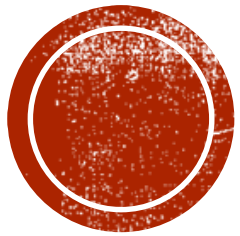
IT125 SQL: DATA, DATA TYPES, AND CONVERSIONS

Bill Barry

TONIGHT

- Laying some Groundwork:
  - Referential Integrity & MySQL Safe Update Mode
  - Full syntax description notation
- Part I: Adding, Editing, and Removing Data
  - Remember how to recreate sample databases if you need to
  - Table operations: learn how to…
    - Copy tables in order to mess with them
    - Delete them when you're done
    - Zap all data from a table but leave its structure intact
  - Table data: learn how to…
    - Add, edit, and delete data from tables
    - Understand the impact of null and auto-increment columns
- Part II: More deeply understand SQL data types and sizes
- Part III: Use implicit and explicit conversions between data types



LAYING SOME GROUNDWORK

DML VS. DDL

So far in the course, we've only studied DML, but are headed toward some DDL soon. What does that mean?

- DML is **Data Manipulation Language**. This covers…
  - …all SQL that works with data
  - …queries
  - …adding, editing, and deleting data
- DDL is **Data Definition Language**. This covers…
  - …all SQL that works with schema (definitions)
  - …creating databases
  - …creating tables
  - …creating columns and assigning their properties/attributes
  - …creating constraints like primary keys and foreign key relationships
  - …destroying tables and databases

REFERENTIAL INTEGRITY

- Definition: a core feature enforced by RDBMSs that prevents data from entering an inconsistent state
- Referential Integrity is the linchpin for trusting data; without it, we couldn't depend on our data and the relationships that bind it together
- It requires that all FK values refer to valid, existing PK values in the parent table
- A tagline for it might be, "no orphans!"
- Examples:
  - Duplicate primary keys will be rejected
  - FK data can't be entered in child table doesn't match existing PK data in parent table
  - PK data deletion is prohibited where there are FK children
  - PK data editing is prohibited where that causes a mismatch between PK and FK
- SQL uses *Declarative* Referential Integrity (see reference slide); rules are part of schema

MYSQL SAFE UPDATE MODE

- It's easy to write short SQL commands that mess up **lots** of data
- Many destructive commands affect **all** rows by default; they are Weapons of Mass Destruction when in inexperienced hands
  - Changing existing data
  - Deleting data
- MySQL provides some "training wheels" to make it less likely you'll destroy stuff; it makes you use PKs in destructive commands
- This is MySQL's "Safe Update Mode;" it is on by default
- If you want to turn it off: Edit \ Preferences \ SQL Editor \ Safe Updates option, then restart Workbench

This is not the same as referential integrity; please know the difference!

SYNTAX DESCRIPTION CONVENTIONS #1

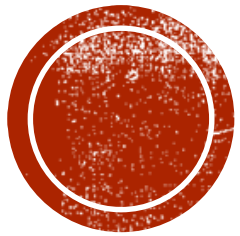
- Terms in **UPPER CASE** denote **keywords**; type these exactly as shown
  - Example: `INSERT`
  - Meaning: type "i-n-s-e-r-t" – doesn't have to be in caps, but must be spelled this way
- Terms in **lower case** denote **fill-ins**; replace with something that makes sense
  - Example: WHERE `condition`
  - Meaning: after typing "w-h-e-r-e", provide a condition (like CustLName = 'Smith')
  - Note: where richer text formatting is available, these are often *italicized* as well
- Terms in **square brackets** indicate **optional portions**; use these if you'd like
  - Example: `INSERT [INTO] tablename`
  - Meaning: You may choose to type "INTO" or leave it out; it's your choice

SYNTAX DESCRIPTION CONVENTIONS #2

- **Vertical bars** denote **options**; pick the *one* that makes sense in your context
  - Example: `[ASC | DESC]`
  - Meaning: You may type either "a-s-c" or "d-e-s-c", or neither (since it's optional)
  - Note: options are often shown in curly braces, e.g., ORDER BY { col_name | expr | pos }; this usually denotes that you *must* choose one of the options
- **Ellipses** mean to **continue the pattern**
  - Example: `expression1 [, expression2]…`
  - Meaning: supply one expression, optionally a second, then more if you'd like
- **Anything else** should be taken literally
  - Example: `(expression1 [, expression2])`
  - Meaning: type an open parenthesis, then an expression; if you choose to include a second expression, type a comma first; end with a close parenthesis

SYNTAX DESCRIPTIONS: A TEST

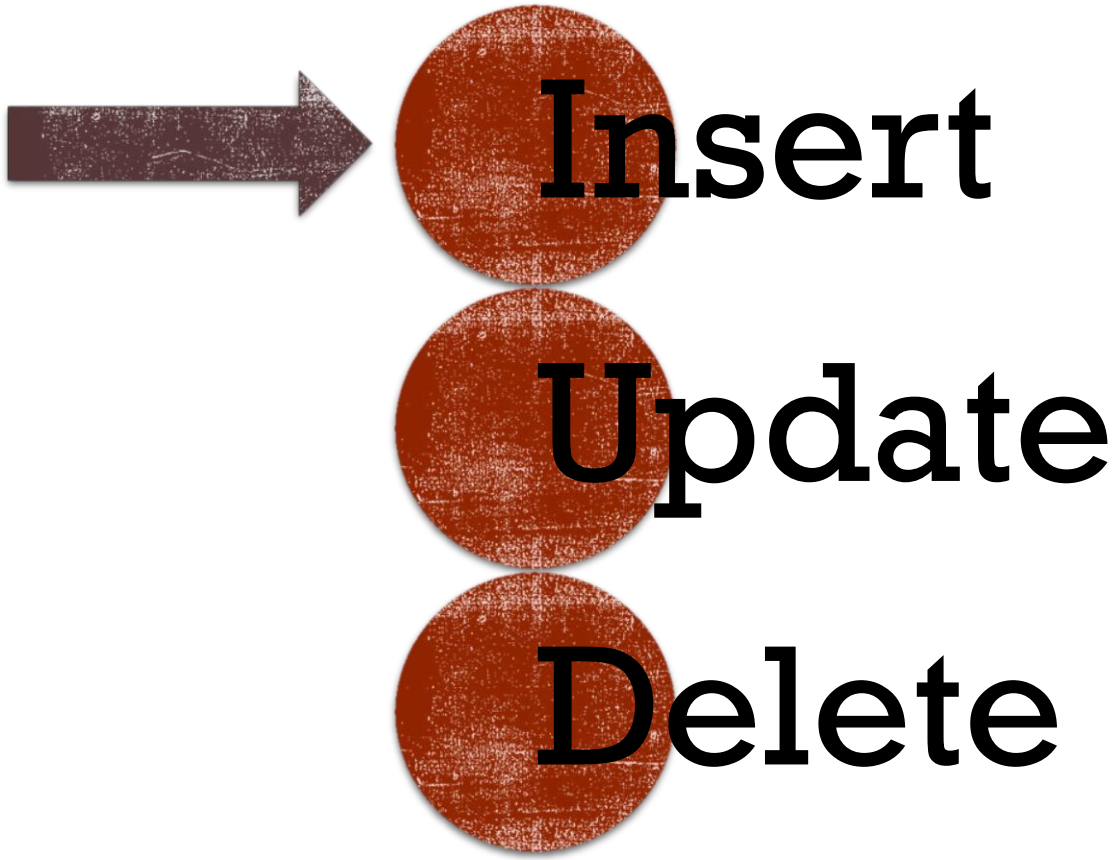
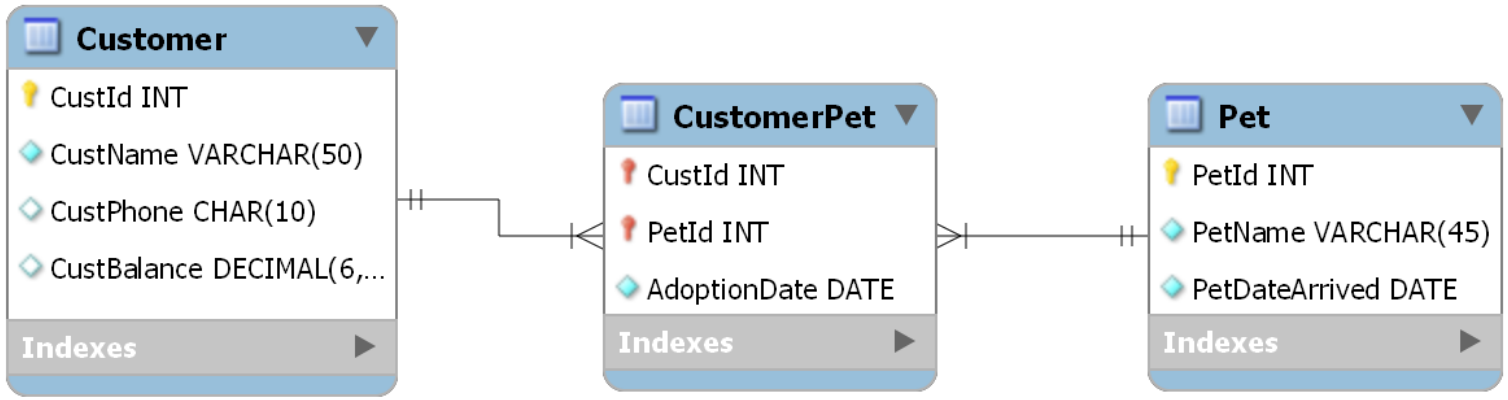
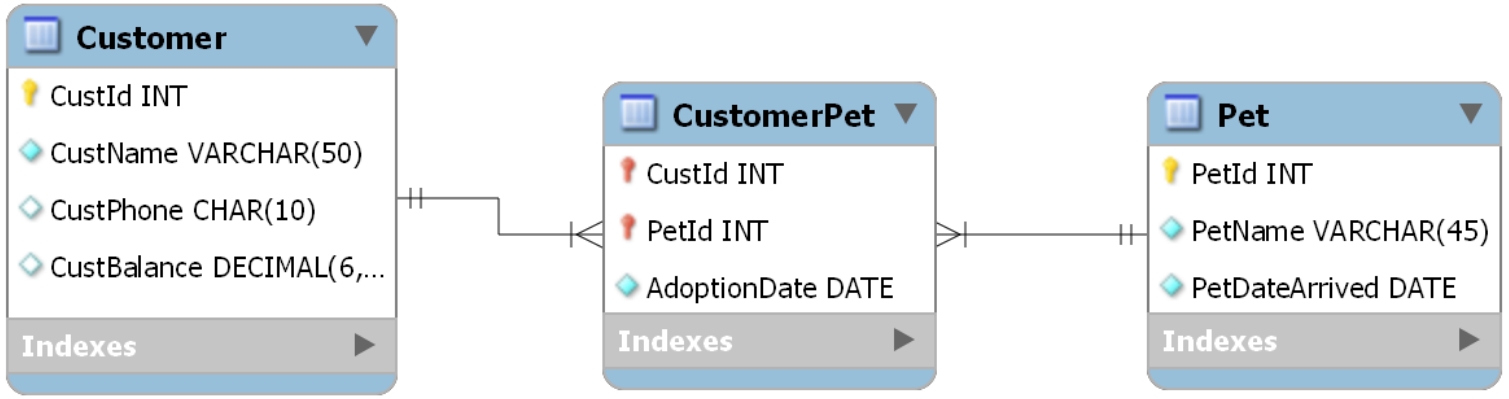
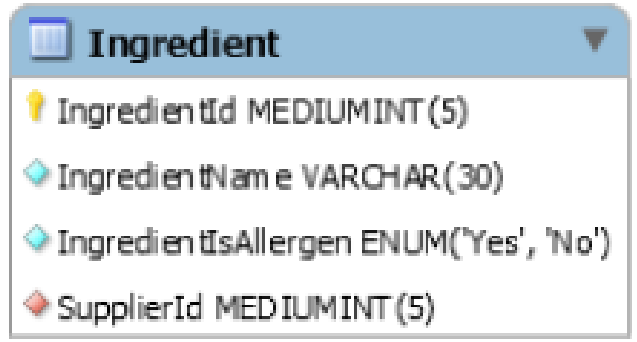
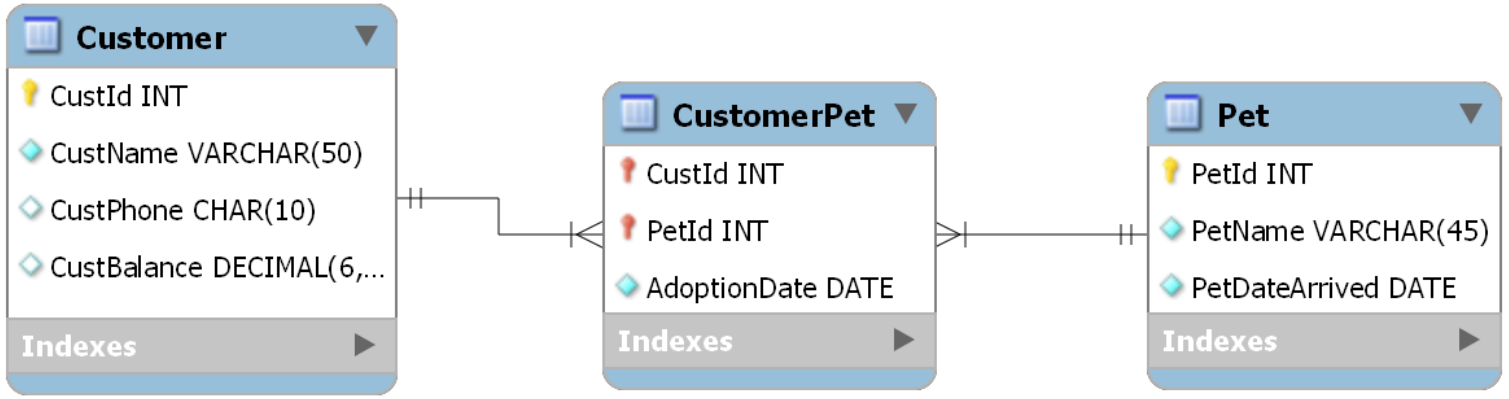
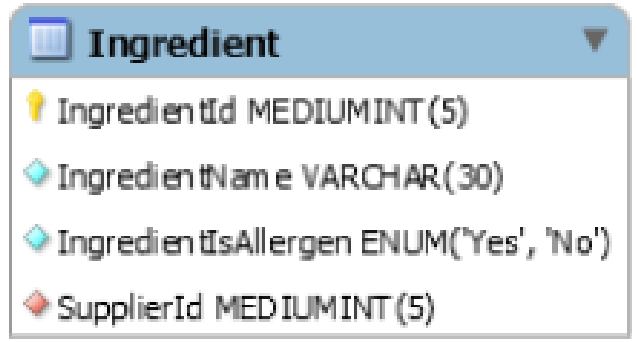
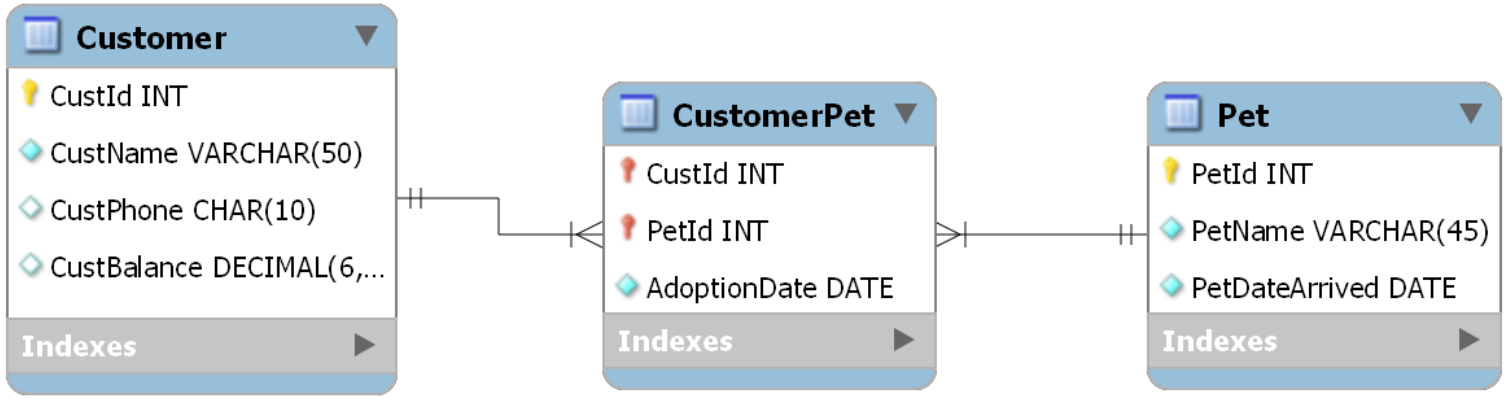
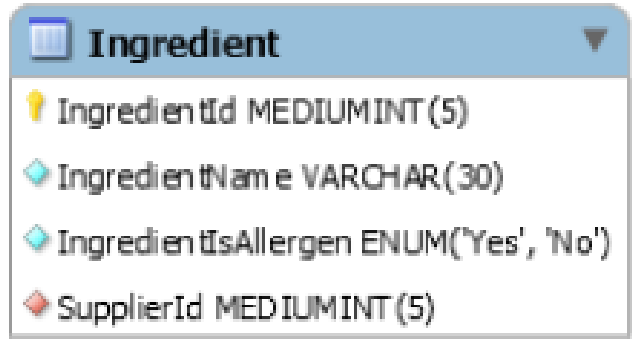
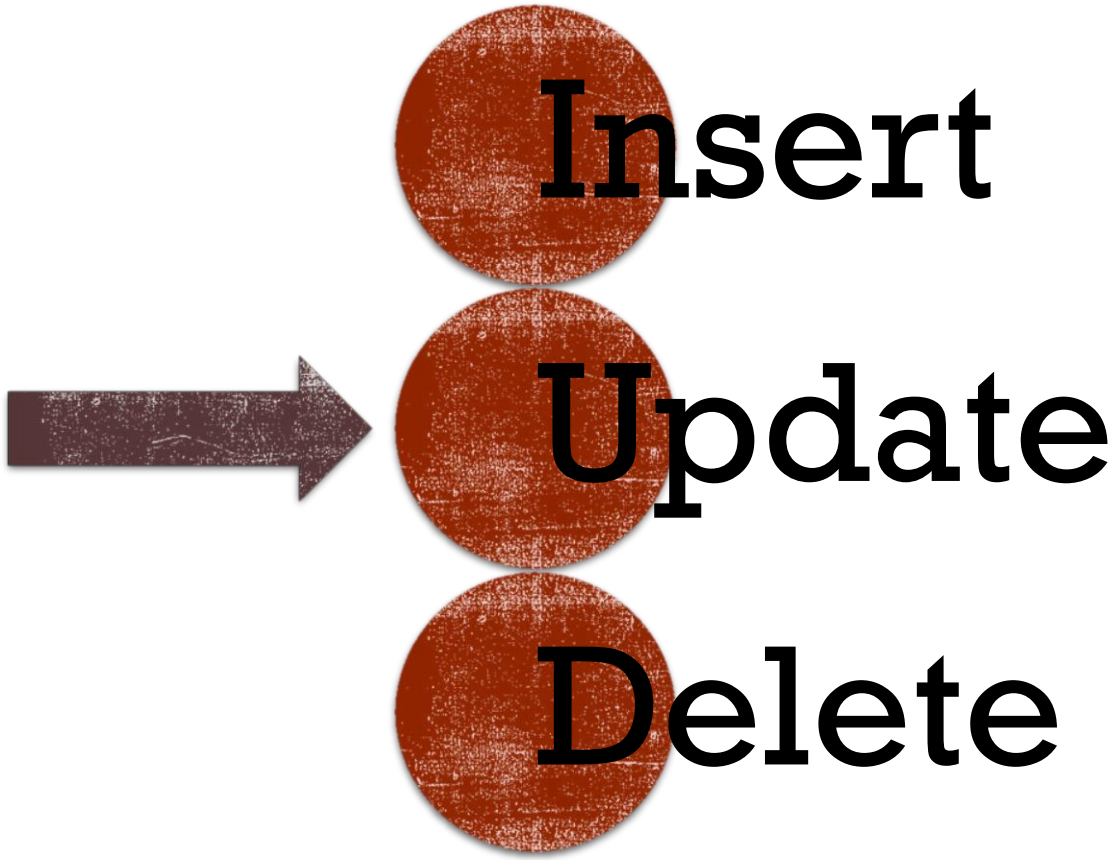
```
SELECT
    [ALL | DISTINCT] select_expr [, select_expr ...]
    [FROM table_references]
    [WHERE filter_condition]
    [ORDER BY {col_name | expr | position} [ASC | DESC], ...]
    [LIMIT row_count]
```



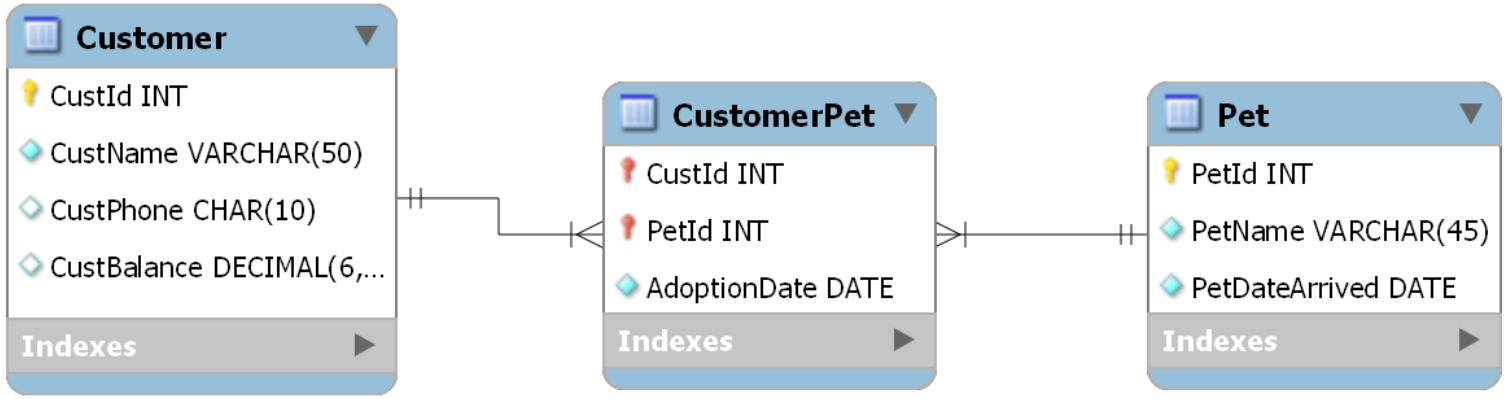
PART I: WORKING WITH DATA

USEFUL TABLE OPERATIONS

- Copy an existing table along with its data (but no constraints, keys, indexes, etc.)
  - ```
    CREATE TABLE new_table_name AS
    SELECT *                          # includes a query; more on this later
    FROM existing_table_name;         # could add WHERE for partial data
    ```
- Delete an existing table along with its data
  - ```
    DROP TABLE table_name;
    ```
- Zap a table's data (but not its structure) with a quick command
  - ```
    TRUNCATE [TABLE] table_name;
    ```


    ```

WORKING WITH DATA



ADDING DATA TO TABLES: INSERT

```
INSERT [INTO] table_name
    [(column_list)]
VALUES (expression1 [, expression2]…) [,
       (expression3...) ]
```

Notes:

- The column list is optional; if you don't supply it, SQL expects the columns in the order in which they are listed in the table definition
- If you use the column list, you don't have to use the same order as the column definitions; just make sure the column order is matched with the data order in the VALUES clause
- Useful keywords: DEFAULT and NULL
  - DEFAULT tells SQL "put in the default or automatic value." This is useful with auto-increment columns and columns where a DEFAULT value has been created in the table schema
  - NULL puts that value into a null-able field; can't use with columns marked Non-Null

PET ADOPTION: IN AN EMPTY DB, WHICH TABLES GET DATA FIRST?

① **Customer**

CustId	CustName	CustPhone	CustBalance
1	Judd Jetson	2068881414	34.45
2	Harriett Hanson	4258882626	0.00
3	Betty Beaumont	2068884747	75.00

② **Pet**

PetId	PetName	PetDateArrived
1	Fluffy	2017-10-13
2	Lucky	2017-10-07
3	Duke	2017-10-31

③ **CustomerPet**

CustId	PetId	AdoptionDate
2	2	2017-11-03
3	1	2017-11-06

Could switch #1 and #2, but CustomerPet must always be last; do you see why?



PET ADOPTION



Customer

CustId	CustName	CustPhone	CustBalance
1	Judd Jetson	2068881414	34.45
2	Harriett Hanson	4258882626	0.00
3	Betty Beaumont	2068884747	75.00

```
INSERT [INTO] table_name
    [(column_list)]
VALUES (expression1 [, expression2]…) [,
        (expression3...) ]
```

- Let's add the first customer using syntax that uses column names
-
```
INSERT Customer
    (CustName, CustBalance, CustPhone, CustId)
VALUES                                  # data order must match
    ('Judd Jetson', 34.45, '2068881414', DEFAULT); # column list order
```
- …but could also be…
-
```
INSERT Customer
    (CustName, CustBalance, CustPhone)  # if CustId is auto-increment
VALUES
    ('Judd Jetson', 34.45, '2068881414');
```

INSERT PRACTICE #1: SINGLE ROW W/COLUMN NAMES

- Insert syntax:
  - `INSERT [INTO] table_name [(column_list)]`
    `VALUES (expression1 [, expression2]…) [,`
    `(expression3 [, expression4]…) ]…`
- In Cathy's Cakes, insert a single row into the Ingredient table, specifying the column names
- Do **not** specify the ingredient id or whether the ingredient is an allergen
- Data:
  - Ingredient name: Maple Syrup
  - Supplier id: 127 (Jakubowski LLC Baking Supply)



PET ADOPTION



Customer

CustId	CustName	CustPhone	CustBalance
1	Judd Jetson	2068881414	34.45
2	Harriett Hanson	4258882626	0.00
3	Betty Beaumont	2068884747	75.00

```
INSERT [INTO] table_name
    [(column_list)]
VALUES (expression1 [, expression2]…) [,
        (expression3...) ]
```

- Let's add the second customer using syntax that uses NO column names
-
```
INSERT Customer
VALUES              # must supply all columns, follow table order
    (DEFAULT, 'Harriett Hanson','4258882626', 0.0);
```
- …but could also be…
-
```
INSERT Customer
VALUES              # works if CustBalance has a DEFAULT of 0.0
    (2, 'Harriett Hanson','4258882626', DEFAULT);
```

INSERT PRACTICE #2: SINGLE ROW W/NO COLUMN NAMES

- Insert syntax:
  - `INSERT [INTO] table_name ~~[(column_list)]~~`
    `VALUES (expression1 [, expression2]…) [,`
    `(expression3 [, expression4]…) ]…`
- Insert a single row into the Ingredient table, **without** specifying the column names
- Data:
  - Ingredient name: Flaked Coconut
  - Allergen? Yes
  - Supplier id: 120



PET ADOPTION



Pet

PetId	PetName	PetDateArrived
1	Fluffy	2017-10-13
2	Lucky	2017-10-07
3	Duke	2017-10-31

```
INSERT [INTO] table_name
    [(column_list)]
VALUES (expression1 [, expression2]…) [,
       (expression3...) ]
```

- Let's add ALL the Pet data in a single statement, without column names
-
```
INSERT Pet
VALUES
    (DEFAULT, 'Fluffy', '2017-10-13'),
    (DEFAULT, 'Lucky',  '2017-10-07'),
    (DEFAULT, 'Duke',   '2017-10-31');
```
- …but could also be…
-
```
INSERT Pet
VALUES
    (1, 'Fluffy','2017-10-13'),
    # and so forth
```

INSERT PRACTICE #3: MULTIPLE ROWS W/NO COLUMN NAMES

- Insert syntax:
  - `INSERT [INTO] table_name [~~(column_list)~~]`
    `VALUES (expression1 [, expression2]…) [,`
    `**(expression3 [, expression4]…)** ]…`
- In a single SQL statement, insert three rows into the Ingredient table, **without** specifying the column names
- Use **valid** data of your choosing
- Don't use the same **supplier** we've used today so far. Don't use the same supplier in all three rows, either



ADDING DATA TO ONE TABLE FROM ANOTHER TABLE

- Concept
  - It's possible to add data to a table *from another table*
  - This one uses a query as part of its work
- Syntax:
  - ```
    INSERT [INTO] table_name [(column_list)]
    select_statement;
    ```
- Example:
  - ```
    INSERT INTO invoice_archive
    SELECT *
    FROM invoices
    WHERE invoice_total - payment_total - credit_total = 0;
    ```

WORKING WITH DATA



CHANGING EXISTING DATA: UPDATE

- Concept
  - If you want to change existing data, use the UPDATE command
  - If you don't specify a WHERE clause, it updates *all* rows
  - MySQL's Safe Update mode won't let you do this unless you use the PK in your WHERE
- Syntax:
  - ```
    UPDATE table_name
    SET columnname1 = expression1 [,
        columnname2 = exprsesion2]...
    WHERE filter_condition;
    ```


    ```

PET ADOPTION

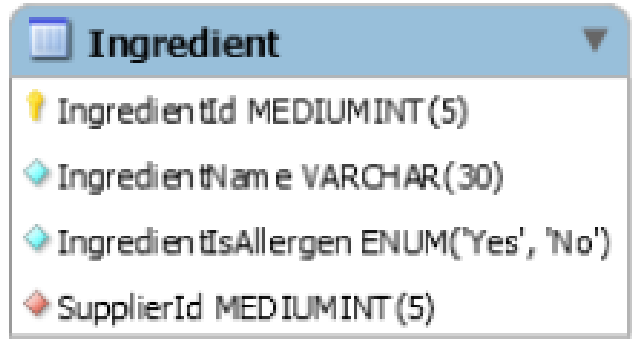


Pet

PetId	PetName	PetDateArrived
1	Fluffy	2017-10-13
2	Lucky	2017-10-07
3	Duke	2017-10-31

```
UPDATE table_name
SET columnname1 = expression1 [,
    columnname2 = exprsesion2]...
WHERE filter_condition;
```

- Duke's arrival date is listed as Oct 31st, but was actually Nov 1st; fix that, being mindful of MySQL's Safe Update Mode
- ```
  UPDATE Pet
  SET PetDateArrived = '2017-11-01'
  WHERE PetId = 3;
  ```


  ```

UPDATE PRACTICE #4

- Update syntax:
  - ```
    UPDATE table_name
    SET columnname1 = expression1 [,
        columnname2 = exprsesion2]...
    WHERE filter_condition
    ```
- In Practice #1 we added the ingredient "Maple Syrup"; it should have been "Maple Syrup, Grade B". Fix that




    ```

UPDATE ON MULTIPLE ROWS

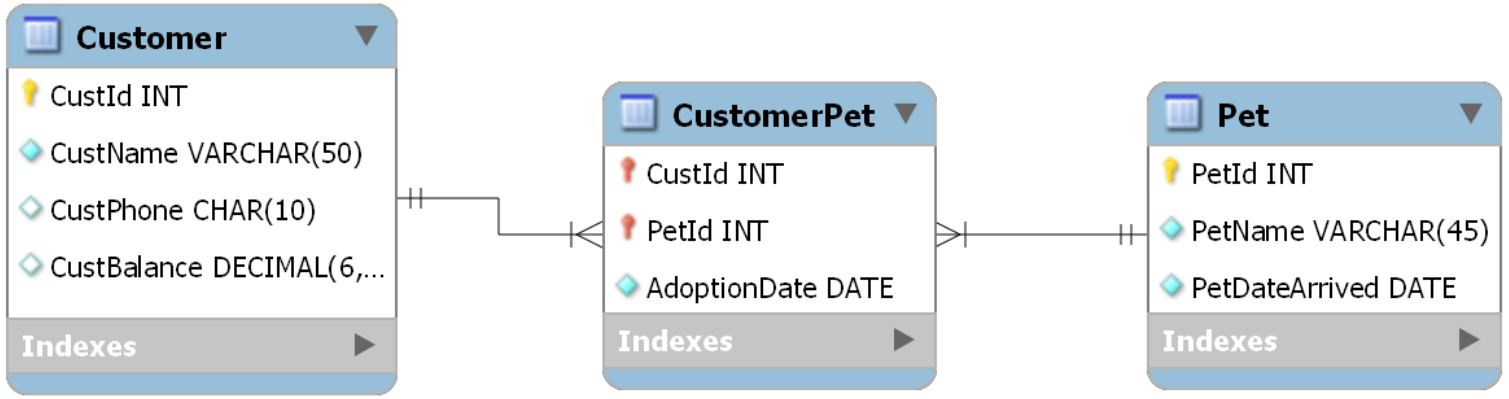
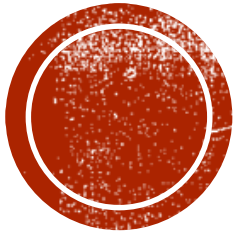
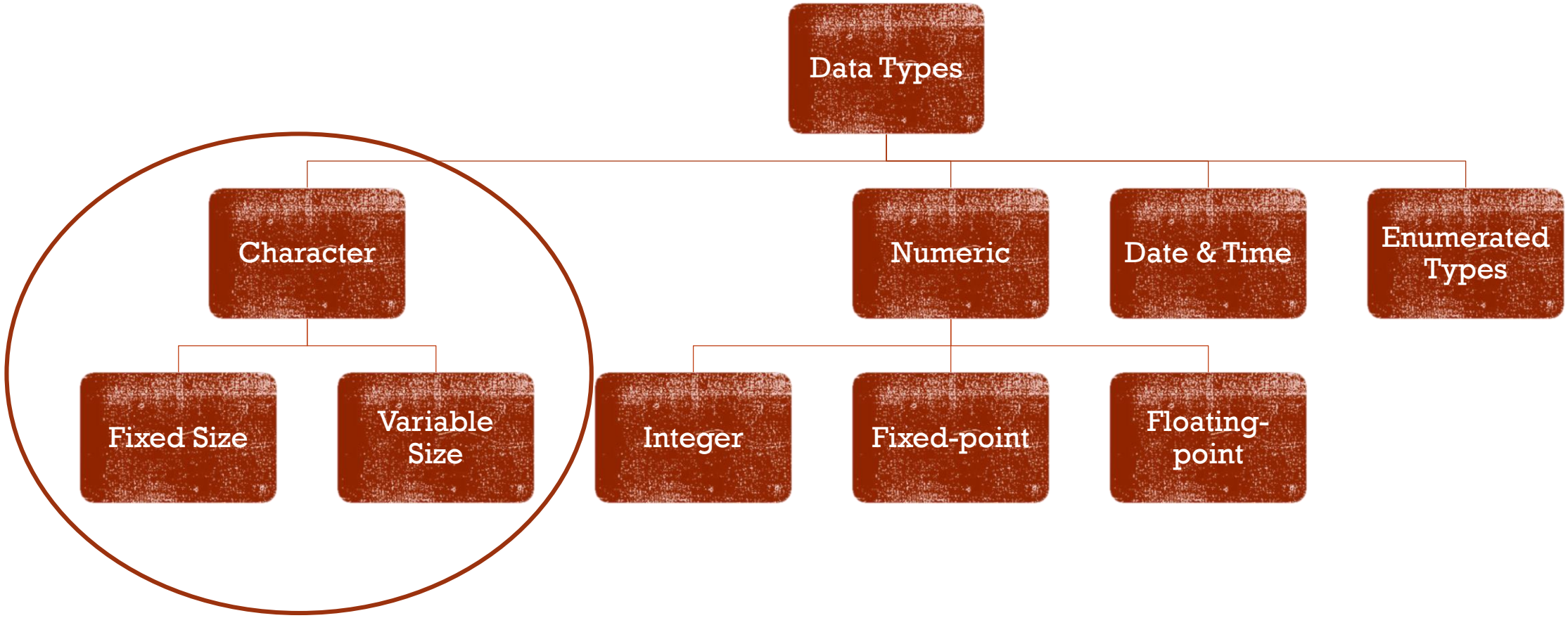
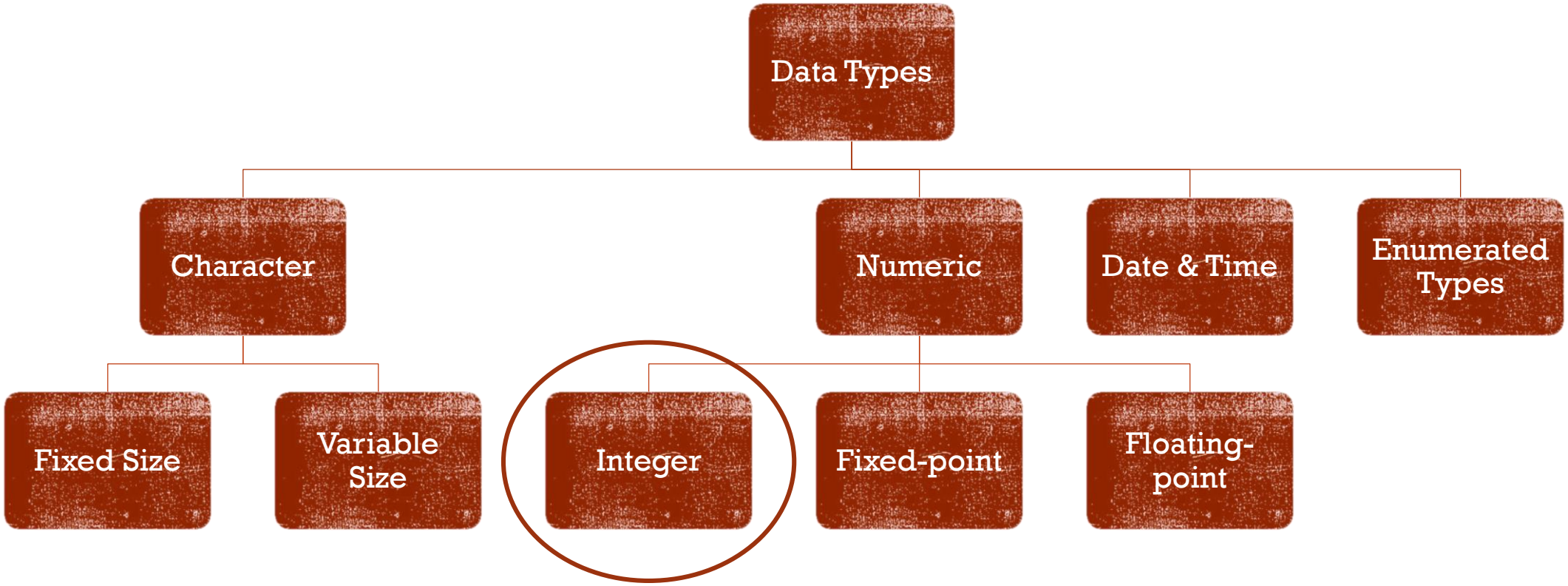
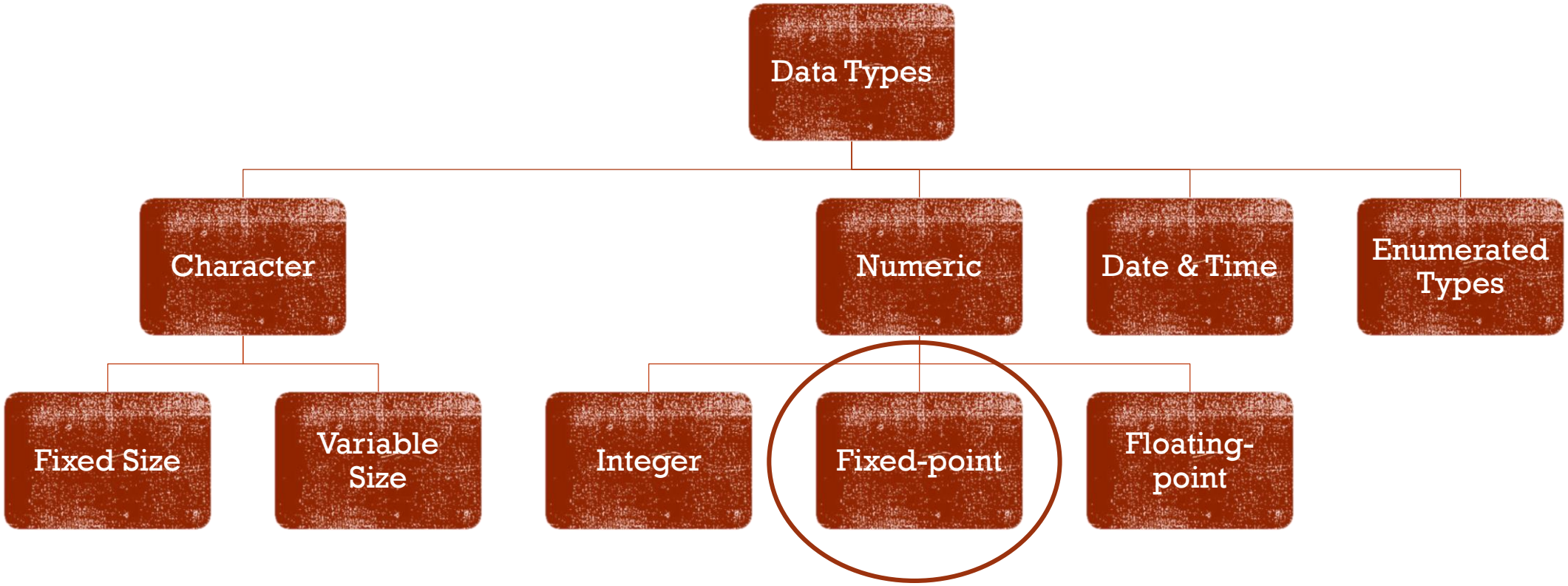
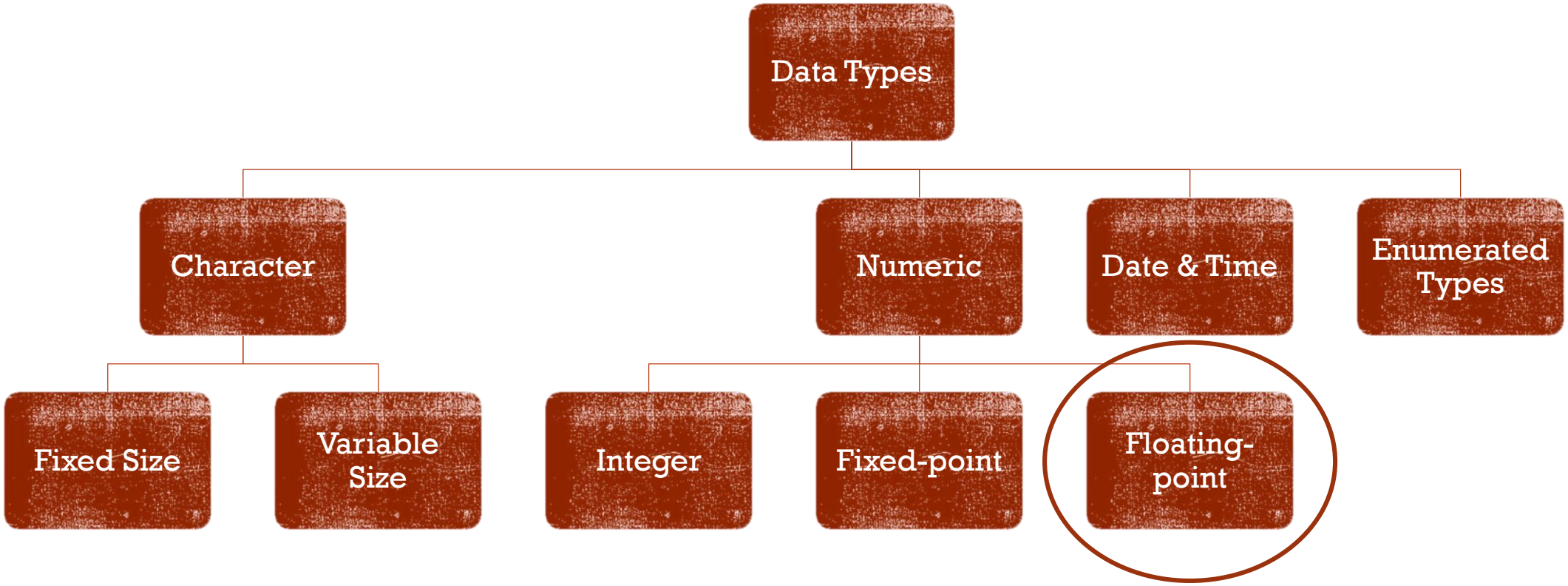
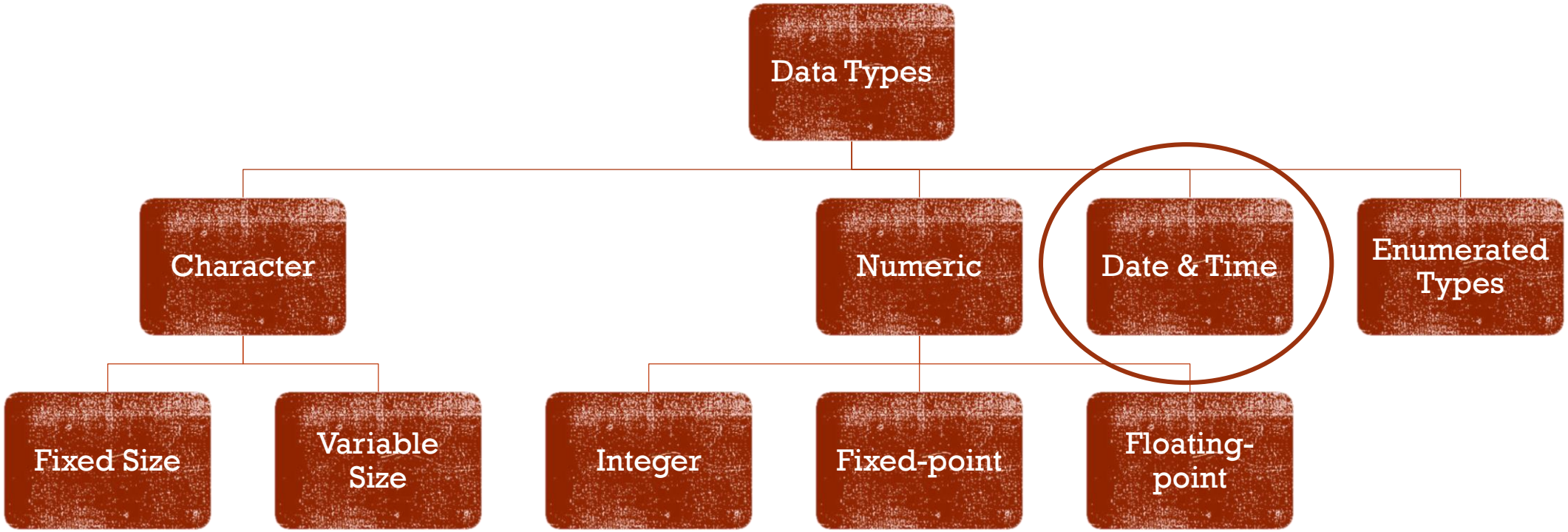
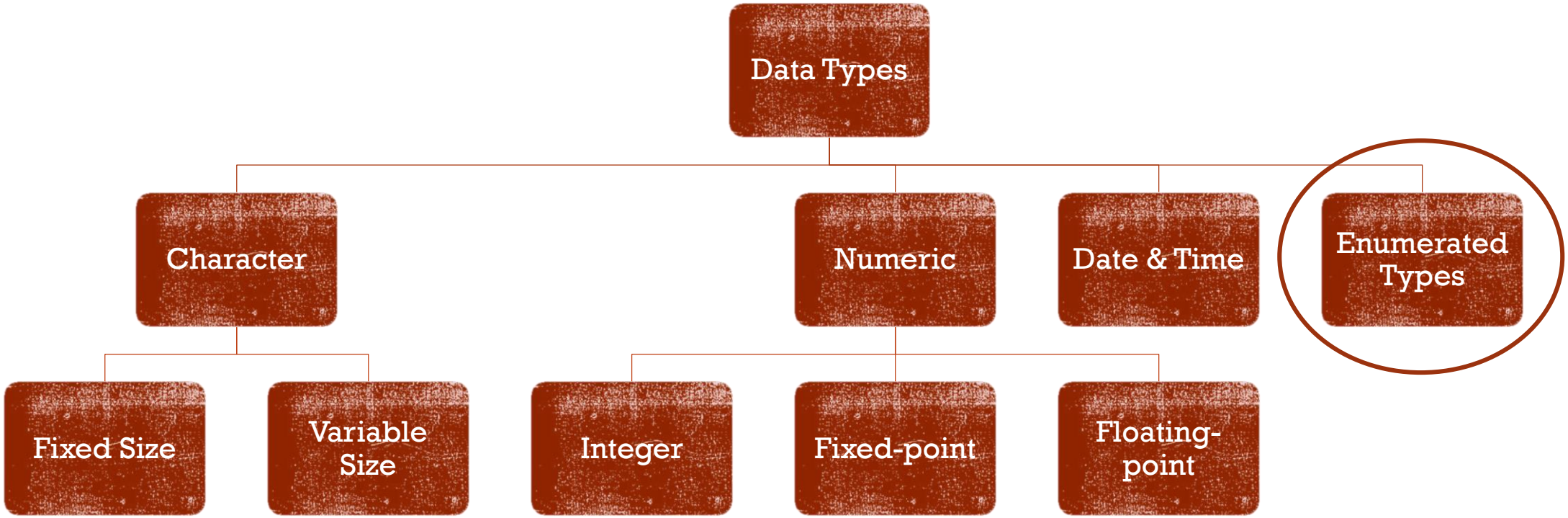
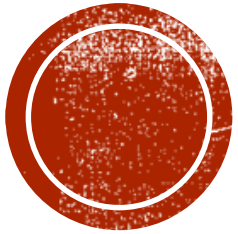
- Concept
  - You can use the update to change multiple rows or all rows; just use a broader WHERE or don't include it at all (if Safe Update Mode is off)
- Examples:
  - ```
    UPDATE Product
    SET ProductPrice = ProductPrice * 1.05;
    ```
  - …but this won't run in MySQL Safe Update Mode, so we need to do this…
  - ```
    UPDATE Product
    SET ProductPrice = ProductPrice * 1.05
    WHERE ProductId >= 0;
    ```

WORKING WITH DATA

- Insert
- Update
- Delete

REMOVE EXISTING ROWS: DELETE

- Concept
  - DELETE lets you remove existing rows from a table
  - Be careful! If you use no WHERE clause, all rows will be deleted (if Safe Update is off)
  - MySQL's Safe Update will refuse to do this unless you have a PK in your WHERE
  - FK constraints may mean SQL refuses to do this for you (e.g., you're trying to delete a row that would make another table's FK invalid)
- Syntax:
  - ```
    DELETE FROM table_name
    [WHERE filter_condition];
    ```


    ```

PET ADOPTION



Customer

CustId	CustName	CustPhone	CustBalance
1	Judd Jetson	2068881414	34.45
2	Harriett Hanson	4258882626	0.00
3	Betty Beaumont	2068884747	75.00

```
DELETE FROM table_name
[WHERE filter_condition];
```

- Add a test customer, then delete that customer to prove you can do so successfully
  - Be mindful of MySQL's Safe Update Mode
-
```
INSERT Customer
VALUES
    (4, 'XXX', NULL, NULL);    # note nullable columns in EER above
```
- …then…
-
```
DELETE FROM Customer
WHERE CustId = 4;
```

What happens to the auto-increment value at this point? Maybe not what you'd guess…

DELETE PRACTICE #5

- Delete syntax:
  - `DELETE FROM table_name`
    `[WHERE filter_condition];`
- In one SQL command, delete all the Ingredient table rows you added today
- Be careful not to delete the whole table's data

REFERENCE: DRI

More on Declarative Referential Integrity:

- https://www.red-gate.com/simple-talk/sql/t-sql-programming/declarative-sql-using-references/
- https://www.jameshbyrd.com/using-declarative-referential-integrity-in-sql-server/
- https://en.wikipedia.org/wiki/Referential_integrity



PART II: DATA TYPES

Reference: MYSQL Data Type Documentation


Data Types
Character
Numeric
Date & Time
Enumerated Types
Fixed Size
Variable Size
Integer
Fixed-point
Floating-point

CHARACTER TYPES

- CHAR(*n*)
  - Always stores *n* characters of data, even for shorter strings
- VARCHAR(*n*)
  - Stores the *actual* characters used plus one byte for the length
- Storage
  - Character storage depends on encoding scheme (e.g., latin1 is 1 byte per character, Unicode is 4 bytes per character); check them out: `SHOW CHARACTER SET`
  - Example below assumes latin1 encoding

Value	CHAR(4)	Storage Required	VARCHAR(4)	Storage Required
''	' '	4 bytes	''	1 byte
'ab'	'ab '	4 bytes	'ab'	3 bytes
'abcd'	'abcd'	4 bytes	'abcd'	5 bytes
'abcdefgh'	'abcd'	4 bytes	'abcd'	5 bytes


Data Types
Character
Numeric
Date & Time
Enumerated Types
Fixed Size
Variable Size
Integer
Fixed-point
Floating-point

INTEGER TYPES

- Numbers without decimal points
- Stored exactly
- By default, can store positive and negative numbers
  - Use UNSIGNED to prevent negatives
- BOOL and BOOLEAN are synonyms for TINYINT(1); that means BOOL isn't as restrictive as we'd like
- Specify default display size in parens, for example INT(4)
  - Has no bearing on *storage*, only *display*

Type	Storage	Minimum Value	Maximum Value
	(Bytes)	(Signed/Unsigned)	(Signed/Unsigned)
TINYINT	1	-128	127
		0	255
SMALLINT	2	-32768	32767
		0	65535
MEDIUMINT	3	-8388608	8388607
		0	16777215
INT	4	-2147483648	2147483647
		0	4294967295
BIGINT	8	-9223372036854775808	9223372036854775807
		0	18446744073709551615


Data Types
Character
Numeric
Date & Time
Enumerated Types
Fixed Size
Variable Size
Integer
Fixed-point
Floating-point

FIXED-POINT TYPES

- DECIMAL allows exact storage of numbers with decimals
  - Great for money; no rounding errors
- The precision is specified after the type, with the scale (digits) afterward
  - Example: DECIMAL(12,2)
  - Maximum precision is 65
- Storage is 4 bytes per 9 decimals; leftovers require a bit more
  - Estimate size using this: precision / 2
- Note: when you specify size in parentheses, you are specifying *precision* and *scale*, for example DECIMAL(5,2) means 5 significant digits with 2 digits after the decimal point. If you then try to insert 1234.56 into this column, you get an error


Data Types
Character
Numeric
Date & Time
Enumerated Types
Fixed Size
Variable Size
Integer
Fixed-point
Floating-point

FLOATING-POINT TYPES

- Stored as *approximations*
  - Take care when comparing; lend themselves to rounding errors
- Can specify precision *bits* hint in parens, e.g., FLOAT(12)
  - Precision 0 to 23 → single-precision (float)
  - Precision 24 to 53 → double-precision (double)
- Can specify display decimals after precision, e.g., FLOAT(5,2)
  - MySQL allows this, but it's non-standard
  - Does limit data entry (1234.56→error) or cause rounding (123.456→123.46)

- "For maximum portability, code requiring storage of approximate numeric data values should use FLOAT or DOUBLE with no specification of precision or number of digits."


Data Types
Character
Numeric
Date & Time
Enumerated Types
Fixed Size
Variable Size
Integer
Fixed-point
Floating-point

DATE/TIME TYPES

- DATE 1/1/1000 to 12/31/9999 (3 bytes)
  - Default format: yyyy-mm-dd
- TIME -838:59:59 to 838:59:59 (3 bytes)
  - Default format: hh:mm:ss
- DATETIME combination of DATE + TIME (8 bytes)
  - Default format: yyyy-mm-dd hh:mm:ss
- TIMESTAMP combination of date & time from midnight 1/1/1970 to 2037 (4 bytes)
  - Default format: yyyy-mm-dd hh:mm:ss
- YEAR 1901 to 2155 (1 byte)

DATE/TIME LITERALS

Literal value	Value stored in DATE column
'2014-08-15'	2014-08-15
'2014-8-15'	2014-08-15
'14-8-15'	2014-08-15
'20140815'	2014-08-15
20140815	2014-08-15
'2014.08.15'	2014-08-15
'14/8/15'	2014-08-15
'8/15/14'	ERROR
'2014-02-31'	ERROR

DATE/TIME LITERALS

Literal value	Value stored in TIME column
'7:32'	07:32:00
'19:32:11'	19:32:11
'193211'	19:32:11
193211	19:32:11
'19:61:11'	ERROR

Literal value	Value stored in DATETIME or TIMESTAMP column
'2014-08-15 19:32:11'	2014-08-15 19:32:11
'2014-08-15'	2014-08-15 00:00:00


Data Types
Character
Numeric
Date & Time
Enumerated Types
Fixed Size
Variable Size
Integer
Fixed-point
Floating-point

ENUM AND SET TYPES

- ENUM lets you specify a number of values (up to 65,535); the cell can contain ONE of them. Uses 1-2 bytes. DEFAULT yields null, if allowed, or first item in list
- SET lets you specify a number of values (up to 64); the cell can contain any COMBINATION of them. Uses 1-8 bytes.

Value	Stored in column ENUM ('Yes', 'No', 'Maybe')
'Yes'	'Yes'
'No'	'No'
'Maybe'	'Maybe'

Value	Stored in column SET ('Pepperoni', 'Mushrooms', 'Olives')
'Pepperoni'	'Pepperoni'
'Mushrooms'	'Mushrooms'
'Pepperoni, Bacon'	'Pepperoni'
'Olives, Pepperoni'	'Pepperoni, Olives'



PART III: CONVERSIONS

IMPLICIT CONVERSIONS

- SQL will *implicitly* convert on your behalf when it needs to
- Examples:
  - `SELECT CONCAT('$', 436.59);  # Result:  $436.59, a string`
  - `SELECT 5346/'5346-43-x';  # Result:  1, an integer`
  - `SELECT '2016-01-21' + 1;  # Result:  20160122, an integer`

EXPLICIT CONVERSIONS

- If you want to convert *explicitly*, use CAST or CONVERT
  - CAST is an ANSI standard; you may see it more
- Types you can cast/convert to:
  - CHAR[(n)]
  - DATE, DATETIME, TIME
  - SIGNED [INTEGER], UNSIGNED [INTEGER]
  - DECIMAL[(M[,D])]
- Examples:
  - ```
    SELECT CAST(invoice_date AS CHAR(10))              # note the 'as'
    FROM invoices;
    ```
  - ```
    SELECT CONVERT(invoice_total, SIGNED INTEGER)      # no 'as', just a comma
    FROM invoices;
    ```

USEFUL STRING CONVERSIONS

- FORMAT converts numbers to comma-delimited strings. Examples:
  - `SELECT FORMAT(1234.567, 2);  # Result:  1,234.57`
  - `SELECT FORMAT(1234.567, 0);  # Result:  1,235`
- CHAR converts numbers into binary strings, mostly used to convert to common control characters:
  - CHAR(9) = Tab, CHAR(10)= Line feed, CHAR(13) = Carriage Return
  - Example:
  - ```
    SELECT CONCAT(vendor_name, CHAR(13,10), vendor_address1, CHAR(13,10),
        vendor_city, ', ', vendor_state, '  ', vendor_zip_code)
    FROM vendors;
    ```

US Postal Service
Attn: Supt. Window Services
Madison, WI 53707

Note: output in some front-ends won't show special characters, e.g., WorkBench results won't, a command window will


    ```

WHAT SHOULD I DO NEXT?

- Take the quiz on Chapters 5 and 8
- Start Proj04
  - Insert, update, delete
  - Constraint violations
  - Safe Update Mode
  - Data types
- Before we meet again…
  - Submit Proj04
  - Read next week's material: Chapter 10, Designing a Database
  - Start thinking about the Midterm Exam; it's not that far away!

QUESTIONS?