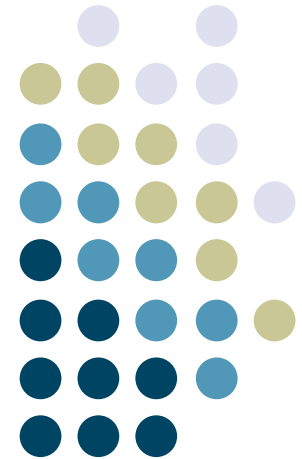


# Data Management

---



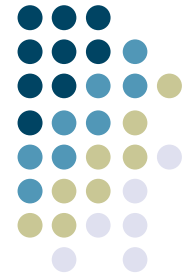
**Georgia  
Tech**



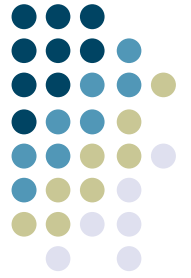
Week 14

# Why Focus on Data Management?

Georgia  
Tech



- Lots of data to keep track of in many programs
  - E.g., map to keep track of your individual chat windows
- Example of *transient* data
  - You create it while your program runs
  - When your program stops, it goes away
  - Not needed between runs

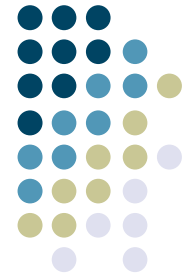


# Persistent Data Management

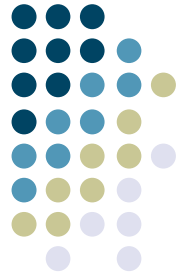
- Sometimes, you need to store data across runs of your program, or among several programs
  - Example: storing preferences information for chat names, icons, etc.
- Simplest strategy: *flat files*
  - Text files, usually containing ASCII, that your program reads and writes
  - “Flat” -- little structure
- What can go wrong when using flat files?

# Pros and Cons of Flat File Data Persistence

Georgia  
Tech

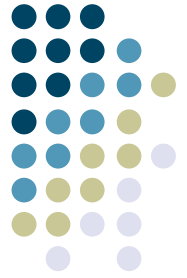


- Pros:
  - Quick 'n' easy
  - Doesn't require any fancy libraries to use
- Cons:
  - Error-prone parsing, especially if you want to save lots of stuff
  - If you need to find just one thing in a file, or update just one thing in a file, can be slow
    - Searches and updates are *linear* in the length of the file... meaning proportional to the length
  - Have to be careful to program defensively
    - File system fills up, program crashes half way through writing: may result in corrupt file
  - Hard to do concurrent access correctly



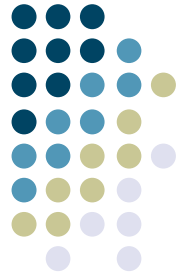
# Other Options

- Option #1: structured files
  - Example: XML-based preferences file
    - `<USER_INFO>`
      - `<NAME>Joe Schmoe</NAME>`
      - `<ICON>/Users/Schmoe/Documents/myicon.jpg</ICON>`
    - `</USER_INFO>`
  - Pros:
    - Takes advantage of XML structure to make parsing easier
    - XML library ensures you've got a well-formatted XML file
  - Cons:
    - Doesn't take care of defensive programming/concurrent access problems, just structure problem
    - Search/update can still be slow



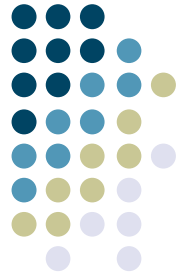
# Other Options

- Option #2: databases
- What's a database?
  - A database is a service that takes care of the persistent storage, management, and retrieval of structured data
  - Hides most of the details of files, filesystems, etc.
  - Provides a highly structured way to read/write data
    - Uses a *language* to let you do this
    - Can *query* your data to find what you want
  - Quick updates: constant time (meaning: same time no matter how big the database gets)
  - Takes care of much of the defensive programming stuff
    - But you can still corrupt a database if you write crap to it



# When to Use a Database

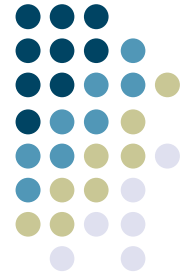
- You need to manage a lot of data (few kilobytes up to a few terabytes) that will persist across runs
- Need to be able to search through it quickly, update it quickly, or perform complex queries
- Need more safety than you can get with just files
- Need to support concurrent access



# Relational Databases

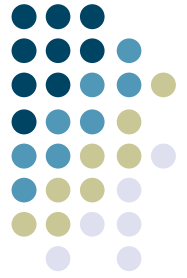
- Most common type of database today
  - Other, less-common type: object-oriented databases
- Basic concepts: **records** are stored in **tables**
  - Think of records as a row in a table
- Generally multiple records in any given table
- Most complex systems will have multiple tables
- Each **column** has a type: string, integer, etc.
- Some columns uniquely identify their records: called **keys** or indices
  - Used to cross-reference rows in different tables

# Example



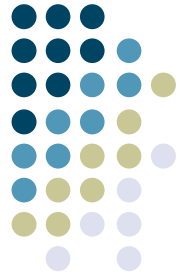
<b>Customer Number (Integer): KEY</b>	<b>Customer Name (String)</b>	<b>Customer Address (String)</b>
52352113	Keith Edwards	85 Fifth Street NW
62352922	Rich DeMillo	801 Atlantic Drive

<b>Product Number (Integer): KEY</b>	<b>Description (String)</b>	<b>List Price (Floating point)</b>
235532	Sterling Martini Shaker	\$69.99
523333	Olives	\$5.99



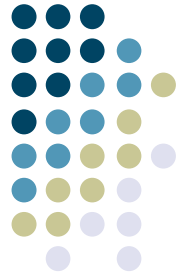
# Keys

- Important concept: *key column(s)*
  - Special column(s) that uniquely identify a row
  - Contains a special number/token/identifier that uniquely identifies the row
    - Might be a “natural” key, or something we have to make up on our own
    - E.g., in a tax database, SSN might suffice as a key
- In previous examples, customer number and product number are keys
  - Likely made up by the company itself
- In some cases, *multiple* columns might be required to uniquely identify a row
- Take care when choosing keys
  - 1994: Brazilian gov’t chose {father’s name, mother’s name, DOB} as a key for voting registration
  - Only unique for siblings born on different dates!
  - Reason why “artificial” keys are often made up (SSN)



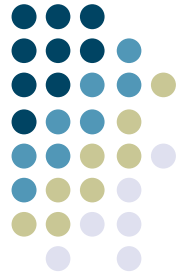
# Database Schemas

- Schemas are the *layout* of the data
  - The set of tables, the columns of those tables, etc
  - Represents the logical structure of your data
- Created by a *database designer*
  - This is the person who creates the schema for a particular application; not the person who writes the database itself
- Once the schema is created, it stays pretty much fixed
  - Applications just add/delete/update rows; not restructure the entire database
- For this project: you'll be taking the role of database designer, as well as application programmer
  - I'll be helping with the database design though



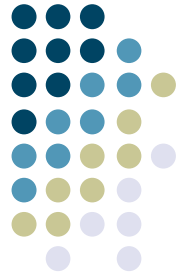
# Database Design

- Database jargon:
  - *Entities*: the real world things that are represented in the database
    - E.g., people, messages, chat sessions, etc.
  - *Attributes*: aspects of an entity that we want to represent
    - E.g., people have names, etc.
  - *Relationships*: Associations among entities
    - E.g., person A sent message B. This is a relationship between person A and message B.



# Designing a Schema

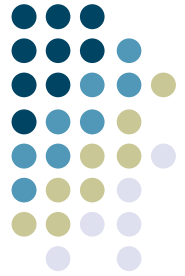
- Entities generally turn into tables
  - Each *instance* of an entity is a new row in that table
  - E.g., a person table, with one row for each person
- Attributes generally turn into columns in tables
  - E.g., a person table might have columns for first, last names, icons, etc.
- Relationships are the glue that holds a database together
  - Cross references between entity tables
  - E.g., a person is involved in multiple chats, while a chat has multiple members
  - Relationships can either be represented as separate columns, or as their own tables
  - More on this later
- **Good design guideline:** minimize redundancy
  - Redundancy is just another chance for tables to get out of sync with each other



# One-to-One Relationships

- What are the ways entities can be related to each other?
- One-to-one relationships
  - E.g., accounts receivable system
  - Each AR can have at most one customer associated with it; each customer can have at most one AR associated with him/her
- One-to-one relationships best represented as columns in a combined table

<b>Account Receivable</b>	<b>Date</b>	<b>Customer Number</b>
1225498	2/15/05	579923512
1225499	2/16/05	999922531



# One-to-Many Relationships

- One-to-many
  - Many instances of one entity are associated with one instance of another entity
  - E.g., library checkout system: One person can have many books checked out, but each book can only be checked out by one person
- Very common. Usually: add column to one table only

Customer Number	Customer Name
12255193	Keith Edwards
93321355	Rich DeMillo

Book ID	Book Name	Checked Out By
333521	History of Typography	12255193
793313	Get Typed!	12255193



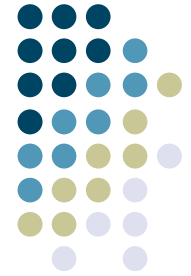
# Many-to-Many Relationships

- Many-to-many
  - E.g., Amazon product/order database
  - Any given order might contain multiple distinct products
  - Any given product might be associated with multiple orders
- Relationships like this result in the creation of a *new table* to hold just the relationship

Product #	Description
12	Martini Shaker
53	Olives

Order #	Date
B29	2/15/05
C33	2/15/05

Product #	Order #
12	B29
53	B29
53	C33



# Database Design Principles

- “Normalizing” data relationships
  - Avoid as much redundancy as possible
  - Structure things so that “anomalies” can’t happen
- Example:

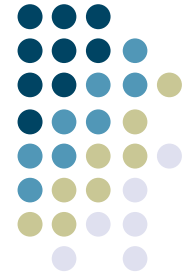
Acme Industries  
INVOICE

Customer Number: 1454  
Customer: W. E. Coyote  
General Delivery  
Falling Rocks, AZ 84211  
(599) 555-9345

Order Date: 11/05/06  
Terms: Net 30  
Ship Via: USPS

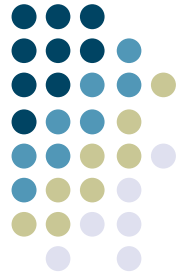
Product No.	Description	Quant.	Unit Price	Ext. Amount
SPR-2290	Super-strength springs	2	24.00	48.00
STR-67	Foot straps, leather	2	2.50	5.00
HLM-45	Deluxe crash helmet	1	67.88	67.88
SFR-1	Rocket, solid fuel	1	128,200.40	128,200.40
ELT-7	Emergency location transmitter	1	79.88	** FREE GIFT **

**Total Order Amount: \$128,321.28**



# An Overly-Simple Approach

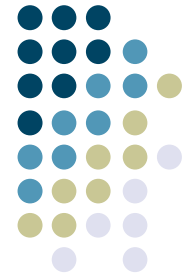
Cust. No.	Cust. Name	Cust. Addr.	Cust. City	Cust. State	Cust. ZIP	Cust. Phone	Order Data	Prod. No.	Desc.	Quant.	Unit Price	Ext. Amount
1454	W. E. Coyote	General Delivery	Falling Rocks	AZ	84211	599-55 5-9345	11/5/2 006	SPR-2290	Super strength springs	2	24.00	48.00
								STR-67	Foot straps, leather	2	2.50	5.00
								HLM-45	Deluxe Crash Helmet	1	67.88	67.88
								SFR-1	Rocket, solid fuel	1	128,200.40	128,200.40
								ELT-1	Emergency Transmitter	1	78.88	0.00



# The Invoice Example

- This simple table structure is *too* simple
- Presents possibility for *anomalies* to arise during use
- Three types of anomalies
  - Insert anomalies
  - Delete anomalies
  - Update anomalies
- We'll work through an example showing how you *normalize* tables to avoid anomalies

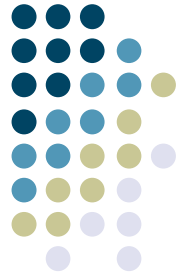




# Delete Anomalies

- *Delete anomalies*: deletion of data about one entity causes unintended loss of data about another
  - Two different entities are mixed in the same relation
  - E.g., deleting last invoice for a customer loses all data for that customer

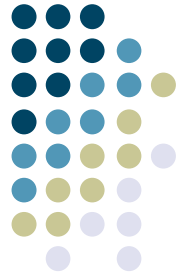
Cust. No.	Cust. Name	Cust. Addr.	Cust. City	Cust. State	Cust. ZIP	Cust. Phone	Order Data	Prod. No.	Desc.	Quant.	Unit Price	Ext. Amount
1454	W. E. Coyote	General Delivery	Falling Rocks	AZ	84211	599-55 5-9345	11/5/2 006	SPR-2290	Super strength springs	2	24.00	48.00
								STR-67	Foot straps, leather	2	2.50	5.00
								HLM-45	Deluxe Crash Helmet	1	67.88	67.88
								SFR-1	Rocket, solid fuel	1	128,200.40	128,200.40
								ELT-1	Emergency Transmitter	1	78.88	0.00



# Update Anomalies

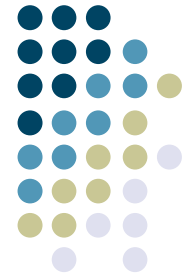
- *Update anomalies*: update of a single data value requires multiple rows of data to be updated
  - Too much redundancy in the database
  - E.g., if we wanted to change customer's address, have to update every invoice for that customer--opportunity for inconsistency

Cust. No.	Cust. Name	Cust. Addr.	Cust. City	Cust. State	Cust. ZIP	Cust. Phone	Order Data	Prod. No.	Desc.	Quant.	Unit Price	Ext. Amount
1454	W. E. Coyote	General Delivery	Falling Rocks	AZ	84211	599-55 5-9345	11/5/2 006	SPR-2290	Super strength springs	2	24.00	48.00
								STR-67	Foot straps, leather	2	2.50	5.00
								HLM-45	Deluxe Crash Helmet	1	67.88	67.88
								SFR-1	Rocket, solid fuel	1	128,200.40	128,200.40
								ELT-1	Emergency Transmitter	1	78.88	0.00
1454	W. E. Coyote	General Delivery	Falling Rocks	AZ	84211	599-55 5-9345	12/30/ 2006	Q-3235	High explosives	2	249.99	499.98
								AJ-92	Anvil	1	149.99	149.99



# In-Class Exercise

- Figure out a correct schema for the Invoice example
- Steps:
  1. Choose a primary key, if one doesn't already exist
  2. Get rid of multivalued attributes
    - Why? Hard to select out what we want
    - Move multivalued attributes to a new table
    - Copy key from original table to the new one
  3. Make sure that non-key attributes only are determined by the *entire* key, not *part* of the key
    - Why? This is a sign that you're mixing unrelated information about multiple entities in a single table
    - Move attribute to new table where it depends on entire key
  4. Make sure that no attributes are determined by non-key attributes
    - Why? Again, clean separation of relations
    - Move determined attributes to new table where they depend only on the key
- Summary: make sure any non-key attribute in a table depends on the key, the whole key, and nothing but the key!

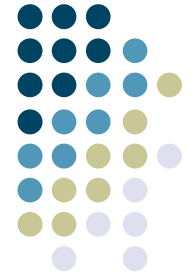


# First Step

I. Choose a primary key, if one doesn't already exist

Invoice													
Invoice No.	Cust. No.	Cust. Name	Cust. Addr.	Cust. City	Cust. State	Cust. ZIP	Cust. Phone	Order Data	Prod. No.	Desc.	Quant.	Unit Price	Ext. Amount
12345	1454	W. E. Coyote	General Delivery	Falling Rocks	AZ	84211	599-55 5-9345	11/5/2006	SPR-2290	Super strength	2	24.00	48.00
									STR-67	springs	2	2.50	5.00
									HLM-45	Foot straps, leather	1	67.88	67.88
									SFR-1	Deluxe Crash Helmet	1	128,200.40	128,200.4
									ELT-1	Rocket, solid fuel	1	78.88	0

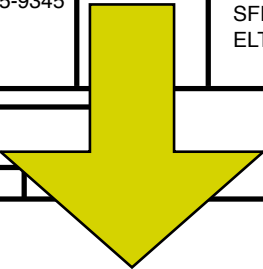
None of the fields here do a good job of uniquely identifying this entire row, so we'll create one, called Invoice Number.



# Second Step

2. Get rid of multivalued attributes: each cell should contain just one data element
  - Make a new table for multivalued attributes; copy the key column

Invoice													
Invoice No.	Cust. No.	Cust. Name	Cust. Addr.	Cust. City	Cust. State	Cust. ZIP	Cust. Phone	Order Date	Prod. No.	Desc.	Quant.	Unit Price	Ext. Amount
12345	1454	W. E. Coyote	General Delivery	Falling Rocks	AZ	84211	599-55 5-9345	11/5/2006	SPR-2290	Super strength springs	2	24.00	48.00
									STR-67	Foot straps, leather	2	2.50	5.00
									HLM-45	Deluxe Crash Helmet	1	67.88	67.88
									SFR-1	Rocket, solid fuel	1	128,200.40	128,200.40
									ELT-1	Emergency Transmitter	1	78.88	0.00



Invoice							
Invoice No.	Cust. No.	Cust. Name	Cust. Addr.	Cust. City	Cust. State	Cust. ZIP	Quant.
12345	1454	W. E. Coyote	General Delivery	Falling Rocks	AZ	84211	5

Invoice Line Item					
Invoice No.	Prod. No.	Desc.	Quantity	Unit Price	Ext. Amount
12345	SPR-2290	Super strength springs	2	24.00	48.00
12345	STR-67	Foot straps	2	2.50	5.00
12345	HLM-45	Deluxe Helmet	1	67.88	67.88
12345	SFR-1	Rocket	1	128,200.40	128,200.40
12345	ELT-1	Emerg. Trans.	1	78.88	0.00



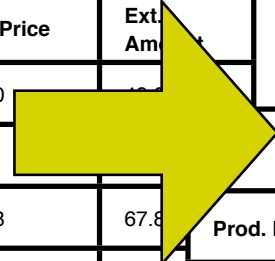
# Third Step

3. Make sure that non-key attributes only are determined by the *entire* key, not *part* of the key
  - In Invoice Line Item, the Invoice No. and Product No. (together) act as the key
    - There can be at most one row with that unique combination of Invoice and Product numbers.
  - But other columns (description, unit price) depend solely on the product number
  - Move to its own table!

Invoice Line Item					
Invoice No.	Prod. No.	Desc.	Quantity	Unit Price	Ext. Amount
12345	SPR-2290	Super strength springs	2	24.00	48.00
12345	STR-67	Foot straps	2	2.50	5.00
12345	HLM-45	Deluxe Helmet	1	67.88	67.88
12345	SFR-1	Rocket	1	128,200.40	128,200.40
12345	ELT-1	Emerg. Trans.	1	78.88	78.88

Invoice Line Item				
Invoice No.	Prod. No.	Quantity	Sales Price	Ext. Amount
12345	SPR-2290	2	24.00	48.00
12345	STR-67	2	2.50	5.00

Product		
Prod. No.	Desc.	List Price
SPR-2290	Super strength springs	24.00
STR-67	Foot straps	2.50
HLM-45	Deluxe Helmet	67.88
SFR-1	Rocket	128,200.40
ELT-1	Emerg. Trans.	78.88

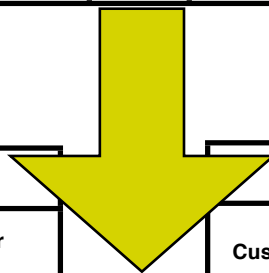




# Fourth Step

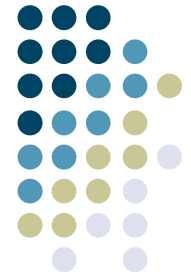
4. Make sure that no attributes are determined by non-key attributes
  - In the Invoice table, Customer Name depends on Invoice Number (each Invoice Number has only one Customer Name associated with it)
  - ... but Customer Name also depends on Customer Number.
  - Same is true for rest of customer attributes
  - Move to its own table!

Invoice								
Invoice No.	Cust. No.	Cust. Name	Cust. Addr.	Cust. City	Cust. State	Cust. ZIP	Cust. Phone	Order Date
12345	1454	W. E. Coyote	General Delivery	Falling Rocks	AZ	84211	599-555-9345	11/5/2006



Invoice		
Invoice No.	Cust. No.	Order Date
12345	1454	11/5/2006

Customer						
Cust. No.	Cust. Name	Cust. Addr.	Cust. City	Cust. State	Cust. ZIP	Cust. Phone
1454	W. E. Coyote	General Delivery	Falling Rocks	AZ	84211	599-555-9345



# Recap of Normalization

- Process is fairly cookbook
  - But still difficult
  - Have to *understand* the data, relationships among them
- Generally, moves toward more smaller tables
- Avoid insert, delete, update anomalies

Customer						
Cust. No.	Cust. Name	Cust. Addr.	Cust. City	Cust. State	Cust. ZIP	Cust. Phone
1454	W. E. Coyote	General Delivery	Falling Rocks	AZ	84211	599-555-9345

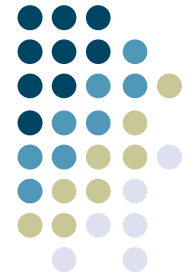
Product		
Prod. No.	Desc.	List Price
SPR-2290	Super strength springs	24.00
STR-67	Foot straps	2.50
HLM-45	Deluxe Helmet	67.88
SFR-1	Rocket	128,200.40
ELT-1	Emerg. Trans.	78.88

Invoice Line Item				
Invoice No.	Prod. No.	Quantity	Sales Price	Ext. Amount
12345	SPR-2290	2	24.00	48.00
12345	STR-67	2	2.50	5.00
12345	HLM-45	1	67.88	67.88
12345	SFR-1	1	128,200.40	128,200.40
12345	ELT-1	1	78.88	0.00

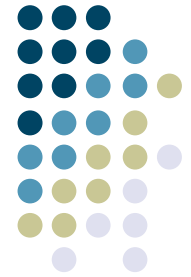
Invoice		
Invoice No.	Cust. No.	Order Date
12345	1454	11/5/2006

# Databases in Practice

Georgia  
Tech

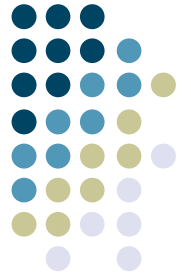


# Database Creation, Queries, and Updates

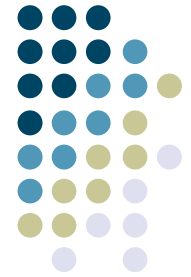


- So far, just talked about the logical structure of databases, and database design
- Now:
  - How to write a program to create that logical structure
  - How to get data into the database
  - How to get data out of a database
- Queries: mechanism for talking to a database
- Many mechanisms; the one we'll focus on here is a common language-based approach

# SQL

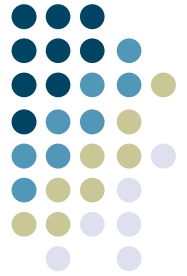


- The *Structured Query Language*
- Textual language for creating database structure, updates, and queries
- Pretty much the universal language for relational databases
  - Supported more-or-less the same across all RDBMS systems
  - Minor variances because vendors want to differentiate their products



# Quick'n'Dirty Overview of SQL

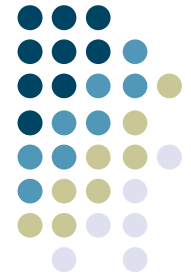
- Basic statements
  - CREATE
  - INSERT
  - UPDATE
  - DELETE
  - SELECT
- Basic column types
  - INT
  - REAL
  - VARCHAR
  - CHAR/CHAR(n)
  - DATE
  - TIME
  - BOOLEAN
  - OBJECT



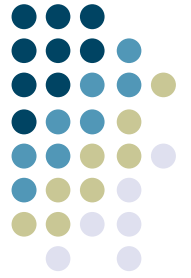
# Creating a New Database Table

- `CREATE CACHED TABLE table_name (col1, col2, ...)`
  - `CACHED` means it actually gets saved to disk. Important!
  - In the parens are the definitions of the columns, separated by commas:
    - `columnName type [GENERATED BY DEFAULT AS IDENTITY] [PRIMARY KEY]`
    - `GENERATED BY DEFAULT AS IDENTITY` means that the database automatically generates a value for this column, and treats it as a key
      - Issue the SQL statement `CALL IDENTITY()` to return the last generated value
    - `PRIMARY KEY` means that the column is considered a key column
- Example:
  - `CREATE CACHED TABLE bookInfo (bookID GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY, title CHAR(64))`

# Storing New Data in the Database

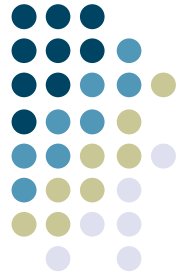


- The INSERT statement adds a new row to a table
- `INSERT INTO table_name (col1, col2, ...) VALUES (val1, val2, ...)`
- Example:
  - `INSERT INTO customerInfo (name, ssn) VALUES ('keith', '123-45-6789')`
- If you leave out the columns you have to provide values for all columns
- Be sure to put single quotes (') around strings!



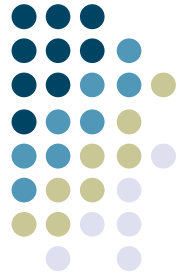
# Removing Rows from a Database

- The DELETE statement removes rows from a database
- DELETE FROM table\_name WHERE condition
- Example:
  - DELETE FROM customerInfo WHERE name = 'keith'
  - DELETE FROM books WHERE publicationYear < 1991
- Without a WHERE clause DELETE removes all rows from the table!



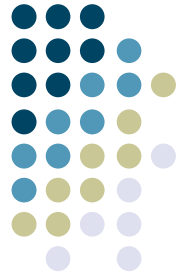
# SQL Conditions

- DELETE and other SQL statements support WHERE conditions that determine which rows they operate on
- Common arithmetic operations:
  - value {=, <, <=, >, >=, <>, !=} value
  - value BETWEEN value AND value
- Other, fancier operations are supported as well
  - Probably not needed for this assignment
  - Check SQL docs for more info



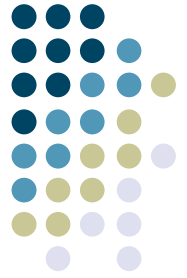
# Updating Cells in Existing Rows

- Use UPDATE to change cells in existing rows
- UPDATE table\_name SET column=expr WHERE condition
- Example:
  - UPDATE bookInfo SET retailPrice='59.99' WHERE isbn='07879312'
- Will update *multiple* rows that match the WHERE condition
- Like DELETE, leaving off the WHERE condition updates all rows in the table!
- Can have multiple *column=expr* separated by a comma, to update multiple fields in matching rows at the same time.



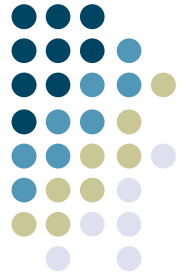
# Querying the Database

- SELECT is the workhorse of SQL: lets you get data out of your database
- SELECT \* FROM table\_name
  - Returns all rows from table\_name
- SELECT columns FROM table\_name
  - Returns only the specified columns from table\_name
- SELECT DISTINCT columns FROM table\_name
  - Removes duplicate values and returns the specified columns



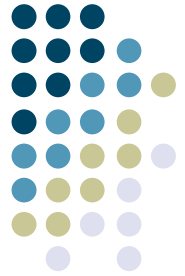
## Querying the Database (cont'd)

- **SELECT columns FROM table\_name WHERE condition**
  - Lets you select out only the rows that match the specified condition, then return only the desired columns from those rows
- **SELECT columns FROM table\_name WHERE condition ORDER BY column**
  - Same as above, only returns data ordered by the specified column
- **SELECT avg(column) FROM table\_name**
  - Can provide a function that applies to the selected column, and aggregates the results
  - Other functions: max(), min(), sum(), count()



# Querying the Database (cont'd)

- Examples:
  - `SELECT * FROM books`
  - `SELECT DISTINCT publicationYear FROM books`
  - `SELECT title, publicationYear FROM books WHERE retailPrice < 50.00 AND publicationYear > 1994`
  - `SELECT avg(retailPrice) FROM books`
  - `SELECT isbn, title, publicationYear FROM books ORDER BY publicationYear`
- Using SELECT in practice:
  - Bad idea: do a very general select, then piece the data together in your program
  - Good idea: let SELECT do the work for you! See if you can get *exactly* the data you need in one SELECT statement

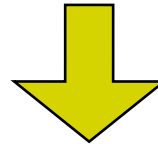


# Advanced SQL: Joins

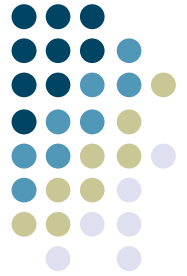
- Everything so far has been about queries over single tables
- Possible to write queries that span tables
  - Very powerful feature of relational databases
- `SELECT * FROM books, publishers WHERE books.publisherID = publishers.publisherID AND publishers.city = 'New York' AND publishers.state = 'NY'`
- Gets details of all books published in NY, even though the information is spread across multiple tables

books			
bookID	publisherID	title	author

publishers			
publisherID	name	city	state

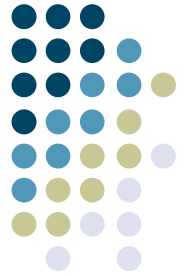


bookID	publisherID	title	author	name	city	state
--------	-------------	-------	--------	------	------	-------



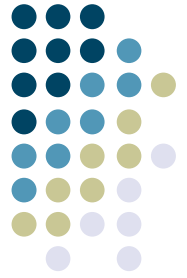
# Databases in the Wild

- Lots of database products and freeware in the world
  - Oracle, Access, MySQL, etc.... literally hundreds
- Most are implemented as *server processes*: your code connects to them over the network
  - Many are super-highend
    - E.g., Oracle
  - Some are more “consumer oriented”
    - Access, Filemaker
- Others are embedded
  - Meaning: the database code lives in a library inside your program, rather than as a server
  - Pro: easy to use, easy to manage (no need to start/stop separate server)
  - Con: can't easily have multiple programs share the database at once



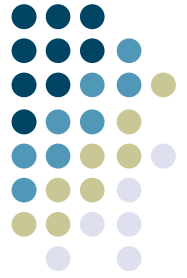
# Hypersonic SQL

- <http://hsqldb.sourceforge.net>
- Freeware database, written in Java
- Can be embedded, or run as a server
- Download hsqldb.jar
  - Available on class website
  - Contains the entire database implementation, along with SQL interpreter, etc.
- Update your classpath to add this in
  - Same process as for the googleapi JAR file



# Using Databases from Java

- Ideally, you'd like to be able to write your code so that you can “swap out” whatever database its using
  - To move higher-end, to move cheaper, because of bugs, whatever
- Java has a cool API that works across databases
- JDBC - the Java Database Connector API
  - You initialize JDBC by telling it what specific database you'll be using and it does the rest
  - Hides whether or not you're talking to a database server, an embedded database, etc.
- Since it's so widely used, and easily callable from Java, we'll use it in the next project

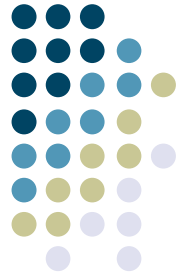


# JDBC Basics

- Load the JDBC driver for your specific database
- Connect to the database
- Issue queries in the form of plain text SQL statements
- Shutdown and disconnect

# Loading the Driver and Connecting to the Database

Georgia  
Tech



```
import java.sql as sql
import java.lang as lang
```

```
try:
```

```
    lang.Class.forName("org.hsqldb.jdbcDriver").newInstance()
```

```
except lang.ClassNotFoundException:
```

```
    print "No JDBC driver found; check classpath?"
```

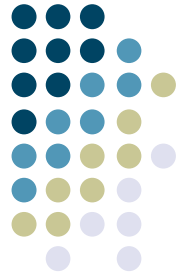
```
try:
```

```
    connection = sql.DriverManager.getConnection("jdbc:hsqldb:file:" + dbname)
```

```
    statement = connection.createStatement()
```

```
except sql.SQLException, ex:
```

```
    print "Couldn't connect to database", ex
```



# Creating the Database

```
# if it already exists, we don't have to create it!
```

```
try:
```

```
    statement.executeQuery("SELECT * FROM CHATS")
```

```
    print "Database already created!"
```

```
    return
```

```
except sql.SQLException:
```

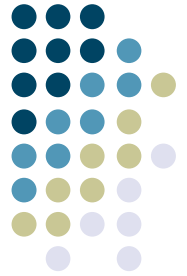
```
    pass
```

```
try:
```

```
    statement.executeQuery("CREATE CACHED TABLE CHATS (CHAT_ID INT  
GENERATED BY DEFAULT AS IDENTITY (START WITH 1) PRIMARY KEY,  
INITIATOR CHAR(64), START REAL)")
```

```
except sql.SQLException, ex:
```

```
    print "Trouble creating CHATS table", ex
```



# Adding Data to the Database

try:

```
initiator = "Keith"
```

```
startTime = time.time()
```

```
statement.executeQuery("INSERT INTO CHATS (INITIATOR, START) VALUES (" +  
initiator + ", " + str(startTime) + ")")
```

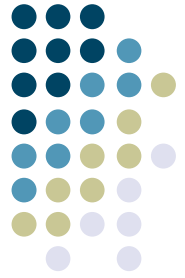
except sql.SQLException, ex:

```
print "Trouble inserting new row", ex
```

- Note use of quotes around string values, and use of `str()` to concatenate non-string values into a string!

# How do you know the key of the data you just added?

Georgia  
Tech



- Special trick:
  - CALL IDENTITY()
  - Returns the key value for the most recently added row
    - As a ResultSet with one row and one column containing an integer value
- Example:

try:

```
statement.executeQuery("INSERT INTO CHATS (INITIATOR, START) VALUES (" +  
    + initiator + ", " + str(startTime) + ")")
```

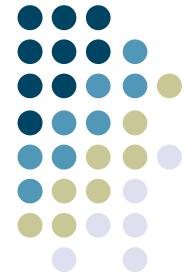
```
rs = statement.executeQuery("CALL IDENTITY()")
```

```
if rs.next():
```

```
    key = rs.getInt(1)
```

```
except sql.SQLException, ex:
```

```
    print "Trouble inserting new row", ex
```



# Updating the Database

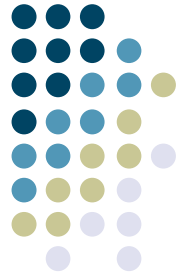
try:

```
newTime = time.time()
```

```
statement.executeQuery("UPDATE CHATS (START) VALUES (" +  
str(newTime) + ")")
```

except sql.SQLException, ex:

```
print "Trouble updating row", ex
```



# Querying the Database

```
try:
```

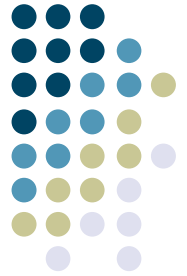
```
    rs = statement.executeQuery("SELECT * FROM CHATS")
```

```
    while rs.next():
```

```
        print "Chat ID:", rs.getInt("CHAT_ID"), "Initiator:",  
              rs.getString("INITIATOR"), "Start Time:", rs.getInt("START")
```

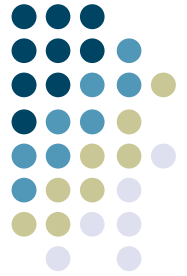
```
except sql.SQLException, ex:
```

```
    print "Trouble executing query", ex
```



# Using ResultSets

- ResultSets: the Java objects returned from `executeQuery()`
- Think of it as a data structure with a built in cursor
  - Use `next()` to move to the next row in the returned data
  - `next()` will return a *true* value as long as there is more data
    - So, *while rs.next()* will work great
    - Need to use it once to position it on the first row
    - If no rows are returned, the first call to `rs.next()` will return a *false* value
- Operations to get data out of the current row
  - `getInt()`, `getString()`, etc. You need to know what the data type is for the row you want
  - Can pass in the name of the column
  - Can pass in the integer index of the column
    - **IMPORTANT:** SQL indices start with 1, not 0!

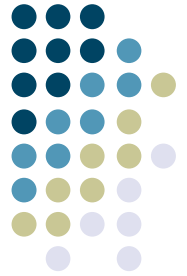


# Shutting Down Cleanly

- Important: You need to shut down cleanly to make sure all data gets written out!
  - If you just Control-C your program, you may lose the last updates
- To be safe, follow the following steps:

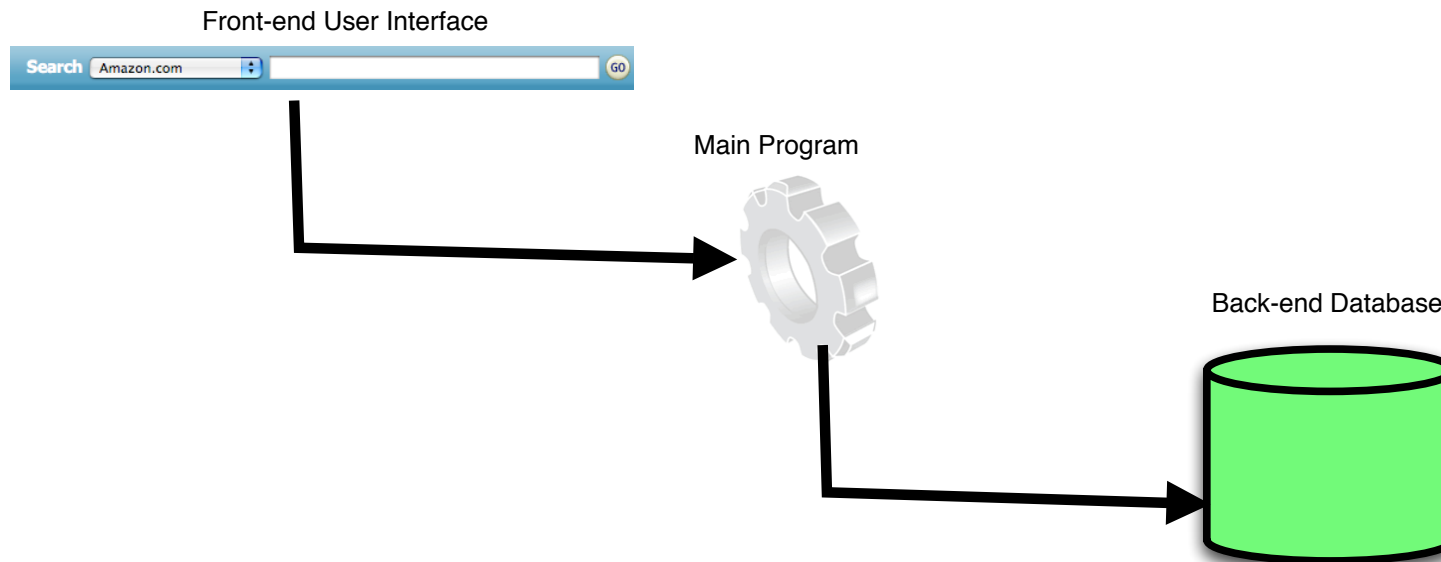
```
statement.executeQuery("SHUTDOWN")
statement.close()
connection.close()
```
- SHUTDOWN writes everything to disk and cleans up nicely
- If you want to make sure everything is flushed out each time you do something, you can also call:

```
statement.executeQuery("CHECKPOINT")
```

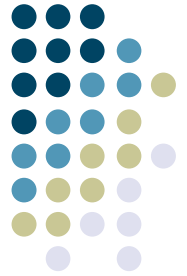


# “Cleaning” Your Input

- There’s often a path from user input all the way down into an SQL statement

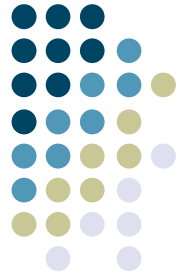


```
searchString = label.text  
stmt.executeQuery("SELECT * FROM PRODUCTS WHERE Name = ' " + searchString + " ' ")
```



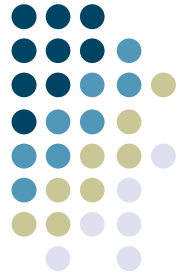
# Cleaning Your Input (cont'd)

- `stmt.executeQuery("SELECT * FROM PRODUCTS WHERE Name = '" + searchString + "'")`
- This line produces a string of SQL that embeds the contents of `searchString`:
  - `SELECT * FROM PRODUCTS WHERE Name = 'Canon Powershot SD300'`
- But what if the user enters a string that contains an apostrophe?
  - Example: user enters: *Canon's Powershot SD300*
  - Concatenating this into the SQL query yields an ill-formed string:
    - `SELECT * FROM PRODUCTS WHERE Name = 'Canon's Powershot SD300'`
- The apostrophe (single quote mark) prematurely terminates the string, leaving random garbage after
- Solution:
  - Pre-filter out any special characters (especially quotations) that might screw up an SQL statement when concatenated in
- ***This is actually a common security hole!***
- What if the user enters: *Powershot'; DROP TABLE PRODUCTS*
  - The string and single quote terminate the string prematurely
  - The semicolon tells SQL that the SELECT statement is over and a new statement is starting
  - `DROP TABLE PRODUCTS` is interpreted as a new SQL command that deletes the products table
- *SQL Injection Attack.*
  - Solution: Use JDBC *PreparedStatement* class, which automatically filters out special characters
  - (Not required for this assignment)



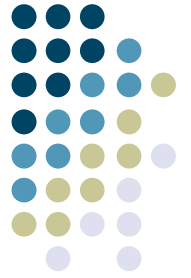
# Hypersonic SQL Pragmatics

- Debugging when using a database
  - Exceptions, exceptions, exceptions
    - Put every call to the database in a try/except block
    - Don't just swallow the exception, *print it!*
  - Print statements are your friend
    - Lots of building long strings of concatenated text, variables, etc.
    - Print out the SQL strings you send to the database, to check for correctness
  - Print the database contents itself
    - Feel like you're doing the right thing, but getting the wrong results?
    - You might have stored weird stuff in your database
    - Take the time to write a function that prints the entire database



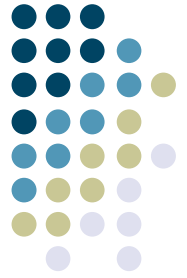
# Hypersonic SQL Pragmatics

- Where does my data live??
- HSQLDB uses a number of files to store the data
  - *These live in the directory from which you run the program!!*
  - For a database named *foo*:
    - *foo.properties* - the database's record of its parameters and configuration
    - *foo.script* - the set of SQL statements executed up until the last session
    - *foo.data* - the actual data for any cached tables
    - *foo.backup* - complete backup of the database from the last session
    - *foo.log* - the set of SQL statements executed in the current session
- After a clean shutdown:
  - *foo.data* is fully updated; *foo.backup* contains what's in *foo.data*; *foo.script* contains the statements executed in the last session; there is no *foo.log*; *foo.properties* contains "modified = no"
- After a non-clean shutdown:
  - *foo.data* may be corrupt; *foo.backup* is ok; *foo.script* contains the SQL commands needed to create that backup; *foo.log* contains the info needed to get produce *foo.data*; *foo.properties* contains "modified = yes"



# Hypersonic SQL Pragmatics

- Starting Over
- Sometimes you just need to recreate the whole database
- Two solutions:
  - `DROP TABLE table_name`
  - By hand, remove all of the database files mentioned previously

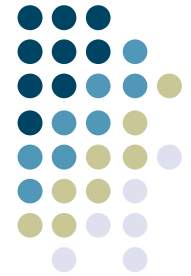


# The Next Programming Project

- We'll modify the client to store information about past chats in a database
- Create db.py to hold the code that interacts with the database
  - Connect to the database; create the necessary structure, update it as chats come and go
  - Will likely be called from both gui.py and net.py
- Augment gui.py to add a “history” button
  - Should show all of the people you've ever chatted with
  - Selecting one should show a history of all chats, their text, etc.
- **Create one database per user!**
  - Name should match the logged in user

# Suggested Chat Database Schema

Georgia  
Tech

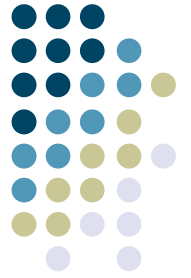


Chats		
ChatID INT <i>Key, automatically generated. NOTE different than server's Chat ID!</i>	Initiator CHAR(64) <i>Use name of participant</i>	Start REAL <i>Use time.time()</i>

Members_in_Chats	
ChatID INT	Member CHAR(64)

Messages			
MessageID INT <i>Key, automatically generated.</i>	Sender CHAR(64) <i>Use name of participant</i>	Text CHAR(256) <i>Sent text</i>	Time REAL <i>Time at which text is sent. Use time.time()</i>

Messages_in_Chats	
ChatID INT	MessageID INT



## Extra Tips

- Doing the time-related stuff:

```
import time
```

```
# get a floating point number with the number of seconds since
```

```
# January 1, 1970 (the start of UNIX time). This value is suitable for
```

```
# storing in the database in columns of type REAL
```

```
now = time.time()
```

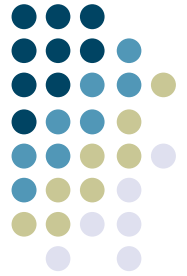
```
# convert it to a more reasonable format (for printing, debugging, etc.).
```

```
# time.localtime() takes one of the floating point things and returns a
```

```
# tuple
```

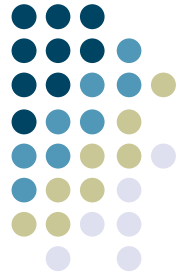
```
year, month, day, hour, minute, second, dayOfWeek, dayOfYear, daylightSavings =
```

```
time.localtime(now)
```



# Suggested Coding Strategy

- Make a new file db.py
- Create functions for all of the basic operations:
  - connect()
  - create()
  - disconnect()
- Create some low-level functions to access the database:
  - getAllMembers()
  - getChatIDsByMember(member\_name)
  - getMembersByChatID(chat\_id)
  - getMessagesByChatID(chat\_id)
  - addChat(initiator, start\_time)
  - addUser(chat\_id, new\_user)
  - addMessage(chat\_id, initiator, text, time)
- Create a getHistory function that your GUI will call out to to get the data to build the history window
- Wrap *everything* in exception handling! Print exceptions, print SQL statements before you send them
- Create some debugging functions to print result sets, print the entire database



# Suggested Coding Strategy

- $\text{difficulty}(\text{google}) < \text{difficulty}(\text{gui}) < \text{difficulty}(\text{db}) \ll \text{difficulty}(\text{net})$
- Less than three weeks to go. Start early, finish early!