D} $A \cap B$ is written as \texttt{A INTERSECT B}, just as in SQL, and is merely an alternative for \texttt{A JOIN B} in the special case where $A$ and $B$ are relations having the same heading. Again, though, relational intersection and SQL intersection are not quite the "same animal". The difference is as with \texttt{UNION} — in SQL the columns are paired by ordinal position, not by column name. SQL is more or less forced into adopting this approach because it does not require every column to have a name; also, it permits two or more columns to have the same name.

Example of a column with no name:

\begin{verbatim}
SELECT COUNT(*) FROM T
\end{verbatim}

returns a table with a single column that has no name.

Example of a table with two columns of the same name:

\begin{verbatim}
SELECT C1 AS X, C2 AS X FROM T
\end{verbatim}

has two columns, both named $X$.

Another example of a table with two columns of the same name:

\begin{verbatim}
SELECT T1.C, T2.C FROM T1, T2
\end{verbatim}

has two columns, both named $C$. 
Difference

Use SQL keyword **MINUS**.

(SELECT artist FROM Pop_albums)

MINUS

(SELECT artist FROM Band_members);

Selects everything but *U2* and *Foo Fighters - The Verve* and *Underworld*.

As you know, relational MINUS is supported by **Tutorial D**, but in **Tutorial D** A MINUS B is merely an alternative way of writing A NOT MATCHING B in the special case where A and B are relations of the same heading. SQL's MINUS applies the same column-matching rule as for UNION and INTERSECT. SQL does not have an immediate counterpart of NOT MATCHING. Instead of T1 NOT MATCHING T2 you would have to write

```
SELECT * FROM T1 WHERE NOT EXISTS (SELECT * FROM T2 WHERE T1.c1 = T2.c1 AND T1.c2 = T2.c2 ...)
```

where $c1$, $c2$, ... are the common columns of T1 and T2.
**Product**

**Product A × B**

Part of the SELECT statement - list more than one table after keyword FROM.

```sql
SELECT f1, f2 FROM Forward, Reverse;
```

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Forward</strong></td>
<td><strong>Reverse</strong></td>
<td><strong>Forward × Reverse</strong></td>
</tr>
<tr>
<td>f1</td>
<td>f2</td>
<td>f1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

In **Tutorial D** this would be Forward TIMES Reverse. Note, though, that although Forward and reverse in this example have different names for their only columns, SQL would allow those names to be equal, resulting in a table with two identical column names.
Project

Project A[a, b …]

Using keyword SELECT as from day 1.

```
SELECT DISTINCT a, b FROM A;
```

```
SELECT DISTINCT artist, album FROM Pop_albums;
```

Actually, it is somewhat inaccurate to say that SQL's SELECT is its counterpart of relational projection. It is so only when each of the following conditions applies:

1. The key word DISTINCT is used immediately after SELECT. If DISTINCT is omitted, the result may be such that the same row can appear more than once.
2. Each of the SELECTed items is a simple column reference, possibly qualified by AS column-name.
3. No two columns in the result have the same name.
Join

Join $A(A.a = B.b)$ JOIN B

Using SELECT as in last seminar:

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

```
SELECT DISTINCT * FROM A, B
WHERE A.a = B.b;
```

Notice how join is a combination of the product of tables and a predicate selection.

The examples show how joins are commonly done in SQL. Note that the first example is more like the COMPOSE operator of Tutorial D, because the common column (barcode) is eliminated in the result.
Advanced Joins

We covered a simple (equi-)join between two tables earlier in the module.
Joins combine rows from two or more tables to create a single result.
Columns are compared with a Join Condition.
Pairs of rows each containing one row from each table are combined when the join condition evaluates to \text{T}RU\text{E}.

Please note that the join could specify also which additional tuples to include, e.g., in the outer join.
Join Types

(SQL) Joins can be classified into the following categories:

• Cartesian Products
• Inner Joins (Equijoins)
• Self Joins
• Outer Joins (Left, Right and Full)

Please note: We don't formally classify joins in relational theory and we don't have outer joins at all! (because of NULLs).
Cartesian Products (Cross Join)

• join without a Join Condition
• generate many rows of data, e.g., test data.
• the base of all the other types of join.

If a join is performed without a Join Condition then the Cartesian product of the specified tables is returned.

Cartesian products can generate many rows of (meaningless?) data, for this reason they are often used for creating test data. Apart from this, they have few practical uses.

Cartesian products form the base of all the other types of join.
Inner Joins (Equijoins)

An Inner Join (or Equijoin) is a join with a condition (e.g., that compares columns for equality =). Rows are combined that have equal values in the specified columns.

The order of the tables listed in the FROM clause should have no significance.

In SQL, the order of elements in the FROM clause determines the order of columns in the result.
Inner Join Example

The example from before:

```
SELECT DISTINCT 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>
</tbody>
</table>

Actually, you can also use directly the SQL inner join as below:

```
SELECT artist, album, year
FROM Pop_albums INNER JOIN CD_year ON Pop_albums.barcode = CD_year.barcode
```

that it is just an alternative way of writing:

```
SELECT artist, album, year
FROM Pop_albums, CD_year
WHERE Pop_albums.barcode = CD_year.barcode
```
Self Join

A Self Join is a join of a table to itself. Put the table in the `FROM` clause twice. Self joins are very useful. Use aliases to distinguish columns in the `WHERE` clause.

Recall that in relational algebra the JOIN operator is *idempotent*: $r \ JOIN \ r = r$, for every relation $r$. So the concept of "self-join" does not usefully arise. In SQL, however, `FROM t as t1, t as t2` yields the Cartesian product of $t$ with itself, and SQL allow the "aliases" $t1$ and $t2$ to be used to qualify column names in the subsequent `WHERE` and `SELECT` clauses. Although the SQL approach here is logically flawed, it does allow many queries of this kind to be expressed more easily in SQL than in *Tutorial D*, where the RENAME operator has to be used to generate distinct attribute names, in place of SQL's aliases on tables.
Self Join Example

Determine artists that have released more than one album:

```sql
SELECT DISTINCT a.artist
FROM Pop_albums a, Pop_albums b
WHERE a.artist = b.artist
AND a.album <> b.album;
```

Can you think of alternative way of expressing this in SQL?

```sql
alternative ways. E.g.
SELECT artist
FROM Pop_albums
GROUP BY artist
HAVING COUNT(*) > 1;
```

or

```sql
SELECT DISTINCT artist
FROM Pop_albums as pa1
WHERE NOT EXISTS
(SELECT * FROM Pop_albums as pa2
 WHERE pa1.album <> pa2.album AND pa1.artist = pa2.artist);
```
Hierarchical Data

Self-joins can be useful when working with hierarchical data.
Consider the following table (EMPLOYEES):

<table>
<thead>
<tr>
<th>EMPLOYEEID</th>
<th>SUPERVISORID</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>NULL</td>
<td>John Balfour</td>
</tr>
<tr>
<td>101</td>
<td>100</td>
<td>Susan Saronnen</td>
</tr>
<tr>
<td>102</td>
<td>100</td>
<td>Eric La Sold</td>
</tr>
<tr>
<td>103</td>
<td>100</td>
<td>Martin Murphy</td>
</tr>
<tr>
<td>104</td>
<td>103</td>
<td>Erica Strange</td>
</tr>
<tr>
<td>105</td>
<td>103</td>
<td>Noah Tamil</td>
</tr>
</tbody>
</table>
How do we write a query to find the name of each employee's supervisor? We can use a self-join:

```
SELECT staff.name, supervisor.name
FROM EMPLOYEES staff, EMPLOYEES supervisor
WHERE staff.supervisorid = supervisor.employeeid;
```

<table>
<thead>
<tr>
<th>STAFF.NAME</th>
<th>SUPERVISOR.NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Susan Saronnen</td>
<td>John Balfour</td>
</tr>
<tr>
<td>Eric La Sold</td>
<td>John Balfour</td>
</tr>
<tr>
<td>Martin Murphy</td>
<td>John Balfour</td>
</tr>
<tr>
<td>Erica Strange</td>
<td>Martin Murphy</td>
</tr>
<tr>
<td>Noah Tamil</td>
<td>Martin Murphy</td>
</tr>
</tbody>
</table>

Warwick

CS252 Fundamentals of Relational Databases

What about finding employee pairs such that the second is somewhere in the first’s supervisor chain? That is a problem of a kind that has fascinated researchers ever since Codd first made his proposals back in 1970; for his proposals did not include a solution to this kind of problem. To find the supervisor of a given employee’s supervisor we need two joins; to find the next supervisor up the chain we need three, and so on. Such queries need an element of recursion in the query expression, which neither Codd’s algebra or original SQL includes. But recursive queries are beyond the scope of CS252—sorry!
Outer Join

An Inner Join excludes rows from either table that don't have a matching row in the other table.
An Outer Join allows us to return unmatched rows.
Outer Joins come in three varieties:
• LEFT - only unmatched rows from the left table are kept
• RIGHT - only unmatched rows from the right table are kept
• FULL - unmatched rows from both tables are retained

All these joins are shorthands, so you should know how to express them in longhand, using UNION. The formal definition of RIGHT JOIN falls out readily from that of LEFT JOIN, and FULL JOIN is just the UNION of the LEFT JOIN and the RIGHT JOIN.

Please also note: RIGHT JOIN is not redundant with LEFT JOIN. A first sight it looks as if T1 RIGHT JOIN T2 is the same as T2 LEFT JOIN T1, but this is not the case because the order of columns in the result is different!
**Example data**

Imagine we have two tables defined as:

*CD_company*

<table>
<thead>
<tr>
<th>barcode</th>
<th>company</th>
</tr>
</thead>
<tbody>
<tr>
<td>04228289827</td>
<td>Island</td>
</tr>
<tr>
<td>042284229920</td>
<td>Island</td>
</tr>
<tr>
<td>731451034725</td>
<td>Island</td>
</tr>
<tr>
<td>026734000524</td>
<td>Junior</td>
</tr>
<tr>
<td>724384491321</td>
<td>Virgin</td>
</tr>
<tr>
<td>724385583223</td>
<td>Capital</td>
</tr>
<tr>
<td>724383719020</td>
<td>EMI</td>
</tr>
<tr>
<td>891030505032</td>
<td>Naxos</td>
</tr>
</tbody>
</table>
### 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>042284229920</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>724384491321</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>
<tr>
<td>722323583123</td>
<td>Leftfield</td>
<td>Leftism</td>
</tr>
</tbody>
</table>

Notice not all companies match up to an album and not all albums match to a company.
Outer Left Join Example

```
SELECT DISTINCT company, artist, album
FROM CD_Company LEFT JOIN Pop_albums ON Pop_albums.barcode = CD_Company.barcode;
```

<table>
<thead>
<tr>
<th>COMPANY</th>
<th>ARTIST</th>
<th>ALBUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMI</td>
<td>Capital</td>
<td>The Colour and the Shape</td>
</tr>
<tr>
<td>Island</td>
<td>Foo Fighters</td>
<td>The Unforgettable Fire</td>
</tr>
<tr>
<td>Island</td>
<td>U2</td>
<td>Rattle and Hum</td>
</tr>
<tr>
<td>Island</td>
<td>U2</td>
<td>Achtung Baby</td>
</tr>
<tr>
<td>Junior</td>
<td>Underworld</td>
<td>Second Toughest in the Infants</td>
</tr>
<tr>
<td>Virgin</td>
<td>The Verve</td>
<td>Urban Hymns</td>
</tr>
<tr>
<td>Naxos</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Produces a summary of record labels and the artists they publish.

Notice that the record labels EMI and Naxos are displayed even though there are no albums with these companies in the Pop albums table.
Outer Right Join Example

SELECT DISTINCT year, artist FROM Pop_albums
RIGHT JOIN CD_year
ON Pop_albums.barcode = CD_year.barcode
ORDER BY year;

YEAR ARTIST
---------------
1984 U2
1988 U2
1991 U2
1992
1996 Underworld
1996
1997 Foo Fighters
1997 The Verve
Outer Full Join Example

SELECT DISTINCT company, artist, album
FROM CD_Company FULL JOIN Pop_albums
  ON Pop_albums.barcode = CD_Company.barcode;
<table>
<thead>
<tr>
<th>COMPANY</th>
<th>ARTIST</th>
<th>ALBUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Island</td>
<td>U2</td>
<td>The Unforgettable Fire</td>
</tr>
<tr>
<td>Island</td>
<td>U2</td>
<td>Rattle and Hum</td>
</tr>
<tr>
<td>Island</td>
<td>U2</td>
<td>Achtung Baby</td>
</tr>
<tr>
<td>Junior</td>
<td>Underworld</td>
<td>Second Toughest in the Infants</td>
</tr>
<tr>
<td>Virgin</td>
<td>The</td>
<td>Verve Urban Hymns</td>
</tr>
<tr>
<td>Capital</td>
<td>Foo Fighters</td>
<td>The Colour and the Shape</td>
</tr>
<tr>
<td>Naxos</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EMI</td>
<td>Leftfield</td>
<td>Leftism</td>
</tr>
</tbody>
</table>

As an exercise, check that the outer join actually produces this table. See if you can spot any errors and typos!!
Interim Summary

- Indexes, Sequences,
- Relational Algebra and Advanced Joins