CS 3200 Topic Overview

Resource Suggestions

Read about these topics in your favorite textbook. (See syllabus for recommendations.)

To look up SQL statements and their use in Postgres, do a Web search for *postgres 9 SQL_keyword*, where SQL_keyword is the one you want to learn more about. We are using a Postgres 9.x server. While Postgres, like most DBMS, implements most of the SQL standard, looking at Postgres-specific documentation will tell you about Postgres-specific differences.

Many people also find the SQL tutorial by W3Schools helpful: http://www.w3schools.com/sql/.

Lecture 1

Motivation, introduction, and overview (see slides)

Lecture 2

Entity-Relationship Model (ERM)

What are entities, entity sets, relationships, and relationship sets?

Creating a basic ERM design by approaching the problem from two directions: (1) diagram first, then check if the resulting relations can actually store the data; and (2) desired relations first, then try to create the matching ER diagram.

Basic transformation of an entity set and a relationship set into SQL code:

- CREATE TABLE statement
- Declaration of attributes and their types
- PRIMARY KEY
- UNIQUE
- FOREIGN KEY … REFERENCES
- ON DELETE and ON UPDATE

Lecture 3

ERM continued

Key constraints ("at most one") and how to express them in the ER diagram and in the corresponding SQL code

Participation constraints (“at least one”) and how to express them in the ER diagram and in the corresponding SQL code:

- NOT NULL condition for an attribute
- Need for more powerful constructs (to be discussed in a future lecture)

Weak entities and how to express them in the ER diagram and how to map the diagram to relations

Hierarchies using the “ISA” design element and their transformation into matching relations. Relevant properties of the hierarchy:

- Overlap: can an entity be in multiple subclasses?
- Coverage: does every superclass entity have to be in one of the subclasses?

Lecture 4

ERM completed

Aggregation to let a relationship participate in another relationship

Design choices for ERM

- Entity versus attribute; example: address of a person
- Entity or relationship, and placement of attributes; example: manager and budget
- Arity of a relationship; example: sale (customer, product, store) versus health insurance policy (employee, policy, dependents)

Useful SQL commands for implementing a design:

- DROP TABLE
- ALTER TABLE
- INSERT INTO … VALUES …
- DELETE FROM … WHERE …
- UPDATE … SET … WHERE …

Lecture 5

Relational algebra: useful for representing query plans

Basic relational operators: selection, projection, cross-product, set difference, union

“Convenience” operators, composed from the basic ones: intersection, join (equi-join, natural join)

Algebra expression represented as query plan tree with relations and operators as nodes; simple query optimization by pushing selection and projection “down”

Example Relations

Students:

SID	Name	Age	GPA
1	Alice	18	3.5
2	Bob	27	3.4
3	Carla	20	3.8
4	Dan	20	3.9

Reservations:

SID	BookID	Date
2	B10	01/17/12
3	B11	01/18/12

Books

BookID	Topic	Title
B10	DB	Intro DB
B11	PL	More PL

StudentsXL:

SID	Name	Age	GPA
1	Alice	18	3.5
2	Bob	27	3.4
3	Carla	20	3.8
4	Dan	20	3.9
6	Frank	20	3.8
7	Gina	27	3.8
8	Hal	18	3.5

Lecture 6

Relational calculus: basis for SQL

Basic structure of a domain-relational calculus expression (look up formal definition in a textbook)

Expressing joins, selection, and projection in relational calculus

Equivalence of relational algebra and safe queries in relational calculus

Relational completeness: SQL can express every relational algebra query

Lecture 7

Correspondence between calculus expression and corresponding SQL query

Basic SQL query: SELECT [DISTINCT] … FROM … WHERE …

SQL query semantics: the conceptual evaluation strategy to find the result of a given SQL query

Examples of conditions in the WHERE clause, including LIKE for string types

Nested queries and their conceptual evaluation (nested-loops style)

Nested queries with correlation

SQL keywords:

- IN, NOT IN, EXISTS, NOT EXISTS, UNIQUE, NOT UNIQUE
- op ANY, op ALL; where op can be <, >, <=, >=, =, or <>
- Aggregate operators: COUNT, SUM, AVG, MIN, MAX

Use of aggregate operators in the basic SQL query

Lecture 8

Two SQL versions of the query to find students who reserved all books:

- Start with first-order logic formulation (for all… there exists…), turn it into (not exists… not exists…), then into SQL query with two nesting levels using NOT EXISTS
- Start with set-based analysis (all books minus all books reserved by the student), then create SQL query using EXCEPT

Aggregation queries with GROUP BY and HAVING

SQL query semantics: the conceptual evaluation strategy to find the result of a given SQL query with GROUP BY and HAVING

Difference between conditions in the HAVING clause versus the WHERE clause

Which attributes can appear in the SELECT clause of a GROUP-BY query: grouping attributes and aggregates of other attributes

Subtleties in the HAVING clause: HAVING 2 <= COUNT(*) versus expressing the group-wise COUNT(*) with a sub-query such as (SELECT COUNT(*) FROM Students S2 WHERE S.age = S2.age)

Lecture 9

Composing more complex SQL queries step-by-step: write separate queries to create intermediate results, use the intermediate results as new relations, then create the final query by inlining the intermediate queries into the final query

Missing values: NULL

- Need for a three-valued logic: true, false, unknown
- Semantics with NULL for comparisons, Boolean formulas, duplicate definition, arithmetic operators, and SQL aggregates

Views

- CREATE VIEW and DROP VIEW statements
- Benefits and tradeoffs of views
- Materialized views: speed up queries, but make updates more expensive

Integrity constraints (ICs)

- Domain constraints (attribute types), primary key, foreign key
- CHECK: general version with sub-query versus simple per-tuple constraint
    - Postgres: does not allow sub-queries, but supports functions (and a function can run a sub-query…)

Lecture 10

Integrity constraints (ICs) continued:

- IC involving multiple tables and why not to use CHECK for such ICs
- Looking ahead: triggers can also implement ICs

Another SQL feature: LEFT/RIGHT/FULL OUTER JOIN

SQL functions to execute a list of SQL statements: CREATE FUNCTION statement, specifying input and output parameters, returning an individual tuple or a set, use of $$ to define a string constant, how to call a function and where to use it

User-defined aggregates: CREATE AGGREGATE statement, state value, state transition function, initialization, finalization

Triggers:

- Event, Condition, Action parts
- Possible events, conditions, and actions

Lecture 11

Insert: discussion of the solution for HW 2 (ER diagram, SQL translation); the documents are available on Blackboard

Triggers continued:

- Trigger timing
- Difficulty of reasoning about what will happen when multiple triggers fire, trigger can fire each other, or themselves repeatedly
- CREATE TRIGGER statement in Postgres: BEFORE UPDATE ON, FOR EACH ROW, EXECUTE PROCEDURE
- WHEN clause in triggers
- Use of OLD and NEW
- TG_ARGV
- BEFORE trigger returning row (tuple) different from NEW
- Example of trigger function written in PL/pgSQL

Example from the Postgres 9.1 manual:

```
CREATE TABLE emp (empname text, salary integer, last_date timestamp, last_user text);

CREATE FUNCTION emp_stamp() RETURNS trigger AS $emp_stamp$
  BEGIN
    -- Check that empname and salary are given
    IF NEW.empname IS NULL THEN RAISE EXCEPTION 'empname cannot be null'; END IF;
    IF NEW.salary IS NULL THEN RAISE EXCEPTION '% cannot have null salary', NEW.empname; END IF;

    -- Salary cannot be negative
    IF NEW.salary < 0 THEN RAISE EXCEPTION '% cannot have a negative salary', NEW.empname; END IF;

    -- Remember who changed the payroll when
    NEW.last_date := current_timestamp;
    NEW.last_user := current_user;

    RETURN NEW;
  END;
$emp_stamp$ LANGUAGE plpgsql;

CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
  FOR EACH ROW EXECUTE PROCEDURE emp_stamp();
```

Lecture 12

User-defined functions written in PL/pgSQL:

- PL/pgSQL = SQL + extensions
- DECLARE variables
- Scoping rules for BEGIN … END blocks
- RAISE NOTICE print statement
- Standard assignment statement, e.g., sum := x + y
- IF … THEN … ELSIF … THEN … ELSE … END IF
- CASE … WHEN … THEN … ELSE … END CASE
- LOOP … EXIT … CONTINUE … END LOOP
- WHILE … END LOOP
- FOR … IN … LOOP

Introduction to transactions in databases

- Why should a DBMS support multiple users concurrently
- What is a transaction
- Write a transaction as if there was no concurrent access, let DBMS figure out correct interleaved execution of transactions
- ACID properties
- Example: two transactions modifying account balances

Executing multiple transactions: serial schedule, notion of equivalence of two schedules, serializable schedule

Possible anomalies when interleaving actions of different transactions

- WR anomaly (dirty read, reading uncommitted data)
    - Unrecoverable schedule if transaction reading dirty data commits before the writing transaction completed
    - Possibility of cascading aborts
- RW anomaly (unrepeatable read)

Lecture 13

Midterm exam review

Anomaly discussion continued

- WW anomaly (overwriting uncommitted data)

Preventing anomalies through locking

- Shared lock, a.k.a. S-lock or read-lock
- Exclusive lock, a.k.a. X-lock or write-lock

Two-phase locking (2PL)

- Ensures serializable schedule, but does not prevent dirty reads
- Strict 2PL version to prevent dirty reads

Lecture 14

Deadlocks

- Strict 2PL and deadlocks
- Deadlock detection using the waits-for graph
- Breaking a deadlock by aborting a transaction
- Avoiding deadlocks by using Conservative 2PL
    - Why Conservative 2PL is not really practical

Phantom problem and its solution

- Locking granularity: individual tuples versus tables
- Locking predicates on tables

Performance of locking, lock contention, and thrashing

Controlling locking overhead in user transactions

- SET TRANSACTION READ ONLY
- Isolation levels and their guarantees to prevent anomalies
    - READ UNCOMMITTED
    - READ COMMITTED
    - REPEATABLE READ
    - SERIALIZABLE
- Achieving flexible per-transaction selection of isolation level through appropriate locking protocols

Transactions in SQL: use BEGIN; … COMMIT; or BEGIN; … ROLLBACK;

Lecture 15

Accessing a DBMS from an application

Embedding SQL in a host language: Embedded SQL

```
char SQLSTATE[6];

EXEC SQL BEGIN DECLARE SECTION
        char c_name[20]; short c_minGPA; float c_age;
EXEC SQL END DECLARE SECTION

c_minGPA = random();

EXEC SQL DECLARE info CURSOR FOR
        SELECT S.name, S.age FROM Students S
        WHERE S.GPA > :c_minGPA
        ORDER BY S.name;
do {
        EXEC SQL FETCH info INTO :c_name, :c_age;
        printf(“%s is %d years old\n”, c_sname, c_age);
} while (SQLSTATE != ‘02000’);

EXEC SQL CLOSE info;
```

Cursors (= more powerful iterators) for accessing a DBMS relation in the host language

Database APIs as an alternative to embedding SQL: JDBC

- JDBC type 4 driver (will be used in this course), e.g., postgresql-9.2-1001.jdbc4.jar
- Other drivers: types 1, 2, and 3

Accessing a DBMS using JDBC

```
import java.sql.*; // And some more imports

public class CreateUsers {

	public static void main(String args[]) {
		Statement stmt;
		String inputLine;

		try {
			// Connect to DB server
			Connection con = DriverManager.getConnection(
				"jdbc:postgresql://129.10.112.226:5432/TestDB", “userID", “pwd");

			// Initialize statement object
			stmt = con.createStatement();
			String login = “newLoginName”;
			String pwd = “newPWD”;

			String createUser = "CREATE USER " + login + " WITH PASSWORD '" + pwd + "'";
			String createDB = "CREATE DATABASE " + login + " OWNER " + login;

			stmt.execute(createUser);
			stmt.execute(createDB);
			con.close();
		} catch (Exception e) {
			e.printStackTrace();
		}
	}
}
```

```
// The usual imports go here

public class SimpleQuery {

        public static void main(String args[]) {
                ArrayList<String> userNames = new ArrayList<String>();
                Statement stmt;

                try {
                        // Connect to DB server
                        Connection con = DriverManager.getConnection(
                                "jdbc:postgresql://129.10.112.226:5432/db", “userID", “pwd");

                        // Initialize statement object and then query DBMS
                        stmt = con.createStatement();

                        String colName = "name";
                        String tableName = "users";

                        ResultSet rs = stmt.executeQuery("SELECT DISTINCT " + colName
                                        + " FROM " + tableName + " ORDER BY " + colName);

                        // Copy results into list
                        while (rs.next()) userNames.add(rs.getString(colName));

                        rs.close();
                        con.close();

                        // Output the user names
                        for (String name : userNames) System.out.println(name);
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }
}
```

Important classes and interfaces from java.sql:

- java.sql.Connection
  - getAutoCommit, setAutoCommit
  - getTransactionIsolation, setTransactionIsolation
  - isReadOnly, setReadOnly
  - isClosed, close
  - commit, rollback
- java.sql.DriverManager
- java.sql.ResultSet
  - next
  - getString, getInt, and so on

- java.sql.Exception
- java.sql.Statement
    - execute, executeQuery, executeUpdate
    - PreparedStatement: pre-compiled SQL statement of fixed structure (can already be pre-optimized by DBMS), parameters supplied at runtime, types of parameters enforced (good protection against SQL injection attacks)
    - CallableStatement: to call SQL stored procedures (= functions in Postgres)

Calling stored procedures:

```
CallableStatement cs = null;
try {
        // Procedure without parameters
        cs = con.prepareCall("{call myStoredProcName}");
        cs.execute();

        // Procedure with input parameters only
        cs = connection.prepareCall("{call getReservations(?)}");
        cs.setString(1, "Joe");
        cs.execute();

        // Procedure with input and output parameters
        cs = connection.prepareCall("{call getReservationCnt(?, ?)}");
        cs.setInt(1, 101);
        cs.registerOutParameter(2, Types.INT);

        // For parameters that are used for both input and output,
        // have both the set and registerOutParameter statement
        cs.execute();
        int result = cs.getInt(2);
} catch (SQLException e) { }
```

Processing SQL warnings:

```
try {
	stmt = con.createStatement();
	warning = con.getWarnings();

	while (warning != null) {
		// handle SQLWarnings;
		warning = warning.getNextWarning():
	}

	con.clearWarnings();
	stmt.executeUpdate(queryString);
	warning = con.getWarnings();
	…
}
catch( SQLException SQLe) {
	// handle the exception
}
```

Accessing database metadata: driver name and version, table names, attribute names and types etc.:

```
DatabaseMetaData md = con.getMetaData();

// Print information about the driver
System.out.println("Name:" + md.getDriverName() + "version: " + md.getDriverVersion());


// Get all table names: parameters allow narrowing down table selection
ResultSet trs = md.getTables(null,null,null,null);

While(trs.next()) {
	String tableName = trs.getString("TABLE_NAME");
	System.out.println("Table: " + tableName);

	// Print all attributes
	ResultSet crs = md.getColumns(null,null,tableName, null);
	while (crs.next()) {
		System.out.println(crs.getString("COLUMN_NAME") + ", ");
	}
}
```

Alternative to JDBC: Java Persistence API, Hibernate

- Hides explicit DBMS accesses and makes them look like "normal" object manipulations
- SQL-inspired Java Persistence Query Language operates against entity objects, not DB tables directly

Tradeoffs between placing application functionality into application layer (e.g., Java program) versus DBMS

- Programming effort required
- Benefiting from DBMS functionality
- Communication cost: transfer input data or (often smaller) results
- Ease of programming for given task

Lectures 16 and 17

Storage and indexing: important numbers [nano seconds] (source: Google's Jeff Dean @LADIS 2009):

L1 cache reference	0.5
Branch mispredict	5
L2 cache reference	7
Mutex lock/unlock	25
Main memory reference	**100**
Compress 1 KB with Zippy	3,000
Send 2 KB over 1 Gbps network	20,000
Read 1 MB sequentially from memory	**250,000**
Round trip within same data center	500,000
Disk seek	**10,000,000**
Read 1 MB sequentially from disk	**20,000,000**
Send packet CA -> Holland -> CA	150,000,000

Hard disk properties

- Seek time, rotational delay, transfer time
- Data access based on pages (aka blocks), typically 4096 bytes or larger

Simplified cost model

- Count number of page accesses on disk
- Ignore difference between random and sequential access (can be large, e.g., factor of 50)
- Ignore CPU time (can be significant, e.g., for joins)

Assumptions about the data for the cost analysis

- **Students** table
  - 40,000 tuples, 80 tuples per page; hence total of 500 pages
  - Each age between 18 and 22 is equally likely, hence 1/5*40,000 = 8000 students in each age group

- **Reservations** table
    - 100,000 tuples, 100 per page; hence total of 1000 pages

Cost of "SELECT * FROM STUDENTS WHERE age = 19", deletes, and inserts

- Heap file: no sorting on age
    - Query
        - Scan all pages: 500 I/O
    - Delete
        - Scan until found (if unique record is deleted): 250 I/O on average
        - Scan until end of file (e.g., when deleting based on age): 500 I/O
    - Insert
        - Append at end (or known empty slot): 2 I/O (read old version, write new version of page)
- Sorted file: sorted on age
    - Query
        - Binary search to find first match: $\log_2 500 = 9$ I/O
        - Scan all matches (8000 match): 8000 / 80 = 100 I/O
    - Delete
        - Perform query to find all matches, then delete them
        - Need to manage free space: non-trivial to maintain sort order
    - Insert
        - Binary search to find correct location, then update page
        - Need to manage overflow of pages

B+-tree index: improved version of sorted file idea

- Search key: attribute(s) of the relation, used for search; we use the age attribute
- Inner (i.e., non-leaf) nodes contain key-values and pointers to other tree nodes, guiding the search
- Leaf nodes contain **data entries**
    - Clustered index: data entries are the actual data records from the Students table
    - Unclustered index: data entries are (age, pointer) pairs, each pointing to a data record in a heap file where the actual student records are stored
        - We can replace a set of pointers like (19, p_1), (19, p_2),…, (19, p_n) with the more compact (19, [p_1, p_2,…, p_n])

Cost of "SELECT * FROM STUDENTS WHERE age = 19", deletes, and inserts

- Clustered tree index on age
    - Need more information about the index
        - Maximum fanout of a non-leaf index node: F; assume F = 100

        - Number of data entries per leaf node: 80 (clustered index stores actual data records); hence there are 40,000 / 80 = 500 index leaf pages
        - Index height = $\log_F$ (number of leaf pages) = 2
            - B-trees in practice often do not have more than 3-5 levels
    - Query
        - Search index from root to first leaf with match: 2 I/O (tree height)
        - Scan leaf level to get all matches: 8000 / 80 = 100 I/O
    - Delete of single entry
        - Search index from root to leaf, then update leaf node
        - Might sometimes cause cascade of updates on path up to root node
    - Insert of single entry
        - Search index from root to leaf, then update leaf node
        - Might sometimes cause cascade of updates on path up to root node
    - Summary of tree updates: cost of a single insert/delete is limited by one to about three times the height of the tree
        - More expensive when several entries are inserted or deleted
- Unclustered tree index on age
    - Need more information about the index
        - Maximum fanout of a non-leaf index node: F; assume F = 100 as above
        - Number of data entries per leaf node: 160 (unclustered index stores (key, pointer) instead of actual data records); hence there are 40,000 / 160 = 250 index leaf pages
        - Index height = $\log_F$ (number of leaf pages) = 2
    - Query
        - Search index from root to first leaf with match: 2 I/O (tree height)
        - Get all data entries by scanning matches in leaf level: 8000 / 160 = 50 I/O
        - For each matching data entry, follow pointer to access actual data record in heap file
            - 8000 matches, hence 8000 I/O using naïve approach
            - Better: re-order accesses by pointer to reduce this to about 500 I/O (each page most likely contains a 19-year-old student)

Hash index

- Search key: attribute(s) of the relation, used for search; we use the age attribute
- Hash function h(search key(s)) maps each search key value to a “bucket”
    - Different keys might hash to the same bucket
    - If the first bucket (the “primary” page) is full, additional overflow pages are linked to it

Cost of “SELECT * FROM STUDENTS WHERE age = 19”

- Access primary page for age=19, then all overflow pages linked from it

- Number of overflow pages depends on (1) number of data entries per page and (2) collisions, i.e., if other age values hash to the same bucket.
- If unclustered, then hash index only provides RID pointers, i.e., additional accesses to heap file with data records are needed like for the unclustered tree index.
- Delete, insert similar to tree

Lecture 18

Index design choices

- Which relations need an index?
- Which fields should be the index search key?
- Should the index be clustered?
- Hash or tree index?
- Consider index update and storage costs

General guidelines

- Tree index: equality and range selections
- Hash index: equality selections
- Clustered index: there can only be one (per relation)
- Query workload analysis: consider all attributes mentioned in the query
    - Index-only strategies (unclustered index)
    - Clustered index for larger range selections
    - Supporting GROUP-BY computation
    - Unclustered index for selections returning a single result, e.g., looking up single primary key

Composite search keys

- Build index on multiple attributes, e.g., <age, GPA>
    - Total order requirement for tree index
- Search key order versus index effectiveness, e.g., <age, GPA> versus <GPA, age>
    - Visualizing index sort order and query range in a scatter-plot
- Unsing multiple 1-dimensional trees versus using a single 2-dimensional tree, e.g., index on <age> and another on <GPA> versus a single one on <age, GPA>

Index effectiveness

- Given a WHERE condition, which of a given set of hash or tree indexes is (1) not useful (and would be ignored by the DB optimizer), (2) useful but not very effective, and (3) useful and highly effective?

Lecture 19

Join algorithms and their cost

- (Naïve) nested loops join: for each tuple in outer, go through all tuples in inner relation
  - Improved version: block nested loops: for each page from outer, process each page from inner relation by checking all tuple-pairs from that outer and inner page
- Index nested loops: for each tuple in outer, use index on appropriate attribute to find all matches from inner relation
  - Can use any index, including clustered/unclustered, hash, or tree, depending on cost and if it "matches"
- Sort-merge join: sort both files on the join attribute(s), then scan "in lockstep"
- Hash join (for equi-joins): partition both relations by using the same hash function, then join each pair of corresponding buckets

System-R style optimizer

- Many equivalent query plans for a given SQL query: enumerate many candidate plans, estimate cost, then choose winner
- Reducing the plan search space
  - Avoid Cartesian products
  - Only consider left-deep join plans (no need to materialize intermediate join results)
  - Push down selections and projections

Lectures 20 and 21

Discussion of HW 5

Normal forms for better database design

Separate relations for Students, Reservations, and Books versus single "wide" relation (all joined together)

- Redundancy in wide relation
- Challenge to keep multiple copies of the same data in different records in sync for wide relation
- Many more NULL values in wide relation
- Higher query cost whenever join is needed across the separate relations

Functional dependencies (FDs)

- What is an FD?
- Need to be specified by the designer
- From a given database, we can only infer FDs that do **not** hold
- Armstrong's axioms
- Closure F^+ of a given set F of FDs

- Algorithm for computing if a certain FD $X \rightarrow Y$ is in F^+
    - Using this algorithm to determine if some set of attributes is a key candidate

Boyce-Codd Normal Form (BCNF)

- How to check if a relation is in BCNF
- Decomposing a relation by using the FD that violates BCNF
- Decomposition properties
    - Order of decompositions affects final result
    - Lossless join property: has to be satisfied
    - Dependency-preservation: sometimes forces tradeoff between redundancy reduction versus performance penalty
- We can always decompose into BCNF such that lossless-join property holds, but there might be no dependency-preserving BCNF decomposition

Third Normal Form (3NF)

- Slightly weaker than BCNF
- There is an algorithm for performing 3NF decomposition that achieves both lossless-join and dependency preservation property

Footnote about other normal forms: 1NF, 2NF, 3NF, BCNF, 4NF, 5NF

Normal forms as part of the database design process

Overview of recovery in a DBMS: the ARIES approach

- Goal: changes of committed transactions should be durable, partial changes of crashed transactions need to be rolled back
- Write-ahead logging (WAL) to keep record of update operations
    - Log record keeps old and new version of a page
    - Allows re-applying or undoing the update
- REDO operation for committed transactions: read forward in the log
- UNDO operation for failed transactions: read backward in the log
- Checkpointing for speeding up recovery after crash

Lecture 22

Final exam review

Overview of SQL injection attacks and how to prevent them

Overview of MapReduce: word count and equi-join example