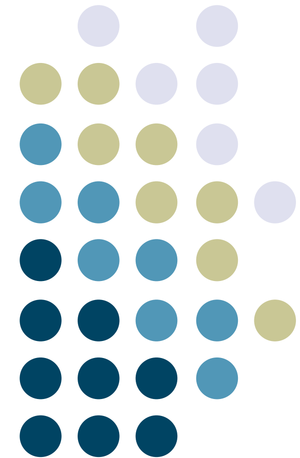
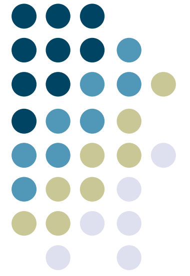


Data Management

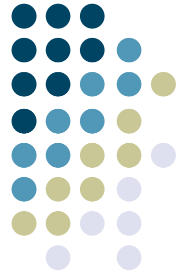
Georgia Tech

Week 14

Why Focus on Data Management?



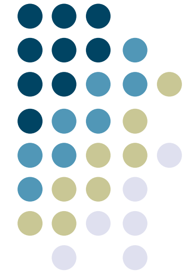
- Lots of data to keep track of in many programs
  - E.g., map to keep track of your individual chat windows
- Example of *transient* data
  - You create it while your program runs
  - When your program stops, it goes away
  - Not needed between runs



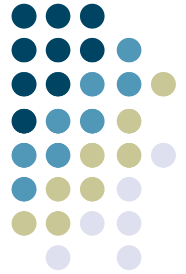
Persistent Data Management

- 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



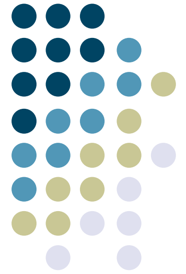
- Pros:
  - Quick 'n' easy
  - Doesn't require any fancy libraries to use
- Cons:
  - Error-prone parsing, especially if you want to save lots of stuff
  - If you need to find just one thing in a file, or update just one thing in a file, can be slow
    - Searches and updates are *linear* in the length 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



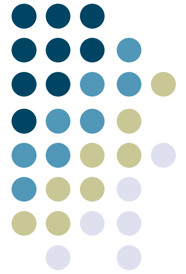
Other Options

- 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

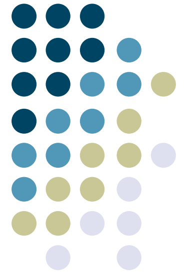


- Option #2: databases
- What's a database?
  - A database is a service that takes care of the persistent storage, management, and retrieval of structured data
  - Hides most of the details of files, filesystems, etc.
  - Provides a highly structured way to read/write data
    - Uses a *language* to 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

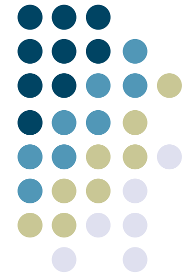
- You need to manage a lot of data (few kilobytes up to a few terabytes) that will persist across runs
- Need to be able to search through it quickly, update it quickly, or perform complex queries
- Need more safety than you can get with just files
- Need to support concurrent access



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

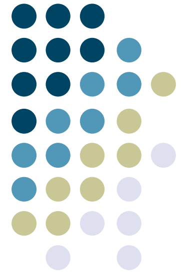
Example



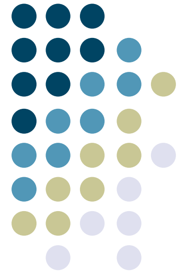
Customer Number	Customer Name	Customer Address
52352113	Keith Edwards	85 Fifth Street NW
62352922	Rich DeMillo	801 Atlantic Drive

Product Number	Description	List Price
2355XR32	Sterling Martini Shaker	$69.99
5233QM33	Olives	$5.99

Keys



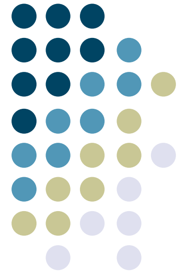
- Important concept: *key column(s)*
  - Special column(s) that uniquely identify a row
  - Contains a special number/token/identifier that uniquely identifies the row
    - Might be a "natural" key, or something we have to make up on our own
    - E.g., in a tax database, SSN might suffice as a key
- 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

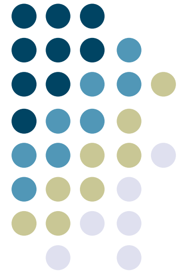
- Schemas are the *layout* of the data
  - The set of tables, the columns of those tables, etc
  - Represents the logical structure of your data
- Created by a *database designer*
  - This is the person who creates the schema for a particular application; not the person who writes the database itself
- 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

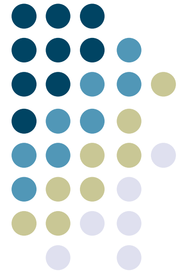


- Database jargon:
    - *Entities*: the real world things that are represented in the database
        - E.g., people, messages, chat sessions, etc.
    - *Attributes*: aspects of an entity that we want to represent
        - E.g., people have names, etc.
    - *Relationships*: Associations among entities
        - 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

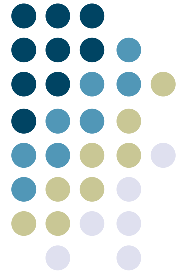


More on Relationships

- 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

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

Georgia Tech



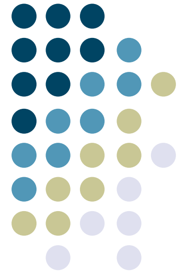
More on Relationships

- 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	Customer Name
12255193	Keith Edwards
93321355	Rich DeMillo

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

More on Relationships



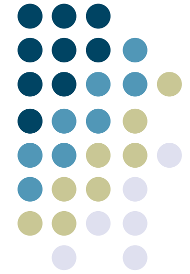
- Many-to-many
  - E.g., Amazon product/order database
  - Any given order might contain multiple distinct products
  - Any given product might be associated with multiple orders
- Relationships like this result in the creation of a *new table* to hold just the relationship

Product #	Description
12	Martini Shaker
53	Olives

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

Product #	Order #
12	B29
53	B29

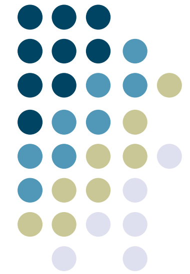
Database Design Principles



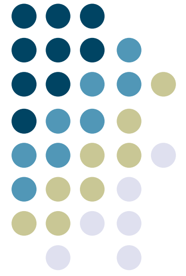
- "Normalizing" data relationships
  - Avoid as much redundancy as possible
  - Structure things so that "anomalies" can't happen
- Example:

```
                                                     Acme Industries
                                                         INVOICE

Customer Number: 1454                              Order Date: 11/05/06
Customer:     W. E. Coyote                         Terms: Net 30
     General Delivery                             Ship Via: USPS
     Falling Rocks, AZ 84211
     (599) 555-9345

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



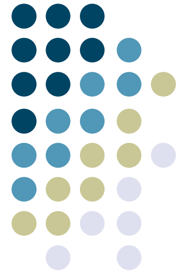
Cust. No.	Cust. Name	Cust. Addr.	Cust. City	Cust. State	Cust. ZIP	Cust. Phone	Order Data	Prod. No.	Desc.	Quant.	Unit Price	Ext. Amount
1454	W. E. Coyote	General Delivery	Falling Rocks	AZ	84211	599 -555 -9345	11/5/ 2006	SPR-2290 STR-67 HLM-45 SFR-1 ELT-1	Super strength springs Foot straps, leather Deluxe Crash Helmet Rocket, solid fuel Emergency Transmitter	2 2 1 1 1	24.00 2.50 67.88 128,200.40 78.88	48.00 5.00 67.88 128,200.40 0.00



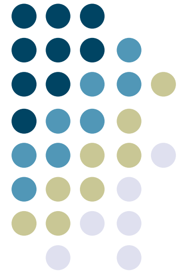
The Invoice Example

- Insert anomalies: can't insert a new row because of an artificial dependency on another relation
  - Two different entities are mixed in the same relation (table)
  - E.g., can't insert a new customer into the database unless they bought something, because all customer data is embedded in the invoice
- 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
- 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

In-Class Exercise

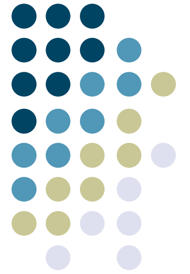


- Figure out a correct schema for the Invoice example
- Steps:
  1. Get rid of multivalued attributes
     - Why? Hard to select out what we want
     - Move multivalued attributes to a new table
     - Copy key from original table to the new one
  2. Make sure that non-key attributes only are determined 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
  3. 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!

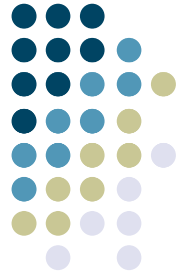


Databases in Practice

Database Creation, Queries, and Updates

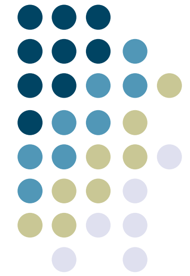


- So far, just talked about the logical structure of databases, and database design
- Now:
  - How to write a program to create that logical structure
  - How to get data into the database
  - How to get data out of a database
- Queries: mechanism for talking to a database
- Many mechanisms; the one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



SQL

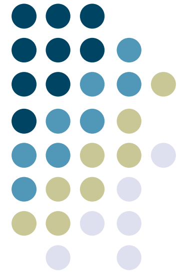
- The *Structured Query Language*
- Textual language for creating database structure, updates, and queries
- Pretty much the universal language for relational databases
  - Supported more-or-less the same across all RDBMS systems
  - Minor variances because vendors want to differentiate their products



Quick'n'Dirty Overview of SQL

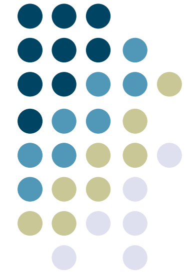
- Basic statements
  - CREATE
  - INSERT
  - UPDATE
  - DELETE
  - SELECT
- Basic column types
  - INT
  - REAL
  - VARCHAR
  - CHAR/CHAR(n)
  - DATE
  - TIME
  - BOOLEAN
  - OBJECT

Creating a New Database Table



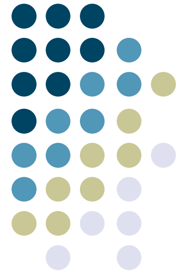
- CREATE CACHED TABLE table_name (col1, col2, ...)
  - CACHED means it actually gets saved to disk. Important!
  - In the parens are the definitions of the columns, separated by commas:
    - columnName type [GENERATED BY DEFAULT AS IDENTITY] [PRIMARY KEY]
    - GENERATED BY DEFAULT AS IDENTITY means 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!

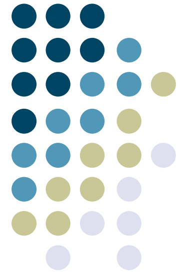
Georgia Tech



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!

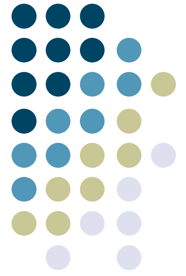
Georgia Tech



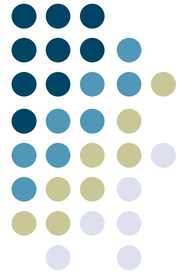
SQL Conditions

- 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



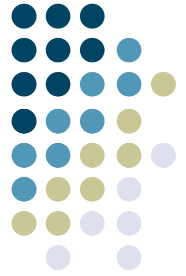
- Use UPDATE to change cells in existing rows
- UPDATE table_name SET column=expr WHERE condition
- Example:
  - UPDATE bookInfo SET retailPrice='59.99' WHERE isbn='07879312'
- Will update *multiple* rows that match the WHERE condition
- Like DELETE, leaving off the WHERE condition updates all rows in 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

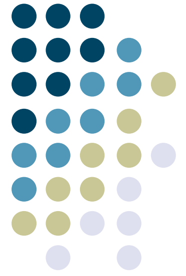
- SELECT is the workhorse of SQL: lets you get data out of your database
- SELECT * FROM table_name
  - Returns all rows from table_name
- SELECT columns FROM table_name
  - Returns only the specified columns from table_name
- SELECT DISTINCT columns FROM table_name
  - Removes duplicate values and returns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

Querying the Database (cont'd)



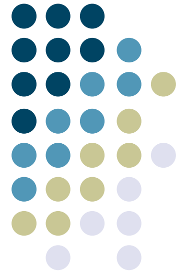
- SELECT columns FROM table_name WHERE condition
  - Lets you select out only the rows that match the specified condition, then return only the desired columns from those rows
- SELECT columns FROM table_name WHERE condition ORDER BY column
  - Same as above, only returns data ordered by the specified 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()

Georgia Tech



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

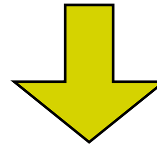


Advanced SQL: Joins

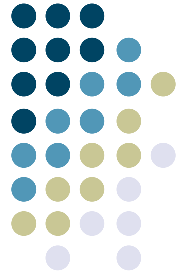
- Everything so far has been about queries over single tables
- Possible to write queries that span tables
  - Very powerful feature of relational databases
- SELECT * FROM books, publishers WHERE books.publisherID = publishers.publisherID AND publishers.city = 'New York' AND publishers.state = 'NY'
- 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



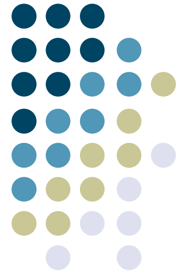
bookID	publisherID	title	author	name	city	state



Databases in the Wild

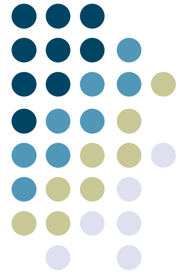
- Lots of database products and freeware in the world
    - Oracle, Access, MySQL, etc.... literally hundreds
- Most are implemented as *server processes*: your code connects to them over the network
    - Many are super-highend
        - E.g., Oracle
    - Some are more "consumer oriented"
        - 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



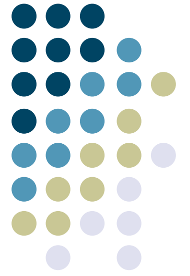
- http://hsqldb.sourceforge.net
- Freeware database, written in Java
- Can be embedded, or run as a server
- Download hsqldb.jar
  - Available on class website
  - Contains the entire database implementation, along with SQL interpreter, etc.
- Update your classpath to add this in
  - Same process as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

Using Databases from Java



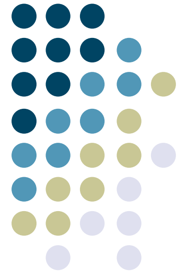
- Ideally, you'd like to be able to write your code so that you can "swap out" whatever database its using
  - To move higher-end, to move cheaper, because of bugs, whatever
- Java has a cool API that works across databases
- JDBC - the Java Database Connector API
  - You initialize JDBC by telling it 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



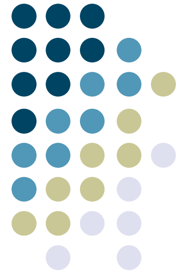
- Load the JDBC driver for your specific database
- Connect to the database
- Issue queries in the form of plain text SQL statements
- Shutdown and disconnect

Loading the Driver and Connecting to the Database

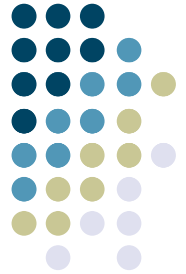


```
import java.sql as sql
import java.lang as lang

try:
    java.lang.Class.forName("org.hsqldb.jdbcDriver").newInstance()
except java.lang.ClassNotFoundException:
    print "No JDBC driver found; check classpath?"

try:
    connection = sql.DriverManager.getConnection("jdbc:hsqldb:file:" + dbname)
    statement = sql.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 DEFALT AS IDENTITY (START WITH 1) PRIMARY KEY,
    INITIATOR CHAR(64), START REAL)")
except sql.SQLException, ex:
    print "Trouble creating CHATS table", ex
```

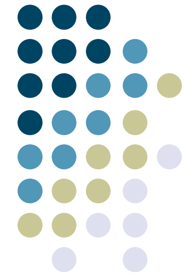


Adding Data to the Database

```
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
```

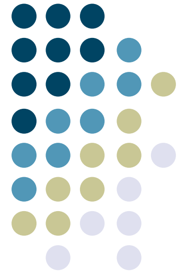
- Note use of quotes around string values, and use of str() to concatenate non-string values into a string!

Updating the Database



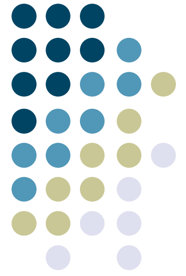
```
try:
    newTime = time.time()
    statement.executeQuery("UPDATE CHATS (START) VALUES (" +
    (str(newTime) + ")")
except sql.SQLException, ex:
    print "Trouble updating row", ex
```

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
```

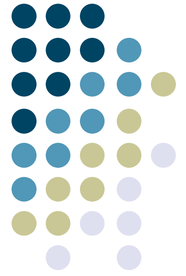
Georgia Tech



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 one 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:

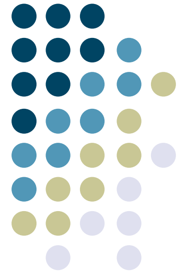
  statement.executeQuery("SHUTDOWN")

  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:

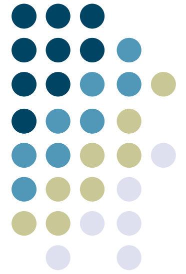
  statement.executeQuery("CHECKPOINT")

Hypersonic SQL Pragmatics



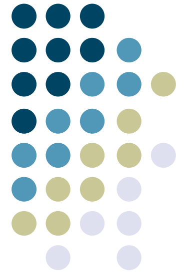
- Debugging when using a database
  - Exceptions, exceptions, exceptions
    - Put *every* call to the database in a try/except block
    - Don't just swallow the exception, *print it!*
  - Print statements are your friend
    - Lots of building long strings of concatenated text, variables, etc.
    - 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"

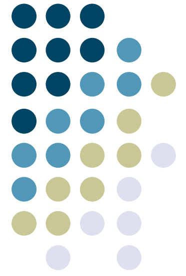
Georgia Tech



Hypersonic SQL Pragmatics

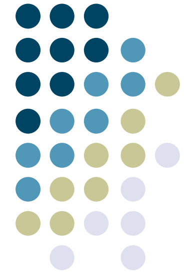
- Starting Over
- Sometimes you just need to recreate the whole database
- Two solutions:
    - DROP TABLE table_name
    - By hand, remove all of the database files mentioned previously

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

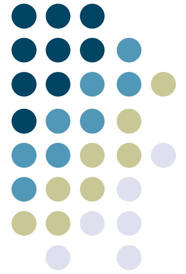


Chats		
ChatID INT *Key, automatically generated. NOTE different than conversation ID!*	Initiator CHAR(64) *Use name of participant*	Start REAL *Use time.time()*

Members_in_Chats	
ChatID INT	Member CHAR(64)

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

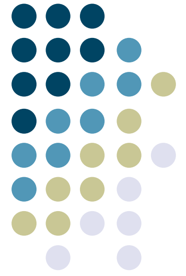
Messages_in_Chats	
ChatID INT	MessageID INT



Extra Tips

- Doing the time-related stuff:

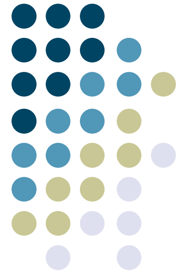
```
import time

# 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

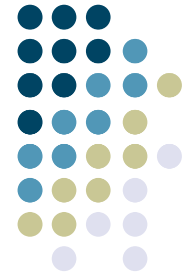
Georgia Tech



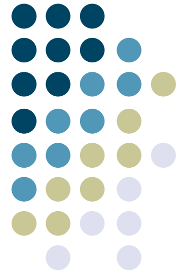
Suggested Coding Strategy

- difficulty(google) < difficulty(gui) < difficulty(db) << difficulty(net)
- Less than three weeks to go. Start early, finish early!

functionality-criteria-4



Feature	Points
Overall Program Structure	**20**
• Previous code still works!	5
• Correct "main" handling	5
• Separate database per user	5
• Good modularity and exception handling	5
Database Creation and Update Functionality	**40**
• Correctly connect to and create database *Use a separate database name per user* *Catch and display any exceptions*	8
• Update chats table when new chats start *Store information in chats table correctly* *Catch and display any exceptions*	8
• Update messages table when each message goes by *Store information in messages table correctly* *Catch and display any exceptions*	8
• Keep intersection tables (messages_in_chats, members_in_chats) correctly updated *Add a row to messages_in_chats to indicate each new message associated with a chat* *Add a row to members_in_chats for each member associated with a chat* *Catch and display any exceptions*	8
• Correctly shutdown database *Correctly issue SHUTDOWN command* *Close the database statement and connection* *Catch and disiplay any exceptions* *Optional: checkpoints throughout the code*	8
Database Retrieval and Display Functionality	**40**
• Produce list of all members you've ever chatted with *Fetch all distinct members from the chats table* *Display correctly in a window* *Catch and display any exceptions*	15
• Display history window for selected user *Fetch all chats associated with a given user* *Fetch all messages associated with those chats* *Display in an organized way in a window* *Catch and display any exceptions*	25
Bonus	**10**
• Fancy (and *correct!!*) schema above and beyond what's been provided	5
• "Statistics" UI giving details about average numbers of chats, average numbers of messages per chat, average chat duration, etc.	5



Schedule for Remaining Weeks

- Monday 4/18: full class devoted to lab
- Monday 4/25: remaining in-class presentations, rest of class devoted to lab
- Friday 4/29: assignment due at 11:59PM
    - Turn in whatever you've got
    - No extensions into finals week
- Remember: no final!