PASSWORDS
TREES AND HIERARCHIES

CS121: Introduction to Relational Database Systems
Fall 2016 – Lecture 24
Account Password Management

- Mentioned a retailer with an online website...
- Need a database to store user account details
  - Username, password, other information
- How to store a user’s password?
- What if the database application’s security is compromised?
  - Can an attacker get a list of all user passwords?
  - Can the DB administrator be trusted?
- Do we actually need to store the original password??
A Naïve Approach

- **A simple solution:**
  - Store each password as plaintext
    ```sql
    CREATE TABLE account (
        username VARCHAR(20) PRIMARY KEY,
        password VARCHAR(20) NOT NULL,
        ...,
    )
    ```

- **Benefits:**
  - If user forgets their password, we can email it to them

- **Drawbacks:**
  - Email is unencrypted – passwords can be acquired by eavesdropping
  - Users tend to use the same password for many different accounts
  - If database security is compromised, attacker gets all users’ passwords
  - Of course, an unreliable administrator can also take advantage of this
A safer approach is to hash user passwords

- Store hashed password, not the original
- For authentication check:
  1. User enters password
  2. Database application hashes the password
  3. If hash matches value stored in DB, authentication succeeds

Example using MD5 hash:

```sql
CREATE TABLE account (
    username VARCHAR(20) PRIMARY KEY,
    pw_hash CHAR(32) NOT NULL,
    ...)
```

To store a password:

```sql
UPDATE account SET pw_hash = md5('new password')
WHERE username = 'joebob';
```
Want a cryptographically secure hash function:
- Easy to compute a hash value from the input text
- Even small changes in input text result in very large changes in the hash value
- Hard to get a specific hash value by choosing input carefully
- Should be collision resistant: hard to find two different messages that generate the same hash function

MD5 is not collision resistant 😞
- “[MD5] should be considered cryptographically broken and unsuitable for further use.” – US-CERT

SHA-1 was also discovered to not be very good
Most people use SHA-2/3 hash algorithms now
Hashed Passwords (3)

- **Benefits:**
  - Passwords aren’t stored in plaintext anymore

- **Drawbacks:**
  - Handling forgotten passwords is a bit trickier
    - Need alternate authentication mechanism for users
  - Isn’t entirely secure! Still prone to **dictionary attacks**.

- Attacker computes a dictionary of common passwords, and each password’s hash value
  - Use hash-values to look up the corresponding password
  - If attacker gets the hash values from the database, can crack some subset of accounts
Hashed, Salted Passwords

- Solution: salt passwords before hashing
- Example:

```
CREATE TABLE account (  
  username VARCHAR(20) PRIMARY KEY,  
  pw_hash CHAR(32) NOT NULL,  
  pw_salt CHAR(6) NOT NULL,  
  ...  
);
```

- Each account is assigned a random salt value
  - Salt is always a specific length, e.g. 6 to 16 characters
- Concatenate plaintext password with salt, before hashing
- Attacker would have to compute a dictionary of hashes for each salt value... Prohibitively expensive!
Password Management

- Basically **no** reason to store passwords in plaintext!!
  - Users almost always use the same passwords in multiple places!
  - Only acceptable in the simplest circumstances
  - (You don’t want to end up on the news because your system got hacked and millions of passwords leaked…)

- Almost always want to employ a secure password storage mechanism
  - Hashing is insufficient! Still need to protect against dictionary attacks by applying salt
  - Also need a good way to handle users that forget their passwords
Many DB schemas need to represent trees or hierarchies of some sort.

Example: parts-explosion diagrams

“How are parts and subsystems assembled?”

“How much does a subsystem weigh?”

Other computations based on parts in subsystems.
Many kinds of relationships between people
- Employee/manager relationships within an organization
- Social graph data from a social network
- Family trees of people, or pedigree data for animals
Trees and Hierarchies (3)

- Most common way of representing trees in the DB is an adjacency list model
  - Each node in the hierarchy specifies its parent node
  - Can represent arbitrary tree depths

- Example: employee database
  - `employee(emp_name, address, salary)`
  - `manages(emp_name, manager_name)`
  - Both attributes of `manages` are foreign keys referencing `employee` relation
Adjacency list model is only one of several ways to represent trees and hierarchies.

Different approaches have different strengths and weaknesses.

Some approaches to consider:
- Adjacency list models
- Nested set models
- Path enumeration models
General Design Questions

- How hard is it to access the tree?
  - Retrieve a specific node
  - Find the parent/children/siblings of a particular node
  - Retrieve all leaf nodes in the tree
  - Retrieve all nodes at a particular level in the tree
  - Retrieve a node and its entire subtree

- Also, path-based queries:
  - Retrieve a node corresponding to a particular path from the root
  - Retrieve nodes matching a path containing wildcards
  - Is a particular path in the hierarchy?
General Design Questions (2)

- How hard is it to modify the tree?
  - Add a single node
  - Delete a leaf node
  - Delete a non-leaf node
    - What to do with subtree of deleted node?
  - Move a subtree within the tree

- How to enforce constraints in the schema?
  - Enforce only one root
  - Disallow cycles in a tree
    - Simplest example: “my parent can’t be myself”
  - Disallow multiple parents (tree vs. directed acyclic graph)
  - Enforce a maximum child-count, maximum depth, etc.
Deleting Nodes

- What happens when a non-leaf node is deleted?

- Option 1:
  - Delete entire subtree as well

- Option 2:
  - Promote one child node into removed node’s position
  - (Or, replace with placeholder)

- Option 3:
  - Make all child nodes into children of deleted node’s parent
Adjacency List Model

- Very common approach for modeling trees
- Each node specifies its parent node
- Relationship between nodes are frequently stored in a separate table
  - e.g. employee and manages
  - Can represent multiple trees without null values
  - Can have employees that are not part of the hierarchy
Adjacency List Model (2)

- **Strengths:**
  - A very flexible model for representing and manipulating trees
  - Easy to add a new node anywhere in the tree
  - Easy to move a whole subtree

- **Weaknesses:**
  - Deleting a node often requires extra steps (to relocate children)
  - Operations on entire subtrees are expensive
  - Operations applied to a particular level across a tree are expensive
  - Looking for a node at a specific path is expensive
  - (All of these operations require multiple queries to identify the nodes affected by the operation.)
Queries on Subtrees

- Example query using subtrees:
  - For every manager reporting to a given Vice President, find the total number of employees under that manager, and their total salaries
  - Computing an aggregate over individual subtrees

- Another example:
  - Database containing parts in a mechanical assembly
    - Parts combined into sub-assemblies
    - Sub-assemblies and parts combined into larger assemblies
    - Top level assembly is the entire system
  - Find number of parts, the total cost, and the total weight of each sub-assembly in the system
Finding Nodes in Subtrees

- To find all nodes in a specific subtree, must iterate a query using a temporary table
  - Example: Find all employees under manager “Jones”
    ```sql
    CREATE TEMPORARY TABLE emps
    ( emp_name VARCHAR(20) NOT NULL );
    INSERT INTO emps VALUES ('Jones');
    INSERT INTO emps SELECT emp_name FROM manages
    WHERE manager_name IN (SELECT * FROM emps)
    AND emp_name NOT IN (SELECT * FROM emps);
    ```
  - Iterate last statement until no new rows added to temp table
  - DATABASES often report a count of how many rows are inserted/modified/deleted by each DML operation
Can also store a “depth” value in the result

```sql
CREATE TEMPORARY TABLE emps (
    emp_name VARCHAR(20) NOT NULL,
    depth    INTEGER NOT NULL
);

INSERT INTO emps VALUES ('Jones', 1);

-- Some variable i, where i = 1 initially:
INSERT INTO emps SELECT emp_name, i + 1 FROM manages
    WHERE manager_name IN
        (SELECT * FROM emps WHERE depth = i);
```

- Each level of the hierarchy has the same depth value
- Slightly more efficient than the previous version
  - Each iteration has fewer rows to consider
Best to implement this as a stored procedure
- Don’t involve command/response round-trips with application code, if possible!
- Perform processing entirely within the DB for best performance

Still acceptable if application has to drive the iteration process
- Most DB connectivity APIs let you create temporary tables, get number of rows changed, etc.
Other Adjacency List Notes

- Must manually order siblings under a node
  - Add another column to the table for ordering siblings

- Adjacency list model is also good for representing graphs
  - Actually easier than using for trees, because fewer constraints are required
  - Traversing the graph requires temporary tables and iterative stored procedures
  - (Other representations we will discuss today aren’t well-suited for graphs at all!)
Recursive Queries in SQL

- For a long time, SQL did not support recursive queries
  - Only way to traverse adjacency-list data model was to use a temporary table and repeated queries, as described

- Now, most databases also support some form of recursive SQL query
  - e.g. PostgreSQL 8.4+, SQLServer, Oracle (not MySQL 😞)
  - If available, makes it much easier to traverse adjacency-list datasets

- Still requires repeated queries against the database…
  - Even though query is easier to write, it’s still slow to execute!
Recursive Queries in PostgreSQL

- Example using SQL99/Postgres 8.4 syntax
- Find all employees directly or indirectly managed by Jones:

```sql
WITH RECURSIVE empl AS (
    SELECT employee_name, manager_name
    FROM manager
    UNION ALL
    SELECT e.employee_name, m.manager_name
    FROM empl AS e JOIN manager AS m
    ON e.manager_name = m.employee_name
)
SELECT * FROM empl
WHERE manager_name = 'Jones';
```
Can compute the depth in the hierarchy, too:

WITH RECURSIVE empl AS (  
    SELECT employee_name, manager_name,
            1 as depth  
    FROM manager  
    UNION ALL  
    SELECT e.employee_name, m.manager_name,
            e.depth + 1 AS depth  
    FROM empl AS e JOIN manager AS m  
    ON e.manager_name = m.employee_name)  
SELECT * FROM empl  
WHERE manager_name = 'Jones';
Microsoft SQLServer 2005/2008 also uses WITH clause for recursive queries

- Similar approach, using a non-recursive subquery and a recursive subquery, combined with UNION [ALL]
- Doesn’t use a RECURSIVE modifier

Neither Postgres nor SQLServer recursive queries can handle cycles in the data...

- Introducing a cycle into the data causes the query to infinite-loop
Oracle Recursive Queries

- Oracle has much more sophisticated recursive query support
- A simple example:
  ```sql
  SELECT employee_name, manager_name, LEVEL
  FROM manager
  CONNECT BY PRIOR employee_name = manager_name;
  ```
  - PRIOR modifier specifies parent in parent/child relationship
  - LEVEL is a pseudocolumn specifying level in the hierarchy
- Can also do many other things, such as:
  - Specify the initial data-set to iterate on
  - Specify the order of siblings in each level of hierarchy
  - Detect and report which nodes are leaves in the hierarchy
  - Detect and report cycles in the dataset
  - Generate the full path from root to each node in the result
Final Notes on Recursive SQL Queries

- As stated earlier:
  - Recursive SQL queries make it much easier to write the query that traverses adjacency-list data...
  - Still requires the DB engine to repeatedly issue queries until the entire hierarchy is traversed!

- If an application needs to store hierarchical or graph data, and must select entire subtrees quickly:
  - Adjacency-list model is the most expensive model to use!
Nested Set Model

- A model optimized for selecting subtrees
- Represents hierarchy using containment
  - A node “contains” its children
- Give each node a “low” and “high” value
  - Specifies a range
  - Always: low < high
- Use this to represent trees:
  - A parent node’s [low, high] range strictly contains the ranges of all child nodes
  - Sibling nodes have non-overlapping ranges
Example using Nested Sets

- **Employee hierarchy:**

  - **Bill**
    - **Charlie**
      - **Fred**
      - **Gary**
    - **Dave**
      - **John**
      - **Rob**
      - **Sam**
      - **Ted**

- **Nested set hierarchy:**

  - **Bill** (1, 18)
    - **Charlie** (2, 7)
      - **Fred** (3, 4)
      - **Gary** (5, 6)
    - **Dave** (8, 17)
      - **John** (9, 16)
      - **Rob** (10, 11)
      - **Sam** (12, 13)
      - **Ted** (14, 15)
Selecting Subtrees

- Each parent node contains its children:

  - Easy to select an entire subtree
    - Select all nodes with low (or high) value within node’s range
  - Can also select all leaf nodes [relatively] easily
    - If all range values separated by 1, find nodes with low + 1 = high
    - For arbitrary range sizes, find nodes that contain no other node’s range values (requires self-join)
Nested Set Model

- **Strengths:**
  - Very easy to select a whole subtree
    - Very important for generating reports against subtrees
  - Tracking the order of siblings is built in

- **Weaknesses:**
  - Some selections are more expensive
    - e.g. finding all leaf nodes in tree requires self-join
  - Constraints on range values are expensive
    - `CHECK` constraint to ensure low < high is cheap…
    - `CHECK` constraint to verify other interval characteristics is expensive!
  - Pretty costly to insert nodes or restructure trees
    - Have to update node bounds properly
Adding Nodes In Nested Set Model

- If adding a node:
  - Must choose range values for new node correctly
  - Often need to update range values of many other nodes, even for simple updates

- Example:
  - Add new employee Harry under Charlie
  - Must update ranges of most nodes in the tree!
Supporting Faster Additions

- Can separate range values by larger amounts
  - e.g. spacing of 100 instead of 1
  - Situations requiring range-adjustments of many nodes will be far less frequent

- Can implement tree-manipulation operations as stored procedures
  - “Add a node,” “move a subtree,” etc.
Path Enumeration Models

- For each node in hierarchy:
  - Represent node’s position in hierarchy as the path from the root to that node
  - Entire path is stored as a single string value

- Node enumeration:
  - Each node has some unique ID or name
  - A path contains the IDs of all nodes between root and a particular node
  - If ID values are fixed size, don’t need a delimiter
  - If ID values are variable size, choose a delimiter that won’t appear in node IDs or names
PathEnumerationModel (2)

- Path enumeration model is fastest when nodes are retrieved using full path
  - “Is a specified node in the hierarchy?”
  - “What are the details for a specified node?”
  - Adjacency list model and nested set model simply can’t answer these queries quickly!

- Example:
  - A database-backed directory service (e.g. LDAP or Microsoft Active Directory)
  - Objects and properties form a hierarchy
  - Properties are accessed using full path names
    - “sales.printers.colorprint550.queue”
Strengths and Weaknesses

- Optimized for read performance
  - Retrieving a specific node using its path is very fast
  - Retrieving an entire subtree is also pretty fast
    - Requires text pattern-matching, but matching a prefix is fast, especially with a suitable index on the string
    - Example: Find all sales print servers
      - Use a condition: path LIKE 'sales.printers.%'

- Adding leaf nodes is fast

- Restructuring a tree can be very slow
  - Have to reconstruct many paths...

- Operations rely on string concatenation and string manipulation
Edge Enumeration

- Paths can enumerate edges instead of nodes
  - Each level of path specifies index of node to select
- Primary method used in books
  - Example:

<table>
<thead>
<tr>
<th>Edge Enumeration</th>
<th>Section Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>SQL</td>
</tr>
<tr>
<td>3.1</td>
<td>Background</td>
</tr>
<tr>
<td>3.2</td>
<td>Data Definition</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Basic Domain Types</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Basic Schema Definition in SQL</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- Like node enumeration, requires string manipulation for most operations
Summary: Trees and Hierarchies

- Can represent trees and hierarchies in several different ways
  - Adjacency list model
  - Nested set model
  - Path enumeration model

- Each approach has different strengths
  - Each is optimized for different kinds of usage

- When designing schemas that require hierarchy:
  - Consider functional and non-functional requirements
  - Choose a model that best suits the problem