Embedded SQL

- SQL is used inside programs
  - Actually built into some languages
  - Create a string representing the query
  - Pass that string to the database to interpret
- Concepts
  - **Connection**: connection to the database
    - Accessed by a URL giving type, host, database, user, pwd, ...
  - **Statement**: set of SQL statements to be executed as one
    - Typically a single query or set of updates
  - **ResultSet**: iterator over a returned table
    - Can iterate over tuples in the returned values
    - Can access fields of the current tuple
Using Embedded SQL Safely

- Queries and updates are built as strings
  - Provides flexibility
  - Lets program add user-entered values to the queries
  - Can be problematic if user values are non-standard
- Prepared statements
  - Queries with variables are defined as strings
  - Variables in the query are represented by $, $i, or ?
  - Values are passed when the query is used
  - Can be faster (database can optimize the query once)
  - Safer and more secure
  - **Use when possible (and its always possible)**

```python
import cx_Oracle
con = cx_Oracle.connect('pythonhol/welcome@localhost/orcl')
cur = con.cursor()
cur.prepare('select * from departments where department_id = :id')
cur.execute(None, {'id': 218})
res = cur.fetchall()
print res
cur.execute(None, {'id': 118})
res = cur.fetchall()
print res
cur.close()
con.close()
[pythonhol@localhost ~]$`
SQL INSERT Statement

- **INSERT INTO** table ( field, ..., field )
  VALUES ( val,...,val)
  - List of fields is optional, but should be there
    - Avoids problems in future, reminds reader of what is what
  - Values can be
    - DEFAULT
    - Constants [ true, false, 0, ... ]
    - Built-in Functions [ CURRENT_TIMESTAMP ]
    - Variables – use $, $i, ? and pass values separately
    - Results of queries [ (SELECT id FROM Artist A WHERE A.name = 'nsync') ]

- **INSERT INTO** table (field, ..., field) SELECT ...
SQL UPDATE Statement

- UPDATE table SET field = value WHERE condition
  - SET field = value, field = value, ...
  - Values as in an insert statement
  - WHERE as in normal select
- Can update multiple rows at once
  - Be careful with the condition
SQL DELETE Statement

- DELETE FROM table WHERE condition
  - Removes all rows in table where condition is true
  - If condition is omitted, deletes all rows
Relational Database Schema

- The tables define the **database schema**
  - Table names
  - Set of attributes/fields for each table
  - Each attribute is typed
  - Tables can have a KEY
  - Tables can have permissions
  - Tables can have constraints (e.g. NOT NULL, FOREIGN KEY)
- Schema can contain other information
  - Indices on the tables
  - Virtual tables (VIEWS)
Rules for Table Organization

• How the tables are organized is important
• Based on the relationships among data items
  • If \( x \Rightarrow y \) (e.g. DiskId \Rightarrow Title, length,...) put in same table
  • If \( x \) is used multiple times, make it a separate table
    • Especially if \( x \) has multiple values or long values
  • If \( x \) can occur multiple times for \( y \), create a separate table
    • Just for relating \( x \) and \( y \)
  • If table will be referenced by other tables, add an id field
    • As a key field
Database Manager Position

- In general this is a difficult problem
  - Need to understand semantics of the data
  - Need to understand how the data will evolve
- If you get it wrong, what happens?
  - With only queries, not much (performance)
  - If the data is update, it might become inconsistent
    - Artist information for example
  - If the data relationships are updated, might not fit structure
    - Multiple artist CDs
- Database manager position
  - Handles setting up, changing, updating, etc. the database
Transactions

- Multiple operations might need to be done together
  - BEGIN TRANSACTION name
    - Operation
    - Operation
    - Operation
  - COMMIT TRANSACTION name
- Example: withdraw and deposit to transfer funds
- Database guarantees that transactions appear sequential
  - No intermediate changes by others
  - Operations all done or all not done (atomic)
  - ACID semantics
- Use any-db-transaction when using Node.js
Batch Processing

• Large numbers of inserts (or updates) can be inefficient
  • But doing them all at once can be a lot faster
  • Can create BATCH updates
    • Like transactions (can use transactions for this)
    • Part of Statement interface for embedded SQL
  • Most databases provide a means for batch inserts
    • Insert multiple tuples into a relation
    • Data comes from a file
Schema Updates

- Tables are going to change
  - Applications evolve over time
  - New data is needed, old data no longer needed

- Adding tables is easy
- Modifying tables is possible
  - ALTER TABLE ADD newfield int DEFAULT 0
  - Can remove fields, add constraints, add indices, ...
  - Beware of constraints between tables

- This can be an expensive operation
  - And might require other computation (computing initial values)

- Can be tricky
  - Might involve dependencies, indices, constraints
  - Maintaining a test database and a production database
Permissions

- External database systems control access
  - Users (independent of system users)
  - Permissions (on a table-by-table basis)
- When you set up the database you can set up users
  - With different permissions on different tables
- This can provide a safeguard in your application
  - Application normally runs as a limited access user
  - Administrative tasks run with more privileges
- Think about permissions when you set up your database
  - Create a separate user for your application
Which SQL Database System to Use

- **Sqlite3, Derby**
  - Only for experimentation, small-scale applications
  - Embedded use only (single access point)

- **MySQL, PostgreSQL**
  - Most common for web applications
    - Extensions to both to handle scaling are available
  - Roughly equivalent
    - PostgreSQL has better extensibility

- **Db2, SqlServer, Oracle**
  - For larger databases, better support
SQL Databases

• Pros
  • ACID semantics (transactions, guaranteed consistency)
  • Queries handled by the database engine (little code)
  • Portable
  • Data consistency through the schema
  • Supports complex indices, triggers, constraints

• Cons
  • Replication of the database is difficult
  • Don’t scale that well to huge databases
  • Not good at storing unstructured documents (blobs)
  • Complex Database Manager issues
  • Changing schema requires thought and effort

• Can we do better for web applications?
Suppose we want to go beyond SQL databases. NoSQL databases are a response to the needs of modern web applications. Which is **not** a characteristic of such databases?

A. They can be accessed directly from HTML5 using database extensions
B. They support eventual consistency rather than immediate consistency
C. They typically use a simple or specialized query language.
D. They provide the ability to easily shard or replicate data
E. They provide specialized implementations suitable for particular types of applications.
NOSQL Databases

- **No SQL**
  - Essentially key-value stores
    - Arbitrary values, single key; look up by key
    - Indexed by key
  - Eventual consistency (no transactions)
- **Not-Only SQL**
  - Trend to make these more like SQL databases
  - NewSQL databases combine SQL queries with No-SQL-style stores
- **Index on particular fields**
  - Field value -> { key } is what is in index
- **Simple queries**
  - Given a set of field values, find corresponding data values
  - Done my intersecting key sets
  - Take resultant key set and allow iteration over values
MongoDB: a simple (sophisticated) NoSQL

- Store JSON objects

```javascript
db.cds.insert(
    {
        diskid: '2a04b804',
        title: 'Body & Soul',
        artist: 'SPEED',
        length: 1210,
        genre: 'jpop',
        year: 1996,
        tracks: [
            { name: 'Body & Soul', artist: 'SPEED', length: 302, number: 0, offset: 187 },
            { name: 'I Remember', artist: 'SPEED', length: 278, number: 1, offset: 22892 },
            { name: 'Body & Soul (hand Bag Mix)', artist: 'SPEED', length: 323, number: 2, offset: 43777 },
            { name: 'Body & Soul (instrumental)', artist: 'SPEED', length: 302, number: 3, offset: 68070 }
        ]
    }
);
```
MongoDB Collections

• **Basic Storage is a collection**
  • Set of json objects
  • Each has a unique identifier (generated by Mongo)
  • Fast lookup based on identifier

• **Access methods on a collection**
  • `db.<collection>.find( query, projection)`
  • Query: `{ field : { $eq : <value> }, ... }`  ($eq,$gt, ...)
    • Field can be “field” or “field.subfield”
    • Can do most queries over a single collection
  • Projection: `{ field : 0|1, .... }`
    • 1 => include field, 0 => exclude field
MongoDB Example Queries

- `db.cds.count()`
- `db.cds.find( { artist : ‘Taylor Swift’ } )`
- `db.cds.find( { “tracks.artist” : ‘Jacques Brel’, “tracks.name” : “Mathilde” } )`
- `db.cds.runCommand(“text”, { search: ‘Paris’ } )`
MongoDB Indices

- `db.<collection>.createIndex(keys,options)`
  - Keys: `{ field : 1 }` (for ascending)
  - `ensureIndex` in some versions
- Mongo supports text indexing
  - Keys: `{ field : `text` }`
  - Keys: `{ `$$**`` : `text`` }`
- Text query
  - `db.<collection>.runCommand(`text`, { search: `words` })`)
  - Words: if multiple, then OR of the words (ala Google, w/ ranking)
  - Can use `"word1" "word2" "word phrase"` as well (and)
MongoDB CRUD

- `db.<collection>.insert({ ... })`
  - Newer versions support `insertMany`
- `db.<collection>.update({ query}, { update })`
  - Query: similar to a find query
  - Update: `{ $set: { field: 'value' } }`
- `db.<collection>.remove({ query })`
- Bulk operations via `db.<collection>.bulkWrite([ ... ])`
Distributed Mongo

- Mongo makes it easy to split up a collection
  - Into distinct portions called shards
  - Based on mongo-generated key or on user key
    - Can shard database to match different servers
  - Makes very large databases possible
  - Makes very large databases faster
- Also does distributed processing
  - Handles scaling to very large databases
Mongo Transactions

• No ACID guarantees
  • Clients may see writes before they are committed
  • If a query updates multiple documents, might read some updates and not others

• BASE instead
  • High availability
  • Eventual consistency

• Newer versions provide transactions
NOSQL Databases

• **Pros**
  • Can store arbitrary objects
  • Easy to set up and use (little management)
  • Easy to change
  • Scales nicely

• **Cons**
  • No transactional guarantees
  • Data consistency is up to the user
  • Complex queries need to be coded explicitly
What Type of Database to Use

• How important are ACID guarantees
  • Newer versions of Mongo provide some of this
• How important is data consistency
  • SQL provides constraints and triggers
• How important is scalability
  • Newer versions of SQL database provide sharding, etc.
• How varied and complex are the queries
  • Complex, unanticipated NoSQL queries require code
• How complex is the data and how often does the schema change
  • Complex data is much easier in NoSQL
• Might want a combination of databases
  • Some for complex data
  • Some for users, authentication, etc. where ACID is useful
Elevator Talks

• Two Minute Sales Pitch (includes 30 seconds for questions)
• Convince a someone to invest in your project
• Important points
  • What is the problem
  • What is your solution
  • Why is this important
  • Why should the audience be interested
Next Time

• Database LAB
  • Do the pre labs
  • Look up Six Degrees of Kevin Bacon
• Monday/Wednesday: Elevator talks