Administrivia

BEFORE lab, please read and highlight

• Assignment 8
• Database Design (on learn.uwaterloo.ca)
  Adapted from Access Database Design & Programming by Steven Roman
• The SQL tutorial notes (on learn.uwaterloo.ca)
• Slides for this lecture

START the lab this week by doing the SQL tutorial

Topics for today

• relational database design
• when you need another table, and why
• SQL (Structured Query Language)
• a model for relational databases

Please ask questions!
Assumptions for Today’s Lecture

You’ve seen a two-table relational database

• you’ve been exposed to
  forms, reports, queries, sorting, & data validation

• you’re familiar with the terms
  database
  table
  record (aka row)
  field (aka column)

If this wasn’t true at the beginning of the term,

• by now you’ve completed the Filemaker Intro

Things to Think about

How does a DBMS differ from a spreadsheet?

Why would I choose to use a DBMS?

How does SQL differ from FileMaker?

What are the reasons for needing more than one database table?
**Why SQL?**

It’s an excellent MODEL for how relational DBMS’s work

Modern “big” DBMS’s are SQL-based

Many PC databases are not

• but can often be used as “front-ends” to mainframe SQL systems
• though FileMaker Pro 11,12, 13 and MS Access are based on SQL

**OS X actually comes with two SQLs!**

• “SQL Lite,” which is designed to be embedded in programs (including the O/S)
• “My SQL,” a (very) popular open source SQL server (used for the Math Faculty’s inventory database)

**Often you can**

import data from an SQL database into software with a nice GUI (eg FileMaker)

by crafting an appropriate “SQL select statement”

---

**Databases – Structured Data**

**Why use a database at all?**

Structuring data allows us to do things we can’t do efficiently, or can’t feasibly do, with unstructured data

• The added power & flexibility aren’t free
  
  it takes time and effort to create (and maintain) the structure

• You have to decide if that effort is worthwhile

**Obvious questions:**

• What do I mean by “structure” in a database?
• What’s the payoff?
The visa worksheet in the Excel assignment

- is an example of a 1–table database, although we built it in Excel, not with a DBMS
- each ROW ("record") holds data for a particular transaction
- each COLUMN holds a particular piece of data about that transaction (a "field")
- we could have used FileMaker
  though for what we wanted to do, it wouldn’t have been worth the effort of learning FileMaker
- indeed, we could have placed the data in a Word table or even in a text processor (eg BBEdit)...
  separate fields by tabs, separate records by ¶
  but working with the data would have been MUCH harder — think about implementing the Actual Balance and Statement Balance columns!

We’ll look at databases in Excel later
Key DBMS Functionality

Data entry *validation*

Sophisticated searching (aka “queries”)

Sophisticated summarizing and reporting

Safe simultaneous updates by multiple users

The REAL power of a relational database

• appears when you have multiple related tables
• what does “related” mean?
• why have multiple tables?

A Music Library

The goal — to refine our understanding of why/when multiple tables are necessary

Suppose you want to keep track of your music

• Album Title
• Artist
• Medium (CD, Tape, LP record, ...)
• Category (Jazz, Classical, Hard Rock, ...)
• Price
• Purchase Date
• Copyright
• Label

Well, you could do it with a word processor

• but ... how to find all the recordings by Led Zeppelin?
• & ... what’s the value of your Charlie Parker albums?
• & ... how to avoid entry of a bogus Medium, Category, etc.

How well would Excel work?

Consider how we might manage our music in FileMaker
Albums in FileMaker (1)

Filemaker

* requires datatypes — Text, Number, Date, etc — why?
* provides data entry options for data validation (default values, value lists, range checks, etc)

Albums in FileMaker (2)

Aside: looks a lot like a spreadsheet, eh?
But as you know, FileMaker has a lot more layout flexibility than Excel.
Also, FileMaker restricts the way you can inter-connect fields via computation (= formulas) ["structure"...]

Aside: looks a lot like a spreadsheet, eh?
But as you know, FileMaker has a lot more layout flexibility than Excel.
Also, FileMaker restricts the way you can inter-connect fields via computation (= formulas) ["structure"...]
Here’s a simple language that lets us describe matching between (database) tables

```
select field_list from table_list where conditions
```

EG

```
select Title, Artist, Price from Albums
```

**Another Example of an SQL Select Statement**

```
select Album_ID, Mins, Secs, Title from Songs
```
Asking for albums below a certain price

\texttt{select Title, Artist, Price from Albums where Price < 10.00}

Asking for albums with a particular album title

\texttt{select Title, Artist, Medium, Category, Price from Albums where Title = 'North Country'}
Asking for albums by a particular artist

```sql
select Title, Artist, Medium, Category, Price from Albums where Artist = 'The Rankin Family'
```

<table>
<thead>
<tr>
<th>Title</th>
<th>Artist</th>
<th>Medium</th>
<th>Category</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>North Country</td>
<td>The Rankin Family</td>
<td>CD</td>
<td>Folk</td>
<td>19.54000</td>
</tr>
<tr>
<td>Fare Thee Well</td>
<td>The Rankin Family</td>
<td>CD</td>
<td>Folk</td>
<td>25.29000</td>
</tr>
</tbody>
</table>

Suppose you want songs too?

Containing such data as

- Title
- Side
- Track
- Playing Time

Maybe you're the librarian for a radio station...

Can we just add Song fields to the Albums Table?

- It's a lot of work (145 additional fields!)
- And how to find a song?
- Or list all the songs that are more than 3 minutes long?
- Or make an alphabetical list of the songs!
- How many songs should you set the table up for?
  - If too few ... you run out
  - If too many ... you waste effort & space
The Songs for each album ... as a list

Laborious to set up...

... as a form (1)

This looks pretty good . . .

... though it's also laborious to set up . . .

... but try another record, . . .
As a Form (2)

... and we can see there are a lot of empty fields

The General Problem Just Illustrated

We want to have multiple copies of some field(s) and we can’t know in advance how many copies — especially difficult if there’s no limit!

Replicating fields is bad because

• it’s a lot of work to set up
• it makes searching difficult
• you waste a lot of space
• you must modify the database structure to add more copies if you run out
How about a table with a separate record for each song?

Now it's easy to find a song, but ... look at all that space wasted in repetitive album info! (See next slide.)

And consider changing the Category for an album...
Be careful
• to get ALL the songs for that album

One Table, One Song/Record – The Data (1)

Space wasted in repetitive album info!
Consider changing the Category for an album
One Table, One Song/Record – The Data (2)

Notice that Album Title, Artist, Medium, Category, ... & Label are completely determined by the Album ID. That is, if you know the Album ID, you know what the Album Title, Group, Medium, Category, ... & Label are. They're always the same for a given Album ID. So why store the Album Title, Group, Medium, ... & Label repeatedly? Why not store them once somewhere else, and keep just the Album ID with each song?

One Table, One Song/Record – The Data (3)

So the idea is to keep (just) the songs in a separate table:

- With just an Album ID field for each song record to locate the album information for each song, it’s just an integer, so it doesn’t take much space.
- From the Album Table, use Album ID to find Songs in the Song Table. DBMS’s do this for you automatically.
- From the Song Table, use Album ID to find album info in the Album Table. DBMS’s do this for you automatically.
- Avoids wasted space
- Searching is straightforward
- Adapts automatically and efficiently to ANY number of songs / album

Important!
Don’t be confused by album data shown in the Song Table — it’s temporarily copied from the Album Table just for
The Song Table in FileMaker

as a list, showing all the songs on an album

<table>
<thead>
<tr>
<th>Title</th>
<th>Once Upon a Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Album ID</td>
<td>3</td>
</tr>
<tr>
<td>Side</td>
<td>Verdi Group</td>
</tr>
<tr>
<td>Track</td>
<td>CD Medium</td>
</tr>
<tr>
<td>Minutes</td>
<td>5:51</td>
</tr>
<tr>
<td>Seconds</td>
<td>19.50</td>
</tr>
<tr>
<td>Album ID</td>
<td>3</td>
</tr>
<tr>
<td>Date</td>
<td>26 Jan 1994</td>
</tr>
<tr>
<td>Copyright</td>
<td>1992</td>
</tr>
<tr>
<td>Vendor</td>
<td>Label</td>
</tr>
<tr>
<td>Songs</td>
<td>Album Title</td>
</tr>
</tbody>
</table>

The Album info is temporarily copied from the Album table.

The Album Table in FileMaker

as a list, showing album information only

<table>
<thead>
<tr>
<th>ID</th>
<th>Mediun Category</th>
<th>Price</th>
<th>Per Date</th>
<th>Label</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CD</td>
<td>Soft</td>
<td>14.99</td>
<td>Fine</td>
<td>Live at the Republic</td>
</tr>
<tr>
<td>2</td>
<td>CD</td>
<td>Soft</td>
<td>19.99</td>
<td>Fine</td>
<td>Live at the Republic</td>
</tr>
<tr>
<td>3</td>
<td>CD</td>
<td>Soft</td>
<td>19.99</td>
<td>Fine</td>
<td>Live at the Republic</td>
</tr>
<tr>
<td>4</td>
<td>CD</td>
<td>Soft</td>
<td>19.99</td>
<td>Fine</td>
<td>Live at the Republic</td>
</tr>
<tr>
<td>5</td>
<td>CD</td>
<td>Soft</td>
<td>19.99</td>
<td>Fine</td>
<td>Live at the Republic</td>
</tr>
<tr>
<td>6</td>
<td>CD</td>
<td>Soft</td>
<td>19.99</td>
<td>Fine</td>
<td>Live at the Republic</td>
</tr>
<tr>
<td>7</td>
<td>CD</td>
<td>Soft</td>
<td>19.99</td>
<td>Fine</td>
<td>Live at the Republic</td>
</tr>
<tr>
<td>8</td>
<td>CD</td>
<td>Soft</td>
<td>19.99</td>
<td>Fine</td>
<td>Live at the Republic</td>
</tr>
<tr>
<td>9</td>
<td>CD</td>
<td>Soft</td>
<td>19.99</td>
<td>Fine</td>
<td>Live at the Republic</td>
</tr>
<tr>
<td>10</td>
<td>CD</td>
<td>Soft</td>
<td>19.99</td>
<td>Fine</td>
<td>Live at the Republic</td>
</tr>
<tr>
<td>11</td>
<td>CD</td>
<td>Soft</td>
<td>19.99</td>
<td>Fine</td>
<td>Live at the Republic</td>
</tr>
<tr>
<td>12</td>
<td>CD</td>
<td>Soft</td>
<td>19.99</td>
<td>Fine</td>
<td>Live at the Republic</td>
</tr>
<tr>
<td>13</td>
<td>CD</td>
<td>Soft</td>
<td>19.99</td>
<td>Fine</td>
<td>Live at the Republic</td>
</tr>
<tr>
<td>14</td>
<td>CD</td>
<td>Soft</td>
<td>19.99</td>
<td>Fine</td>
<td>Live at the Republic</td>
</tr>
<tr>
<td>15</td>
<td>CD</td>
<td>Soft</td>
<td>19.99</td>
<td>Fine</td>
<td>Live at the Republic</td>
</tr>
<tr>
<td>16</td>
<td>CD</td>
<td>Soft</td>
<td>19.99</td>
<td>Fine</td>
<td>Live at the Republic</td>
</tr>
<tr>
<td>17</td>
<td>CD</td>
<td>Soft</td>
<td>19.99</td>
<td>Fine</td>
<td>Live at the Republic</td>
</tr>
<tr>
<td>18</td>
<td>CD</td>
<td>Soft</td>
<td>19.99</td>
<td>Fine</td>
<td>Live at the Republic</td>
</tr>
<tr>
<td>19</td>
<td>CD</td>
<td>Soft</td>
<td>19.99</td>
<td>Fine</td>
<td>Live at the Republic</td>
</tr>
<tr>
<td>20</td>
<td>CD</td>
<td>Soft</td>
<td>19.99</td>
<td>Fine</td>
<td>Live at the Republic</td>
</tr>
</tbody>
</table>

as a form, showing info about one album and a list of all the songs on that album in the Songs table

The Songs info is temporarily copied from the Songs table.
# Terminology

**Album ID**

- is a “primary key” for the Album Table because it uniquely identifies an album
- is a “foreign key” of the Songs Table because it contains a primary key of the Album Table and thus links a Song record to a unique Album record

**“One-to-many” and “many-to-one”**

- wrt Album ID
  - Albums is the “one table”
  - Songs is the “many table” because for a given Album ID there is only ONE Album record — but are (usually) MANY Song records
  - “many-to-many” can happen, too
    - though not by matching a primary key in each of two tables!
    - it is often useful
- we’ll see an example next week

---

**The Albums & Songs Database**

<table>
<thead>
<tr>
<th>Albums</th>
<th>Songs</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a primary key)</td>
<td></td>
</tr>
<tr>
<td><strong>Album_ID</strong></td>
<td><strong>Song_ID</strong></td>
</tr>
<tr>
<td>Title</td>
<td>Side</td>
</tr>
<tr>
<td>Group</td>
<td>Track</td>
</tr>
<tr>
<td>Medium</td>
<td>Title</td>
</tr>
<tr>
<td>Category</td>
<td>Mins</td>
</tr>
<tr>
<td>Price</td>
<td>Secs</td>
</tr>
<tr>
<td>Purchase Date</td>
<td>Album_ID (a foreign key)</td>
</tr>
<tr>
<td>Copyright</td>
<td></td>
</tr>
<tr>
<td>Label</td>
<td></td>
</tr>
<tr>
<td>Sample</td>
<td></td>
</tr>
</tbody>
</table>

the relationship from Albums to Songs on Album_ID is one-to-many
the relationship from Songs to Albums on Album_ID is many-to-one
Summary

Relational databases
• keep data in multiple tables
• each of which has a primary key
• and link those tables
• by matching field values (though not necessarily via a foreign key)

This works in both directions
• given a song, get the album info
• given an album, list the songs on that album
  — just by matching field values (a “relationship”)

Criteria for when you need another table...

1. You have more than one “entity” (eg cars and drivers); fields for one are empty for the other (& vice-versa)
2. You can have multiple values of some field, ESPECIALLY when you can’t predict how many
   (such as having multiple songs on each album)
3. Given the value of one field A ...

eg in a Course Offerings table:
Listing the songs on an album ("dot notation") . . . and matching songs to a particular album

```
select Side, Track, Songs.Title from Albums, Songs
where (Albums.Title = 'North Country') and (Albums.Album_ID = Songs.Album_ID)
```

---

Listing the songs in alphabetical order ("order by")

```
select Side, Track, Songs.Title from Albums, Songs
where (Albums.Title = 'North Country') and (Albums.Album_ID = Songs.Album_ID)
order by Songs.Title
```
More on Matching (1)

Here’s the precise syntax of a select statement

```sql
select field_listA  
from table_list  
[where conditions]  
[order by field_listB]  
[group by field_listC]
```

You must type each clause in the order shown

just as “The red bounces ball.” is incorrect English

[ ... ] means that ... is optional

Fields in the various field_lists must exist in a table of table_list

you can use “*” as field_listA to mean “all the fields”

If two tables use the same field name, you must write TableName.FieldName to indicate which field you mean

they aren’t necessarily a (foreignKey, primaryKey) matchup

More on Matching (2)

Repeated from the previous slide...

```sql
select field_listA  
from table_list  
[where conditions]  
[order by field_listB]  
[group by field_listC]
```

The ordering specified by “order by” is

* first by the leftmost field in field_listB
* then by the second leftmost field in field_listB
* etc, from left to right

The “where” clause can accomplish two things

* extract only specific records
  
  eg where (Albums.Title = ‘North Country’)

* specify a connection between two tables
  
  eg where (Albums.AlbumID = Songs.AlbumID)

Actually, these two actions aren’t really different . . .
Consider the two tables

<table>
<thead>
<tr>
<th>Students</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDN</td>
<td>IDN</td>
</tr>
<tr>
<td>Name</td>
<td>Course</td>
</tr>
<tr>
<td></td>
<td>Term</td>
</tr>
<tr>
<td></td>
<td>Mark</td>
</tr>
</tbody>
</table>

with Students data and Register data

<table>
<thead>
<tr>
<th>IDN</th>
<th>Name</th>
<th>IDN</th>
<th>Course</th>
<th>Term</th>
<th>Mark</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Aaron</td>
<td>10</td>
<td>CS200</td>
<td>9701</td>
<td>89</td>
</tr>
<tr>
<td>11</td>
<td>Sarah</td>
<td>10</td>
<td>Biol458</td>
<td>9701</td>
<td>75</td>
</tr>
<tr>
<td>12</td>
<td>Jose</td>
<td>11</td>
<td>CS200</td>
<td>9701</td>
<td>81</td>
</tr>
<tr>
<td>13</td>
<td>Marie</td>
<td>11</td>
<td>Econ335</td>
<td>9701</td>
<td>94</td>
</tr>
</tbody>
</table>

What are the primary keys for these tables? The foreign keys?

---

Suppose you said select * from Students as S, Register as R

Here's what that produces —

<table>
<thead>
<tr>
<th>S.IDN</th>
<th>S.Name</th>
<th>R.IDN</th>
<th>R.Course</th>
<th>R.Term</th>
<th>R.Mark</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Aaron</td>
<td>10</td>
<td>CS200</td>
<td>9701</td>
<td>89</td>
</tr>
<tr>
<td>10</td>
<td>Aaron</td>
<td>10</td>
<td>Biol458</td>
<td>9701</td>
<td>75</td>
</tr>
<tr>
<td>10</td>
<td>Aaron</td>
<td>11</td>
<td>CS200</td>
<td>9701</td>
<td>81</td>
</tr>
<tr>
<td>11</td>
<td>Sarah</td>
<td>10</td>
<td>CS200</td>
<td>9701</td>
<td>89</td>
</tr>
<tr>
<td>11</td>
<td>Sarah</td>
<td>10</td>
<td>Biol458</td>
<td>9701</td>
<td>75</td>
</tr>
<tr>
<td>11</td>
<td>Sarah</td>
<td>11</td>
<td>CS200</td>
<td>9701</td>
<td>81</td>
</tr>
<tr>
<td>11</td>
<td>Sarah</td>
<td>11</td>
<td>Econ335</td>
<td>9701</td>
<td>94</td>
</tr>
<tr>
<td>12</td>
<td>Jose</td>
<td>10</td>
<td>CS200</td>
<td>9701</td>
<td>89</td>
</tr>
<tr>
<td>12</td>
<td>Jose</td>
<td>10</td>
<td>Biol458</td>
<td>9701</td>
<td>75</td>
</tr>
<tr>
<td>12</td>
<td>Jose</td>
<td>11</td>
<td>CS200</td>
<td>9701</td>
<td>81</td>
</tr>
<tr>
<td>12</td>
<td>Jose</td>
<td>11</td>
<td>Econ335</td>
<td>9701</td>
<td>94</td>
</tr>
<tr>
<td>13</td>
<td>Marie</td>
<td>10</td>
<td>CS200</td>
<td>9701</td>
<td>89</td>
</tr>
<tr>
<td>13</td>
<td>Marie</td>
<td>10</td>
<td>Biol458</td>
<td>9701</td>
<td>75</td>
</tr>
<tr>
<td>13</td>
<td>Marie</td>
<td>11</td>
<td>CS200</td>
<td>9701</td>
<td>81</td>
</tr>
<tr>
<td>13</td>
<td>Marie</td>
<td>11</td>
<td>Econ335</td>
<td>9701</td>
<td>94</td>
</tr>
</tbody>
</table>

Every possible combination of a Student and a Register.
A model for how the where clause works (3)

More likely what you want is something like

```sql
select * from Students as S, Register as R
where S.IDN = R.IDN
```

which produces

<table>
<thead>
<tr>
<th>S.IDN</th>
<th>S.Name</th>
<th>R.IDN</th>
<th>R.Course</th>
<th>R.Term</th>
<th>R.Mark</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Aaron</td>
<td>10</td>
<td>CS200</td>
<td>9701</td>
<td>89</td>
</tr>
<tr>
<td>10</td>
<td>Aaron</td>
<td>10</td>
<td>Biol458</td>
<td>9701</td>
<td>75</td>
</tr>
<tr>
<td>11</td>
<td>Sarah</td>
<td>11</td>
<td>CS200</td>
<td>9701</td>
<td>81</td>
</tr>
<tr>
<td>11</td>
<td>Sarah</td>
<td>11</td>
<td>Econ335</td>
<td>9701</td>
<td>94</td>
</tr>
</tbody>
</table>

by constructing the previous table
and then throwing out rows that don’t satisfy the where clause

---

How the where clause works (4)

Or more elegantly,

```sql
select S.IDN, Name, Course from Students as S, Register as R
where S.IDN = R.IDN
```

which produces

<table>
<thead>
<tr>
<th>S.IDN</th>
<th>S.Name</th>
<th>R.Course</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Aaron</td>
<td>CS200</td>
</tr>
<tr>
<td>10</td>
<td>Aaron</td>
<td>Biol458</td>
</tr>
<tr>
<td>11</td>
<td>Sarah</td>
<td>CS200</td>
</tr>
<tr>
<td>11</td>
<td>Sarah</td>
<td>Econ335</td>
</tr>
</tbody>
</table>
**Just the courses for Aaron**

```sql
select S.IDN, Name, Course from Students as S, Register as R
where (Name = 'Aaron') and (S.IDN = R.IDN)
```

which produces

<table>
<thead>
<tr>
<th>S.IDN</th>
<th>S.Name</th>
<th>R.Course</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Aaron</td>
<td>CS200</td>
</tr>
<tr>
<td>10</td>
<td>Aaron</td>
<td>Bio458</td>
</tr>
</tbody>
</table>

**List the students taking CS 200**

```sql
select Course, R.IDN, Name
from Students as S, Register as R
where (Course = 'CS200') and (R.IDN = S.IDN)
```

which produces

<table>
<thead>
<tr>
<th>R.Course</th>
<th>R.IDN</th>
<th>S.Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS200</td>
<td>10</td>
<td>Aaron</td>
</tr>
<tr>
<td>CS200</td>
<td>11</td>
<td>Sarah</td>
</tr>
</tbody>
</table>
Why Stop at 2?

Suppose we add a third table

<table>
<thead>
<tr>
<th>Students</th>
<th>Register</th>
<th>Courses</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDN</td>
<td>IDN</td>
<td>Name</td>
</tr>
<tr>
<td>Name</td>
<td>Course</td>
<td>Room</td>
</tr>
<tr>
<td>Term</td>
<td>Time</td>
<td>Description</td>
</tr>
</tbody>
</table>

NB: Students.Name and Courses.Name hold different things

Here's some data for Courses

<table>
<thead>
<tr>
<th>Name</th>
<th>Room</th>
<th>Time</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS100</td>
<td>DC1351</td>
<td>M 1230</td>
<td>Introduction to Computer Usage</td>
</tr>
<tr>
<td>CS200</td>
<td>MC4060</td>
<td>M 1230</td>
<td>Advanced Concepts for Computer Usage</td>
</tr>
<tr>
<td>Biol458</td>
<td>B2 350</td>
<td>MWF 830</td>
<td>Behavioural Ecology</td>
</tr>
<tr>
<td>Econ335</td>
<td>ML212</td>
<td>TR 1000</td>
<td>Economic Development</td>
</tr>
</tbody>
</table>

What's the primary key for Courses?

A list of the courses for Aaron, with description

```sql
SELECT S.Name, Course, Description
FROM Students as S, Register as R, Courses as C
WHERE (S.Name = 'Aaron')
AND (S.IDN = R.IDN)
AND (R.Course = C.Name)
```

which produces

<table>
<thead>
<tr>
<th>S.Name</th>
<th>R.Course</th>
<th>C.Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aaron</td>
<td>CS200</td>
<td>Advanced Concepts for Computer Usage</td>
</tr>
<tr>
<td>Aaron</td>
<td>Biol458</td>
<td>Behavioural Ecology</td>
</tr>
</tbody>
</table>

Incidentally,

it would be better to use Aaron's IDN than his name (why?)

if it makes sense, you can use <, ≤, >, ≥ or <> instead of =

you can use as many tables as you want
Adding New Records – Insert

Insert Into `table_name` ( `list_of_fields` ) Values ( `list_of_values` )

EG

    `insert into` Students ( IDN, Name ) values ( 14, 'Barbara' )

Altering Existing Data – Update

Update `table_name` Set `field` = `value` Where `condition`

EG

    `update Students set Name = 'Mike' where IDN = 10`
    `update Register set Mark = 100 where Course = 'CS200'`
    `update Register set Mark = 0`
Removing Records – Delete

Delete From table_name Where condition

EG

delete from Students where IDN = 14

delete from Students

Creating Tables and Fields (1)

CREATE TABLE Students
(
    IDN integer NOT NULL,
    Name varchar(10) NOT NULL,

    PRIMARY KEY ( IDN ),
    CHECK( IDN between 10 and 99 ),
);

CREATE TABLE Courses
(
    Name varchar(10) NOT NULL,
    Room varchar(10) NOT NULL,
    Thyme varchar(10) NOT NULL,
    Description varchar(40) NOT NULL,

    PRIMARY KEY ( Name ),
);

Each field has a type (Integer, Char, ...)

allocate storage

know how to manipulate (eg compare)

"NOT NULL"

you must supply a value

PRIMARY KEYs are identified
Creating Tables (2)

```
CREATE TABLE Register
(
    IDN integer NOT NULL,
    Course varchar(10) NOT NULL,
    Term integer NOT NULL,
    Mark integer,
    PRIMARY KEY ( IDN, Course, Term ),
    FOREIGN KEY ( IDN ) REFERENCES Students ( IDN ) ON DELETE CASCADE,
    FOREIGN KEY ( Course ) REFERENCES Courses ( Name ) ON DELETE CASCADE,
    CHECK( IDN between 10 and 99 ),
    CHECK( Term between 5800 and 9999 ),
    CHECK( Mark between 0 and 100 ),
);
```

FOREIGN KEY

identify the target table
what happens on deletion or updates?

Indices (1)

Consider
• select Name, IDN from Students where Name = 'Marie'

How long does it take to find the right record?
• suppose you have 15,000 students
• examine the records one-by-one?
• is that how you look up somebody's phone number?

No!
• if the records were sorted by Name
  it would be much faster

Can we assume records are sorted in the order we want?
• not if we might look things up on any of two or more fields!
• also, we don’t want to have to enter data in sorted order,
or have to sort all the data before looking something up
Indices (2)

The solution: create indices

• and update them whenever a record is added
• or a name is changed

Indices are “auxiliary tables” you can’t (directly) manipulate

• the DBMS updates them when you change an indexed table

For our example database

• Create Index ByName On Students( Name )
• Create Index ByIDN On Students( IDN )
• Create Index ByCourse On Register( Course )
• etc.

If an index exists and would be useful

• SQL will use it automatically

SQL won’t create an index for you, however

• except for primary keys

Other SQL Commands

EG

• delete a table
• delete an index
• add a field
• delete a field

We won’t worry about them because it’s easier to use the Sybase Central
Sybase Central — Creating a New Field

Sybase Central — Utilities (Creating a New Database, Backups, etc)
Sub-Selects (1)

**SQL Statements**

```sql
SELECT Albums.Title, Purchase_Price
FROM Albums
WHERE
  Purchase_Price = (SELECT Purchase_Price
                      FROM Albums
                      WHERE Title = 'Led Zeppelin')
```

**Results**

<table>
<thead>
<tr>
<th>Title</th>
<th>Purchase_Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Led Zeppelin II</td>
<td>9.95000</td>
</tr>
<tr>
<td>Led Zeppelin</td>
<td>9.95000</td>
</tr>
</tbody>
</table>

Selects albums whose purchase price is equal to the purchase price of the album “Led Zeppelin.”

Sub-Selects (2)

**SQL Statements**

```sql
SELECT Title, Purchase_Price
FROM Albums
WHERE
  Purchase_Price > (SELECT AVG(Purchase_Price)
                      FROM Albums)
```

**Results**

<table>
<thead>
<tr>
<th>Title</th>
<th>Purchase_Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dare to Dream</td>
<td>19.50000</td>
</tr>
<tr>
<td>Bad</td>
<td>16.99000</td>
</tr>
<tr>
<td>The Phantom of the Opera</td>
<td>19.95000</td>
</tr>
<tr>
<td>Les Miserables</td>
<td>19.95000</td>
</tr>
<tr>
<td>Over 60 Minutes With Lightfoot</td>
<td>16.99000</td>
</tr>
<tr>
<td>August and Everything After</td>
<td>17.99000</td>
</tr>
<tr>
<td>North Country</td>
<td>19.54000</td>
</tr>
<tr>
<td>Live: Right here, right now</td>
<td>29.95000</td>
</tr>
<tr>
<td>Fare Thee Well Love</td>
<td>25.29000</td>
</tr>
</tbody>
</table>

Selects albums whose purchase price is greater than the average purchase price of all albums.
Selects albums for which there's at least one whose title is identical to the album's title.

Selects albums for which there's no song whose title is identical to the album's title.

There are other variations...
What’s Next?

Comparing FileMaker and SQL

Remember to read

* “Database Design” on learn.uwaterloo.ca