Table Design Considerations

GEOG/ECOL/ENR 5050

Fall 2016

Instructor: Dr. Shannon E. Albeke

Some Naming Conventions

- When naming table I prefer to use a prefix, followed by the type of data stored, first letter as a capital letter for easier reading (prefixName)
- tblXxxx – I like to create tables that store the data with the prefix ‘tbl’. These are often the Parent tables as well
- xrefXxxx – When I need to store data that has a One-to-Many relationship, I sometimes use this prefix, but also use the ‘tbl’ prefix…it just depends…
- luXxxx – I use this prefix when creating a lookup table that will constrain the values in a column of a related table (i.e. luSpecies for a table containing species codes)
- fcXxxx – I use this prefix when storing a Feature Class, also known as a shapefile or GIS layer

Primary Keys – Child Keys

- Each table MUST have column(s) that create a unique row of data. These column(s) create a Primary key for that table, this helps the database software index/sort the data and assure that data are not duplicated. Thus ensuring **Data Integrity**, our primary objective. Parent tables often allow us to store 'less' data by using codes coupled with a description of the code. (The States example)

- Child tables will almost always contain the same columns of data as the Parent, but also have additional column(s) to create unique rows, plus additional attribute data. (The Cities within States example)

- Common/related fields between two tables MUST have the same Data Type/Length

Horizontal Table Structure

- Also known as Flat File
- Positives
  - Most intuitive
    - 1 record contains all of the attributes describing the 'thing'
  - Matches format for most Reports, Stats packages, GIS
- Negatives
  - Not flexible when adding new fields
  - Null values (what do they mean?)
    - Dummy values (e.g. -9999, NA, etc.)
  - Require many lookup tables

Design Example

Horizontal/Flat Table

AID	Sex	Age	Height
M_1	F	2	53
M_2	M		60

Notice the NULL value for M_2 for age…

Vertical Table Structure

- Stores data going down instead of across
  - Can use Crosstab Query to make flat
- Positives
  - Flexible, can easily add in new Attributes (field)
  - No Null values stored in the table
  - Fewer lookup tables
  - I find it easier to calculate statistics on the data
- Negatives
  - Loss of strong data typing (general field type (text) to store all information (Number, Text, DateTime)
  - Less control over data

Design Example

Horizontal/Flat Table

AID (PK)	Sex	Age	Height
M_1	F	2	53
M_2	M		60

Notice the NULL value for M_2 for age…

AID (PK)	Parameter (PK)	Measurement (Text)
M_1	Sex	F
M_1	Age	2
M_1	Height	53
M_2	Sex	M
M_2	Height	60

Notice the NULL value doesn't need to be stored, use a PIVOT command to transform back into a Flat table for analyses

SQL Query Types

Basic Query Statement Structure

SQL:

“Some CLAUSE statement” & “Followed by FROM where” & “Followed by WHAT criteria” & “Followed by the GROUPING or ORDERING options”

e.g.

SELECT *[table_columns]* **FROM** *[table_name]*
WHERE *[table_column] = {your criteria}*

SQL Clause

- Each query statement begins with a CLAUSE and tells the db engine what action to perform
  - **SELECT** – return rows matching the criteria
  - **UPDATE** – update existing rows with new data values
  - **INSERT** – append new rows into a table
  - **INTO** – create a new data table from rows returned by query (e.g. SELECT * INTO new_table FROM yourTable)
  - **DELETE** – delete rows matching the criteria

Basic SELECT Query Statement

SQL:

SELECT *[StateName]* **FROM** *[tblStates]*
WHERE *[StateName] = 'Wyoming'*

Basic SELECT Query Statement

SQL:

SELECT *[StateName]* **FROM** *[tblStates]*
WHERE *[StateName] = "Wyoming"*

Alias = Rename your column

SELECT Avg(*[sqKM])* ***AS*** *MeanArea* **FROM** *[tblStates]*
GROUP BY StateCode

FROM WHERE stuff

- FROM – the table(s) and/or query(ies) from which you would like to return rows of data
- WHERE – the criteria used to filter which data are ‘fetched’ from the FROM command
  - Uses operators and is data-type specific

WHERE CLAUSE Operators

- Equality Operators:
  - Equal to: '**=**' or '**IS**' or **IN()**
  - Not equal to: '**<>**' or '**!=**' or '**IS NOT**' or **NOT IN()**
  - Less than: '**<**' or '**<=**'
  - Greater than: '**>**' or '**>=**'
- Mathematical Operators
  - **+** for addition
  - **-** for subtraction
  - ***** for multiplication
  - **/** for division
  - **%** for modulus (the remainder of the division)
- Logical Operators:
  - **AND** – combines conditional statements
  - **OR** – treats each conditional statement separately
    - Can use IN to string together multiple OR statements
    - E.g. WHERE State = 'WY' OR State= 'CO'

      can be WHERE State IN('WY', 'CO')
  - **BETWEEN** for numeric or datetime data
  - Treat WHERE's like math equation

GROUP BY CLAUSE

- **Aggregates** (consolidates and calculates) column values into a single record value
  - If a column is part of the SELECT statement and is not performing an expression of some sort, it must be included in the GROUP BY
    - Expressions: Avg, StDev, Count, Sum, etc.
  - **SELECT** plantType, Count(plantSpecies) **AS** NumInd
    FROM tblPlantSurvey
    WHERE plantType = 'forbs'
    GROUP BY plantType
    - This returns the Count (# of rows) for each unique plantType

ORDER BY CLAUSE

- **Sorts** rows as they are returned from a SELECT command
  - Ascending is the default, use DESC to sort in reverse order

  - **SELECT** plantType, plantSpecies, Genus
    FROM tblPlantSurvey
    WHERE plantType = “forbs”
    ORDER BY plantSpecies, Genus **DESC**

Joining multiple tables/queries

- **JOIN** joins together two tables on a matching column(s)
- E.g. our database has 2 tables (tblState, tblCounty)

```
SELECT tblCounty.county, tblState.state
FROM tblState JOIN tblCounty
ON tblState.state = tblCounty.state
WHERE tblState.state = “WY”
ORDER BY tblCounty.county
```