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

- Redundant but can improve the performance of a DBMS (look up rows)
- Updates can take longer
- Not required, do not affect query results
- Automatically added to any column declared as a Primary Key.
- DB implementations: own specific schemes; in some you can specify what kind to use.

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 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.

Creating a Sequence

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

```
CREATE SEQUENCE seq1;
```

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

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

To delete a sequence use:

```
DROP SEQUENCE seq2;
```

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');
```

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
- 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.

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 removed.
  - product $A \times B$ Cartesian product of A and B.
  - projection $\pi_{a \ b \ ...} A$ Select only attributes $a, b, ...$ from relation $A$.
  - select $\sigma_C A$ Select only those tuples satisfying the specified Boolean condition $C$, where $C$ is constructed from arithmetic comparisons involving attributes by using propositional connectives.

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.
- divideby - this attempts to invert product, in so far as this is possible.
Codd’s completeness criterion: a query language is complete if it can express each of the five primitive relational operators.

Union

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

(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.

Intersection

Intersection $A \cap B$
Use SQL keyword `INTERSECT`. Tables must be compatible.
Query selects U2 and Foo Fighters:

(SELECT artist FROM Pop_albums)
INTERSECT
(SELECT artist FROM Band_members);

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.

Product

Product $A \times B$
Part of the SELECT statement - list more than one table after keyword `FROM`.

<table>
<thead>
<tr>
<th>Forward</th>
<th>Reverse</th>
<th>Forward x Reverse</th>
</tr>
</thead>
<tbody>
<tr>
<td>f1</td>
<td>f2</td>
<td>f1 f2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1 2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2 1</td>
</tr>
</tbody>
</table>

Project

Project $A[a, b ...]$
Using keyword `SELECT` as from day 1.

SELECT DISTINCT artist, album FROM Pop_albums;
Join

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

Using SELECT as in last seminar:

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

```sql
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.

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 TRUE.

Join Types

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

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

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.

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.

Inner Join Example

The example from before:

```sql
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>
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.

Self Join Example

Determine artists that have released more than one album:

```
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?

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;
```

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

Example data

Imagine we have two tables defined as:

<table>
<thead>
<tr>
<th>barcode</th>
<th>company</th>
</tr>
</thead>
<tbody>
<tr>
<td>04228289827</td>
<td>Island</td>
</tr>
<tr>
<td>04228429920</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>724383718920</td>
<td>EMI</td>
</tr>
<tr>
<td>891030505032</td>
<td>Naxos</td>
</tr>
</tbody>
</table>
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>
</tr>
</thead>
<tbody>
<tr>
<td>Island</td>
</tr>
<tr>
<td>Junior</td>
</tr>
<tr>
<td>Capital</td>
</tr>
<tr>
<td>EMI</td>
</tr>
<tr>
<td>Naxos</td>
</tr>
</tbody>
</table>

Notice not all companies match up to an album and not all albums match to a company.

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;
```

<table>
<thead>
<tr>
<th>YEAR</th>
<th>ARTIST</th>
</tr>
</thead>
<tbody>
<tr>
<td>1984</td>
<td>U2</td>
</tr>
<tr>
<td>1988</td>
<td>U2</td>
</tr>
<tr>
<td>1991</td>
<td>U2</td>
</tr>
<tr>
<td>1992</td>
<td>U2</td>
</tr>
<tr>
<td>1996</td>
<td>Underworld</td>
</tr>
<tr>
<td>1996</td>
<td>U2</td>
</tr>
<tr>
<td>1997</td>
<td>Foo Fighters</td>
</tr>
<tr>
<td>1997</td>
<td>The Verve</td>
</tr>
</tbody>
</table>

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>
</tr>
</thead>
<tbody>
<tr>
<td>Island</td>
</tr>
<tr>
<td>Junior</td>
</tr>
<tr>
<td>Capital</td>
</tr>
<tr>
<td>EMI</td>
</tr>
<tr>
<td>Naxos</td>
</tr>
<tr>
<td>Leftfield</td>
</tr>
</tbody>
</table>

Notice not all companies match up to an album and not all albums match to a company.

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.
Interim Summary

• Indexes, Sequences,
• Relational Algebra and Advanced Joins