Course Overview

- This course focuses on the relational database model from a beginning perspective.
- Many IT professionals have learned what they know about databases through rumor, trial-and-error, and painful experience.
- Over the years, some develop an intuitive feel for what makes a good database design but they may still not understand the reasons why a design is good or bad, and they may leave behind a trail of rickety, poorly constructed programs built on shaky database foundations.

- The course provides the tools students need to design a database.
  - It explains how to determine what should go in a database and how a database should be organized to ensure data integrity and a reasonable level of performance.
  - It explains techniques for designing a database that is strong enough to store data safely and consistently, flexible enough to allow the application to retrieve the data it needs quickly and reliably, and adaptable enough to accommodate a realistic amount of change.

Introduction to Databases

- What a Database Management System (DBMS) provides for applicactions?
- It provides a means of handling large amounts of data.

Introduction to Databases

- Database Management System (DBMS) provides….

  . . . efficient, reliable, convenient, and safe, multi-user storage of and access to massive amounts of persistent data.

- DBMSs are extremely common in the prevalent in the world today.

- They sit behind many websites running your banking systems, your telecommunications, scientific experiments, and much, much more.

- Massive – *terabytes*
- Persistent – *data in the DB outlives the programs that execute on that data*
- Safe – *since database systems run critical applicatons, such as e-commerce, telecommunications, and banking systems, they need to have guarantees that the data managed by the system will stay in a consistent state, it won't be lost or overwritten in case of failures:*
  - *hardware, software, power, malicious users trying to corrupt data*
  - *So, database systems have a number of built-in mechanisms ensuring the data remains consistent, regardless of what happens*
- Multi-user – *concurrency control*
- Convenient – database systems are designed to make it easy to work with large amounts of data and to do very powerful processing on that data.
  - *Physical Data Independence*
  - *HIgh-level Query Languages*
- Efficient – *thousands of queries per second*
- Reliable – 99.9999% uptime

- Database applications may be programmed via “frameworks”
  - Django
  - Ruby on Rails

- DBMS may run in conjunction with “middleware”

- Data-intensive applications may not use DBMS at all
  - Files, excel spreadsheets

- We are going to focus on
  - the DBMS itself
  - storing and operating data through a DBMS.

- Using an envelope of business cards is useful as long as it doesn’t contain too many cards.
  - You can find a particular piece of data (for example, a person’s phone number) by looking through all of the cards.
  - The database is easy to expand by shoving more cards into the envelope, at least up to a point. If you have more than a dozen or so business cards, finding a particular card can be time consuming.
  - You can even rearrange the cards a bit to improve performance for cards you use often. Each time you use a card, move it to the front of the pile. Over time, those that are used most will be in front.

- A notebook is small, easy to use, easy to carry, doesn't require electricity, and doesn't need to boot before you can use it.
  - A notebook database is also easily extensible because you can buy another notebook to add to your collection when the first one is full.
  - However, a notebook's contents are arranged sequentially.
  - If you want to find information about a particular topic, you'll have to look through the pages one at a time until you find what you want.
  - The more data you have, the harder this kind of search becomes.

- A filing cabinet can store a lot more information than a notebook and you can easily expand the database by adding more files or cabinets.
  - Finding a particular piece of information in the filing cabinet can be easier than finding it in a notebook as long as you are searching for the type of data used to arrange the records.
  - If the filing cabinet is full of customer information sorted by customer name, and you want to find a particular customer’s data, you’re in luck. If you want to find all of the customers that live in a certain city, you’ll have to dig through the files one at a time.

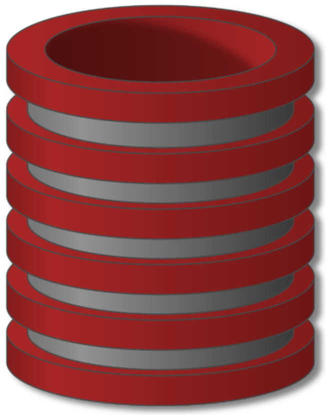
Key Concepts

- Data model
  - description of how the data is structured
  - one of the most common data models is the relational data model
  - in the relational data model, the database is thought to of as a set of records
- Schema versus data
  - schema is the structure of the database
  - data is the actual data stored within the schema
- Data definition language (DDL)
  - To set up the schema
- Data manipulation or query language (DML)
  - To query and modify data

Key People

- DBMS implementer
  - Builds system
- Database designer
  - Establishes schema for a database
  - figures out how to structure data before building an application
- Database application developer
  - Write programs that operate on database
- Database administrator
  - loads data
  - keeps running smoothly

Whether you know it or not, you're using a database every day



Goals of Effective Database Design

- Using modern database tools, just about anyone can build a database. The question is, will the resulting database be useful?
  - A database won't do you much good if you can't get data out of it quickly, reliably, and consistently.
  - It won't be useful if it's full of incorrect or contradictory data.
  - It also won't be useful if it is stolen, lost, or corrupted by data that was only half written when the system crashed.

- You can address all of these potential problems by using modern database tools, a good database design, and a pinch of common sense, but only if you understand what those problems are so you can avoid them.

- Building an application is often compared to building a house or skyscraper.
  - You probably wouldn't start building a multibillion dollar skyscraper without a comprehensive design that is based on well-established architectural principles.
- Unfortunately software developers often rush off to start coding as soon as they possibly can.
  - Coding is more fun and interesting than design is.
  - Coding also lets developers tell management and customers how many lines of code they have written so it seems like they are making progress even if the lines of code are corrupted by false assumptions.
- Only later do they realize that the underlying design is flawed, the code they wrote is worthless, and the project is in serious trouble.

- Now back to database design.
- Few parts of an application’s design are as critical as the database’s design.
- The database is the repository of the information that the rest of the application manages and displays to the users.
- If the database doesn’t store the right data, doesn’t keep the data safe, or doesn’t let the application find the data it needs, then the application has little chance for success.
- Here the GIGO (Garbage In, Garbage Out) principle is in full effect.
- If the underlying data is unsound, it doesn’t matter what the application that uses it does; the results will be suspect at best.

- For example, imagine that you've built an order tracking system that can quickly fetch information about a customer's past orders.
  - Unfortunately every time you ask the program to fetch a certain customer's records it returns a slightly different result.
  - Though the program can find data quickly, the results are not trustworthy enough to be usable.

- Or imagine that you have built an amazing program that can track the thousands of tasks that make up a single complex job such as building a cruise liner or passenger jet. It can track each task's state of completion, determine when you need to order new parts for them to be ready for future phases of construction, and can even determine the present value of future purchases so you can decide whether it is better to buy parts now or wait until they are needed.
  - Unfortunately the program takes hours to recalculate the complex task schedule and pricing details.
  - Though the calculations are correct, they are so slow that users cannot reasonably make any changes. Changing the color of the fabric of a plane's seats or the tile used in a cruise liner's hallways could delay the whole project.

- Or suppose you have built an efficient subscription application that lets customers subscribe to your company's quarterly newsletters and data services.
  - It lets you quickly find and update any customer's subscriptions and it always shows the same values for a particular customer consistently.
  - Unfortunately, when you change the price of one of your publications you find that not all of the customers' records show the updated price.
  - Some customers' subscriptions are at the new rate, some are at the old rate, and some seem to be at a rate you've never seen before.
- Some systems allow you to offer sale prices or special incentives to groups of customers, or they allow sales reps to offer special prices to particular customers.
- That kind of system requires careful design if you want to be able to do things like change standard prices without messing up customized pricing.

- Poor database design can lead to these and other annoying and potentially expensive scenarios.
- A good design creates a solid foundation on which you can build the rest of the application.
- Experienced developers know that the longer a bug remains in a system the harder it is to find and fix.
- From that it logically follows that it is extremely important to get the design right before you start building on top of it.
- Database design is no exception.
- A flawed database design can doom a project to failure before it has begun as surely as ill-conceived software architecture, poor implementation, or incompetent programming can.

Relational Databases

- Used by all major commercial database systems
- Very simple model
- Query with high-level languages: simple yet expressive
- Efficient implementations

Schema – structural description of relations in database

Instance – actual contents at given point in time

Student

ID	name	GPA	photo
123	Amy	3.9	🙂
234	Bob	3.4	NULL
345	Craig	NULL	😐
	⋮		

College

name	state	enr
Stanford	CA	15,000
Berkeley	CA	36,000
MIT	MA	10,000
	⋮	

Database = set of named **relations** (or **tables**)
Each relation has a set of named **attributes** (or **columns**)
Each **tuple** (or **row**) has a value for each attribute
Each attribute has a **type** (or **domain**)

Student

ID	name	GPA	photo
123	Amy	3.9	🙂
234	Bob	3.4	NULL
345	Craig	NULL	😐
	⋮		

College

name	state	enr
Stanford	CA	15,000
Berkeley	CA	36,000
MIT	MA	10,000
	⋮	

NULL – special value for "unknown" or "undefined"

Student

ID	name	GPA	photo
123	Amy	3.9	
234	Bob	3.4	NULL
345	Craig	NULL	
	⋮		

College

name	state	enr
Stanford	CA	15,000
Berkeley	CA	36,000
MIT	MA	10,000
	⋮	

Key – attribute whose value is unique in each tuple

Or set of attributes whose combined values are unique

Student

ID	name	GPA	photo
123	Amy	3.9	☺
234	Bob	3.4	NULL
345	Craig	NULL	😐
	⋮		

College

name	state	enr
Stanford	CA	15,000
Berkeley	CA	36,000
MIT	MA	10,000
	⋮	

Tables

- Tables are the basic structure where data is stored in the database.
- Given that in most cases, there is no way for the database vendor to know ahead of time what your data storage needs are, chances are that you will need to create tables in the database yourself.
- Many database tools allow you to create tables without writing SQL, but given that tables are the container of all the data, it is important to include the **CREATE TABLE** syntax in this lecture.

What goes into a table?

- Tables are divided into rows and columns.
- Each row represents one piece of data, and each column can be thought of as representing a component of that piece of data.
- So, for example, if we have a table for recording customer information, then the columns may include information such as First Name, Last Name, Address, City, Country, Birth Date, and so on.
- As a result, when we specify a table, we include the column headers and the data types for that particular column.

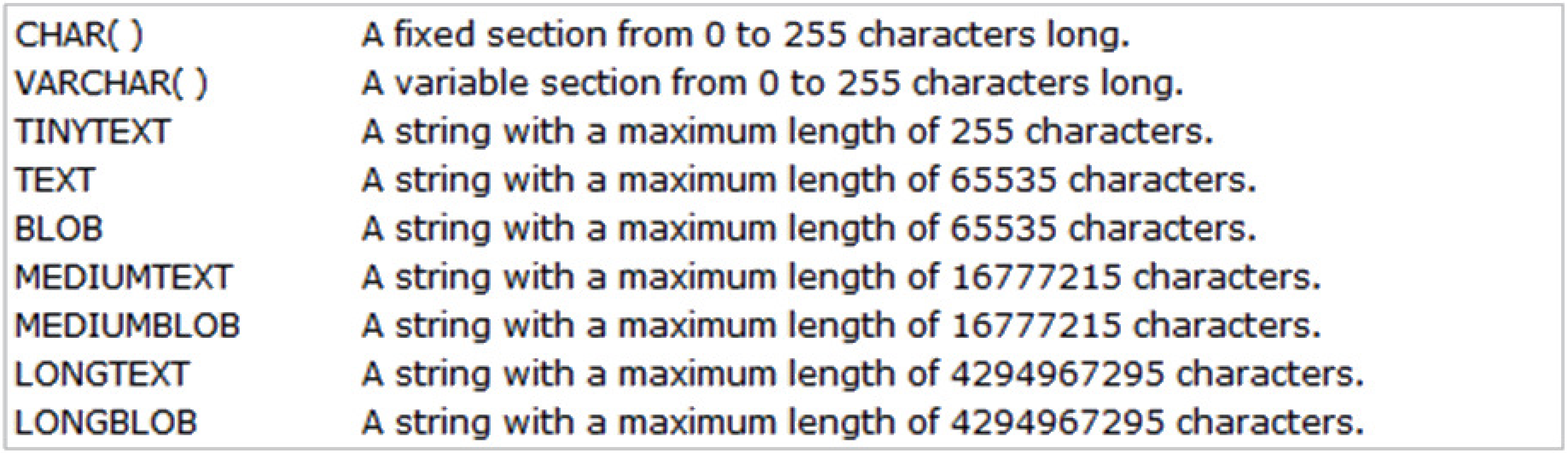
What are data types?

- Typically, data comes in a variety of forms. It could be
  - an integer (such as 1),
  - a real number (such as 0.55),
  - a string (such as 'sql'),
  - a date/time expression (such as '2000-JAN-25 03:22:22'),
  - or even in binary format.

- When we specify a table, we need to specify the data type associated with each column (i.e., we will specify that 'First Name' is of type char(50) - meaning it is a string with 50 characters).

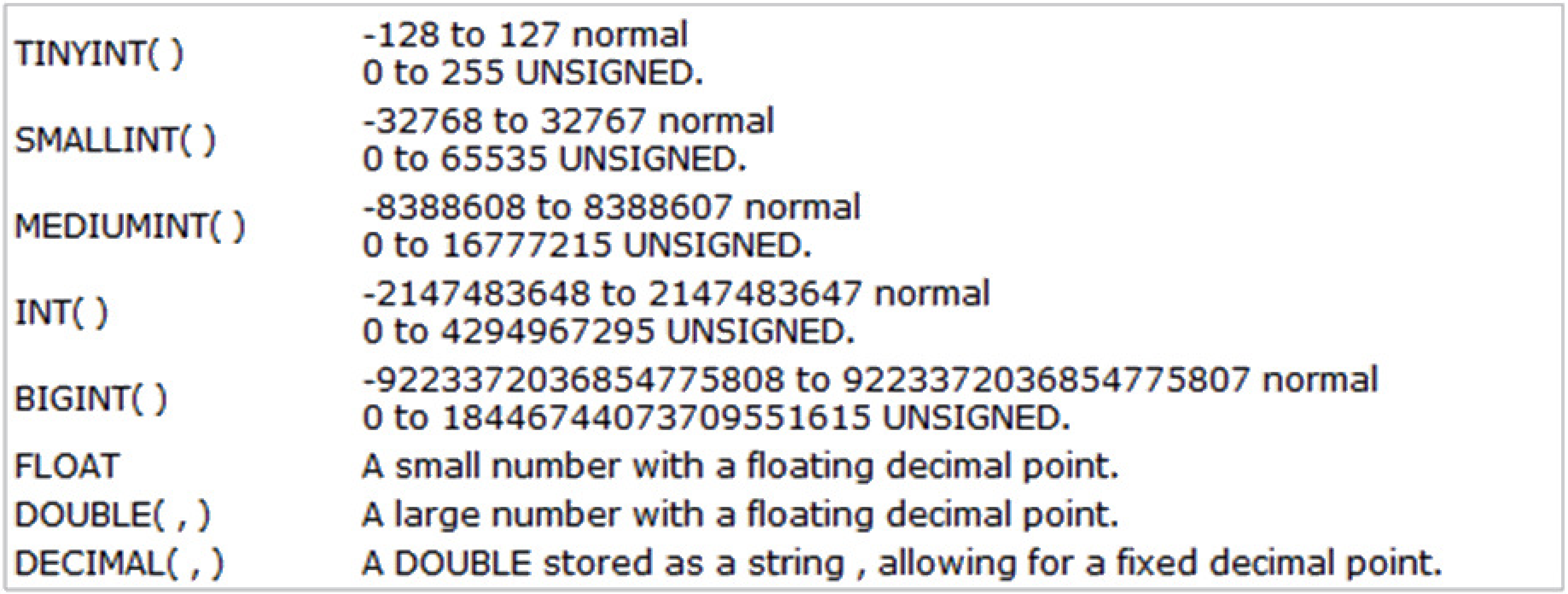
- One thing to note is that different relational databases allow for different data types, so it is wise to consult with a database-specific reference first.

MySQL Text Types



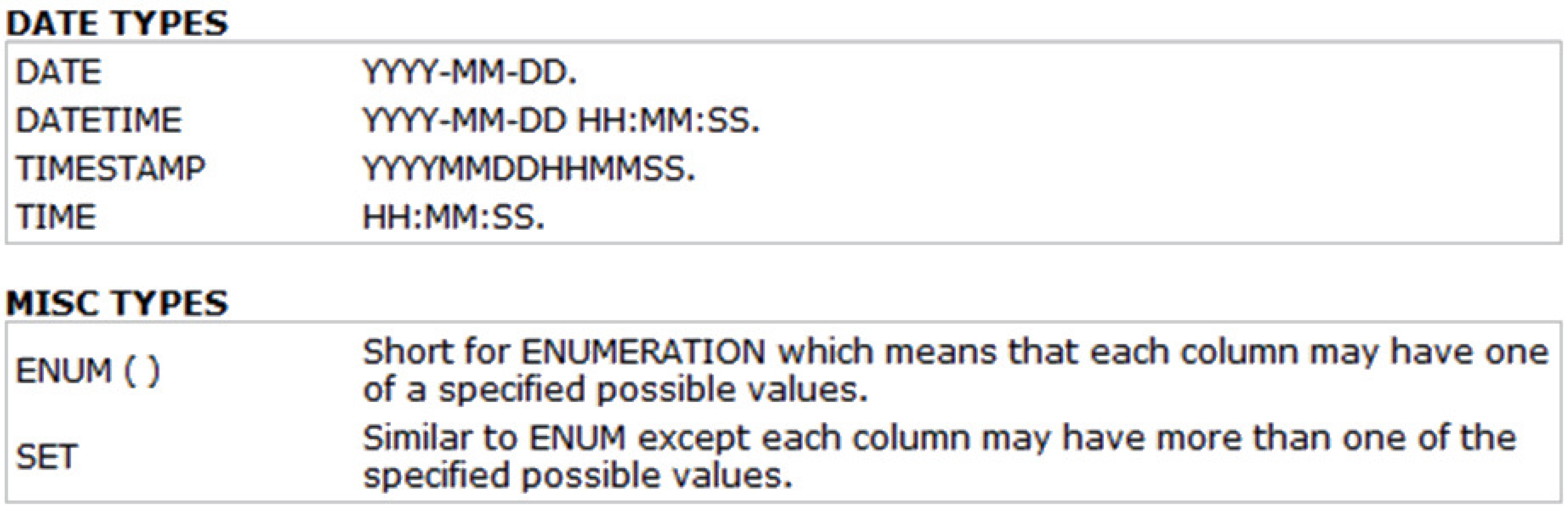
CHAR( )	A fixed section from 0 to 255 characters long.
VARCHAR( )	A variable section from 0 to 255 characters long.
TINYTEXT	A string with a maximum length of 255 characters.
TEXT	A string with a maximum length of 65535 characters.
BLOB	A string with a maximum length of 65535 characters.
MEDIUMTEXT	A string with a maximum length of 16777215 characters.
MEDIUMBLOB	A string with a maximum length of 16777215 characters.
LONGTEXT	A string with a maximum length of 4294967295 characters.
LONGBLOB	A string with a maximum length of 4294967295 characters.

MySQL Number Types



TINYINT( )	-128 to 127 normal 0 to 255 UNSIGNED.
SMALLINT( )	-32768 to 32767 normal 0 to 65535 UNSIGNED.
MEDIUMINT( )	-8388608 to 8388607 normal 0 to 16777215 UNSIGNED.
INT( )	-2147483648 to 2147483647 normal 0 to 4294967295 UNSIGNED.
BIGINT( )	-9223372036854775808 to 9223372036854775807 normal 0 to 18446744073709551615 UNSIGNED.
FLOAT	A small number with a floating decimal point.
DOUBLE(,)	A large number with a floating decimal point.
DECIMAL(,)	A DOUBLE stored as a string , allowing for a fixed decimal point.

MySQL Date and Misc Types



DATE TYPES

DATE	YYYY-MM-DD.
DATETIME	YYYY-MM-DD HH:MM:SS.
TIMESTAMP	YYYYMMDDHHMMSS.
TIME	HH:MM:SS.

MISC TYPES

ENUM ( )	Short for ENUMERATION which means that each column may have one of a specified possible values.
SET	Similar to ENUM except each column may have more than one of the specified possible values.

Storage Requirements for Numeric Types

Data Type	Storage Required
TINYINT	1 byte
SMALLINT	2 bytes
MEDIUMINT	3 bytes
INT, INTEGER	4 bytes
BIGINT	8 bytes
FLOAT(*p*)	4 bytes if 0 <= *p* <= 24, 8 bytes if 25 <= *p* <= 53
FLOAT	4 bytes
DOUBLE [PRECISION], REAL	8 bytes
DECIMAL(*M*,*D*), NUMERIC(*M*,*D*)	Varies; see following discussion
BIT(*M*)	approximately (*M*+7)/8 bytes

Storage Requirements for Date and Time Types

Data Type	Storage Required
DATE	3 bytes
TIME	3 bytes
DATETIME	8 bytes
TIMESTAMP	4 bytes
YEAR	1 byte

Storage Requirements for String Types

In the following table, *M* represents the declared column length in characters for nonbinary string types and bytes for binary string types. *L* represents the actual length in bytes of a given string value.

Data Type	Storage Required
`CHAR(M)`	$M \times w$ bytes, $0 <= M <= 255$, where w is the number of bytes required for the maximum-length character in the character set
`BINARY(M)`	M bytes, $0 <= M <= 255$
`VARCHAR(M)`, `VARBINARY(M)`	$L + 1$ bytes if column values require 0 – 255 bytes, $L + 2$ bytes if values may require more than 255 bytes
`TINYBLOB`, `TINYTEXT`	$L + 1$ bytes, where $L < 2^{8}$
`BLOB`, `TEXT`	$L + 2$ bytes, where $L < 2^{16}$
`MEDIUMBLOB`, `MEDIUMTEXT`	$L + 3$ bytes, where $L < 2^{24}$
`LONGBLOB`, `LONGTEXT`	$L + 4$ bytes, where $L < 2^{32}$
`ENUM('value1','value2',...)`	1 or 2 bytes, depending on the number of enumeration values (65,535 values maximum)
`SET('value1','value2',...)`	1, 2, 3, 4, or 8 bytes, depending on the number of set members (64 members maximum)

Variable-length string types are stored using a length prefix plus data. The length prefix requires from one to four bytes depending on the data type, and the value of the prefix is *L* (the byte length of the string). For example, storage for a `MEDIUMTEXT` value requires *L* bytes to store the value plus three bytes to store the length of the value.

The SQL syntax for **CREATE TABLE**

```
CREATE TABLE "table_name"
  ("column 1" "data_type_for_column_1",
  "column 2" "data_type_for_column_2",
  ... )
```

Reserved Words in MySQL 5.5.28

ACCESSIBLE	ADD	ALL	LOCALTIME	LOCALTIMESTAMP	LOCK
ALTER	ANALYZE	AND	LONG	LONGBLOB	LONGTEXT
AS	ASC	ASENSITIVE	LOOP	LOW_PRIORITY	MASTER_SSL_VERIFY_SERVER_CERT
BEFORE	BETWEEN	BIGINT	MATCH	MAXVALUE	MEDIUMBLOB
BINARY	BLOB	BOTH	MEDIUMINT	MEDIUMTEXT	MIDDLEINT
BY	CALL	CASCADE	MINUTE_MICROSECOND	MINUTE_SECOND	MOD
CASE	CHANGE	CHAR	MODIFIES	NATURAL	NOT
CHARACTER	CHECK	COLLATE	NO_WRITE_TO_BINLOG	NULL	NUMERIC
COLUMN	CONDITION	CONSTRAINT	ON	OPTIMIZE	OPTION
CONTINUE	CONVERT	CREATE	OPTIONALLY	OR	ORDER
CROSS	CURRENT_DATE	CURRENT_TIME	OUT	OUTER	OUTFILE
CURRENT_TIMESTAMP	CURRENT_USER	CURSOR	PRECISION	PRIMARY	PROCEDURE
DATABASE	DATABASES	DAY_HOUR	PURGE	RANGE	READ
DAY_MICROSECOND	DAY_MINUTE	DAY_SECOND	READS	READ_WRITE	REAL
DEC	DECIMAL	DECLARE	REFERENCES	REGEXP	RELEASE
DEFAULT	DELAYED	DELETE	RENAME	REPEAT	REPLACE
DESC	DESCRIBE	DETERMINISTIC	REQUIRE	RESIGNAL	RESTRICT
DISTINCT	DISTINCTROW	DIV	RETURN	REVOKE	RIGHT
DOUBLE	DROP	DUAL	RLIKE	SCHEMA	SCHEMAS
EACH	ELSE	ELSEIF	SECOND_MICROSECOND	SELECT	SENSITIVE
ENCLOSED	ESCAPED	EXISTS	SEPARATOR	SET	SHOW
EXIT	EXPLAIN	FALSE	SIGNAL	SMALLINT	SPATIAL
FETCH	FLOAT	FLOAT4	SPECIFIC	SQL	SQLEXCEPTION
FLOAT8	FOR	FORCE	SQLSTATE	SQLWARNING	SQL_BIG_RESULT
FOREIGN	FROM	FULLTEXT	SQL_CALC_FOUND_ROWS	SQL_SMALL_RESULT	SSL
GRANT	GROUP	HAVING	STARTING	STRAIGHT_JOIN	TABLE
HIGH_PRIORITY	HOUR_MICROSECOND	HOUR_MINUTE	TERMINATED	THEN	TINYBLOB
HOUR_SECOND	IF	IGNORE	TINYINT	TINYTEXT	TO
IN	INDEX	INFILE	TRAILING	TRIGGER	TRUE
INNER	INOUT	INSENSITIVE	UNDO	UNION	UNIQUE
INSERT	INT	INT1	UNLOCK	UNSIGNED	UPDATE
INT2	INT3	INT4	USAGE	USE	USING
INT8	INTEGER	INTERVAL	UTC_DATE	UTC_TIME	UTC_TIMESTAMP
INTO	IS	ITERATE	VALUES	VARBINARY	VARCHAR
JOIN	KEY	KEYS	VARCHARACTER	VARYING	WHEN
KILL	LEADING	LEAVE	WHERE	WHILE	WITH
LEFT	LIKE	LIMIT	WRITE	XOR	YEAR_MONTH
LINEAR	LINES	LOAD	ZEROFILL		

New Reserved Words in MySQL 5.5

GENERAL	IGNORE_SERVER_IDS	MASTER_HEARTBEAT_PERIOD
MAXVALUE	RESIGNAL	SIGNAL
SLOW		

- So, if we are to create the customer table specified as above, we would type in

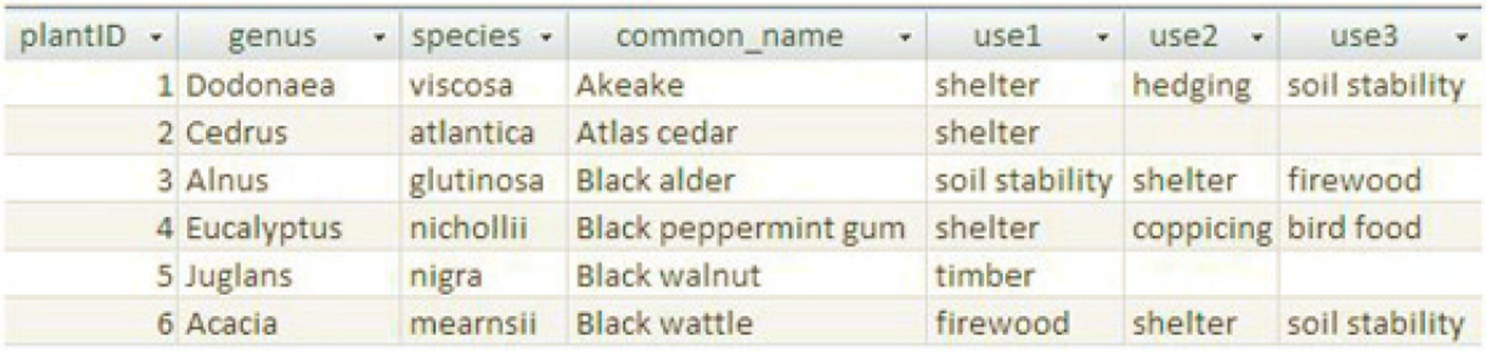
```
CREATE TABLE customer (First_Name char(50),
Last_Name char(50),
Address char(50),
City char(50),
Country char(25),
Birth_Date date)
```

- Sometimes, we want to provide a default value for each column. A default value is used when you do not specify a column's value when inserting data into the table. To specify a default value, add "Default [value]" after the data type declaration. In the above example, if we want to default column "Address" to "Unknown" and City to "Mumbai", we would type in

```
CREATE TABLE customer (First_Name char(50),
Last_Name char(50),
Address char(50) default 'Unknown',
City char(50) default 'Mumbai',
Country char(25),
Birth_Date date)
```

EXAMPLE 1-1. THE PLANT DATABASE

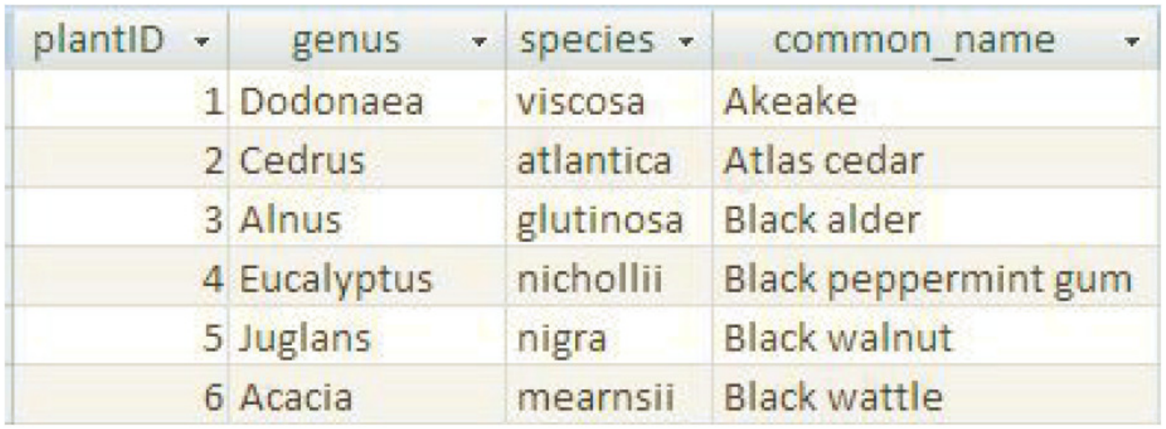
Figure 1-1 shows a small portion of a database table recording information about plants. Along with the botanical and common names of each plant, the developer decides it would be convenient to keep information on the uses for each plant. This is to help prospective buyers decide whether a plant is appropriate for their requirements.



plantID	genus	species	common_name	use1	use2	use3
1	Dodonaea	viscosa	Akeake	shelter	hedging	soil stability
2	Cedrus	atlantica	Atlas cedar	shelter		
3	Alnus	glutinosa	Black alder	soil stability	shelter	firewood
4	Eucalyptus	nichollii	Black peppermint gum	shelter	coppicing	bird food
5	Juglans	nigra	Black walnut	timber		
6	Acacia	mearnsii	Black wattle	firewood	shelter	soil stability

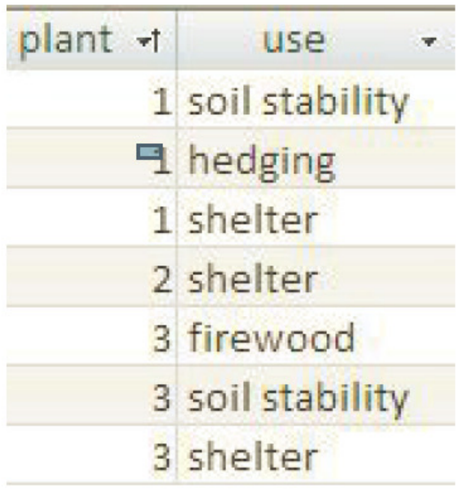
Figure 1-1. *The plant database*

If we look up a plant, we can immediately see what its uses are. However, if we want to find all the plants suitable for hedging, for example, we have a problem.



plantID	genus	species	common_name
1	Dodonaea	viscosa	Akeake
2	Cedrus	atlantica	Atlas cedar
3	Alnus	glutinosa	Black alder
4	Eucalyptus	nichollii	Black peppermint gum
5	Juglans	nigra	Black walnut
6	Acacia	mearnsii	Black wattle

Table Plants



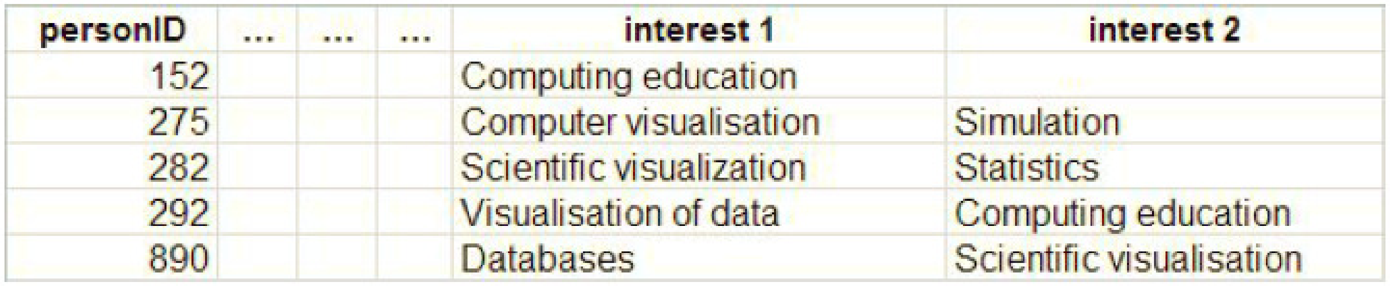
plant	use
1	soil stability
1	hedging
1	shelter
2	shelter
3	firewood
3	soil stability
3	shelter

Table Uses

Figure 1-2. *An improved database design to represent Plants and Uses*

EXAMPLE 1-2. RESEARCH INTERESTS

An employee of a university's liaison team often receives calls asking to speak to a specialist in a particular topic. The liaison team decides to set up a small spreadsheet to maintain data about each staff member's main research interests. Originally, the intention is to record just one main area for each staff member, but academics, being what they are, cannot be so constrained. The problem of an indeterminate number of interests is solved by adding a few extra columns in order to accommodate all the interests each staff member supplies. Part of the spreadsheet is shown in Figure 1-3.



personID	...	...	...	interest 1	interest 2
152				Computing education	
275				Computer visualisation	Simulation
282				Scientific visualization	Statistics
292				Visualisation of data	Computing education
890				Databases	Scientific visualisation

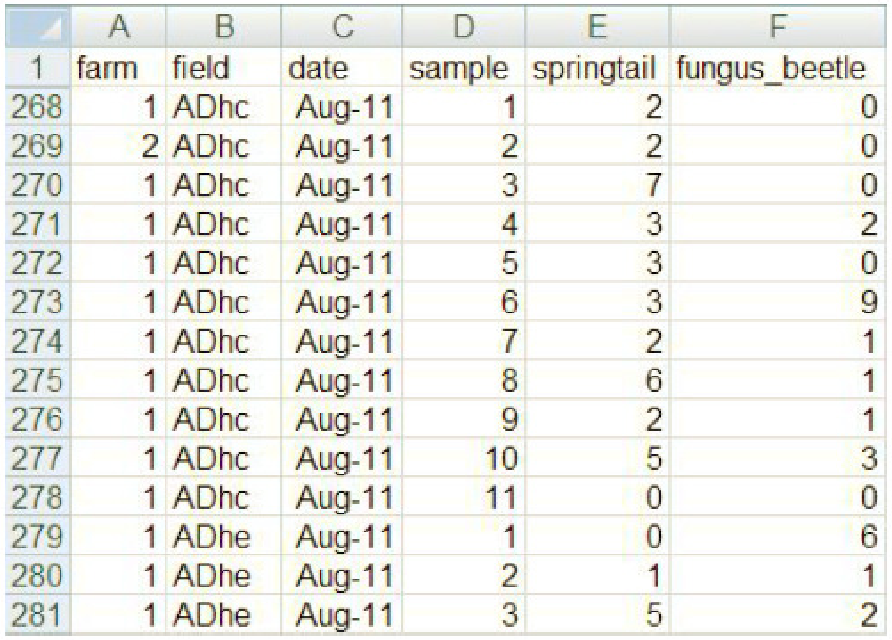
Figure 1-3. *Research interests in a spreadsheet*

We are able to see at a glance the research interests of a particular person, but as was the case in Example 1-1, it is awkward to do the reverse and find who is interested in a particular topic. However, we have an additional problem here. Many of the research interests look similar but they are described differently. How easy will it be to find a researcher who is able to "visualize data"?

EXAMPLE 1-3. INSECT DATA[1]

Team members of a long-term environmental project regularly visit farms and take samples to determine the numbers of particular insect species present. Each field on a farm has been given a unique code, and on each visit to a field a number of representative samples are taken. The counts of each species present in each sample are recorded.

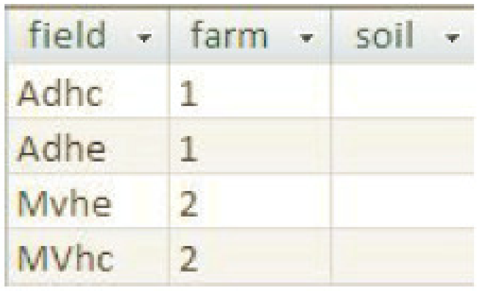
Figure 1-4 shows a portion of the data as it was recorded in a spreadsheet.



	A	B	C	D	E	F
1	farm	field	date	sample	springtail	fungus_beetle
268	1	ADhc	Aug-11	1	2	0
269	2	ADhc	Aug-11	2	2	0
270	1	ADhc	Aug-11	3	7	0
271	1	ADhc	Aug-11	4	3	2
272	1	ADhc	Aug-11	5	3	0
273	1	ADhc	Aug-11	6	3	9
274	1	ADhc	Aug-11	7	2	1
275	1	ADhc	Aug-11	8	6	1
276	1	ADhc	Aug-11	9	2	1
277	1	ADhc	Aug-11	10	5	3
278	1	ADhc	Aug-11	11	0	0
279	1	ADhe	Aug-11	1	0	6
280	1	ADhe	Aug-11	2	1	1
281	1	ADhe	Aug-11	3	5	2

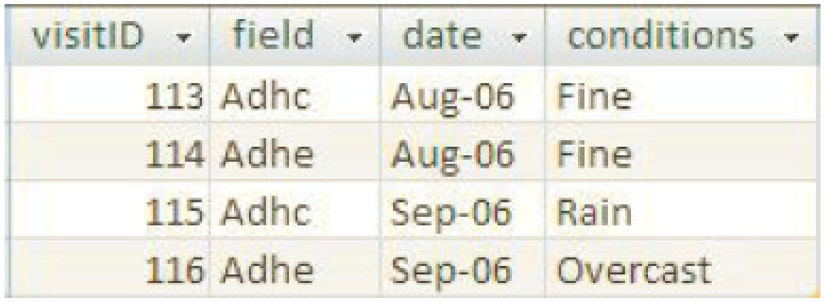
Figure 1-4. *Insect data in a spreadsheet*

The information about each farm was recorded (quite correctly) elsewhere, thus avoiding that data being repeated. However, there are still problems. The fact that field ADhc is on farm 1 is recorded every visit, and it does not take long to find the first data entry error in row 269. (The coding used for the fields raises other issues that we will not address just now.)



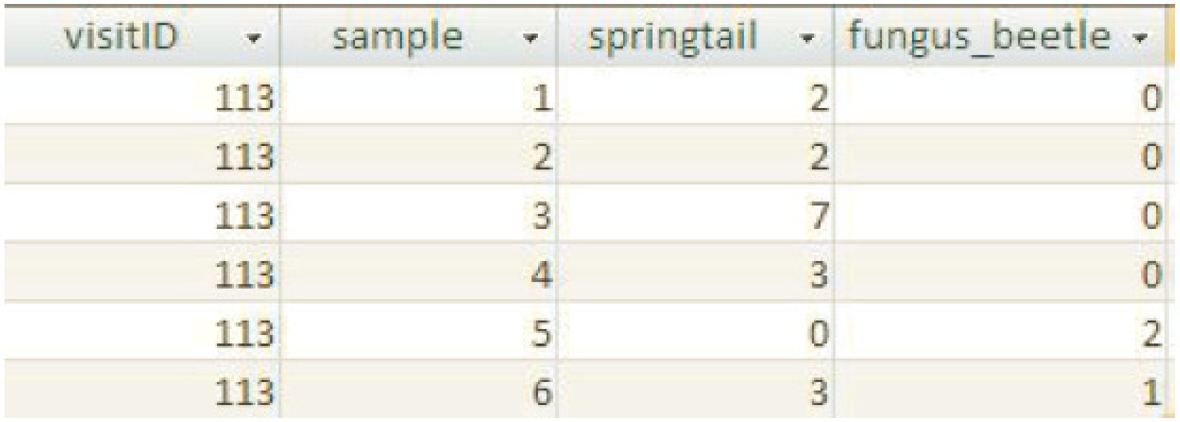
field	farm	soil
Adhc	1	
Adhe	1	
Mvhe	2	
MVhc	2	

Table Fields



visitID	field	date	conditions
113	Adhc	Aug-06	Fine
114	Adhe	Aug-06	Fine
115	Adhc	Sep-06	Rain
116	Adhe	Sep-06	Overcast

Table Visits



visitID	sample	springtail	fungus_beetle
113	1	2	0
113	2	2	0
113	3	7	0
113	4	3	0
113	5	0	2
113	6	3	1

Table Counts

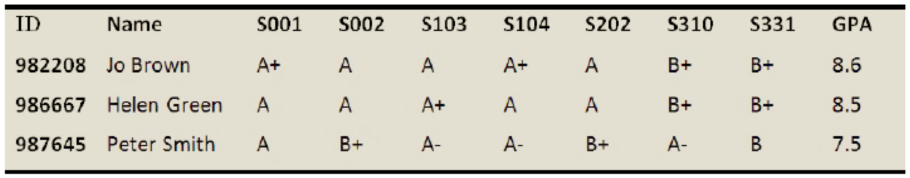
Figure 1-5. *An improved database design for the insect problem*

Designing for a Single Report

Another cause of a problematic database is to design a table to match the requirements of a particular report. A small business might have in mind a format that is required for an invoice. A school secretary may want to see the whereabouts of teachers during the week. Thinking backward from one specific report can lead to a database with many flaws. Example 1-4 is a particular favorite of mine, because the first time I was ever paid real money to fix up a database was because of this problem (clearly student record software has moved on a great deal since then!).

EXAMPLE 1-4. ACADEMIC RESULTS

A university department needs to have its final–year results in a format appropriate for taking along to the examiners' meeting. The course was very rigidly prescribed with all students completing the same subjects, and a report similar to the one in Figure 1-6 was generated by hand prior to the system being computerized. This format allowed each student's performance to be easily compared across subjects, helping to determine honors' boundaries.



ID	Name	S001	S002	S103	S104	S202	S310	S331	GPA
982208	Jo Brown	A+	A	A	A+	A	B+	B+	8.6
986667	Helen Green	A	A	A+	A	A	B+	B+	8.5
987645	Peter Smith	A	B+	A-	A-	B+	A-	B	7.5

Figure 1-6. *Report required for students' results*

A database table was designed to exactly match the report in Figure 1-6, with a field for each column. The first year the database worked a treat. The next year the problems started. Can you anticipate them?

Some students were permitted to replace one of the papers with one of their own choosing. The table was amended to include columns for option name and option mark. Then some subjects were replaced, but the old ones had to be retained for those students who had taken them in the past. The table became messier, but it could still cope with the data.
What the design couldn't handle was students who failed and then reenrolled in a subject. The complete academic record for a student needed to be recorded, and the design of the table made it impossible to record more than one mark if a student completed a subject several times. That problem wasn't noticed until the second year in operation (when the first students started failing). By then, a fair amount of effort had gone into development and data entry. The somewhat curious solution was to create a new table for each year, and then to apply some tortuous logic to extract a student's marks from the appropriate tables. When the original developer left for a new job, several years' worth of data were left in a state that no one else could comprehend. And that's how I got my first database job (and the database coped with changing requirements over several years).