COURSE OVERVIEW
THE RELATIONAL MODEL

CS121: Introduction to Relational Database Systems

Fall 2014 – Lecture 1

Course Overview

- Introduction to relational database systems
  - Theory and use of relational databases
- Focus on:
  - The Relational Model and relational algebra
  - SQL (the Structured Query Language)
  - The Entity-Relationship model
  - Database schema design and normal forms
  - Various common uses of database systems
- By end of course:
  - Should be very comfortable using relational databases
  - Familiar with basic relational database theory

Textbook

- No textbook is required for the course
  - The lecture slides contain most of the relevant details
  - Other essential materials are provided with the assignments
- A great book: Database System Concepts, 5^{th} ed.
  - Silberschatz, Korth, Sudarshan
  - (The current edition is 6^{th}; they messed a lot of things up…)
  - Covers theory, use, and implementation of relational databases, so good to have for 121/122/123 sequence
- I will also make recordings of the lectures available

Assignments

- Assignments are given approximately weekly
  - Set of problems focusing on that week's material
  - Most include hands-on practice with real databases
  - Made available on Wednesdays
  - Due approx. one week later: Thursdays at 2am
    - That's the <u>start</u> of Thursday, not the end of Thursday
- Midterm and final exam are typically 4-6 hours long
- Assignment and exam weighting:
  - 8 assignments, comprising 65% of your grade
  - Midterm counts for 15% of your grade
  - Final exam counts for 20% of your grade

Course Website and Submissions

- CS121 is on the Caltech Moodle
  - https://courses.caltech.edu/course/view.php?id=1684
  - 2014 enrollment key: unionall (as one word)
- Please enroll in the course as soon as possible!
  - I will make class announcements via Moodle
  - You will submit your assignments via Moodle
- Most assignments will be submitted on the Moodle
  - We suggest you do HW1 and HW5 by hand, rather than on the computer, unless you are awesome at $\LaTeX$
  - (Trust us, you will finish them much faster.)

Grading Policies

- Submit assignments <u>on time</u>!
- Late assignments and exams will be penalized!
  - Up to 1 day (24 hours) late: 10% penalty
  - Up to 2 days (48 hours) late: 30% penalty
  - Up to 3 days (72 hours) late: 60% penalty
  - After 3 days, don't bother. ☹
- But, extensions are available:
  - <u>Must</u> provide a note from Dean's Office or Health Center
  - You also have 3 "late tokens" to use however you want
    - Each late token is worth a 24-hour extension
    - **Can't use late tokens on the final exam without my permission**

Other Administrivia

- I will be away from Caltech for weeks 2 and 3 ☹

- We do have lecture recordings for those weeks
- We will have plenty of TAs to help with the work
  - Will likely have extended office hours for questions during this time

- Will discuss this more next time

Database Terminology

- Database – an organized collection of information
  - A *very* generic term…
  - Covers flat text-files with simple records…
  - …all the way up to multi-TB data warehouses!
  - Some means to query this set of data as a unit, and usually some way to update it as well
- Database Management System (DBMS)
  - Software that manages databases
    - Create, modify, query, backup/restore, etc.
  - Sometimes just "database system"

Before DBMSes Existed…

- Typical approach:
  - Ad-hoc or purpose-built data files
  - Special-built programs implemented various operations against the database
- Want to perform new operations?
  - Create new programs to manipulate the data files!
- Want to change the data model?
  - Update all the programs that access the data!
- How to implement transactions? Security? Integrity constraints?

Enter the DBMS

- Provide layers of abstraction to isolate users, developers from database implementation
  - Physical level: how values are stored/managed on disk
  - Logical level: specification of records and fields
  - View level: queries and operations that users can perform (typically through applications)
- Provide <u>generic</u> database capabilities that specific applications can utilize
  - Specification of database schemas
  - Mechanism for querying and manipulating records

Kinds of Databases

- *Many* kinds of databases, based on usage
- Amount of data being managed
  - embedded databases: small, application-specific systems (e.g. SQLite, BerkeleyDB)
  - data warehousing: vast quantities of data (e.g. Oracle)
- Type/frequency of operations being performed
  - OLTP: Online Transaction Processing
    - "Transaction-oriented" operations like buying a product or booking an airline flight
  - OLAP: Online Analytical Processing
    - Storage and analysis of *very* large amounts of data
    - e.g. "What are my top selling products in each sales region?"

Data Models

- Databases must represent:
  - the data itself (typically structured in some way)
  - associations between different data values
- What kind of data can be modeled?
- What kinds of associations can be represented?
- The <u>data model</u> specifies:
  - what data can be stored (and sometimes how it is stored)
  - associations between different data values
  - what constraints can be enforced
  - how to access and manipulate the data

Data Models (2)

- Most database systems use the <u>relational model</u>
  - A database is a collection of tables containing records
  - Format of records is fixed
    - It can be changed, but this is infrequent!
  - Data is modeled at logical level, not physical level
- Preceded by hierarchical data model, and the network model
  - Very powerful and complicated models
  - Required much more physical-level specification
  - Queries implemented as programs that navigate the schema
  - Schemas couldn't be changed without heavy costs

Data Models

- This course focuses on the Relational Model
  - SQL (Structured Query Language) draws heavily from the relational model
  - Most database systems use the relational model!
- Also focuses on the Entity-Relationship Model
  - Much higher level model than relational model
  - Useful for modeling abstractions
  - Very useful for database design!
  - Not supported by most databases, but used in many database design tools
  - Easy to translate into the relational model

Other Data Models

- Relational model is not the only one in use!
  - By far the most widely used, at this point
- Object model, object-relational model
  - Model data records as “objects” that store references to related objects and values
  - Very similar to the network model, but with a much higher level of abstraction
- XML data models
  - Optimized for XML document storage
  - Queries using XPath, XQuery, etc.
  - XSLT support for transforming XML documents

Other Data Models (2)

- There are also simpler structured storage models
  - Key-value stores, document stores, NoSQL, etc.
  - Relax most of the constraints imposed by relational model
  - Allow for extremely large distributed databases with very flexible schemas
  - (Relational model is one kind of structured storage model)
- Used to manage data for the largest, most heavily used websites
  - Performance and scaling requirements simply disallow the use of the relational model
  - Can't impose constraints without an overwhelming cost

The Relational Model and SQL

Before we start:

- SQL is *loosely* based on the relational model
- Some terms appear in both the relational model and in SQL…

  …but they aren't exactly the same!
- Be careful if you already know some SQL
  - Don't assume that similarly named concepts are identical. They're not!

History of the Relational Model

- Invented by Edgar F. ("Ted") Codd in early 1970s
- Focus was data independence
  - Existing data models required physical level design and implementation
  - Changes were very costly to applications that accessed the database
- IBM, Oracle were first implementers of relational model (1977)
  - Usage spread very rapidly through software industry
  - SQL was a particularly powerful innovation

Relations

- Relations are basically tables of data
  - Each row represents a record in the relation
- A relational database is a set of relations
  - Each relation has a unique name in the database

acct_id	branch_name	balance
A-301	New York	350
A-307	Seattle	275
A-318	Los Angeles	550
…	…	…

The *account* relation

- Each row in the table specifies a <u>relationship</u> between the values in that row
  - The account ID "A-307", branch name "Seattle", and balance "275" are all related to each other

Relations and Attributes

- Each relation has some number of attributes
  - Sometimes called "columns"
- Each attribute has a domain
  - Specifies the set of valid values for the attribute
- The *account* relation:
  - 3 attributes
  - Domain of *balance* is the set of nonnegative integers
  - Domain of *branch_name* is the set of all valid branch names in the bank

acct_id	branch_name	balance
A-301	New York	350
A-307	Seattle	275
A-318	Los Angeles	550
…	…	…

account

Tuples and Attributes

- Each row is called a <u>tuple</u>
  - A fixed-size, ordered set of name-value pairs
- A <u>tuple variable</u> can refer to any valid tuple in a relation
- Each attribute in the tuple has a unique name
- Can also refer to attributes by index
  - Attribute 1 is the first attribute, etc.
- Example:
  - Let tuple variable *t* refer to first tuple in *account* relation
  - *t*[*balance*] = 350
  - *t*[2] = "New York"

acct_id	branch_name	balance
A-301	New York	350
A-307	Seattle	275
A-318	Los Angeles	550
…	…	…

account

Tuples and Relationships

- In the *account* relation:
  - Domain of *acct_id* is D_1
  - Domain of *branch_name* is D_2
  - Domain of *balance* is D_3

acct_id	branch_name	balance
A-301	New York	350
A-307	Seattle	275
A-318	Los Angeles	550
…	…	…

account

- The *account* relation is a subset of the tuples in the Cartesian product $D_1 \times D_2 \times D_3$
- Each tuple included in *account* specifies a <u>relationship</u> between that set of values
  - Hence the name, "relational model"
  - Tuples in the *account* relation specify the details of valid bank accounts

Tuples and Relations

- A relation is a set of tuples
  - Each tuple appears exactly once
    - *Note: SQL tables are multisets! (Sometimes called bags.)*
  - If two tuples t_1 and t_2 have the same values for all attributes, then t_1 and t_2 are the *same tuple* (i.e. $t_1 = t_2$)
- The order of tuples in a relation is not relevant

Relation Schemas

- Every relation has a <u>schema</u>
  - Specifies the type information for relations
  - Multiple relations can have the same schema
- A relation schema includes:
  - an ordered set of attributes
  - the domain of each attribute
- Naming conventions:
  - Relation names are written as all lowercase
  - Relation schema's name is capitalized
- For relation *r* and relation schema *R*:
  - Write *r*(*R*) to indicate that the schema of *r* is *R*

Schema of *account* Relation

- The relation schema of *account* is:

  Account_schema = (*acct_id*, *branch_name*, *balance*)

- To indicate that *account* has schema *Account_schema*:

  account(*Account_schema*)

acct_id	branch_name	balance
A-301	New York	350
A-307	Seattle	275
A-318	Los Angeles	550
…	…	…

account

- Important note:
  - Domains are not stated explicitly in this notation!

Relation Schemas

- Relation schemas are ordered <u>sets</u> of attributes
  - Can use set operations on them
- Examples:

  Relations $r(R)$ and $s(S)$
    - Relation r has schema R
    - Relation s has schema S

  $R \cap S$
    - The set of attributes that R and S have in common

  $R - S$
    - The set of attributes in R that are not also in S

  $K \subseteq R$
    - K is some subset of the attributes in relation schema R

Attribute Domains

- The relational model constrains attribute domains to be atomic
  - Values are indivisible units
- Mainly a simplification
  - Virtually all relational database systems provide non-atomic data types
- Attribute domains may also include the *null* value
  - *null* = the value is unknown or unspecified
  - *null* can often complicate things. Generally considered good practice to avoid wherever reasonable to do so.

Relations and Relation Variables

- More formally:
- *account* is a <u>relation variable</u>
  - A name associated with a specific schema, and a set of tuples that satisfies that schema
  - (sometimes abbreviated "relvar")
- A specific set of tuples with the same schema is called a <u>relation value</u> (sometimes abbreviated "relval")
  - (Formally, this can also be called a relation)
  - Can be associated with a relation variable
  - Or, can be generated by applying relational operations to one or more relation variables

acct_id	branch_name	balance
A-301	New York	350
A-307	Seattle	275
A-318	Los Angeles	550
…	…	…

The *account* relation

Relations and Relation Variables (2)

- Problem:
  - The term "relation" is often used in slightly different ways
- "Relation" normally means the collection of tuples
  - i.e. "relation" usually means "relation value"
- It is often used less formally to refer to a relation variable and its associated relation value
  - e.g. "the *account* relation" is really a relation variable that holds a specific relation value

acct_id	branch_name	balance
A-301	New York	350
A-307	Seattle	275
A-318	Los Angeles	550
…	…	…

The *account* relation

Distinguishing Tuples

- Relations are *sets* of tuples…
  - No two tuples can have the same values for *all* attributes…
  - But, some tuples might have the same values for *some* attributes
- Example:
  - Some accounts have the same balance
  - Some accounts are at the same branch

acct_id	branch_name	balance
A-301	New York	350
A-307	Seattle	275
A-318	Los Angeles	550
A-319	New York	80
A-322	Los Angeles	275

account

Keys

- Keys are used to distinguish individual tuples
  - A superkey is a set of attributes that uniquely identifies tuples in a relation

acct_id	branch_name	balance
A-301	New York	350
A-307	Seattle	275
A-318	Los Angeles	550
A-319	New York	80
A-322	Los Angeles	275

account

- Example:
  {*acct_id*} is a superkey
- Is {*acct_id*, *balance*} a superkey?
  - Yes! Every tuple will have a unique set of values for this combination of attributes.
- Is {*branch_name*} a superkey?
  - No. Each branch can have multiple accounts

Superkeys and Candidate Keys

- A superkey is a set of attributes that uniquely identifies tuples in a relation
- Adding attributes to a superkey produces another superkey
  - If {*acct_id*} is a superkey, so is {*acct_id*, *balance*}
  - If a set of attributes $K \subseteq R$ is a superkey, so is any superset of K
  - Not all superkeys are equally useful…
- A *minimal* superkey is called a <u>candidate key</u>
  - A superkey for which no proper subset is a superkey
  - For *account*, only {*acct_id*} is a candidate key

Primary Keys

- A relation might have several candidate keys
- In these cases, one candidate key is chosen as the primary means of uniquely identifying tuples
  - Called a primary key
- Example: *customer* relation
  - Candidate keys could be:

    {*cust_id*}

    {*cust_ssn*}
  - Choose primary key:

    {*cust_id*}

cust_id	cust_name	cust_ssn
23-652	Joe Smith	330-25-8822
15-202	Ellen Jones	221-30-6551
23-521	Dave Johnson	005-81-2568
…	…	…

customer

Primary Keys (2)

- Keys are a property of the relation schema, not individual tuples
  - Applies to *all* tuples in the relation
- Primary key attributes are listed first in relation schema, and are underlined
- Examples:

  Account_schema = (*<u>acct_id</u>*, *branch_name*, *balance*)

  Customer_schema = (*<u>cust_id</u>*, *cust_name*, *cust_ssn*)
- Only indicate primary keys in this notation
  - Other candidate keys are not specified

Primary Keys (3)

- Multiple records cannot have the same values for a primary key!
  - …or any candidate key, for that matter…
- Example: *customer*(*<u>cust_id</u>*, *cust_name*, *cust_ssn*)

cust_id	cust_name	cust_ssn
23-652	Joe Smith	330-25-8822
15-202	Ellen Jones	221-30-6551
23-521	Dave Johnson	005-81-2568
15-202	Albert Stevens	450-22-5869
…	…	…

customer

  - Two customers cannot have the same ID.
- This is an example of an <u>invalid relation</u>
  - Set of tuples doesn't satisfy the required constraints

Keys Constrain Relations

- Primary keys *constrain* the set of tuples that can appear in a relation
  - Same is true for *all* superkeys
- For a relation r with schema R
  - If $K \subseteq R$ is a superkey then
    $\langle \forall\ t_1, t_2 \in r(R) : t_1[K] = t_2[K] : t_1[R] = t_2[R] \rangle$
  - i.e. if two tuple-variables have the same values for the superkey attributes, then they refer to the same tuple
    - $t_1[R] = t_2[R]$ is equivalent to saying $t_1 = t_2$

Choosing Candidate Keys

- Since candidate keys constrain the tuples that can be stored in a relation…
  - Attributes that would make good (or bad) candidate keys depend on *what is being modeled*
- Example: customer name as candidate key?
  - Very likely that multiple people will have same name
  - Thus, not a good idea to use it as a candidate key
- These constraints motivated by external requirements
  - Need to understand what we are modeling in the database

Foreign Keys

- One relation schema can include the attributes of another schema's primary key
- Example: *depositor* relation
  - *Depositor_schema* = (*cust_id*, *acct_id*)
  - Associates customers with bank accounts
  - *cust_id* and *acct_id* are both <u>foreign keys</u>
    - *cust_id* references the primary key of *customer*
    - *acct_id* references the primary key of *account*
  - *depositor* is the <u>referencing relation</u>
    - It refers to the *customer* and *account* relations
  - *customer* and *account* are the <u>referenced relations</u>

depositor Relation

cust_id	cust_name	cust_ssn
23-652	Joe Smith	330-25-8822
15-202	Ellen Jones	221-30-6551
23-521	Dave Johnson	005-81-2568
…	…	…

customer

acct_id	branch_name	balance
A-301	New York	350
A-307	Seattle	275
A-318	Los Angeles	550
…	…	…

account

cust_id	acct_id
15-202	A-301
23-521	A-307
23-652	A-318
…	…

depositor

- *depositor* relation references *customer* and *account*
- Represents relationships between customers and their accounts
- Example: Joe Smith's accounts
  - "Joe Smith" has an account at the "Los Angeles" branch, with a balance of 550.

Foreign Key Constraints

- Tuples in *depositor* relation specify values for *cust_id*
  - *customer* relation <u>must</u> contain a tuple corresponding to each *cust_id* value in *depositor*
- Same is true for *acct_id* values and *account* relation
- Valid tuples in a relation are also constrained by foreign key references
  - Called a <u>foreign-key constraint</u>
- Consistency between two dependent relations is called <u>referential integrity</u>
  - Every foreign key value must have a corresponding primary key value

Foreign Key Constraints (2)

- Given a relation $r(R)$
  - A set of attributes $K \subseteq R$ is the primary key for R
- Another relation $s(S)$ references r
  - $K \subseteq S$ too
  - $\langle\ \forall\ t_s \in s : \exists\ t_r \in r : t_s[K] = t_r[K]\ \rangle$
- Notes:
  - K is not required to be a candidate key for S, only R
  - K may also be part of a larger candidate key for S

Primary Key of *depositor* Relation?

- *Depositor_schema* = (*cust_id*, *acct_id*)
- If {*cust_id*} is the primary key:
  - A customer can only have one account
    - Each customer's ID can appear only once in *depositor*
  - An account could be owned by multiple customers
- If {*acct_id*} is the primary key:
  - Each account can be owned by only one customer
    - Each account ID can appear only once in *depositor*
  - Customers could own multiple accounts
- If {*cust_id*, *acct_id*} is the primary key:
  - Customers can own multiple accounts
  - Accounts can be owned by multiple customers
- Last option is how most banks really work

cust_id	acct_id
15-202	A-301
23-521	A-307
23-652	A-318
…	…

depositor

RELATIONAL ALGEBRA

CS121: Introduction to Relational Database Systems

Fall 2014 – Lecture 2

Administrivia

- First assignment will be available today
  - Due next Thursday, October 9, 2:00 AM
- We have TAs:
  - Solomon Chang
  - Daniel Kong
  - Ryan Langman
  - Eric Pelz
  - Daniel Wang
- See Moodle for contact info and office hours
  - Can send questions to cs121tas@caltech.edu (Donnie + TAs)

Query Languages

- A query language specifies how to access the data in the database
- Different kinds of query languages:
  - Declarative languages specify what data to retrieve, but not how to retrieve it
  - Procedural languages specify what to retrieve, as well as the process for retrieving it
- Query languages often include updating and deleting data as well
- Also called data manipulation language (DML)

The Relational Algebra

- A procedural query language
- Comprised of relational algebra operations
- Relational operations:
  - Take one or two relations as input
  - Produce a relation as output
- Relational operations can be composed together
  - Each operation produces a relation
  - A query is simply a relational algebra expression
- Six "fundamental" relational operations
- Other useful operations can be composed from these fundamental operations

“Why is this useful?”

- SQL is only loosely based on relational algebra
- SQL is much more on the “declarative” end of the spectrum
- *Many* relational database implementations use relational algebra operations as basis for representing execution plans
  - Simple, clean, effective abstraction for representing how results will be generated
  - Relatively easy to manipulate for query optimization

Fundamental Relational Algebra Operations

- Six fundamental operations:

σ	select operation
Π	project operation
$\cup$	set-union operation
$-$	set-difference operation
$\times$	Cartesian product operation
ρ	rename operation

- Each operation takes one or two relations as input
- Produces another relation as output
- Important details:
  - What tuples are included in the result relation?
  - Any constraints on input schemas? What is schema of result?

Select Operation

- Written as: $\sigma_P(r)$
- P is the predicate for selection
  - P can refer to attributes in *r* (but no other relation!), as well as literal values
  - Can use comparison operators: $=, \neq, <, \leq, >, \geq$
  - Can combine multiple predicates using: $\wedge$ (and), $\vee$ (or), $\neg$ (not)
- *r* is the input relation
- Result relation contains all tuples in *r* for which P is true
- Result schema is identical to schema for *r*

Select Examples

Using the *account* relation:

acct_id	branch_name	balance
A-301	New York	350
A-307	Seattle	275
A-318	Los Angeles	550
A-319	New York	80
A-322	Los Angeles	275

account

"Retrieve all tuples for accounts in the Los Angeles branch."

$\sigma_{branch\_name=\text{"Los Angeles"}}(account)$

acct_id	branch_name	balance
A-318	Los Angeles	550
A-322	Los Angeles	275

"Retrieve all tuples for accounts in the Los Angeles branch, with a balance under \$300."

$\sigma_{branch\_name=\text{"Los Angeles"} \wedge balance<300}(account)$

acct_id	branch_name	balance
A-322	Los Angeles	275

Project Operation

- Written as: $\Pi_{a,b,\ldots}(r)$
- Result relation contains only specified attributes of *r*
  - Specified attributes must actually be in schema of *r*
  - Result's schema only contains the specified attributes
  - Domains are same as source attributes' domains
- Important note:
  - Result relation may have fewer rows than input relation!
  - Why?
    - Relations are *sets* of tuples, not multisets

Project Example

Using the *account* relation:

acct_id	branch_name	balance
A-301	New York	350
A-307	Seattle	275
A-318	Los Angeles	550
A-319	New York	80
A-322	Los Angeles	275

account

"Retrieve all branch names that have at least one account."

$\Pi_{branch\_name}(account)$

branch_name
New York
Seattle
Los Angeles

- Result only has three tuples, even though input has five
- Result schema is just (*branch_name*)

Composing Operations

- Input can also be an expression that evaluates to a relation, instead of just a relation
- $\Pi_{acct\_id}(\sigma_{balance \geq 300}(account))$
  - Selects the account IDs of all accounts with a balance of $300 or more
  - Input relation's schema is:

    $Account\_schema = (\underline{acct\_id}, branch\_name, balance)$
  - Final result relation's schema?
    - Just one attribute: ($acct\_id$)
- Distinguish between <u>base</u> and <u>derived</u> relations
  - *account* is a base relation
  - $\sigma_{balance \geq 300}(account)$ is a derived relation

Set-Union Operation

- Written as: $r \cup s$
- Result contains all tuples from *r* and *s*
  - Each tuple is unique, even if it's in both *r* and *s*
- Constraints on schemas for *r* and *s* ?
- *r* and *s* must have <u>compatible</u> schemas:
  - *r* and *s* must have same <u>arity</u>
    - (same number of attributes)
  - For each attribute i in *r* and *s*, *r*[i] must have the same domain as *s*[i]
  - (Our examples also generally have same attribute names, but not required! Arity and domains are what matter.)

Set-Union Example

- More complicated schema:

acct_id	branch_name	balance
A-301	New York	350
A-307	Seattle	275
A-318	Los Angeles	550
A-319	New York	80
A-322	Los Angeles	275

account

cust_name	acct_id
Johnson	A-318
Smith	A-322
Reynolds	A-319
Lewis	A-307
Reynolds	A-301

depositor

loan_id	branch_name	amount
L-421	San Francisco	7500
L-445	Los Angeles	2000
L-437	Las Vegas	4300
L-419	Seattle	2900

loan

cust_name	loan_id
Anderson	L-437
Jackson	L-419
Lewis	L-421
Smith	L-445

borrower

Set-Union Example (2)

- Find names of all customers that have either a bank account or a loan at the bank

acct_id	branch_name	balance
A-301	New York	350
A-307	Seattle	275
A-318	Los Angeles	550
A-319	New York	80
A-322	Los Angeles	275

account

cust_name	acct_id
Johnson	A-318
Smith	A-322
Reynolds	A-319
Lewis	A-307
Reynolds	A-301

depositor

loan_id	branch_name	amount
L-421	San Francisco	7500
L-445	Los Angeles	2000
L-437	Las Vegas	4300
L-419	Seattle	2900

loan

cust_name	loan_id
Anderson	L-437
Jackson	L-419
Lewis	L-421
Smith	L-445

borrower

Set-Union Example (3)

- Find names of all customers that have either a bank account or a loan at the bank
  - Easy to find the customers with an account:
    $\Pi_{cust\_name}(depositor)$
  - Also easy to find customers with a loan:
    $\Pi_{cust\_name}(borrower)$

cust_name
Johnson
Smith
Reynolds
Lewis

$\Pi_{cust\_name}(depositor)$

cust_name
Anderson
Jackson
Lewis
Smith

$\Pi_{cust\_name}(borrower)$

- Result is set-union of these expressions:
  $\Pi_{cust\_name}(depositor) \cup \Pi_{cust\_name}(borrower)$
  - Note that inputs have 8 tuples, but result has 6 tuples.

cust_name
Johnson
Smith
Reynolds
Lewis
Anderson
Jackson

Set-Difference Operation

- Written as: $r - s$
- Result contains tuples that are only in *r*, but not in *s*
  - Tuples in both *r* and *s* are excluded
  - Tuples only in *s* do not affect the result
- Constraints on schemas of *r* and *s*?
  - Schemas must be compatible
  - (Exactly like set-union.)

Set-Difference Example

acct_id	branch_name	balance
A-301	New York	350
A-307	Seattle	275
A-318	Los Angeles	550
A-319	New York	80
A-322	Los Angeles	275

account

cust_name	acct_id
Johnson	A-318
Smith	A-322
Reynolds	A-319
Lewis	A-307
Reynolds	A-301

depositor

loan_id	branch_name	amount
L-421	San Francisco	7500
L-445	Los Angeles	2000
L-437	Las Vegas	4300
L-419	Seattle	2900

loan

cust_name	loan_id
Anderson	L-437
Jackson	L-419
Lewis	L-421
Smith	L-445

borrower

"Find all customers that have an account but not a loan."

Set-Difference Example (2)

- Again, each component is easy
  - All customers that have an account:

    $\Pi_{cust\_name}(depositor)$
  - All customers that have a loan:

    $\Pi_{cust\_name}(borrower)$

cust_name
Johnson
Smith
Reynolds
Lewis

$\Pi_{cust\_name}(depositor)$

cust_name
Anderson
Jackson
Lewis
Smith

$\Pi_{cust\_name}(borrower)$

- Result is set-difference of these expressions

  $\Pi_{cust\_name}(depositor) - \Pi_{cust\_name}(borrower)$

cust_name
Johnson
Reynolds

Cartesian Product Operation

- Written as: $r \times s$
  - Read as "*r* cross *s*"
- <u>No</u> constraints on schemas of *r* and *s*
- Schema of result is *concatenation* of schemas for *r* and *s*
- If *r* and *s* have overlapping attribute names:
  - <u>All</u> overlapping attributes are included; none are eliminated
  - Distinguish overlapping attribute names by prepending the source relation's name
- Example:
  - Input relations: $r(a, b)$ and $s(b, c)$
  - Schema of $r \times s$ is $(a, r.b, s.b, c)$

Cartesian Product Operation (2)

- Result of $r \times s$
  - Contains *every* tuple in *r*, combined with *every* tuple in *s*
  - If *r* contains N_r tuples, and *s* contains N_s tuples, result contains $N_r \times N_s$ tuples
- Allows two relations to be compared and/or combined
  - If we want to correlate tuples in relation *r* with tuples in relation *s*…
  - Compute $r \times s$, then select out desired results with an appropriate predicate

Cartesian Product Example

- Compute result of *borrower* × *loan*

cust_name	loan_id
Anderson	L-437
Jackson	L-419
Lewis	L-421
Smith	L-445

borrower

loan_id	branch_name	amount
L-421	San Francisco	7500
L-445	Los Angeles	2000
L-437	Las Vegas	4300
L-419	Seattle	2900

loan

- Result will contain 4 × 4 = 16 tuples

Cartesian Product Example (2)

- Schema for borrower is:

  Borrower_schema = (*cust_name*, *loan_id*)

- Schema for loan is:

  Loan_schema = (*loan_id*, *branch_name*, *amount*)

- Schema for result of *borrower* × *loan* is:

  (*cust_name*, *borrower.loan_id*,
  loan.loan_id, *branch_name*, *amount*)

  - Overlapping attribute names are distinguished by including name of source relation

Cartesian Product Example (3)

Result:

cust_name	borrower. loan_id	loan. loan_id	branch_name	amount
Anderson	L-437	L-421	San Francisco	7500
Anderson	L-437	L-445	Los Angeles	2000
Anderson	L-437	L-437	Las Vegas	4300
Anderson	L-437	L-419	Seattle	2900
Jackson	L-419	L-421	San Francisco	7500
Jackson	L-419	L-445	Los Angeles	2000
Jackson	L-419	L-437	Las Vegas	4300
Jackson	L-419	L-419	Seattle	2900
Lewis	L-421	L-421	San Francisco	7500
Lewis	L-421	L-445	Los Angeles	2000
Lewis	L-421	L-437	Las Vegas	4300
Lewis	L-421	L-419	Seattle	2900
Smith	L-445	L-421	San Francisco	7500
Smith	L-445	L-445	Los Angeles	2000
Smith	L-445	L-437	Las Vegas	4300
Smith	L-445	L-419	Seattle	2900

Cartesian Product Example (4)

- Can use Cartesian product to associate related rows between two tables
  - …but, a lot of extra rows are included!

cust_name	borrower. loan_id	loan. loan_id	branch_name	amount
…	…	…	…	…
Jackson	L-419	L-437	Las Vegas	4300
Jackson	L-419	L-419	Seattle	2900
Lewis	L-421	L-421	San Francisco	7500
Lewis	L-421	L-445	Los Angeles	2000
…	…	…	…	…

- Combine Cartesian product with a select operation

$$\sigma_{borrower.loan\_id=loan.loan\_id}(borrower \times loan)$$

Cartesian Product Example (5)

- "Retrieve the names of all customers with loans at the Seattle branch."

cust_name	loan_id
Anderson	L-437
Jackson	L-419
Lewis	L-421
Smith	L-445

borrower

loan_id	branch_name	amount
L-421	San Francisco	7500
L-445	Los Angeles	2000
L-437	Las Vegas	4300
L-419	Seattle	2900

loan

- Need both *borrower* and *loan* relations
- Correlate tuples in the relations using *loan_id*
- Then, computing result is easy.

Cartesian Product Example (6)

26

- Associate customer names with loan details, using Cartesian product and a select:

  $\sigma_{borrower.loan\_id=loan.loan\_id}(borrower \times loan)$

- Select out loans at Seattle branch:

  $\sigma_{branch\_name=\text{“Seattle”}}(\sigma_{borrower.loan\_id=loan.loan\_id}(borrower \times loan))$

  Simplify:

  $\sigma_{borrower.loan\_id=loan.loan\_id \wedge branch\_name=\text{“Seattle”}}(borrower \times loan)$

- Project results down to customer name:

  $\Pi_{cust\_name}(\sigma_{borrower.loan\_id=loan.loan\_id \wedge branch\_name=\text{“Seattle”}}(borrower \times loan))$

- Final result:

cust_name
Jackson

Rename Operation

- Results of relational operations are unnamed
  - Result has a schema, but the relation itself is unnamed
- Can give result a name using the rename operator
- Written as: $\rho_x(E)$
  - E is an expression that produces a relation
  - E can also be a named relation or a relation-variable
  - x is new name of relation
- More general form is: $\rho_{x(A_1,\ A_2,\ \ldots,\ A_n)}(E)$
  - Allows renaming of relation's attributes
  - Requirement: E has arity n

Scope of Renamed Relations

- Rename operation ρ only applies within a specific relational algebra expression
  - This does not create a new relation-variable!
  - The new name is only visible to enclosing relational-algebra expressions
- Rename operator is used for two main purposes:
  - Allow a derived relation and its attributes to be referred to by enclosing relational-algebra operations
  - Allow a base relation to be used multiple ways in one query
    - $r \times \rho_s(r)$
- In other words, rename operation ρ is used to resolve ambiguities within a specific relational algebra expression

Rename Example

- "Find the ID of the loan with the largest amount."

loan_id	branch_name	amount
L-421	San Francisco	7500
L-445	Los Angeles	2000
L-437	Las Vegas	4300
L-419	Seattle	2900

loan

  - Hard to find the loan with the largest amount!
    - (At least, with the tools we have so far…)
  - Much easier to find all loans that have an amount *smaller* than some other loan
  - Then, use set-difference to find the largest loan

Rename Example (2)

- How to find all loans with an amount smaller than some other loan?
  - Use Cartesian Product of *loan* with itself:

    $loan \times loan$
  - Compare each loan's amount to all other loans
- Problem: Can't distinguish between attributes of left and right *loan* relations!
- Solution: Use rename operation

  $loan \times \rho_{test}(loan)$
  - Now, right relation is named *test*

Rename Example (3)

- Find IDs of all loans with an amount smaller than some other loan:

  $\Pi_{loan.loan\_id}(\sigma_{loan.amount < test.amount}(loan \times \rho_{test}(loan)))$

- Finally, we can get our result:

  $\Pi_{loan\_id}(loan)$ –

  $\Pi_{loan.loan\_id}(\sigma_{loan.amount < test.amount}(loan \times \rho_{test}(loan)))$

loan_id
L-421

- What if multiple loans have max value?
  - All loans with max value appear in result.

Additional Relational Operations

- The fundamental operations are sufficient to query a relational database…
- Can produce some large expressions for common operations!
- Several additional operations, defined in terms of fundamental operations:

$\cap$	set-intersection
$\bowtie$	natural join
$\div$	division
$\leftarrow$	assignment

Set-Intersection Operation

- Written as: $r \cap s$
- $r \cap s = r - (r - s)$

  $r - s$ = the rows in r, but not in s

  $r - (r - s)$ = the rows in both r and s
- Relations must have compatible schemas
- Example: find all customers with both a loan and a bank account

  $\Pi_{cust\_name}(borrower) \cap \Pi_{cust\_name}(depositor)$

Natural Join Operation

- Most common use of Cartesian product is to correlate tuples with same key-values
  - Called a join operation
- The natural join is a shorthand for this operation
- Written as: $r \bowtie s$
  - *r* and *s* must have common attributes
  - The common attributes are usually a key for *r* and/or *s*, but certainly don't have to be

Natural Join Definition

- For two relations $r(R)$ and $s(S)$
- Attributes used to perform natural join:

  $R \cap S = \{A_1, A_2, \ldots, A_n\}$
- Formal definition:

  $r \bowtie s = \Pi_{R \cup S}(\sigma_{r.A_1=s.A_1 \wedge r.A_2=s.A_2 \wedge \ldots \wedge r.A_n=s.A_n}(r \times s))$
  - r and s are joined on their common attributes
  - Result is projected so that common attributes only appear once

Natural Join Example

- Simple example:

  "Find the names of all customers with loans."

- Result:

  $\Pi_{cust\_name}(\sigma_{borrower.loan\_id=loan.loan\_id}(borrower \times loan))$

- Rewritten with natural join:

  $\Pi_{cust\_name}(borrower \bowtie loan)$

Natural Join Characteristics

- Very common to compute joins across multiple tables
- Example: *customer* ⋈ *borrower* ⋈ *loan*
- Natural join operation is associative:
  - (*customer* ⋈ *borrower*) ⋈ *loan* is equivalent to *customer* ⋈ (*borrower* ⋈ *loan*)

- Note:
  - Even though these expressions are equivalent, order of join operations can dramatically affect query cost!
  - (Keep this in mind for later…)

Division Operation

- Binary operator: $r \div s$
- Implements a "for each" type of query
  - "Find all rows in *r* that have one row corresponding to each row in *s*."
  - Relation *r* divided by relation *s*
- Easiest to illustrate with an example:
- Puzzle Database

  puzzle_list(*puzzle_name*)
    - Simple list of puzzles by name

  completed(*person_name*, *puzzle_name*)
    - Records which puzzles have been completed by each person

Puzzle Database

"Who has solved every puzzle?"

- Need to find every person in *completed* that has an entry for every puzzle in *puzzle_list*
- Divide *completed* by *puzzle_list* to get answer:

$completed \div puzzle\_list =$

person_name
Alex
Carl

- Only Alex and Carl have completed every puzzle in *puzzle_list*.

person_name	puzzle_name
Alex	altekruse
Alex	soma cube
Bob	puzzle box
Carl	altekruse
Bob	soma cube
Carl	puzzle box
Alex	puzzle box
Carl	soma cube

completed

puzzle_name
altekruse
soma cube
puzzle box

puzzle_list

Puzzle Database (2)

"Who has solved every puzzle?"

completed ÷ *puzzle_list* =

person_name
Alex
Carl

- Very reminiscent of integer division
  - Result relation contains tuples from *completed* that are evenly divided by *puzzle_name*
- Several other kinds of relational division operators
  - e.g. some can compute "remainder" of the division operation

person_name	puzzle_name
Alex	altekruse
Alex	soma cube
Bob	puzzle box
Carl	altekruse
Bob	soma cube
Carl	puzzle box
Alex	puzzle box
Carl	soma cube

completed

puzzle_name
altekruse
soma cube
puzzle box

puzzle_list

Division Operation

For $r(R) \div s(S)$

- Required: $S \subseteq R$
  - All attributes in S must also be in R
- Result has schema $R - S$
  - Result has attributes that are in R but not also in S
  - (Probably best if $S \subset R$…)
- Every tuple t in result satisfies these conditions:

  $t \in \Pi_{R-S}(r)$

  $\langle\ \forall\ t_s \in s : \exists\ t_r \in r : t_r[S] = t_s[S] \cap t_r[R{-}S] = t\ \rangle$

    - Every tuple in the result has a row in r corresponding to every row in s

Puzzle Database

person_name	puzzle_name
Alex	altekruse
Alex	soma cube
Bob	puzzle box
Carl	altekruse
Bob	soma cube
Carl	puzzle box
Alex	puzzle box
Carl	soma cube

completed = *r*

puzzle_name
altekruse
soma cube
puzzle box

puzzle_list = *s*

For *completed* ÷ *puzzle_list*

- Schemas are compatible
- Result has schema (*person_name*)
  - Attributes in *completed* schema, but not also in *puzzle_list* schema

person_name
Alex
Carl

completed ÷ *puzzle_list*

- Every tuple *t* in result satisfies these conditions:

$t \in \Pi_{R-S}(r)$

$\langle\ \forall\ t_s \in s : \exists\ t_r \in r : t_r[S] = t_s[S] \cap t_r[R-S] = t\ \rangle$

Division Operation

- Not provided natively in most SQL databases
  - Rarely needed!
  - Easy enough to implement in SQL, if needed

- Will see it in the homework assignments, and on the midterm… ☺
  - Often a very nice shortcut for more involved queries

Relation-Variables

- Recall: relation variables refer to a specific relation
  - A specific set of tuples, with a particular schema
- Example: *account* relation

acct_id	branch_name	balance
A-301	New York	350
A-307	Seattle	275
A-318	Los Angeles	550
A-319	New York	80
A-322	Los Angeles	275

account

  - *account* is actually technically a relation-variable, as are all our named relations so far

Assignment Operation

- Can assign a relation-value to a relation-variable
- Written as: *relvar* $\leftarrow$ *E*
  - *E* is an expression that evaluates to a relation
- Unlike ρ, the name *relvar* persists in the database
- Often used for temporary relation-variables:

  $temp1 \leftarrow \Pi_{R-S}(r)$

  $temp2 \leftarrow \Pi_{R-S}((temp1 \times s) - \Pi_{R-S,S}(r))$

  $result \leftarrow temp1 - temp2$

  - Query evaluation becomes a sequence of steps
  - (This is an implementation of the $\div$ operator)
- Can also use to represent data updates
  - More about updates next time…

RELATIONAL ALGEBRA II

CS121: Introduction to Relational Database Systems

Fall 2014 – Lecture 3

Last Lecture

- Query languages provide support for retrieving information from a database
- Introduced the relational algebra
  - A procedural query language
  - Six fundamental operations:
    - select, project, set-union, set-difference, Cartesian product, rename
  - Several additional operations, built upon the fundamental operations
    - set-intersection, natural join, division, assignment

Extended Operations

- Relational algebra operations have been extended in various ways
  - More generalized
  - More useful!
- Three major extensions:
  - Generalized projection
  - Aggregate functions
  - Additional join operations
- *All* of these appear in SQL standards

Generalized Projection Operation

- Would like to include computed results into relations
  - e.g. "Retrieve all credit accounts, computing the current 'available credit' for each account."
  - Available credit = credit limit – current balance
- Project operation is generalized to include computed results
  - Can specify *functions* on attributes, as well as attributes themselves
  - Can also assign names to computed values
  - (Renaming attributes is also allowed, even though this is also provided by the ρ operator)

Generalized Projection

- Written as: $\Pi_{F_1, F_2, \ldots, F_n}(E)$
  - F_i are arithmetic expressions
  - E is an expression that produces a relation
  - Can also name values: F_i **as** *name*
- Can use to provide <u>derived attributes</u>
  - Values are always computed from other attributes stored in database
- Also useful for updating values in database
  - (more on this later)

Generalized Projection Example

- "Compute available credit for every credit account."

$$\Pi_{\textit{cred\_id},\ (\textit{limit} - \textit{balance})\ \textbf{as}\ \textit{available\_credit}}(\textit{credit\_acct})$$

cred_id	limit	balance
C-273	2500	150
C-291	750	600
C-304	15000	3500
C-313	300	25

credit_acct

⇨

cred_id	available_credit
C-273	2350
C-291	150
C-304	11500
C-313	275

Aggregate Functions

- Very useful to apply a function to a collection of values to generate a single result
- Most common aggregate functions:

sum	sums the values in the collection
avg	computes average of values in the collection
count	counts number of elements in the collection
min	returns minimum value in the collection
max	returns maximum value in the collection

- Aggregate functions work on <u>multisets</u>, not sets
  - A value can appear in the input multiple times

Aggregate Function Examples

"Find the total amount owed to the credit company."

$$\mathcal{G}_{\textbf{sum}(balance)}(credit\_acct)$$

4275

cred_id	limit	balance
C-273	2500	150
C-291	750	600
C-304	15000	3500
C-313	300	25

credit_acct

"Find the maximum available credit of any account."

$$\mathcal{G}_{\textbf{max}(available\_credit)}(\Pi_{(limit\ -\ balance)\ \textbf{as}\ available\_credit}(credit\_acct))$$

11500

Grouping and Aggregation

- Sometimes need to compute aggregates on a *per-item* basis
- Back to the puzzle database:

  puzzle_list(*puzzle_name*)

  completed(*person_name*, *puzzle_name*)

puzzle_name
altekruse
soma cube
puzzle box

puzzle_list

- Examples:
  - How many puzzles has *each person* completed?
  - How many people have completed each puzzle?

person_name	puzzle_name
Alex	altekruse
Alex	soma cube
Bob	puzzle box
Carl	altekruse
Bob	soma cube
Carl	puzzle box
Alex	puzzle box
Carl	soma cube

completed

Grouping and Aggregation (2)

puzzle_name
altekruse
soma cube
puzzle box

puzzle_list

person_name	puzzle_name
Alex	altekruse
Alex	soma cube
Bob	puzzle box
Carl	altekruse
Bob	soma cube
Carl	puzzle box
Alex	puzzle box
Carl	soma cube

completed

"How many puzzles has each person completed?"

$_{person\_name}\mathcal{G}_{\mathbf{count}(puzzle\_name)}(completed)$

- First, input relation *completed* is grouped by unique values of *person_name*
- Then, **count**(*puzzle_name*) is applied separately to each group

Grouping and Aggregation (3)

$${}_{person\_name}\mathcal{G}_{\mathbf{count}(puzzle\_name)}(completed)$$

Input relation is grouped by *person_name*

Aggregate function is applied to each group

person_name	puzzle_name
Alex	altekruse
Alex	soma cube
Alex	puzzle box
Bob	puzzle box
Bob	soma cube
Carl	altekruse
Carl	puzzle box
Carl	soma cube

person_name	count(puzzle_name)
Alex	3
Bob	2
Carl	3

Distinct Values

- Sometimes want to compute aggregates over sets of values, instead of multisets

Example:

- Chage puzzle database to include a *completed_times* relation, which records multiple solutions of a puzzle

- How many puzzles has each person completed?
  - Using *completed_times* relation this time

person_name	puzzle_name	seconds
Alex	altekruse	350
Alex	soma cube	45
Bob	puzzle box	240
Carl	altekruse	285
Bob	puzzle box	215
Alex	altekruse	290

completed_times

Distinct Values (2)

"How many puzzles has each person completed?"

- Each puzzle appears multiple times now.

person_name	puzzle_name	seconds
Alex	altekruse	350
Alex	soma cube	45
Bob	puzzle box	240
Carl	altekruse	285
Bob	puzzle box	215
Alex	altekruse	290

completed_times

- Need to count distinct occurrences of each puzzle's name

$${}_{person\_name}\mathcal{G}_{\textbf{count-distinct}(puzzle\_name)}(completed\_times)$$

Eliminating Duplicates

- Can append **-distinct** to any aggregate function to specify elimination of duplicates
  - Usually used with **count:** **count-distinct**
  - Makes no sense with **min, max**

General Form of Aggregates

- General form: $_{G_1, G_2, \ldots, G_n}\mathcal{G}_{F_1(A_1), F_2(A_2), \ldots, F_m(A_m)}(E)$
  - E evalutes to a relation
  - Leading G_i are attributes of E to group on
  - Each F_i is aggregate function applied to attribute A_i of E
- First, input relation is divided into groups
  - If no attributes G_i specified, no grouping is performed (it's just one big group)
- Then, aggregate functions applied to each group

General Form of Aggregates (2)

- General form: ${}_{G_1, G_2, \ldots, G_n}\mathcal{G}_{F_1(A_1), F_2(A_2), \ldots, F_m(A_m)}(E)$
- Tuples in E are grouped such that:
  - All tuples in a group have same values for attributes $G_1, G_2, \ldots, G_n$
  - Tuples in different groups have different values for $G_1, G_2, \ldots, G_n$
- Thus, the values $\{g_1, g_2, \ldots, g_n\}$ in each group uniquely identify the group
  - $\{G_1, G_2, \ldots, G_n\}$ are a superkey for the result relation

General Form of Aggregates (3)

- General form: $_{G_1, G_2, \ldots, G_n}\mathcal{G}_{F_1(A_1), F_2(A_2), \ldots, F_m(A_m)}(E)$
- Tuples in result have the form:
  $\{g_1, g_2, \ldots, g_n, a_1, a_2, \ldots, a_m\}$
  - g_i are values for that particular group
  - a_i is result of applying F_i to the multiset of values of A_i in that group
- Important note: $F_i(A_i)$ attributes are unnamed!
  - Informally we refer to them as $F_i(A_i)$ in results, but they have no name.
  - Specify a name, same as before: $F_i(A_i)$ **as** *attr_name*

One More Aggregation Example

puzzle_name
altekruse
soma cube
puzzle box

puzzle_list

person_name	puzzle_name
Alex	altekruse
Alex	soma cube
Bob	puzzle box
Carl	altekruse
Bob	soma cube
Carl	puzzle box
Alex	puzzle box
Carl	soma cube

completed

"How many people have completed each puzzle?"

$${}_{puzzle\_name}\mathcal{G}_{\mathbf{count}(person\_name)}(completed)$$

- What if nobody has tried a particular puzzle?
  - Won't appear in *completed* relation

One More Aggregation Example

puzzle_name
altekruse
soma cube
puzzle box
clutch box

puzzle_list

person_name	puzzle_name
Alex	altekruse
Alex	soma cube
Bob	puzzle box
Carl	altekruse
Bob	soma cube
Carl	puzzle box
Alex	puzzle box
Carl	soma cube

completed

- New puzzle added to *puzzle_list* relation
  - Would like to see { "clutch box", 0 } in result…
  - "clutch box" won't appear in result!
- Joining the two tables doesn't help either
  - Natural join won't produce any rows with "clutch box"

Outer Joins

- Natural join requires that both left and right tables have a matching tuple

  $r \bowtie s = \Pi_{R \cup S}(\sigma_{r.A_1=s.A_1 \wedge r.A_2=s.A_2 \wedge \ldots \wedge r.A_n=s.A_n}(r \times s))$

- Outer join is an extension of join operation
  - Designed to handle *missing information*
- Missing information is represented by *null* values in the result
  - *null* = unknown or unspecified value

Forms of Outer Join

- Left outer join: $r ⟕ s$
  - If a tuple $t_r \in r$ doesn't match any tuple in s, result contains $\{ t_r, null, \ldots, null \}$
  - If a tuple $t_s \in s$ doesn't match any tuple in r, it's excluded
- Right outer join: $r ⟖ s$
  - If a tuple $t_r \in r$ doesn't match any tuple in s, it's excluded
  - If a tuple $t_s \in s$ doesn't match any tuple in r, result contains $\{ null, \ldots, null, t_s \}$

Forms of Outer Join (2)

- Full outer join: $r ⟗ s$
  - Includes tuples from *r* that don't match *s*, as well as tuples from *s* that don't match *r*
- Summary:

$r =$

attr1	attr2
a	r1
b	r2
c	r3

$s =$

attr1	attr3
b	s2
c	s3
d	s4

$r \bowtie s$

attr1	attr2	attr3
b	r2	s2
c	r3	s3

$r ⟕ s$

attr1	attr2	attr3
a	r1	*null*
b	r2	s2
c	r3	s3

$r ⟖ s$

attr1	attr2	attr3
b	r2	s2
c	r3	s3
d	*null*	s4

$r ⟗ s$

attr1	attr2	attr3
a	r1	*null*
b	r2	s2
c	r3	s3
d	*null*	s4

Effects of *null* Values

- Introducing *null* values affects *everything!*
  - *null* means "unknown" or "nonexistent"
- Must specify effect on results when *null* is present
  - These choices are *somewhat* arbitrary…
  - (Read your database user's manual! ☺)
- Arithmetic operations (+, –, *, /) involving *null* evaluate to *null*
- Comparison operations involving *null* evaluate to <u>*unknown*</u>
  - *unknown* is a third truth-value
  - **Note:** Yes, even *null* = *null* evaluates to *unknown*.

Boolean Operators and *unknown*

- and

  true $\wedge$ *unknown* = *unknown*

  false $\wedge$ *unknown* = false

  unknown $\wedge$ *unknown* = *unknown*

- or

  true $\vee$ *unknown* = true

  false $\vee$ *unknown* = *unknown*

  unknown $\vee$ *unknown* = *unknown*

- not

  $\neg$ *unknown* = *unknown*

Relational Operations

- For each relational operation, need to specify behavior with respect to *null* and *unknown*
- Select: $\sigma_P(E)$
  - If P evaluates to *unknown* for a tuple, that tuple is excluded from result (i.e. definition of σ doesn't change)
- Natural join: $r \bowtie s$
  - Includes a Cartesian product, then a select
  - If a common attribute has a *null* value, tuples are excluded from join result
  - Why?
    - *null* = (anything) evaluates to *unknown*

Project and Set-Operations

- Project: $\Pi(E)$
  - Project operation must eliminate duplicates
  - *null* value is treated like any other value
  - Duplicate tuples containing *null* values are also eliminated
- Union, Intersection, and Difference
  - *null* values are treated like any other value
  - Set union, intersection, difference computed as expected
- These choices are somewhat arbitrary
  - *null* means "value is unknown or missing"…
  - …but in these cases, two *null* values are considered equal.
  - Technically, two *null* values aren't the same. (oh well)

Grouping and Aggregation

- In grouping phase:
  - *null* is treated like any other value
  - If two tuples have same values (including *null*) on the grouping attributes, they end up in same group
- In aggregation phase:
  - *null* values are removed from the input multiset before aggregate function is applied!
    - Slightly different from arithmetic behavior; it keeps one *null* value from wiping out an aggregate computation.
  - If aggregate function gets an empty multiset for input, the result is *null*…
    - …except for **count**! In that case, **count** returns 0.

Generalized Projection, Outer Joins

- Generalized Projection operation:
  - A combination of simple projection and arithmetic operations
  - Easy to figure out from previous rules
- Outer joins:
  - Behave just like natural join operation, except for padding missing values with *null*

Back to Our Puzzle!

"How many people have completed each puzzle?"

puzzle_name
altekruse
soma cube
puzzle box
clutch box

puzzle_list

person_name	puzzle_name
Alex	altekruse
Alex	soma cube
Bob	puzzle box
Carl	altekruse
Bob	soma cube
Carl	puzzle box
Alex	puzzle box
Carl	soma cube

completed

- Use an outer join to include <u>all</u> puzzles, not just solved ones

  puzzle_list ⟕ *completed*

puzzle_name	person_name
altekruse	Alex
soma cube	Alex
puzzle box	Bob
altekruse	Carl
soma cube	Bob
puzzle box	Carl
puzzle box	Alex
soma cube	Carl
clutch box	*null*

Counting the Solutions

- Now, use grouping and aggregation
  - Group on puzzle name
  - Count up the people!

$$_{puzzle\_name}\mathcal{G}_{\mathbf{count}(person\_name)}(puzzle\_list ⟕ completed)$$

puzzle_name	person_name
altekruse	Alex
soma cube	Alex
puzzle box	Bob
altekruse	Carl
soma cube	Bob
puzzle box	Carl
puzzle box	Alex
soma cube	Carl
clutch box	*null*

puzzle_name	person_name
altekruse	Alex
altekruse	Carl
soma cube	Alex
soma cube	Bob
soma cube	Carl
puzzle box	Bob
puzzle box	Carl
puzzle box	Alex
clutch box	*null*

puzzle_name	count
altekruse	2
soma cube	3
puzzle box	3
clutch box	0

Database Modification

- Often need to modify data in a database
- Can use assignment operator $\leftarrow$ for this
- Operations:
  - $r \leftarrow r \cup E$ Insert new tuples into a relation
  - $r \leftarrow r - E$ *D*elete tuples from a relation
  - $r \leftarrow \Pi(r)$ Update tuples already in the relation

- Remember: *r* is a relation-variable
  - Assignment operator assigns a new relation-value to *r*
  - Hence, RHS expression may need to include existing version of *r*, to avoid losing unchanged tuples

Inserting New Tuples

- Inserting tuples simply involves a union:

  $r \leftarrow r \cup E$

  - *E* has to have correct arity

- Can specify actual tuples to insert:

  completed $\leftarrow$ *completed* $\cup$
  { ("Bob", "altekruse"), ("Carl", "clutch box") }

  constant relation

  - Adds two new tuples to *completed* relation

- Can specify constant relations as a set of values
  - Each tuple is enclosed with parentheses
  - Entire set of tuples enclosed with curly-braces

Inserting New Tuples (2)

- Can also insert tuples generated from an expression
- Example:

  "Dave is joining the puzzle club. He has done every puzzle that Bob has done."

  - Find out puzzles that Bob has completed, then construct new tuples to add to *completed*

Inserting New Tuples (3)

- How to construct new tuples with name "Dave" and each of Bob's puzzles?
  - Could use a Cartesian product:

    $\{ (\text{"Dave"}) \} \times \Pi_{puzzle\_name}(\sigma_{person\_name=\text{"Bob"}}(completed))$
  - Or, use generalized projection:

    $\Pi_{\text{"Dave"}\ \textbf{as}\ person\_name,\ puzzle\_name}(\sigma_{person\_name=\text{"Bob"}}(completed))$
- Add new tuples to *completed* relation:

  $completed \leftarrow completed \cup$
  $\Pi_{\text{"Dave"}\ \textbf{as}\ person\_name,\ puzzle\_name}(\sigma_{person\_name=\text{"Bob"}}(completed))$

Deleting Tuples

- Deleting tuples uses the – operation:

  $r \leftarrow r - E$

- Example:

  Get rid of the "soma cube" puzzle.

puzzle_name
altekruse
soma cube
puzzle box

puzzle_list

Problem:

- *completed* relation references the *puzzle_list* relation
- To respect referential integrity constraints, should delete from *completed* first.

person_name	puzzle_name
Alex	altekruse
Alex	soma cube
Bob	puzzle box
Carl	altekruse
Bob	soma cube
Carl	puzzle box
Alex	puzzle box
Carl	soma cube

completed

Deleting Tuples (2)

- *completed* references *puzzle_list*
  - *puzzle_name* is a key
  - *completed* shouldn't have any values for *puzzle_name* that don't appear in *puzzle_list*
  - Delete tuples from *completed* first.
  - <u>Then</u> delete tuples from *puzzle_list*.

$completed \leftarrow completed - \sigma_{puzzle\_name=\text{"soma cube"}}(completed)$

$puzzle\_list \leftarrow puzzle\_list - \sigma_{puzzle\_name=\text{"soma cube"}}(puzzle\_list)$

Of course, could also write:

$completed \leftarrow \sigma_{puzzle\_name\neq\text{"soma cube"}}(completed)$

Deleting Tuples (3)

- In the relational model, we have to think about foreign key constraints ourselves…
- Relational database systems take care of these things for us, automatically.
  - Will explore the various capabilities and options in a few weeks

Updating Tuples

- General form uses generalized projection:

  $r \leftarrow \Pi_{F_1, F_2, \ldots, F_n}(r)$

- Updates <u>all</u> tuples in *r*

acct_id	branch_name	balance
A-301	New York	350
A-307	Seattle	275
A-318	Los Angeles	550
A-319	New York	80
A-322	Los Angeles	275

account

- Example:

  "Add 5% interest to all bank account balances."

  $account \leftarrow \Pi_{acct\_id,\ branch\_name,\ (balance*1.05)}(account)$

  - **Note:** Must include unchanged attributes too

Updating *Some* Tuples

- Updating only *some* tuples is more verbose
  - Relation-variable is set to the *entire result* of the evaluation
  - Must include both updated tuples, and non-updated tuples, in result
- Example:

  "Add 5% interest to accounts with a balance less than $10,000."

  $$account \leftarrow \Pi_{acct\_id,\ branch\_name,\ (balance*1.05)}(\sigma_{balance<10000}(account)) \cup \sigma_{balance \geq 10000}(account)$$

Updating *Some* Tuples (2)

Another example:

"Add 5% interest to accounts with a balance less than $10,000, and 6% interest to accounts with a balance of $10,000 or more."

$$account \leftarrow \Pi_{acct\_id, branch\_name, (balance*1.05)}(\sigma_{balance<10000}(account)) \cup$$
$$\Pi_{acct\_id, branch\_name, (balance*1.06)}(\sigma_{balance \geq 10000}(account))$$

- Don't forget to include any non-updated tuples in your update operations!

Relational Algebra Summary

- Very expressive query language for retrieving information from a relational database
  - Simple selection, projection
  - Computing correlations between relations using joins
  - Grouping and aggregation operations
- Can also specify changes to the contents of a relation-variable
  - Inserts, deletes, updates
- The relational algebra is a <u>procedural</u> query language
  - State a sequence of operations for computing a result

Relational Algebra Summary (2)

- Benefit of relational algebra is that it can be formally specified and reasoned about
- Drawback is that it is *very* verbose!
- Database systems usually provide much simpler query languages
  - Most popular *by far* is SQL, the Structured Query Language
- However, many databases use relational algebra-like operations internally!
  - Great for representing execution plans, due to its procedural nature

Next Time

- Transition from relational algebra to SQL
- Start working with “real” databases ☺

SQL OVERVIEW

CS121: Introduction to Relational Database Systems

Fall 2014 – Lecture 4

SQL

- SQL = Structured Query Language
- Original language was "SEQUEL"
  - IBM's System R project (early 1970's)
  - "Structured English Query Language"
- Caught on very rapidly
  - Simple, declarative language for writing queries
  - Also includes many other features
- Standardized by ANSI/ISO
  - SQL-86, SQL-89, SQL-92, SQL:1999, SQL:2003, SQL:2008, SQL:2011
  - Most implementations *loosely* follow the standards (plenty of portability issues)

SQL Features

- Data Definition Language (DDL)
  - Specify relation schemas (attributes, domains)
  - Specify a variety of integrity constraints
  - Access constraints on data
  - Indexes and other storage “hints” for performance
- Data Manipulation Language (DML)
  - Generally based on relational algebra
  - Supports querying, inserting, updating, deleting data
  - Very sophisticated features for multi-table queries
- Other useful tools
  - Defining views, transactions, etc.

SQL Basics

- SQL language is case-insensitive
  - both keywords and identifiers (for the most part)
- SQL statements end with a semicolon
- SQL comments have two forms:
  - Single-line comments start with two dashes

```
-- This is a SQL comment.
```

  - Block comments follow C style

```
/*
 * This is a block comment in SQL.
 */
```

SQL Databases

- SQL relations are contained within a database
  - Each application usually works against its own database
  - Several applications may share the same database, too
- An example from MySQL:

```
CREATE DATABASE bank;
USE bank;
```

  - Creates a new, empty database called **bank**
  - **USE** statement makes **bank** the "default" database for the current connection
  - DDL and DML operations will be evaluated in the context of the connection's default database

Creating a SQL Table

- In SQL, relations are called "tables"
  - Not *exactly* like relational model "relations" anyway
- Syntax:

```
CREATE TABLE t (
    attr1 domain1,
    attr2 domain2,
    ... ,
    attrN domainN
);
```

  - **`t`** is name of relation (table)
  - **`attr1`**, … are names of attributes (columns)
  - **`domain1`**, … are domains (types) of attributes

SQL Names

- Tables, columns, etc. require names
- Rules on valid names can vary dramatically across implementations

- Good, portable rules:
  - First character should be alphabetical
  - Remaining characters should be alphanumeric or underscore '_'
  - Use same the case in DML that you use in DDL

SQL Attribute Domains

- Some standard SQL domain types:

  `CHAR(N)`
  - A character field, fixed at N characters wide
  - Short for **`CHARACTER(N)`**

  `VARCHAR(N)`
  - A variable-width character field, with maximum length N
  - Short for **`CHARACTER VARYING(N)`**

  `INT`
  - A signed integer field (typically 32 bits)
  - Short for **`INTEGER`**
  - Also **`TINYINT`**, **`SMALLINT`**, **`BIGINT`**, etc.
  - Also unsigned variants
    - Non-standard, only supported by some vendors

CHAR vs. VARCHAR

- Both **CHAR** and **VARCHAR** have a size limit
- **CHAR** is a fixed-length character field
  - Can store shorter strings, but storage layer pads out the value to the full size
- **VARCHAR** is a variable-length character field
  - Storage layer doesn't pad out shorter strings
  - String's length must also be stored for each value
- Use **CHAR** when all values are approximately (or *exactly*) the same length
- Use **VARCHAR** when values can be very different lengths

SQL Attribute Domains (2)

- More standard SQL domain types:

  `NUMERIC(P,D)`
  - A fixed-point number with user-specified precision
  - P total digits; D digits to right of decimal place
  - Can exactly store numbers

  `DOUBLE PRECISION`
  - A double-precision floating-point value
  - An approximation! Don't use for money! ☺
  - **`REAL`** is sometimes a synonym

  `FLOAT(N)`
  - A floating-point value with at least N bits of precision

SQL Attribute Domains (3)

- Other useful attribute domains, too:

  `DATE`, `TIME`, `TIMESTAMP`
  - For storing temporal data
- Large binary/text data fields

  `BLOB`, `CLOB`, `TEXT`
  - Binary Large Objects, Character Large Objects
  - Large text fields
  - **`CHAR`, `VARCHAR`** tend to be very limited in size
- Other specialized types
  - Enumerations, geometric or spatial data types, etc.
  - User-defined data types

Choosing the Right Type

- Need to think carefully about what type makes most sense for your data values
- Example: storing ZIP codes
  - US postal codes for mail routing
  - 5 digits, e.g. 91125 for Caltech
- Does `INTEGER` make sense?
- **Problem 1:** Some ZIP codes have leading zeroes!
  - Many east-coast ZIP codes start with 0.
  - Numeric types won't include leading zeros.
- **Problem 2:** US mail also uses ZIP+4 expanded ZIP codes
  - e.g. 91125-8000
- **Problem 3:** Many foreign countries use non-numeric values

Choosing the Right Type (2)

- Better choice for ZIP codes?
  - A **`CHAR`** or **`VARCHAR`** column makes much more sense
- For example:
  - **`CHAR(5)`** or **`CHAR(9)`** for US-only postal codes
  - **`VARCHAR(20)`** for US + international postal codes
- Another example: monetary amounts
  - Floating-point representations cannot exactly represent all values
    - e.g. 0.1 is an infinitely-repeating binary decimal value
  - Use **`NUMERIC`** to represent monetary values

Example SQL Schema

- Creating the account relation:

```
CREATE TABLE account (
    acct_id      CHAR(10),
    branch_name  CHAR(20),
    balance      NUMERIC(12, 2)
);
```

  - Account IDs can't be more than 10 chars
  - Branch names can't be more than 20 chars
  - Balances can have 10 digits left of decimal, 2 digits right of decimal
    - Fixed-point, exact precision representation of balances

Inserting Rows

- Tables are initially empty
- Use **INSERT** statement to add rows

```
INSERT INTO account
   VALUES ('A-301', 'New York', 350);
INSERT INTO account
   VALUES ('A-307', 'Seattle', 275);
...
```

  - String values are single-quoted
  - (In SQL, double-quoted strings refer to column names)
  - Values appear in same order as table's attributes

Inserting Rows (2)

- Can specify which attributes in **INSERT**

```
INSERT INTO account (acct_id, branch_name, balance)
  VALUES ('A-301', 'New York', 350);
```

  - Can list attributes in a different order
  - Can exclude attributes that have a default value
- Problem: We can add multiple accounts with same account ID!

```
INSERT INTO account
  VALUES ('A-350', 'Seattle', 800);
INSERT INTO account
  VALUES ('A-350', 'Los Angeles', 195);
```

Primary Key Constraints

- The **CREATE TABLE** syntax also allows integrity constraints to be specified
  - Are often specified after all attributes are listed
- Primary key constraint:

```
CREATE TABLE account (
    acct_id       CHAR(10),
    branch_name   CHAR(20),
    balance       NUMERIC(12, 2),

    PRIMARY KEY (acct_id)
);
```

  - Database won't allow two rows with same account ID

Primary Key Constraints (2)

- A primary key can have multiple attributes

```
CREATE TABLE depositor (
  customer_name  VARCHAR(30),
  acct_id        CHAR(10),
  PRIMARY KEY (customer_name, acct_id)
);
```

  - Necessary because SQL tables are multisets
- A table cannot have multiple primary keys
  - (obvious)
- *Many* other kinds of constraints too
  - Will cover in future lectures!

Removing Rows, Tables, etc.

- Can delete rows with **DELETE** command
  - Delete bank account with ID A-307:

    ```
    DELETE FROM account WHERE acct_id = 'A-307';
    ```
  - Delete all bank accounts:

    ```
    DELETE FROM account;
    ```
- Can drop tables and databases:
  - Remove account table:

    ```
    DROP TABLE account;
    ```
  - Remove an entire database, including all tables!

    ```
    DROP DATABASE bank;
    ```

Issuing SQL Queries

- SQL queries use the **SELECT** statement
- *Very* central part of SQL language
  - Concepts appear in all DML commands
- General form is:

```
SELECT A1, A2, ...
  FROM r1, r2, ...
  WHERE P;
```

  - r_i are the relations (tables)
  - A_i are attributes (columns)
  - P is the selection predicate

SELECT Operations

- **SELECT** A_1, A_2, ...
  - Corresponds to a relational algebra project operation $\Pi_{A_1,\, A_2,\, \ldots}(\ \ldots\ )$
  - Some books call σ "restrict" because of this name mismatch
- **FROM** r_1, r_2, ...
  - Corresponds to Cartesian product of relations r_1, r_2, … $r_1 \times r_2 \times \ldots$

`SELECT` Operations (2)

- **`WHERE P`**
  - Corresponds to a selection operation

    $\sigma_P(\ldots)$
  - Can be omitted. When left off, P = true
- Assembling it all:

  `SELECT` $\boldsymbol{A_1}$**`,`** $\boldsymbol{A_2}$**`, ... FROM`** $\boldsymbol{r_1}$**`,`** $\boldsymbol{r_2}$**`, ...`**
  `WHERE P;`
  - Equivalent to: $\Pi_{A_1, A_2, \ldots}(\sigma_P(r_1 \times r_2 \times \ldots))$

SQL and Duplicates

- Biggest difference between relational algebra and SQL is use of multisets
  - In SQL, relations are <u>multisets</u> of tuples, not sets
- Biggest reason is practical:
  - Removing duplicate tuples is time consuming!
- Must revise definitions of relational algebra operations to handle duplicates
  - Mainly affects set-operations: $\cup$, $\cap$, $-$
  - (Book explores this topic in depth)
- SQL provides ways to exclude duplicates for all operations

Example Queries

"Find all branches with at least one bank account."

```
SELECT branch_name
  FROM account;
```

- Equivalent to typing:

```
SELECT ALL branch_name
  FROM account;
```

```
+--------------+
| branch_name  |
+--------------+
| New York     |
| Seattle      |
| Los Angeles  |
| New York     |
| Los Angeles  |
+--------------+
```

- To eliminate duplicates:

```
SELECT DISTINCT branch_name
  FROM account;
```

```
+--------------+
| branch_name  |
+--------------+
| New York     |
| Seattle      |
| Los Angeles  |
+--------------+
```

Selecting Specific Attributes

- Can specify one or more attributes to appear in result

  "Find ID and balance of all bank accounts."

  ```
  SELECT acct_id, balance
    FROM account;
  ```

acct_id	balance
A-301	350.00
A-307	275.00
A-318	550.00
A-319	80.00
A-322	275.00

- Can also specify * to mean "all attributes"

  ```
  SELECT * FROM account;
  ```

  - Returns all details of all accounts.

acct_id	branch_name	balance
A-301	New York	350.00
A-307	Seattle	275.00
A-318	Los Angeles	550.00
A-319	New York	80.00
A-322	Los Angeles	275.00

Computing Results

- The **SELECT** clause is a *generalized projection* operation
  - Can compute results based on attributes

```
SELECT cred_id, credit_limit - balance
  FROM credit_account;
```

  - Computed values don't have a (standard) name!
    - Many DBMSes name the 2nd column "**credit_limit - balance**"
- Can also name (or rename) values

```
SELECT cred_id,
       credit_limit - balance AS available_credit
  FROM credit_account;
```

WHERE Clause

- The **WHERE** clause specifies a selection predicate
  - Can use comparison operators:

    =, <> equals, not-equals (**!=** also usually supported)

    <, <= less than, less or equal

    >, >= greater than, greater or equal
  - Can refer to any attribute in **FROM** clause
  - Can include arithmetic expressions in comparisons

WHERE Examples

"Find IDs and balances of all accounts in the Los Angeles branch."

```
SELECT acct_id, balance FROM account
  WHERE branch_name = 'Los Angeles';
```

acct_id	balance
A-318	550.00
A-322	275.00

"Retrieve all details of bank accounts with a balance less than $300."

```
SELECT * FROM account
  WHERE balance < 300;
```

acct_id	branch_name	balance
A-307	Seattle	275.00
A-319	New York	80.00
A-322	Los Angeles	275.00

Larger Predicates

- Can use **AND**, **OR**, **NOT** in **WHERE** clause

```
SELECT acct_id, balance FROM account
  WHERE branch_name = 'Los Angeles' AND
        balance < 300;
```

```
SELECT * FROM account
  WHERE balance >= 250 AND balance <= 400;
```

- SQL also has **BETWEEN** and **NOT BETWEEN** syntax

```
SELECT * FROM account
  WHERE balance BETWEEN 250 AND 400;
```

  - Note that **BETWEEN** <u>includes</u> interval endpoints!

String Comparisons

- String values can be compared
  - Lexicographic comparisons
  - Default is often to ignore case!

    `SELECT 'HELLO' = 'hello';  -- Evaluates to true`
- Can also do pattern matching with **`LIKE`** expression

  `string_attr` **`LIKE`** ***`pattern`***
  - **`pattern`** is a string literal enclosed in single-quotes
    - `%` (percent) matches a substring
    - `_` (underscore) matches a single character
    - Can escape `%` or `_` with a backslash `\`

String-Matching Example

"Find all accounts at branches with 'le' somewhere in the name."

- Why? I don't know…

```
SELECT * FROM account
  WHERE branch_name LIKE '%le%';
```

```
+---------+-------------+---------+
| acct_id | branch_name | balance |
+---------+-------------+---------+
| A-307   | Seattle     |  275.00 |
| A-318   | Los Angeles |  550.00 |
| A-322   | Los Angeles |  275.00 |
+---------+-------------+---------+
```

String Operations

- Regular-expression matching is also part of the SQL standard (SQL:1999)
- String-matching operations tend to be expensive
  - Especially patterns with a leading wildcard, e.g. `'%abc'`
- Try to avoid heavy reliance on pattern-matching

- If string searching is required, try to pre-digest text and generate search indexes
  - Some databases provide "full-text search" capabilities, but such features are vendor-specific!

FROM Clause

- Can specify one or more tables in **FROM** clause
- If multiple tables:
  - Select/project against Cartesian product of relations

```
-- Produces a row for every combination
-- of input tuples.
SELECT * FROM borrower, loan;
```

```
+-----------+---------+---------+---------------+---------+
| cust_name | loan_id | loan_id | branch_name   | amount  |
+-----------+---------+---------+---------------+---------+
| Anderson  | L-437   | L-419   | Seattle       | 2900.00 |
| Jackson   | L-419   | L-419   | Seattle       | 2900.00 |
| Lewis     | L-421   | L-419   | Seattle       | 2900.00 |
| Smith     | L-445   | L-419   | Seattle       | 2900.00 |
| Anderson  | L-437   | L-421   | San Francisco | 7500.00 |
| Jackson   | L-419   | L-421   | San Francisco | 7500.00 |
| Lewis     | L-421   | L-421   | San Francisco | 7500.00 |
| ...                                                     |
```

FROM Clause (2)

- If tables have overlapping attributes, use **tbl_name.attr_name** to distinguish

```
SELECT * FROM borrower, loan
  WHERE borrower.loan_id = loan.loan_id;
```

```
+-----------+---------+---------+---------------+---------+
| cust_name | loan_id | loan_id | branch_name   | amount  |
+-----------+---------+---------+---------------+---------+
| Jackson   | L-419   | L-419   | Seattle       | 2900.00 |
| Lewis     | L-421   | L-421   | San Francisco | 7500.00 |
| Anderson  | L-437   | L-437   | Las Vegas     | 4300.00 |
| Smith     | L-445   | L-445   | Los Angeles   | 2000.00 |
+-----------+---------+---------+---------------+---------+
```

  - All columns can be referred to by **tbl_name.attr_name**
- This kind of query is called an equijoin
- Databases optimize equijoin queries *very* effectively.

SQL and Joins

- SQL provides several different options for performing joins across multiple tables
- This form is the most basic usage
  - Was in earliest versions of SQL
  - Doesn't provide natural joins
  - Can't do outer joins either
- Will cover other forms of SQL join syntax soon…

Renaming Tables

- Can specify alternate names in **FROM** clause too
  - Write: **`table AS name`**
  - (The **AS** is optional, but it's clearer to leave it in.)
- Previous example:

  "Find the loan with the largest amount."
  - Started by finding loans that have an amount smaller than some other loan's amount
  - Used Cartesian product and rename operation

```
SELECT DISTINCT loan.loan_id
  FROM loan, loan AS test
  WHERE loan.amount < test.amount;
```

```
+---------+
| loan_id |
+---------+
| L-445   |
| L-419   |
| L-437   |
+---------+
```

Renaming Tables (2)

- When a table is renamed in **FROM** clause, can use the new name in both **SELECT** and **WHERE** clauses
- Useful for long table names! ☺

```
SELECT c.cust_name, l.amount
  FROM customer AS c, borrower AS b,
       loan AS l
  WHERE c.cust_name = b.cust_name AND
        b.loan_id = l.loan_id;
```

Set Operations

- SQL also provides set operations, like relational algebra
- Operations take two relations and produce an output relation
- Set-union:

  ***select₁* UNION *select₂* ;**
- Set-intersection:

  ***select₁* INTERSECT *select₂* ;**
- Set-difference:

  ***select₁* EXCEPT *select₂* ;**
- **Note:** ***select$_i$*** are complete **SELECT** statements!

Set-Operation Examples

- Find customers with an account or a loan:

```
SELECT cust_name FROM depositor UNION
SELECT cust_name FROM borrower;
```

  - Database automatically eliminates duplicates

- Find customers with an account but not a loan:

```
SELECT cust_name FROM depositor EXCEPT
SELECT cust_name FROM borrower;
```

  - Can also put parentheses around **SELECT** clauses for readability

```
(SELECT cust_name FROM depositor)
EXCEPT
(SELECT cust_name FROM borrower);
```

Set Operations and Duplicates

- By default, SQL set-operations <u>eliminate</u> duplicate tuples
  - Opposite to default behavior of **`SELECT`**!
- Can keep duplicate tuples by appending **`ALL`** to set operation:

  select$_1$ **UNION ALL** ***select***$_2$ **;**

  select$_1$ **INTERSECT ALL** ***select***$_2$ **;**

  select$_1$ **EXCEPT ALL** ***select***$_2$ **;**

How Many Duplicates?

- Need to define behavior of "set operations" on multisets
- Given two <u>multiset</u> relations r_1 and r_2
  - r_1 and r_2 have same schema
  - Some tuple t appears c_1 times in r_1, and c_2 times in r_2

$\boldsymbol{r}_1 \cup_{\text{ALL}} \boldsymbol{r}_2$

contains $c_1 + c_2$ copies of t

$\boldsymbol{r}_1 \cap_{\text{ALL}} \boldsymbol{r}_2$

contains $min(c_1, c_2)$ copies of t

$\boldsymbol{r}_1 -_{\text{ALL}} \boldsymbol{r}_2$

contains $max(c_1 - c_2, 0)$ copies of t

Other Relational Operations

- Can actually update definitions of all relational operations to support multisets
- Necessary for using relational algebra to model execution plans
- Not terribly interesting though… ☺

- If you're curious, see book for details

SQL Style Guidelines

- Follow good coding style in SQL!
- Some recommendations:
  - Use lowercase names for tables, columns, etc.
  - Put a descriptive comment above every table
  - Write all SQL keywords in uppercase
  - Follow standard indentation scheme
    - e.g. indent columns in table declarations by 2-4 spaces
  - Keep lines to 80 characters or less!
    - wrap lines in reasonable places

- **Note:** You <u>will</u> lose points for sloppy SQL.

Next Time

- Sorting results
- Grouping and aggregate functions
- Nested queries and many more set operations
- How to update SQL databases

SQL QUERIES

CS121: Introduction to Relational Database Systems

Fall 2014 – Lecture 5

SQL Queries

- SQL queries use the **SELECT** statement
- General form is:

```
SELECT A1, A2, ...
  FROM r1, r2, ...
  WHERE P;
```

  - r_i are the relations (tables)
  - A_i are attributes (columns)
  - P is the selection predicate
- Equivalent to: $\Pi_{A_1, A_2, \ldots}(\sigma_P(r_1 \times r_2 \times \ldots))$

Ordered Results

- SQL query results can be ordered by particular attributes
- Two main categories of query results:
  - "Not ordered by anything"
    - Tuples can appear in *any* order
  - "Ordered by attributes A_1, A_2, …"
    - Tuples are sorted by specified attributes
    - Results are sorted by A_1 first
    - Within each value of A_1, results are sorted by A_2
    - etc.
- Specify an **`ORDER BY`** clause at end of **`SELECT`** statement

Ordered Results (2)

- Find bank accounts with a balance under $700:

```
SELECT account_number, balance
  FROM account
  WHERE balance < 700;
```

```
+----------------+---------+
| account_number | balance |
+----------------+---------+
| A-102          |  400.00 |
| A-101          |  500.00 |
| A-444          |  625.00 |
| A-305          |  350.00 |
+----------------+---------+
```

- Order results in increasing order of bank balance:

```
SELECT account_number, balance
  FROM account
  WHERE balance < 700
  ORDER BY balance;
```

```
+----------------+---------+
| account_number | balance |
+----------------+---------+
| A-305          |  350.00 |
| A-102          |  400.00 |
| A-101          |  500.00 |
| A-444          |  625.00 |
+----------------+---------+
```

  - Default order is ascending order

Ordered Results (3)

- Say **ASC** or **DESC** after attribute name to specify order
  - **ASC** is redundant, but can improve readability in some cases
- Can list multiple attributes, each with its own order

"Retrieve a list of all bank branch details, ordered by branch city, with each city's branches listed in reverse order of holdings."

```
SELECT * FROM branch
  ORDER BY branch_city ASC, assets DESC;
```

```
+-------------+-------------+------------+
| branch_name | branch_city | assets     |
+-------------+-------------+------------+
| Pownal      | Bennington  |  400000.00 |
| Brighton    | Brooklyn    | 7000000.00 |
| Downtown    | Brooklyn    |  900000.00 |
| Round Hill  | Horseneck   | 8000000.00 |
| Perryridge  | Horseneck   | 1700000.00 |
| Mianus      | Horseneck   |  400200.00 |
| Redwood     | Palo Alto   | 2100000.00 |
| ...         | ...         |        ... |
```

Aggregate Functions in SQL

- SQL provides grouping and aggregate operations, just like relational algebra
- Aggregate functions:

`SUM`	sums the values in the collection
`AVG`	computes average of values in the collection
`COUNT`	counts number of elements in the collection
`MIN`	returns minimum value in the collection
`MAX`	returns maximum value in the collection

- **SUM** and **AVG** require numeric inputs (obvious)

Aggregate Examples

- Find average balance of accounts at Perryridge branch

```
SELECT AVG(balance) FROM account
  WHERE branch_name = 'Perryridge';
```

```
+--------------+
| AVG(balance) |
+--------------+
|   650.000000 |
+--------------+
```

- Find maximum amount of any loan in the bank

```
SELECT MAX(amount) AS max_amt FROM loan;
```

  - Can name computed values, like usual

```
+---------+
| max_amt |
+---------+
| 7500.00 |
+---------+
```

Aggregate Examples (2)

- This query produces an error:

```
SELECT branch_name,
       MAX(amount) AS max_amt
  FROM loan;
```

- Aggregate functions compute a *single value* from a multiset of inputs
  - Doesn't make sense to combine individual attributes and aggregate functions like this
- This does work:

```
SELECT MIN(amount) AS min_amt,
       MAX(amount) AS max_amt
  FROM loan;
```

```
+---------+---------+
| min_amt | max_amt |
+---------+---------+
|  500.00 | 7500.00 |
+---------+---------+
```

Eliminating Duplicates

- Sometimes need to eliminate duplicates in SQL queries
  - Can use **DISTINCT** keyword to eliminate duplicates
- Example:

"Find the number of branches that currently have loans."

```
SELECT COUNT(branch_name) FROM loan;
```

  - Doesn't work, because branches may have multiple loans
  - Instead, do this:

```
SELECT COUNT(DISTINCT branch_name) FROM loan;
```

  - Duplicates are eliminated from input multiset before aggregate function is applied

Computing Counts

- Can count individual attribute values

```
COUNT(branch_name)
COUNT(DISTINCT branch_name)
```

- Can also count the total number of tuples

```
COUNT(*)
```

  - If used with grouping, counts total number of tuples in each group
  - If used without grouping, counts total number of tuples
- Counting a specific attribute is useful when:
  - Need to count (possibly distinct) values of a particular attribute
  - Cases where some values in input multiset may be **NULL**
    - As before, **COUNT** ignores **NULL** values (more on this next week)

Grouping and Aggregates

- Can also perform grouping on a relation before computing aggregates
  - Specify a **GROUP BY** $A_1, A_2, \ldots$ clause at end of query
- Example:

"Find the average loan amount for each branch."

```
SELECT branch_name, AVG(amount) AS avg_amt
  FROM loan GROUP BY branch_name;
```

- First, tuples in **loan** are grouped by **branch_name**
- Then, aggregate functions are applied to each group

branch_name	avg_amt
Central	570.000000
Downtown	1250.000000
Mianus	500.000000
North Town	7500.000000
Perryridge	1400.000000
Redwood	2000.000000
Round Hill	900.000000

Grouping and Aggregates (2)

- Can group on multiple attributes
  - Each group has unique values for the entire set of grouping attributes
- Example:

  "How many accounts does each customer have at each branch?"
  - Group by both customer name *and* branch name
  - Compute count of tuples in each group
  - Can write the SQL statement yourself, and try it out

Grouping and Aggregates (3)

- Note the difference between relational algebra notation and SQL syntax
- Relational algebra syntax:

  $_{G_1,\, G_2,\, \ldots,\, G_n}\mathcal{G}_{F_1(A_1),\, F_2(A_2),\, \ldots,\, F_m(A_m)}(E)$

  - Grouping attributes only appear on left of $\mathcal{G}$
- SQL syntax:

  SELECT $G_1, G_2, \ldots,\ F_1(A_1), F_2(A_2), \ldots$
  FROM $r_1, r_2, \ldots$ **WHERE P**
  GROUP BY $G_1, G_2, \ldots$

  - Frequently, grouping attributes are specified in both the **SELECT** clause <u>and</u> **GROUP BY** clause

Grouping and Aggregates (4)

- SQL doesn't <u>require</u> that you specify the grouping attributes in the **SELECT** clause
  - Only requirement is that the grouping attributes are specified in the **GROUP BY** clause
  - e.g. if you only want the aggregated results, could do this:

```
SELECT F1(A1),F2(A2),...
  FROM r1,r2,... WHERE P
  GROUP BY G1,G2,...
```

- Also, can use expressions for grouping and aggregates
  - Example (very uncommon, but also valid):

```
SELECT MIN(a + b) – MAX(c)
FROM t GROUP BY d * e;
```

Filtering Tuples

- The **WHERE** clause is applied *before* any grouping occurs

```
SELECT G1,G2,..., F1(A1),F2(A2),...
  FROM r1,r2,... WHERE P
  GROUP BY G1,G2,...
```

  - Translates into relational algebra expression:

    $\Pi_{\ldots}({}_{G_1, G_2, \ldots}\mathcal{G}_{F_1(A_1), F_2(A_2), \ldots}(\sigma_P(r_1 \times r_2 \times \ldots)))$
  - A **WHERE** clause constrains the set of tuples that grouping and aggregation are applied to

Filtering Results

- To apply filtering to the results of grouping and aggregation, use a **HAVING** clause
  - Exactly like **WHERE** clause, except applied *after* grouping and aggregation

```
SELECT G1,G2,..., F1(A1),F2(A2),...
  FROM r1,r2,... WHERE PW
  GROUP BY G1,G2,...
  HAVING PH
```

  - Translates into:

$$\Pi_{\ldots}(\sigma_{P_H}(_{G_1, G_2, \ldots}\mathcal{G}_{F_1(A_1), F_2(A_2), \ldots}(\sigma_{P_W}(r_1 \times r_2 \times \ldots))))$$

The **HAVING** Clause

- The **HAVING** clause can use aggregate functions in its predicate
  - It's applied after grouping/aggregation, so those values are available
  - The **WHERE** clause *cannot* do this, of course
- Example:

"Find all customers with more than one loan."

```
SELECT customer_name, COUNT(*) AS num_loans
  FROM borrower GROUP BY customer_name
  HAVING COUNT(*) > 1;
```

```
+---------------+-----------+
| customer_name | num_loans |
+---------------+-----------+
| Smith         |         3 |
+---------------+-----------+
```

Nested Subqueries

- SQL provides broad support for nested subqueries
  - A SQL query is a "select-from-where" expression
  - Nested subqueries are "select-from-where" expressions embedded within another query
- Can embed queries in **`WHERE`** clauses
  - Sophisticated selection tests
- Can embed queries in **`FROM`** clauses
  - Issuing a query against a derived relation
- Can even embed queries in **`SELECT`** clauses!
  - Appeared in SQL:2003 standard; many DBs support this
  - Makes many queries easier to write, but can be slow too

Kinds of Subqueries

- Some subqueries produce only a single result

```
SELECT MAX(assets) FROM branch;
```

  - Called a <u>scalar subquery</u>
  - Still a relation, just with one attribute and one tuple
- Most subqueries produce a relation containing multiple tuples
  - Nested queries often produce relation with single attribute
    - Very common for subqueries in **WHERE** clause
  - Nested queries can also produce multiple-attribute relation
    - Very common for subqueries in **FROM** clause
    - Can also be used in the **WHERE** clause in some cases

Subqueries in **WHERE** Clause

- Widely used:
  - Direct comparison with scalar-subquery results
  - Set-membership tests: **IN**, **NOT IN**
  - Empty-set tests: **EXISTS**, **NOT EXISTS**
- Less frequently used:
  - Set-comparison tests: **ANY**, **SOME**, **ALL**
  - Uniqueness tests: **UNIQUE**, **NOT UNIQUE**
- (Can also use these in the **HAVING** clause)

Comparison with Subquery Result

- Can use scalar subqueries in **WHERE** clause comparisons
- Example:
  - Want to find the name of the branch with the smallest number of assets.
  - Can easily find the smallest number of assets:

```
SELECT MIN(assets) FROM branch;
```

  - This is a scalar subquery; can use it in **WHERE** clause:

```
SELECT branch_name FROM branch
WHERE assets = (SELECT MIN(assets) FROM branch);
```

```
+-------------+
| branch_name |
+-------------+
| Pownal      |
+-------------+
```

Set Membership Tests

- Can use **`IN (...)`** and **`NOT IN (...)`** for set membership tests
- Example:
  - Find customers with both an account and a loan.
  - Before, did this with a **`INTERSECT`** operation
  - Can also use a set-membership test:
    "Select all customer names from depositor relation, that also appear somewhere in borrower relation."

```
SELECT DISTINCT customer_name FROM depositor
 WHERE customer_name IN (
          SELECT customer_name FROM borrower)
```

  - **`DISTINCT`** necessary because a customer might appear multiple times in **`depositor`**

Set Membership Tests (2)

- **`IN (...)`** and **`NOT IN (...)`** support subqueries that return multiple columns (!!!)
- Example: "Find the ID of the largest loan at each branch, including the branch name and the amount of the loan."
  - First, need to find the largest loan at each branch

```
SELECT branch_name, MAX(amount)
  FROM loan GROUP BY branch_name
```

  - Use this result to identify the rest of the loan details

```
SELECT * FROM loan
WHERE (branch_name, amount) IN (
      SELECT branch_name, MAX(amount)
      FROM loan GROUP BY branch_name);
```

Empty-Set Tests

- Can test whether or not a subquery generates any results at all
  - `EXISTS (...)`
  - `NOT EXISTS (...)`
- Example:

  "Find customers with an account but not a loan."

```
SELECT DISTINCT customer_name FROM depositor d
WHERE NOT EXISTS (
    SELECT * FROM borrower b
    WHERE b.customer_name = d.customer_name);
```

  - Result includes every customer that appears in depositor table, that *doesn't also* appear in the borrower table.

Empty-Set Tests (2)

"Find customers with an account but not a loan."

```
SELECT DISTINCT customer_name FROM depositor d
WHERE NOT EXISTS (
    SELECT * FROM borrower b
    WHERE b.customer_name = d.customer_name);
```

- Inner query refers to an attribute in outer query's relation

- In general, nested subqueries can refer to enclosing queries' relations.
- However, enclosing queries cannot refer to the nested queries' relations.

Correlated Subqueries

"Find customers with an account but not a loan."

```
SELECT DISTINCT customer_name FROM depositor d
WHERE NOT EXISTS (
    SELECT * FROM borrower b
    WHERE b.customer_name = d.customer_name);
```

- When a nested query refers to an enclosing query's attributes, it is a <u>correlated subquery</u>
  - The inner query must be evaluated once *for each tuple* considered by the enclosing query
  - Generally to be avoided! *Very* slow.

Correlated Subqueries (2)

- Many correlated subqueries can be restated using a join or a Cartesian product
  - Often the join operation will be *much* faster
  - More advanced DBMSes will automatically decorrelate such queries, but some can't…
- Certain conditions, e.g. **`EXISTS`**/**`NOT EXISTS`**, usually indicate presence of a correlated subquery
- If it's easy to decorrelate the subquery, do that! ☺
- If not, test the query for its performance.
  - If the database can decorrelate it, you're done!
  - If the database can't decorrelate it, may need to come up with an alternate formulation.

Set Comparison Tests

- Can compare a value to a set of values
  - Is a value larger/smaller/etc. than *some* value in the set?
- Example:

  "Find all branches with assets greater than at least one branch in Brooklyn."

```
SELECT branch_name FROM branch
  WHERE assets > SOME (
    SELECT assets FROM branch
      WHERE branch_name='Brooklyn');
```

Set Comparison Tests (2)

- General form of test:

  `attr` *compare_op* **`SOME ( `** *subquery* **`)`**

  - Can use any comparison operation

    `= SOME` is same as **`IN`**

  - **`ANY`** is a synonym for **`SOME`**

- Can also compare a value with *all* values in a set
  - Use **`ALL`** instead of **`SOME`**

    `<> ALL` is same as **`NOT IN`**

Set Comparison Tests (3)

- Example:

  "Find branches with assets greater than *all* branches in Brooklyn."

```
SELECT branch_name FROM branch
  WHERE assets > ALL (
    SELECT assets FROM branch
      WHERE branch_name='Brooklyn');
```

  - Could also write this with a scalar subquery

```
SELECT branch_name FROM branch
  WHERE assets >
         (SELECT MAX(assets) FROM branch
            WHERE branch_name='Brooklyn');
```

Uniqueness Tests

- Can test whether a nested query generates any duplicate tuples
  - `UNIQUE (...)`
  - `NOT UNIQUE (...)`
- Not widely implemented
  - Expensive operation!
- Can emulate in a number of ways
  - `GROUP BY ... HAVING COUNT(*) = 1` or `GROUP BY ... HAVING COUNT(*) > 1` is one approach

Subqueries in **FROM** Clause

- Often need to compute a result in multiple steps
- Can query against a subquery's results
  - Called a <u>derived relation</u>
- A trivial example:
  - A **HAVING** clause can be implemented as a nested query in the **FROM** clause

HAVING vs. Nested Query

"Find all cities with more than two customers living in the city."

```
SELECT customer_city, COUNT(*) AS num_customers
FROM customer GROUP BY customer_city
HAVING COUNT(*) > 2;
```

- Or, can write:

```
SELECT customer_city, num_customers
FROM (SELECT customer_city, COUNT(*)
      FROM customer GROUP BY customer_city)
      AS counts (customer_city, num_customers)
WHERE num_customers > 2;
```

  - Grouping and aggregation is computed by inner query
  - Outer query selects desired results generated by inner query

Derived Relation Syntax

- Subquery in **FROM** clause must be given a name
  - Many DBMSes also require attributes to be named

```
SELECT customer_city, num_customers
FROM (SELECT customer_city, COUNT(*)
      FROM customer GROUP BY customer_city)
      AS counts (customer_city, num_customers)
WHERE num_customers > 2;
```

  - Nested query is called **counts**, and specifies two attributes
  - Syntax varies from DBMS to DBMS…
    - MySQL requires a name for derived relations, but *doesn't* allow attribute names to be specified.

Using Derived Relations

- More typical is a query against aggregate values
- Example:

  "Find the largest *total* account balance of any branch."

  - Need to compute total account balance for each branch first.

    ```
    SELECT branch_name, SUM(balance) AS total_bal
    FROM account GROUP BY branch_name;
    ```

  - Then we can easily find the answer:

    ```
    SELECT MAX(total_bal) AS largest_total
    FROM (SELECT branch_name,
                 SUM(balance) AS total_bal
          FROM account GROUP BY branch_name)
          AS totals (branch_name, tot_bal);
    ```

Aggregates of Aggregates

- <u>Always</u> take note when computing aggregates of aggregates!

  "Find the <u>largest</u> <u>total</u> account balance of any branch."

  - Two nested aggregates: max of sums
- A very common mistake:

```
SELECT branch_name, SUM(balance) AS tot_bal
FROM account GROUP BY branch_name
HAVING tot_bal = MAX(tot_bal)
```

  - A **SELECT** query can only perform <u>one</u> level of aggregation
  - Need a second **SELECT** to find the maximum total
  - Unfortunately, MySQL accepts this and returns bogus result

More Data Manipulation Operations

- SQL provides many other options for inserting, updating, and deleting tuples
- All commands support **`SELECT`**-style syntax
- Can insert individual tuples into a table:

```
INSERT INTO table VALUES (1, 'foo', 50);
```

- Can also insert the result of a query into a table:

```
INSERT INTO table SELECT ...;
```

  - Only constraint is that generated results must have a compatible schema

Deleting Tuples

- SQL **DELETE** command can use a **WHERE** clause

  `DELETE FROM table;`

  - Deletes all rows in the table

  `DELETE FROM table WHERE ...;`

  - Only deletes rows that satisfy the conditions
  - The **WHERE** clause can use anything that **SELECT**'s **WHERE** clause supports
    - Nested queries, in particular!

Updating Tables

- SQL also has an **UPDATE** command for modifying existing tuples in a table
- General form:

```
UPDATE table
   SET attr1=val1, attr2=val2, ...
 WHERE condition;
```

  - Must specify the attributes to update
  - Attributes being modified *must* appear in table being updated (obvious)
  - The **WHERE** clause is optional! If unspecified, *all* rows are updated.
  - **WHERE** condition can contain nested queries, etc.

Updating Tables (2)

- Values in **UPDATE** can be arithmetic expressions
  - Can refer to any attribute in table being updated
- Example:
  - Add 2% interest to all bank account balances with a balance of $500 or less.

```
UPDATE account
  SET balance = balance * 1.02
  WHERE balance <= 500;
```

Review

- SQL query syntax is *very* rich
  - Can state a wide range of complex queries
  - Many ways to state a particular query
- SQL supports nested queries
  - Often essential for computing particular results
  - Can sometimes be *very* inefficient
- SQL also provides similar capability for inserting, deleting, and updating tables

Next Time

- **`NULL`** values in SQL
- Additional SQL join operations
  - Natural join
  - Outer joins
- SQL views

SUBQUERIES AND VIEWS

CS121: Introduction to Relational Database Systems

Fall 2014 – Lecture 6

String Comparisons and GROUP BY

- Last time, introduced many advanced features of SQL, including **GROUP BY**
- Recall: string comparisons using **=** are *case-insensitive* by default

```
SELECT 'HELLO' = 'hello';   -- Evaluates to true
```

- This can also cause unexpected results with SQL grouping and aggregation
- Example: table of people's favorite colors
  - ```
    CREATE TABLE favorite_colors (
      name  VARCHAR(30) PRIMARY KEY,
      color VARCHAR(30)
    );
    ```
    ```

String Compares and GROUP BY (2)

- Add data to our table:

```
INSERT INTO favorite_colors VALUES ('Alice', 'BLUE');
INSERT INTO favorite_colors VALUES ('Bob', 'Red');
INSERT INTO favorite_colors VALUES ('Clara', 'blue');
...
```

- How many people like each color?
  - **`SELECT color, COUNT(*) num_people FROM favorite_colors GROUP BY color;`**
  - Even though "BLUE" and "blue" differ in case, they will still end up in the same group!

Null Values in SQL

- Like relational algebra, SQL represents missing information with *null* values
  - **NULL** is a keyword in SQL
  - Typically written in all-caps
- Use **IS NULL** and **IS NOT NULL** to check for *null* values
  - **attr = NULL** is *never* true! (It is *unknown*.)
  - **attr <> NULL** is also *never* true! (Also *unknown*.)
  - Instead, write: **attr IS NULL**
- Aggregate operations ignore **NULL** input values
  - **COUNT** returns 0 for an empty input multiset
  - All others return **NULL** for an empty input (even **SUM** !)

Comparisons and Unknowns

- Relational algebra introduced the *unknown* truth-value
  - Produced by comparisons with *null*
- SQL also has tests for *unknown* values

```
comp IS UNKNOWN
comp IS NOT UNKNOWN
```

  - ***comp*** is some comparison operation

NULL in Inserts and Updates

- Can specify **NULL** values in **INSERT** and **UPDATE** statements

```
INSERT INTO account
  VALUES ('A-315', NULL, 500);
```

  - Can clearly lead to some problems…
  - Primary key attributes are not allowed to have **NULL** values
  - Other ways to specify constraints on **NULL** values for specific attributes

Additional Join Operations

- SQL-92 introduces additional join operations
  - natural joins
  - left/right/full outer joins
  - theta joins
- Syntax varies from the basic “Cartesian product” join syntax
  - All changes are in **`FROM`** clause
  - Varying levels of syntactic sugar…

Theta Join

- One relational algebra operation we skipped
- Theta join is a generalized join operation
  - Sometimes called a "condition join"
- Written as: $r \bowtie_{\theta} s$
- Abbreviation for: $\sigma_{\theta}(r \times s)$
- Doesn't include project operation like natural join and outer joins do
- No *null*-padded results, like outer joins have

SQL Theta Joins

- SQL provides a syntax for theta joins
- Example:

  Associate customers and loan balances

```
SELECT * FROM borrower INNER JOIN loan ON
 borrower.loan_number = loan.loan_number;
```

  - Result:

customer_name	loan_number	loan_number	branch_name	amount
Smith	L-11	L-11	Round Hill	900.00
Jackson	L-14	L-14	Downtown	1500.00
Hayes	L-15	L-15	Perryridge	1500.00
Adams	L-16	L-16	Perryridge	1300.00
Jones	L-17	L-17	Downtown	1000.00
...	...	...	...	...

SQL Theta Joins (2)

- Syntax in **`FROM`** clause:

```
table1 INNER JOIN table2 ON condition
```

  - **`INNER`** is optional; just distinguishes from outer joins
- No duplicate attribute names are removed
  - Can specify relation name, attribute names

```
table1 INNER JOIN table2 ON condition
  AS rel (attr1, attr2, ...)
```

- Very similar to a derived relation

Theta Joins on Multiple Tables

- Can join across multiple tables with this syntax
- Example: join customer, borrower, loan tables
  - Nested theta-joins:

```
SELECT * FROM customer AS c
  JOIN borrower AS b ON
         c.customer_name = b.customer_name
  JOIN loan AS l ON
         b.loan_number = l.loan_number;
```

  - Generally evaluated left to right
  - Can use parentheses to specify join order
  - Order usually doesn't affect results or performance
    (if outer joins are involved, results can definitely change)

Theta Joins on Multiple Tables (2)

Join customer, borrower, loan tables: take 2

- One Cartesian product and one theta join:

```
SELECT * FROM customer AS c
  JOIN borrower AS b JOIN loan AS l
  ON c.customer_name = b.customer_name
     AND b.loan_number = l.loan_number;
```

- Database will optimize this anyway, but it really isn't two theta joins

Join Conditions

- Can specify *any* condition (including nested subqueries) in **`ON`** clause
  - Even conditions that aren't related to join itself

- Guideline:
  - Use **`ON`** clause for join conditions
  - Use **`WHERE`** clause for selecting rows
  - Mixing the two can cause lots of confusion!

Cartesian Products

- Cartesian product can be specified as **CROSS JOIN**
  - Can't specify an **ON** condition for a **CROSS JOIN**
- Cartesian product of *borrower* and *loan*:

```
SELECT * FROM borrower CROSS JOIN loan;
```

  - Same as a theta join with no condition:

```
SELECT * FROM borrower INNER JOIN loan
  ON TRUE;
```

  - Or, simply:

```
SELECT * FROM borrower JOIN loan;
SELECT * FROM borrower, loan;
```

Outer Joins

- Can specify outer joins in SQL as well:

```
SELECT * FROM table1
  LEFT OUTER JOIN table2 ON condition;
SELECT * FROM table1
  RIGHT OUTER JOIN table2 ON condition;
SELECT * FROM table1
  FULL OUTER JOIN table2 ON condition;
```

  - **OUTER** is implied by **LEFT**/**RIGHT**/**FULL**, and can therefore be left out

```
SELECT * FROM table1 LEFT JOIN table2 ON
  condition;
```

Common Attributes

- **ON** syntax is clumsy for simple joins
  - Also, it's tempting to include conditions that should be in the **WHERE** clause
- Often, schemas are designed such that join columns have the same names
  - e.g. *borrower.loan_number* and *loan.loan_number*
- **USING** clause is a simplified form of **ON**

```
SELECT * FROM t1 LEFT OUTER JOIN t2
  USING (a1, a2, ...);
```

  - *Roughly* equivalent to:

```
SELECT * FROM t1 LEFT OUTER JOIN t2
  ON (t1.a1 = t2.a1 AND t1.a2 = t2.a2 AND ...);
```

Common Attributes (2)

- `USING` also eliminates duplicate join attributes
  - Result of join with `USING (a1, a2, ...)` will only have one instance of each join column in the result
  - This is fine, because `USING` requires equal values for the specified attributes
- Example: tables *r*(*a*, *b*, *c*) and *s*(*a*, *b*, *d*)
  - `SELECT * FROM r JOIN s USING (a)`
  - Result schema is: (*a*, *r.b*, *r.c*, *s.b*, *s.d*)
- Can use `USING` clause with `INNER` / `OUTER` joins
  - *No* condition allowed for `CROSS JOIN`

Natural Joins

- SQL natural join operation:

  ```
  SELECT * FROM t1 NATURAL INNER JOIN t2;
  ```

  - **INNER** is optional, as usual
  - No **ON** or **USING** clause is specified
- *All* common attributes are used in natural join operation
  - To join on a *subset* of common attributes, use a regular **INNER JOIN**, with a **USING** clause

Natural Join Example

Join borrower and loan relations:

```
SELECT * FROM borrower NATURAL JOIN loan;
```

- Result:

```
+-------------+---------------+-------------+---------+
| loan_number | customer_name | branch_name | amount  |
+-------------+---------------+-------------+---------+
| L-11        | Smith         | Round Hill  |  900.00 |
| L-14        | Jackson       | Downtown    | 1500.00 |
| L-15        | Hayes         | Perryridge  | 1500.00 |
| L-16        | Adams         | Perryridge  | 1300.00 |
| L-17        | Jones         | Downtown    | 1000.00 |
| L-17        | Williams      | Downtown    | 1000.00 |
| L-20        | McBride       | North Town  | 7500.00 |
| L-21        | Smith         | Central     |  570.00 |
| L-23        | Smith         | Redwood     | 2000.00 |
| L-93        | Curry         | Mianus      |  500.00 |
+-------------+---------------+-------------+---------+
```

  - Could also use inner join, **`USING (loan_number)`**

Natural Outer Joins

- Can also specify natural outer joins
  - **NATURAL** specifies how the rows/columns are matched
  - All overlapping columns are used for join operation
  - Unmatched tuples from (left, right, or both) tables are **NULL**-padded and included in result
- Example:

```
SELECT * FROM customer
  NATURAL LEFT OUTER JOIN borrower;
SELECT * FROM customer
  NATURAL LEFT JOIN borrower;
```

Outer Joins and Aggregates

- Outer joins can generate **NULL** values
- Aggregate functions ignore **NULL** values
  - **COUNT** has most useful behavior!
- Example:
  - Find out how many loans each customer has
  - Include customers with *no* loans; show 0 for those customers
  - Need to use *customer* and *borrower* tables
  - Need to use an outer join to include customers with no loans

Outer Joins and Aggregates (2)

- First step: left outer join customer and borrower tables

```
SELECT customer_name, loan_number
FROM customer LEFT OUTER JOIN borrower
     USING (customer_name);
```

- Generates result:
  - Customers with no loans have **NULL** for *loan_number* attribute

```
+---------------+-------------+
| customer_name | loan_number |
+---------------+-------------+
| Adams         | L-16        |
| Brooks        | NULL        |
| Curry         | L-93        |
| Glenn         | NULL        |
| Green         | NULL        |
| Hayes         | L-15        |
| ...                         |
+---------------+-------------+
```

Outer Joins and Aggregates (3)

- Finally, need to count number of accounts for each customer
  - Use grouping and aggregation for this
  - Grouping, aggregation is applied to *results* of **`FROM`** clause; won't interfere with join operation
- What's the difference between **`COUNT(*)`** and **`COUNT(loan_number)`** ?
  - **`COUNT(*)`** simply counts number of tuples in each group
  - **`COUNT(*)`** won't produce any counts of 0!
  - **`COUNT(loan_number)`** is what we want

Outer Joins and Aggregates (4)

- Final query:

```
SELECT customer_name,
       COUNT(loan_number) AS num_loans
FROM customer LEFT OUTER JOIN borrower
     USING (customer_name)
GROUP BY customer_name
ORDER BY COUNT(loan_number) DESC;
```

  - Sort by count, just to make it easier to analyze

```
+---------------+-----------+
| customer_name | num_loans |
+---------------+-----------+
| Smith         |         3 |
| Jones         |         1 |
| Curry         |         1 |
| McBride       |         1 |
| Hayes         |         1 |
| Jackson       |         1 |
| Williams      |         1 |
| Adams         |         1 |
| Brooks        |         0 |
| Lindsay       |         0 |
| ...                       |
```

Views

- So far, have used SQL at logical level
  - Queries generally use actual relations
  - …but they don't need to!
  - Can also write queries against derived relations
    - Nested subqueries or **`JOIN`**s in **`FROM`** clause
- SQL also provides view-level operations
- Can define <u>views</u> of the logical model
  - Can write queries directly against views

Why Views?

- Two main reasons for using views
- Reason 1: Performance and convenience
  - Define a view for a widely used derived relation
  - Write simple queries against the view
  - DBMS automatically computes view's contents when it is used in a query
- Some databases provide <u>materialized views</u>
  - View's result is pre-computed and stored on disk
  - DBMS ensures that view is "up to date"
    - Might update view's contents immediately, or periodically

Why Views? (2)

- Reason 2: Security!
  - Can specify access constraints on both tables and views
  - Can specify strict access constraints on a table with sensitive information
  - Can provide a view that excludes sensitive information, with more lenient access
- Example: employee information database
  - Logical-level tables might have SSN, salary info, other private information
  - An "employee directory" view could limit this down to employee name and professional contact information

Creating a View

- SQL syntax for creating a view is very simple
  - Based on **SELECT** syntax, as always

    ```
    CREATE VIEW viewname AS select_stmt;
    ```
  - View's columns are columns in **SELECT** statement
  - Column names must be unique, just like any table's columns
  - Can specify view columns in **CREATE VIEW** syntax:

    ```
    CREATE VIEW viewname (attr1, attr2, ...) AS
      select_stmt;
    ```
- Even easier to remove:

  ```
  DROP VIEW viewname;
  ```

Example View

- Create a view that shows *total* account balance of each customer.
  - The **SELECT** statement would be:

```
SELECT customer_name,
       SUM(balance) AS total_balance
FROM depositor NATURAL JOIN account
GROUP BY customer_name;
```

  - The view is just as simple:

```
CREATE VIEW customer_deposits AS
  SELECT customer_name,
         SUM(balance) AS total_balance
  FROM depositor NATURAL JOIN account
  GROUP BY customer_name;
```

- With views, good attribute names are a *must*.

Updating a View?

- A view is a derived relation…
- What to do if an **`INSERT`** or **`UPDATE`** refers to a view?
- One simple solution: Don't allow it! ☺
- Could also allow the database designer to specify what operations to perform when a modification is attempted against a view
  - Very flexible approach
  - Default is still to forbid updates to views

Updatable Views

- Can actually define updates for certain kinds of views
- A view is <u>updatable</u> if:
  - The **`FROM`** clause only uses one relation
  - The **`SELECT`** clause only uses attributes in the relation, and doesn't perform any computations
  - Attributes not listed in the **`SELECT`** clause can be set to **`NULL`**
  - The view's query doesn't perform any grouping or aggregation
- In these cases, **`INSERT`**s, **`UPDATE`**s, and **`DELETE`**s can be performed

Updatable Views (2)

- Example view:
  - All accounts at Downtown branch.

```
CREATE VIEW downtown_accounts AS
  SELECT account_number, branch_name, balance
  FROM account WHERE branch_name='Downtown';
```

- Is this view updatable?
  - **FROM** uses only one relation
  - **SELECT** includes all attributes from the relation
  - No computations, aggregates, distinct values, etc.
  - Yes, it is updatable!

Updatable Views?

- Issue a query against the view:

```
SELECT * FROM downtown_accounts;
```

```
+----------------+-------------+---------+
| account_number | branch_name | balance |
+----------------+-------------+---------+
| A-101          | Downtown    |  500.00 |
+----------------+-------------+---------+
```

- Insert a new tuple:

```
INSERT INTO downtown_accounts
  VALUES ('A-600', 'Mianus', 550);
```

- Look at the view again:

```
SELECT * FROM downtown_accounts;
```

```
+----------------+-------------+---------+
| account_number | branch_name | balance |
+----------------+-------------+---------+
| A-101          | Downtown    |  500.00 |
+----------------+-------------+---------+
```

  - **Where's my tuple?!**

Checking Inserted Rows

- Can add **WITH CHECK OPTION** to the view declaration
  - Inserted rows are checked against the view's **WHERE** clause
  - If a row doesn't satisfy the **WHERE** clause, it is rejected
- Updated view definition:

```
CREATE VIEW downtown_accounts AS
  SELECT account_number, branch_name, balance
  FROM account WHERE branch_name='Downtown'
WITH CHECK OPTION;
```

SQL DATA DEFINITION: KEY CONSTRAINTS

CS121: Introduction to Relational Database Systems

Fall 2014 – Lecture 7

Data Definition

- Covered most of SQL data manipulation operations
- Continue exploration of SQL data definition features
  - Specifying tables and their columns (lecture 4)
  - Declaring views of the logical-level schema (lecture 6)
  - Specifying constraints on individual columns, or entire tables
  - Providing stored procedures to manipulate data
  - Specifying security access constraints
  - …and more!

Data Definition (2)

- We will focus on the mechanics of data definition
- For now, ignoring a *very* important question:
  - Exactly what *is* a "good" database schema, anyway??!
- General design goals:
  - Should be able to fully represent all necessary details and relationships in the schema
  - Try to *eliminate* the ability to store invalid data
  - Many other design goals too (security, performance)
    - Sometimes these design goals conflict with each other…
- DBMSes can enforce *many* different constraints
  - Want to leverage this capability to ensure correctness

Catalogs and Schemas

- SQL provides hierarchical grouping capabilities for managing collections of tables
  - Also separate namespaces for different collections of tables
- Standard mechanism has three levels:
  - Catalogs
  - Schemas
  - Tables
  - Each level is assigned a name
  - Within each container, names must be unique
- Allows multiple applications to use the same server
  - Even multiple instances of a particular application

Catalogs and Schemas (2)

- Every table has a full name:
  - catalog.schema.table
- Database systems vary <u>widely</u> on implementation of these features!
  - Catalog functionality not covered by SQL specification
  - Schema and table levels are specified
  - Most DBMSes offer some kind of grouping
- Common behaviors:
  - "Databases" generally correspond to catalogs
    - `CREATE DATABASE web_db;`
  - Schema-level grouping is usually provided
    - `CREATE SCHEMA blog_schema;`

Using a Database

- Normally, must connect to a database server to use it
  - Specify a username and password, among other things
- Each database connection has its own environment
  - "Session state" associated with that client
  - Can specify the catalog and schema to use
    - e.g. `USE bank;` to use the banking database
    - e.g. Specifying database *`user_db`* to the MySQL client
  - All operations will use that catalog and schema by default
  - Can frequently override using full names for tables, etc.

Creating Tables

- General form:

```
CREATE TABLE name (
  attr1 type1,
  attr2 type2,
  ...
);
```

- SQL provides a variety of standard column types
  - **INT**, **CHAR(N)**, **VARCHAR(N)**, **DATE**, etc.
  - (see Lecture 4 for more details about basic column types)
- Table and column names must follow specific rules
- Table must have a unique name within schema
- All columns must have unique names within the table

Table Constraints

- By default, SQL tables have *no* constraints
  - Can insert multiple copies of a given row
  - Can insert rows with **NULL** values in any column
- Can specify columns that comprise primary key

```
CREATE TABLE account (
  account_number CHAR(10),
  branch_name    VARCHAR(20),
  balance        NUMERIC(12, 2),
  PRIMARY KEY (account_number)
);
```

  - No two rows can have same values for primary key
  - A table can have only one primary key

Primary Key Constraints

- Alternate syntax for primary keys

```
CREATE TABLE account (
  account_number CHAR(10)       PRIMARY KEY,
  branch_name    VARCHAR(20),
  balance        NUMERIC(12, 2)
);
```

  - Can only be used for single-column primary keys!

- For multi-column primary keys, must specify primary key after column specifications

```
CREATE TABLE depositor (
  customer_name  VARCHAR(30),
  account_number CHAR(10),
  PRIMARY KEY (customer_name, account_number)
);
```

Null-Value Constraints

- Every attribute domain contains *null* by default
  - Same with SQL: every column can be set to **NULL**, if it isn't part of a primary key
- Often, **NULL** is not an acceptable value!
  - e.g. bank accounts must <u>always</u> have a balance
- Can specify **NOT NULL** to exclude **NULL** values for particular columns
  - **NOT NULL** constraint specified in column declaration itself
- Stating **NOT NULL** for primary key columns is unnecessary and redundant

Account Relation

- Account number is a primary key
  - Already cannot be **NULL**
- Branch name and balance also should always be specified
  - Add **NOT NULL** constraints to those columns
- SQL:

```
CREATE TABLE account (
  account_number CHAR(10)        PRIMARY KEY,
  branch_name    VARCHAR(20)     NOT NULL,
  balance        NUMERIC(12, 2)  NOT NULL
);
```

Other Candidate Keys

- Some relations have multiple candidate keys
- Can specify candidate keys with **`UNIQUE`** constraints
  - Like primary key constraints, can specify candidate keys in the column declaration, or after all columns
  - Can only specify multi-column candidate key after the column specifications
- Unlike primary keys, **`UNIQUE`** constraints do not exclude **`NULL`** values!
  - This constraint considers **`NULL`** values to be unequal!
  - If some attributes in the **`UNIQUE`** constraint allow **`NULL`**s, DB will allow multiple rows with the same values!

UNIQUE Constraints

- Example: An employee relation

```
CREATE TABLE employee (
  emp_id   INT           PRIMARY KEY,
  emp_ssn  CHAR(9)       NOT NULL UNIQUE,
  emp_name VARCHAR(40)   NOT NULL,
  ...
);
```

  - Employee's ID is the primary key
  - All employees need a SSN, but no two employees should have the same SSN
    - Don't forget **NOT NULL** constraint too!
  - All employees should have a name, but multiple employees might have same name

UNIQUE and NULL

- Example:

```
CREATE TABLE customer (
  cust_name   VARCHAR(30) NOT NULL,
  address     VARCHAR(60),
  UNIQUE (cust_name, address)
);
```

- Try inserting values:

```
INSERT INTO customer
  VALUES ('John Doe', '123 Spring Lane');
INSERT INTO customer
  VALUES ('John Doe', '123 Spring Lane');
```

  - Second insert fails, as expected:

```
Duplicate entry 'John Doe-123 Spring Lane' for
  key 'cust_name'
```

UNIQUE and NULL (2)

- Example:

```
CREATE TABLE customer (
  cust_name  VARCHAR(30) NOT NULL,
  address    VARCHAR(60),
  UNIQUE (cust_name, address)
);
```

- Try inserting more values:

```
INSERT INTO customer VALUES ('Jane Doe', NULL);
INSERT INTO customer VALUES ('Jane Doe', NULL);
```

  - Both inserts succeed!
- **Be careful using nullable columns in UNIQUE constraints!**
  - Usually, you *really* want to specify **NOT NULL** for <u>all</u> columns that appear in **UNIQUE** constraints

CHECK Constraints

- Often want to specify other constraints on values
- Can require values in a table to satisfy some predicate, using a **CHECK** constraint
  - Very effective for constraining columns' domains, and eliminating obviously bad inputs
- **CHECK** constraints must appear after the column specifications
- In theory, can specify any expression that generates a Boolean result
  - This includes nested subqueries!
  - In practice, DBMS support for **CHECK** constraints varies widely, and is often quite limited

CHECK Constraint Examples

- Can constrain values in a particular column:

```
CREATE TABLE employee (
  emp_id   INT           PRIMARY KEY,
  emp_ssn  CHAR(9)       NOT NULL UNIQUE,
  emp_name VARCHAR(40)   NOT NULL,
  pay_rate NUMERIC(5,2)  NOT NULL,
  CHECK (pay_rate > 5.25)
);
```

- Ensures that all employees have a minimum wage

CHECK Constraint Examples (2)


```
CREATE TABLE employee (
  emp_id   INT            PRIMARY KEY,
  emp_ssn  CHAR(9)        NOT NULL UNIQUE,
  emp_name VARCHAR(40)    NOT NULL,
  status   VARCHAR(10)    NOT NULL,
  pay_rate NUMERIC(5,2)   NOT NULL,
  CHECK (pay_rate > 5.25),
  CHECK (status IN
         ('active', 'vacation', 'suspended'))
);
```

- Employee status must be one of the specified values
  - Like an enumerated type
  - (Many DBs provide similar support for enumerated types)

Another **CHECK** Constraint

- Depositor relation:

```
CREATE TABLE depositor (
  customer_name   VARCHAR(30),
  account_number CHAR(10),
  PRIMARY KEY (customer_name, account_number),
  CHECK (account_number IN
           (SELECT account_number FROM account))
);
```

- Rows in depositor table should only contain valid account numbers!
  - The valid account numbers appear in **account** table
  - This is a referential integrity constraint

Another **CHECK** Constraint (2)

- Depositor relation:

```
CREATE TABLE depositor (
  customer_name  VARCHAR(30),
  account_number CHAR(10),
  PRIMARY KEY (customer_name, account_number),
  CHECK (account_number IN
         (SELECT account_number FROM account))
);
```

- When does this constraint need to be checked?
  - When changes are made to `depositor` table
  - Also when changes are made to `account` table!

CHECK Constraints

- Easy to write *very* expensive **CHECK** constraints
- **CHECK** constraints aren't used very often
  - Lack of widespread support; using them limits portability
  - When used, they are usually *very* simple
    - Enforce more specific constraints on data values, or enforce string format constraints using regular expressions, etc.
  - Avoid huge performance impacts!
- Don't use **CHECK** constraints for referential integrity ☺
  - There's a better way!

Referential Integrity Constraints

- Referential integrity constraints are very important!
  - These constraints span multiple tables
  - Allow us to associate data across multiple tables
  - One table's values are constrained by another table's values
- A relation can specify a primary key
  - A set of attributes that uniquely identifies each tuple in the relation
- A relation can also include attributes of another relation's primary key
  - Called a <u>foreign key</u>
  - Referencing relation's values for the foreign key must also appear in the referenced relation

Referential Integrity Constraints (2)

- Given a relation $r(R)$
  - $K \subseteq R$ is the primary key for R
- Another relation $s(S)$ references r
  - $K \subseteq S$ too
  - $\langle\ \forall\ t_s \in s : \exists\ t_r \in r : t_s[K] = t_r[K]\ \rangle$
- Also called a subset dependency
  - $\Pi_K(s) \subseteq \Pi_K(r)$
  - Foreign-key values in s must be a subset of primary-key values in r

SQL Foreign Key Constraints

- Like primary key constraints, can specify in multiple ways
- For a single-column foreign key, can specify in column declaration
- Example:

```
CREATE TABLE depositor (
  customer_name  VARCHAR(30) REFERENCES customer,
  account_number CHAR(10)    REFERENCES account,
  PRIMARY KEY (customer_name, account_number),
);
```

  - Foreign key refers to primary key of referenced relation
- Foreign-key constraint does *NOT* imply **NOT NULL**!
  - Must explicitly add this, if necessary
  - In this example, **PRIMARY KEY** constraint eliminates **NULL**s

Foreign Key Constraints (2)

- Can also specify the column in the referenced relation
- Especially useful when referenced column is a candidate key, but not the primary key
- Example:
  - Employees have both company-assigned IDs and social security numbers
  - Health benefit information in another table, tied to social security numbers

Foreign Key Example

- Employee information:

```
CREATE TABLE employee (
  emp_id   INT           PRIMARY KEY,
  emp_ssn  CHAR(9)       NOT NULL UNIQUE,
  emp_name VARCHAR(40)   NOT NULL,
  ...
);
```

- Health plan information:

```
CREATE TABLE healthplan (
  emp_ssn  CHAR(9)      PRIMARY KEY
                        REFERENCES employee (emp_ssn),
  provider VARCHAR(20)  NOT NULL,
  pcp_id   INT          NOT NULL,
  ...
);
```

Multiple Constraints

- Can combine several different constraints

```
emp_ssn CHAR(9) PRIMARY KEY
                REFERENCES employee (emp_ssn)
```

  - *emp_ssn* is primary key of *healthplan* relation
  - *emp_ssn* is also a foreign key to *employee* relation
  - Foreign key references the candidate-key *employee.emp_ssn*

Self-Referencing Foreign Keys

- A relation can have a foreign key reference to itself
  - Common for representing hierarchies or graphs
- Example:

```
CREATE TABLE employee (
  emp_id      INT            PRIMARY KEY,
  emp_ssn     CHAR(9)        NOT NULL UNIQUE,
  emp_name    VARCHAR(40)    NOT NULL,
  ...
  manager_id  INT            REFERENCES employee
);
```

  - **manager_id** and **emp_id** have the same domain – the set of valid employee IDs
  - Allow **NULL** manager IDs for employees with no manager

Alternate Foreign Key Syntax

- Can also specify foreign key constraints after all column specifications
  - Required for multi-column foreign keys
- Example:

```
CREATE TABLE employee (
  emp_id      INT,
  emp_ssn     CHAR(9)     NOT NULL,
  emp_name    VARCHAR(40) NOT NULL,
  ...
  manager_id INT,

  PRIMARY KEY (emp_id),
  UNIQUE (emp_ssn),
  FOREIGN KEY (manager_id) REFERENCES employee
);
```

Multi-Column Foreign Keys

- Multi-column foreign keys can also be affected by **`NULL`** values
  - Individual columns may allow **`NULL`** values
- If all values in foreign key are non-**`NULL`** then the foreign key constraint is enforced
- If any value in foreign key is **`NULL`** then the constraint cannot be enforced!
  - Or, "the constraint is defined to hold" (*lame*…)

Example Bank Schema

- Account relation:

```
CREATE TABLE account (
  account_number VARCHAR(15)   NOT NULL,
  branch_name    VARCHAR(15)   NOT NULL,
  balance        NUMERIC(12,2) NOT NULL,
  PRIMARY KEY (account_number)
);
```

- Depositor relation:

```
CREATE TABLE depositor (
  customer_name  VARCHAR(15) NOT NULL,
  account_number VARCHAR(15) NOT NULL,
  PRIMARY KEY (customer_name, account_number),
  FOREIGN KEY (account_number) REFERENCES account,
  FOREIGN KEY (customer_name)  REFERENCES customer
);
```

Foreign Key Violations

- Several ways to violate foreign key constraints
- If referencing relation gets a bad foreign-key value, the operation is simply forbidden
  - e.g. trying to insert a row into *depositor* relation, where the row contains an invalid account number
  - e.g. trying to update a row in *depositor* relation, trying to change customer name to an invalid value
- More subtle issues when the *referenced* relation is changed
  - What to do with *depositor* if a row is deleted from *account*?

Example Bank Data

- *account* data:

```
+-----------------+---------------+-----------+
| account_number  | branch_name   | balance   |
+-----------------+---------------+-----------+
|  ...                                        |
| A-215           | Mianus        |   700.00  |
| A-217           | Brighton      |   750.00  |
| A-222           | Redwood       |   700.00  |
| A-305           | Round Hill    |   350.00  |
|  ...                                        |
+-----------------+---------------+-----------+
```

- *depositor* data:

```
+----------------+----------------+
| customer_name  | account_number |
+----------------+----------------+
|  ...                            |
| Smith          | A-215          |
| Jones          | A-217          |
| Lindsay        | A-222          |
| Turner         | A-305          |
|  ...                            |
+----------------+----------------+
```

Try to delete **`A-222`** from *account*. What should happen?

Foreign Key Violations

- Option 1: Disallow the delete from *account*
  - Force the user to remove all rows in *depositor* relation that refer to A-222
  - Then user may remove row A-222 in *account* relation
  - Default for SQL. Also a pain, but probably a good choice.
- Option 2: <u>Cascade</u> the delete operation
  - If user deletes A-222 from *account* relation, *all* referencing rows in *depositor* should also be deleted
  - Seems reasonable; rows in *depositor* only make sense in context of corresponding rows in *account*

Foreign Key Violations (2)

- Option 3: Set foreign key value to **`NULL`**
  - If primary key goes away, update referencing row to indicate this.
  - Foreign key column can't specify **`NOT NULL`** constraint
  - Doesn't make sense in every situation
    - Doesn't make sense in *account* and *depositor* example!
- Option 4: Set foreign key value to some default
  - Can specify a default value for columns
  - (Haven't talked about how to do this in SQL, yet.)

Cascading Changes

- Can specify behavior on foreign key constraint

```
CREATE TABLE depositor (
  ...
  FOREIGN KEY (account_number) REFERENCES account
                               ON DELETE CASCADE,
  FOREIGN KEY (customer_name)  REFERENCES customer
                               ON DELETE CASCADE
);
```

  - When account A-222 is deleted from *account* relation, corresponding rows in *depositor* will be deleted too
  - Read: "When a row is deleted from referenced relation, corresponding rows are deleted from this relation."
- Similar considerations for updates to primary key values in the referenced relation
  - Can also specify **ON UPDATE** behaviors

Summary

- Integrity constraints are a very powerful feature of the relational model
- SQL provides many ways to specify and enforce constraints
  - Actual support for different kinds of constraints varies among DBMSes
- Allows a database to exclude all invalid values
- Database can also resolve some integrity violations *automatically*
  - e.g. cascade deletion of rows from referencing relations, or setting foreign key values to **`NULL`**

SQL DDL II

CS121: Introduction to Relational Database Systems

Fall 2014 – Lecture 8

Last Lecture

- Covered SQL constraints
  - `NOT NULL` constraints
  - `CHECK` constraints
  - `PRIMARY KEY` constraints
  - `FOREIGN KEY` constraints
  - `UNIQUE` constraints
- Impact of `NULL` values on constraint enforcement
  - Specifically, `FOREIGN KEY` and `UNIQUE`…
- Automatic resolution of constraint violation

Constraint Names

- Can assign names to constraints
  - When constraint is violated, error indicates which constraint
  - Database usually assigns names to constraints if you don't
  - Rules on constraint names vary
- Example:

```
CREATE TABLE employee (
  ...
  CONSTRAINT emp_pk     PRIMARY KEY (emp_id),
  CONSTRAINT emp_ssn_ck UNIQUE (emp_ssn),
  CONSTRAINT emp_mgr_fk FOREIGN KEY (manager_id)
                        REFERENCES employee
```

- Useful for referring to specific constraints

Temporary Constraint Violation

- Constraints take time to enforce
  - Can dramatically impact performance of large data-import operations
- Some operations may need to temporarily violate constraints
  - The operation is performed within a larger transaction (i.e. a batch of operations that should be treated as a unit)
  - During the transaction, constraints are temporarily violated
  - At end of transaction, constraint is restored
- <u>Defer</u> constraint enforcement to end of transaction
  - At end of transaction, all changes are checked against deferred constraints

Deferring Constraint Application

- Can mark constraints as deferrable
- In constraint declaration, specify:
  - `DEFERRABLE` constraints may be deferred to end of transaction
  - `NOT DEFERRABLE` constraints are *always* applied immediately
- For `DEFERRABLE` constraints:
  - `INITIALLY IMMEDIATE` is applied immediately by default
  - `INITIALLY DEFERRED` is applied at end of transaction by default

Temporarily Removing Constraints

- To defer constraints in current transaction:

  SET CONSTRAINTS *c1*, *c2*, ... DEFERRED;

  - Specified constraints must be deferrable
- Not all databases support deferred constraints
  - Only option is to temporarily remove and then reapply constraints
  - Will usually affect all users of database! Safest to ensure exclusive access for this.
  - Remove, then reapply constraints with **ALTER TABLE** syntax

Date and Time Values

- SQL provides data types for dates and times
- **`DATE`**
  - A calendar date, including year, month, and day of month
- **`TIME`**
  - A time of day, including hour, minute, and second value
  - Doesn't include fractional seconds
- **`TIME(P)`**
  - Just like **`TIME`**, but includes P digits of fractional seconds
  - Typically, P = [0, 6]

Date and Time Values (2)

- Can include timezone info as well:
  - **`TIME WITH TIMEZONE`**
  - **`TIME(P) WITH TIMEZONE`**
- **`TIMESTAMP`**
  - A combination of date and time values
  - Includes fractional seconds by default
  - Can also specify **`TIMESTAMP(P)`**
  - P = 6 by default
  - Timestamps can also include time zone info
    - **`TIMESTAMP WITH TIMEZONE`**
    - **`TIMESTAMP(P) WITH TIMEZONE`**

Date and Time Values (3)

- Often a variety of other non-standard types
  - **`DATETIME`** – Like **`TIMESTAMP`** but P = 0 by default
  - **`YEAR`** – Just a 4-digit year value
  - Nonstandard = not portable

Microsoft SQLServer Date Types

- SQLServer 2005 and earlier provide very different date/time support
  - **`DATETIME`** – more like standard **`TIMESTAMP`** type
    - Represents both date and time
    - Jan 1, 1753 – Dec 31, 9999; precision of 3.33ms (???)
  - **`SMALLDATETIME`**
    - Jan 1, 1900 – Jun 6, 2079; precision of 1 minute
  - No ability to represent only a date, or only a time!
- SQLServer 2008 adds more standard-like support
  - **`DATE`**, **`TIME`**, **`DATETIME2`** – similar to standard types
  - **`DATETIMEOFFSET`** – date/time value plus timezone

Date and Time Formats

- Date and time values follow specific formats
  - Enclosed in single-quotes
- Examples: MER-A "Spirit" launch time
  - Timestamp value (UT; +0):
    `'2003-06-10 17:58:46.773'`
  - Date value: `'2003-06-10'`
  - Time value: `'17:58:47'`
- Can have invalid date/time values:
  - Invalid time: `'25:14:68'`
  - Invalid date: `'2001-02-31'`
  - Some DBMSes can allow partial/invalid dates and times, if required by an application

Date and Time Formats (2)

- Most DBMSes support many date/time formats
- Most widely supported is ISO-8601 date/time format
  - ISO-8601 format:
    `'2003-06-10 17:58:46.773'`
    - year-month-day hour:minutes:seconds.milliseconds
    - Sometimes date and time are separated by "T" character
    - Time is in 24-hour time format
    - Optional timezone specification at end
  - Other formats:
    `'June 10, 2003 5:58:46 PM'`
    `'10-Jun-2003 17:58:46.773'`
  - Most databases can parse all of these

"Current Time" Values

- Several functions provide current date and time values

  `CURRENT_DATE()`

  `CURRENT_TIME()`

  `CURRENT_TIMESTAMP()`

  - Include time zone information

  `LOCALTIME()`

  `LOCALTIMESTAMP()`

  - Don't include time zone information
- Usually many other functions too, e.g. **`NOW()`**
  - Nonstandard, but widely supported

Components of Dates and Times

- Date and time values are *not* atomic
  - Not really allowed in the Relational Model…
  - (In reality, many SQL types are not atomic)
- SQL provides a function to extract components of dates and times
  - **`EXTRACT (`*`field`*` FROM `*`value`*`)`**
  - Can specify:
    - **`YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE`, `SECOND`**
    - **`TIMZEONE_HOUR`, `TIMEZONE_MINUTE`**
  - Many other (nonstandard but common) options too
    - week of year, day of year, day of week, quarter, century, …

Example Date Operation

- Sales records:

```
CREATE TABLE salesrecords (
  sale_id INTEGER PRIMARY KEY,
  cust_id INTEGER NOT NULL,
  sale_time TIMESTAMP NOT NULL,
  sales_total NUMERIC(8, 2) NOT NULL,
  ...
);
```

- Compute monthly sales totals:
  - Start by finding month of each sale

```
SELECT sale_id,
  EXTRACT (MONTH FROM sale_time) AS sale_month
  FROM salesrecords;
```

  - Build larger query using this information

Time Intervals

- `INTERVAL`
  - Data type for time intervals
  - Supports operations on dates and times
  - Also supports a precision: `INTERVAL(P)`
- If x and y are date values:

  $x - y$ produces an `INTERVAL`
- If i is an `INTERVAL` value:

  $x + i$ or $x - i$ produces a date value
- Can use `INTERVAL` to specify fixed intervals
  - `INTERVAL 1 WEEK`
  - `INTERVAL '1 WEEK'`

Example Date Schema

- Event database schema:

```
CREATE TABLE event (
  event_id   INTEGER       PRIMARY KEY,
  event_type VARCHAR(20)   NOT NULL,
  event_date DATE          NOT NULL,
  event_desc VARCHAR(200)
);
```

- To generate notices of upcoming events:

```
SELECT * FROM event
 WHERE event_date >= CURRENT_DATE() AND
       event_date <=
           (CURRENT_DATE() + INTERVAL 1 WEEK);
```

Example Date Schema (2)

- Can rewrite to use **BETWEEN** syntax:

```
SELECT * FROM event
 WHERE event_date BETWEEN
     CURRENT_DATE() AND
     (CURRENT_DATE() + INTERVAL 1 WEEK);
```

- Current date/time functions are evaluated only once during a query! ☺
  - e.g. query will see one value for **CURRENT_TIME()** even if it runs for an extended period of time

“Large Object” Types

- SQL **CHAR(N)** and **VARCHAR(N)** types have limited sizes
  - For **CHAR**, usually N < 256
  - For **VARCHAR**, usually N < 65536
- **BLOB** and **CLOB** types support larger data sizes
  - “LOB” = Large Object
  - Useful for storing images, documents, etc.
  - Support varies widely across DBMSes
  - **TEXT** is also rather common
    - Large text fields, e.g. MB or GB of text data

Example Schema

- Schema for storing book reviews:

```
CREATE TABLE bookreview (
  review_id   INT PRIMARY KEY,
  book_title  VARCHAR(50) NOT NULL,
  book_image  BLOB,
  reviewer    VARCHAR(30) NOT NULL,
  pub_time    TIMESTAMP NOT NULL,
  review_text CLOB NOT NULL,
  UNIQUE (book_title, reviewer)
);
```

- Review text can be large
- Can also include a book image, if desired

Large Object Notes

- General support for "large object" types is usually focused on smaller objects
  - No larger than a few 10s of KBs
  - A few MBs is *definitely* pushing it
- Most expensive part is moving large objects into and out of database
  - For simple, general purpose DBMSes, can involve constructing large SQL statements with escaped data
- Databases also don't store this information very efficiently

Large Object Notes (2)

- For objects larger than ~100 KB, should definitely use the filesystem
  - That's what it's designed for!
  - Store *filesystem paths* in the database instead
- For smaller objects that are frequently retrieved, storing on filesystem can take load off database
  - e.g. user icons for a social networking website
  - Let webserver serve them directly from the filesystem – again, it knows how to do that kind of thing more quickly
- Some DBMSes have specialized support for storing and manipulating very large objects
  - Just don't expect your application to be easily portable…

Default Values

- Can specify default values for columns
    - ***`colname type`*** **`DEFAULT`** ***`expr`***
  - Can specify an actual value
    - **`book_rating INT DEFAULT 3`**
  - Can specify an expression
    - **`pub_time TIMESTAMP DEFAULT NOW()`**
- If unspecified, default value is **`NULL`**
- Affects **`INSERT`** statements
  - Columns with default values don't have to be specified
  - Columns without a default value *must* be specified at insert-time!

Serial Primary Key Values

- Many databases offer special support for integer primary keys
  - DB will generate unique values for use as primary keys
- Examples:
  - PostgreSQL and MySQL:

```
CREATE TABLE employee (
  emp_id    SERIAL PRIMARY KEY,
  ...
```

  - Microsoft SQLServer:

```
CREATE TABLE employee (
  emp_id    INT IDENTITY PRIMARY KEY,
  ...
```

Updated Book Review Schema


```
CREATE TABLE bookreview (
  review_id   SERIAL        PRIMARY KEY,
  book_title  VARCHAR(50)  NOT NULL,
  book_image  BLOB,
  reviewer    VARCHAR(30)  NOT NULL,
  pub_time    TIMESTAMP    NOT NULL DEFAULT NOW(),
  book_rating INT          NOT NULL DEFAULT 3,
  review_text CLOB         NOT NULL,
  UNIQUE (book_title, reviewer)
);
```

- Every new review gets a unique ID value
- Publication time is set to current time when review is added to database
- Default book rating is 3 out of 5

Altering Table Schemas

- SQL **`ALTER TABLE`** command allows schema changes
- Wide variety of operations
  - Rename a table
  - Add and remove constraints
  - Add and remove table columns
  - Change the type of a column
  - Change default values for columns
- Very useful for migrating schema to new version
  - Migration process must be carefully designed…
- Again, support varies across DBMSes

Example Alterations

- Rename the *bookreview* table:

```
ALTER TABLE bookreview
  RENAME TO item_review;
```

- Remove the book image column:

```
ALTER TABLE bookreview
  DROP COLUMN book_image;
```

- Add a constraint to the *bookreview* table:

```
ALTER TABLE bookreview
  ADD CHECK (book_rating BETWEEN 1 AND 5);
```

Table Alteration Notes

- Can drop columns from tables
  - What if the column is a key?
  - What if the column is referenced by a view?
  - Can often specify **`CASCADE`** to delete dependent objects, if desired
- Newly added columns <u>must</u> have a default value
  - Existing rows in database get default value for new column
- Changing table schema can be *very* expensive
  - Some operations can require scanning or rewriting the entire table
    - Some DBs do this for all schema-alteration commands, e.g. MySQL
  - e.g. adding a new constraint requires a table scan

Temporary Tables

- Sometimes want to generate and store relations temporarily
  - Complex operations implemented as multiple queries
  - This is relational algebra assignment operation: $\leftarrow$
- SQL provides <u>temporary tables</u> for these cases
  - Table's contents are associated with client's session
  - Clients can't access each others' temp table data
- SQL standard specifies <u>global</u> temporary tables
  - Temporary table has a global name and schema
  - Only the <u>contents</u> of the temporary table are per-client
  - When client disconnects, their temporary data is purged

Temporary Tables (2)

- Many databases also provide local temporary tables
  - Table's schema is also local to client session
  - When client disconnects, the table is dropped
  - Different clients can use same table name with different schemas
- Client can manually purge data from temp tables when needed
  - In case of local temp tables, can also drop them anytime during session

Temporary Table Syntax

- Simple variation of **`CREATE TABLE`** syntax
  - Add **`TEMPORARY`** (or **`GLOBAL TEMPORARY`**) to command
- Example:
  - Make a temporary table to store counts of sales grouped by month

```
CREATE TEMPORARY TABLE salesbymonth (
  sale_month INT NOT NULL,
  num_sales  INT NOT NULL
);
```

Temporary Table Example

- Can populate temp table with computed values

```
INSERT INTO salesbymonth
  SELECT EXTRACT (MONTH FROM sale_time) AS mon,
         COUNT(*)
    FROM salesrecords GROUP BY mon;
```

  - Only need to perform computations once
  - Improves efficiency of large or multi-step operations
  - Temporary results are cleaned up at end of session
- Issue queries against temporary table and use results

```
SELECT sale_month, num_sales, promotion_desc
 FROM salesbymonth
      JOIN promotions USING (sale_month);
```

Temporary Table Contents

- When to flush temporary table contents?
- Two main options:
  - At end of current transaction
  - When entire client session ends
- Can specify behavior with **`ON COMMIT`** clause at end of table declaration
  - To flush temp table at end of each transaction:

    `ON COMMIT DELETE ROWS`
  - To flush temp table at end of session:

    `ON COMMIT PRESERVE ROWS`
  - SQL standard specifies default is **`DELETE ROWS`** !
    - Not all DBMSes follow this, but some do!

Example ON COMMIT Clauses

- To flush rows after each transaction:

```
CREATE TEMPORARY TABLE salesbymonth (
  sale_id    INT NOT NULL,
  sale_month INT NOT NULL
) ON COMMIT DELETE ROWS;
```

- To keep rows until end of session:

```
CREATE TEMPORARY TABLE salesbymonth (
  sale_id    INT NOT NULL,
  sale_month INT NOT NULL
) ON COMMIT PRESERVE ROWS;
```

Using Temporary Tables

- Temporary tables can dramatically improve performance of certain queries
- Approach:
  - Create temporary table to store useful but costly intermediate results
    - Don't use many (or any) constraints – want to be fast!
  - Populate temporary table via **`INSERT ... SELECT`** statement
  - Use temporary table to compute other results
  - Temporary table goes away automatically, at end of transaction, or at end of session

Alternate Temp-Table Syntaxes

- Databases frequently support alternate syntaxes for creating and populating temporary tables
  - Simplify the common case!
- One common syntax (e.g. MySQL, Postgres, Oracle):

```
CREATE TEMPORARY TABLE tblname AS
  select_stmt;
```

- Another common syntax (e.g. Postgres, SQLServer):

```
SELECT ... INTO TEMPORARY TABLE ...;
```

- Both syntaxes can also create non-temporary tables

Real-World Example

- A query run on a MySQL server:

```
SELECT ident, total_a / total_b AS ratio
FROM (SELECT CONCAT(a1, a2) AS ident,
             SUM(val_a) AS total_a
      FROM t1 GROUP BY ident) AS result1,
     (SELECT CONCAT(a1, a2) AS ident,
             SUM(val_b) AS total_b
      FROM t2 GROUP BY ident) AS result2
WHERE result1.ident = result2.ident;
```

- Overall query takes ~15 mins to execute on fast server
- Inner queries complete in << 1 second by themselves

Real-World Example (2)

- MySQL query:

```
SELECT ident, total_a / total_b AS ratio
FROM (SELECT CONCAT(a1, a2) AS ident,
             SUM(val_a) AS total_a
      FROM t1 GROUP BY ident) AS result1,
     (SELECT CONCAT(a1, a2) AS ident,
             SUM(val_b) AS total_b
      FROM t2 GROUP BY ident) AS result2
WHERE result1.ident = result2.ident;
```

- Problem is that MySQL cannot *efficiently* join two derived results using a computed column
  - A limitation of MySQL's join processor ☹

Real-World Example (3)

- A solution:
  - First, create temporary tables to hold intermediate results

```
CREATE TEMPORARY TABLE temp1 AS
  SELECT CONCAT(a1, a2) AS ident,
         SUM(val_a) AS total_a
  FROM t1 GROUP BY ident;
```

    - …same with other inner query…
  - Second, create indexes on temporary tables
  - Finally, issue outer query against temporary tables
- Result:
  - <u>Entire process</u>, including create/drop temp tables, takes < 1 second (as opposed to ~15 minutes)

SQL STORED ROUTINES

CS121: Introduction to Relational Database Systems

Fall 2014 – Lecture 9

SQL Functions

- SQL queries can use sophisticated math operations and functions
  - Can compute simple functions, aggregates
  - Can compute and filter results
- Sometimes, apps require specialized computations
  - Would like to use these in SQL queries, too
- SQL provides a mechanism for defining functions
  - Called User-Defined Functions (UDFs)

SQL Functions (2)

- Can be defined in a procedural SQL language, or in an external language
  - SQL:1999, SQL:2003 both specify a language for declaring functions and procedures
- Different vendors provide their own languages
  - Oracle: PL/SQL
  - Microsoft: Transact-SQL (T-SQL)
  - PostgreSQL: PL/pgSQL
  - MySQL: stored procedure support strives to follow specifications (and mostly does)
  - Some also support external languages: Java, C, C#, etc.
- As usual, lots of variation in features and syntax

Example SQL Function

- A SQL function to count how many bank accounts a particular customer has:

```
CREATE FUNCTION account_count(
    customer_name VARCHAR(20)
) RETURNS INTEGER
BEGIN
  DECLARE a_count INTEGER;

  SELECT COUNT(*) INTO a_count FROM depositor AS d
  WHERE d.customer_name = customer_name;

  RETURN a_count;
END
```

  - Function can take arguments and return values
  - Can use SQL statements and other operations in body

Example SQL Function (2)

- Can use our function for individual accounts:

```
SELECT account_count('Johnson');
```

- Can include in computed results:

```
SELECT customer_name,
       account_count(customer_name) AS accts
FROM customer;
```

- Can include in **WHERE** clause:

```
SELECT customer_name FROM customer
WHERE account_count(customer_name) > 1;
```

Arguments and Return-Values

- Functions can take any number of arguments (even 0)
- Functions *must* return a value
  - Specify type of value in **RETURNS** clause
- From our example:

```
CREATE FUNCTION account_count(
    customer_name VARCHAR(20)
) RETURNS INTEGER
```

  - One argument named **customer_name**, type is **VARCHAR(20)**
  - Returns some **INTEGER** value

Table Functions

- SQL:2003 spec. includes <u>table functions</u>
  - Return a whole table as their result
  - Can be used in **`FROM`** clause
- A generalization of views
  - Can be considered to be parameterized views
  - Call function with specific arguments
  - Result is a relation based on those arguments
- Although SQL:2003 not broadly supported yet, most DBMSes provide a feature like this
  - *…in various ways, of course…*

Function Bodies and Variables

- Blocks of procedural SQL commands are enclosed with **BEGIN** and **END**
  - Defines a compound statement
  - Can have nested **BEGIN** … **END** blocks
- Variables are specified with **DECLARE** statement
  - Must appear at start of a block
  - Initial value is **NULL**
  - Can initialize to some other value with **DEFAULT** syntax
  - Scope of a variable is within its block
  - Variables in inner blocks can shadow variables in outer blocks

Example Blocks and Variables

- Our **account_count** function's body:

```
BEGIN
  DECLARE a_count INTEGER;

  SELECT COUNT(*) INTO a_count FROM depositor AS d
  WHERE d.customer_name = customer_name;

  RETURN a_count;
END
```

- A simple integer variable with initial value:

```
BEGIN
  DECLARE result INTEGER DEFAULT 0;
  ...
END
```

Assigning To Variables

- Can use **SELECT … INTO** syntax
  - For assigning the result of a query into a variable

```
SELECT COUNT(*) INTO a_count
FROM depositor AS d
WHERE d.customer_name = customer_name;
```

  - Query must produce a single row

    Note: **SELECT INTO** sometimes has multiple meanings! This form is specific to the body of stored routines.
    - e.g. frequently used to create a temp table from a **SELECT**
- Can also use **SET** syntax
  - For assigning result of a math expression to a variable

```
SET result = n * (n + 1) / 2;
```

Assigning Multiple Variables

- Can assign to multiple variables using **SELECT INTO** syntax
- Example: Want both the number of accounts and the total balance

```
DECLARE a_count INTEGER;
DECLARE total_balance NUMERIC(12,2);

SELECT COUNT(*), SUM(balance)
INTO a_count, total_balance
FROM depositor AS d NATURAL JOIN account
WHERE d.customer_name = customer_name;
```

Another Example

- Simple function to compute sum of 1..N

```
CREATE FUNCTION sum_n(n INTEGER) RETURNS INTEGER
BEGIN
    DECLARE result INTEGER DEFAULT 0;
    SET result = n * (n + 1) / 2;
    RETURN result;
END
```

- Lots of extra work in that! To simplify:

```
CREATE FUNCTION sum_n(n INTEGER) RETURNS INTEGER
BEGIN
    RETURN n * (n + 1) / 2;
END
```

Dropping Functions

- Can't simply overwrite functions in the database
  - Same as tables, views, etc.
- First, drop old version of function:

```
DROP FUNCTION sum_n;
```

- Then create new version of function:

```
CREATE FUNCTION sum_n(n INTEGER)
RETURNS INTEGER
BEGIN
    RETURN n * (n + 1) / 2;
END
```

SQL Procedures

- Functions have specific limitations
  - Must return a value
  - All arguments are input-only
  - Typically cannot affect current transaction status (i.e. function cannot commit, rollback, etc.)
  - Usually not allowed to modify tables, except in particular circumstances
- Stored procedures are more general constructs without these limitations
  - Generally can't be used in same places as functions
  - e.g. can't use in **`SELECT`** clause
  - Procedures don't return a value like functions do

Example Procedure

- Write a procedure that returns both the number of accounts a customer has, and their total balance
  - Results are passed back using out-parameters

```
CREATE PROCEDURE account_summary(
    IN  customer_name VARCHAR(20),
    OUT a_count INTEGER,
    OUT total_balance NUMERIC(12,2)
)
BEGIN
    SELECT COUNT(*), SUM(balance)
    INTO a_count, total_balance
    FROM depositor AS d NATURAL JOIN account
    WHERE d.customer_name = customer_name;
END
```

- Default parameter type is **IN**

Calling a Procedure

- Use the **CALL** statement to invoke a procedure

```
CALL account_summary(...);
```

- To use this procedure, must also have variables to receive the values
- MySQL SQL syntax:

```
CALL account_summary('Johnson',
                     @j_count, @j_total);
SELECT @j_count, @j_total;
```

  - **@var** declares a temporary session variable

```
+---------+----------+
| @j_cnt  | @j_tot   |
+---------+----------+
| 2       | 1400.00  |
+---------+----------+
```

Conditional Operations

- SQL provides an if-then-else construct

```
IF cond₁ THEN command₁
ELSEIF cond₂ THEN command₂
ELSE command₃
END IF
```

  - Branches can also specify compound statements instead of single statements
    - Enclose compound statements with **BEGIN** and **END**
  - Can leave out **ELSEIF** and/or **ELSE** clauses, as usual

Looping Constructs

- SQL also provides looping constructs
- **WHILE** loop:

```
DECLARE n INTEGER DEFAULT 0;
WHILE n < 10 DO
    SET n = n + 1;
END WHILE;
```

- **REPEAT** loop:

```
REPEAT
    SET n = n - 1;
UNTIL n = 0
END REPEAT;
```

Iteration Over Query Results

- Sometimes need to issue a query, then iterate over each row in result
  - Perform more sophisticated operations than a simple SQL query can perform
- Examples:
  - Many kinds of values that standard OLTP databases can't compute quickly!
  - Assign a dense rank to a collection of rows:
    - Can compare each row to all other rows, typically with a cross-join
    - Or, sort rows then iterate over results, assigning rank values
  - Given web logs containing individual HTTP request records:
    - Compute each client's "visit length," from requests that are within 20 minutes of some other request from the same client

Cursors

- Need to issue a query to fetch specific results
- Then, need to iterate through each row in the result
  - Operate on each row's values individually
- A <u>cursor</u> is an iterator over rows in a result set
  - Cursor refers to one row in query results
  - Can access row's values through the cursor
  - Can move cursor forward through results
- Cursors can provide different features
  - Read-only vs. read-write
  - Forward-only vs. bidirectional
  - Static vs. dynamic (when concurrent changes occur)

Cursor Notes

- Cursors can be expensive
- Can the operation use a normal SQL query instead?
  - (Usually, the answer is yes…)
  - Cursors let you do what databases do, but <u>slower</u>
- Cursors might also hold system resources until they are finished
  - e.g. DB might store query results in a temporary table, to provide a read-only, static view of query result
- Syntax varies widely across DBMSes
- Most external DB connectivity APIs provide cursor capabilities

Stored Procedures and Cursors

- Can use cursors inside stored procedures
- Syntax from the book:

```
DECLARE n INTEGER DEFAULT 0;
FOR r AS SELECT balance FROM account
         WHERE branch_name='Perryridge'
DO
    SET n = n + r.balance;
END FOR
```

  - Iterates over account balances from Perryridge branch, summing balances
  - `r` is implicitly a cursor
    - `FOR` construct automatically moves the cursor forward
  - (Could compute this with a simple SQL query, too…)

MySQL Cursor Syntax

- Must *explicitly* declare cursor variable

```
DECLARE cur CURSOR FOR
    SELECT ... ;
```

- Open cursor to use query results:

```
OPEN cur;
```

- Fetch values from cursor into variables

```
FETCH cur INTO var1, var2, ... ;
```

  - Next row is fetched, and values are stored into specified variables
  - Must specify the same number of variables as columns in the result
  - A specific error condition is flagged to indicate end of results
- Close cursor at end of operation

```
CLOSE cur;
```

  - Also happens automatically at end of enclosing block

Handling Errors

- Many situations where errors can occur in stored procedures
  - Called <u>conditions</u>
  - Includes errors, warnings, other signals
  - Can also include user-defined conditions
- Handlers can be defined for conditions
- When a condition is signaled, its handler is invoked
  - Handler can specify whether to continue running the procedure, or whether to exit procedure instead

Conditions

- Predefined conditions:
  - **`NOT FOUND`**
    - Query fetched no results, or command processed no results
  - **`SQLWARNING`**
    - Non-fatal SQL problem occurred
  - **`SQLEXCEPTION`**
    - Serious SQL error occurred

Conditions (2)

- Can also define application-specific conditions
  - Examples:
    - "Account overdraft!"
    - "Inventory of item hit zero."
- Syntax for declaring conditions:

```
DECLARE acct_overdraft CONDITION
DECLARE zero_inventory CONDITION
```

- Not every DBMS supports generic conditions
  - e.g. MySQL supports assigning names to <u>existing</u> SQL error codes, but not creating new conditions

Handlers

- Can declare handlers for specific conditions
- Handler specifies statements to execute
- Handler also specifies what should happen next:
  - Continue running the procedure where it left off
  - Exit the stored procedure completely
- Syntax:
  - A continue-handler:

    ```
    DECLARE CONTINUE HANDLER FOR condition statement
    ```
  - An exit-handler:

    ```
    DECLARE EXIT HANDLER FOR condition statement
    ```
  - Can also specify a statement-block instead of an individual statement

Handlers (2)

- Handlers can do very simple things
  - e.g. set a flag to indicate some situation
- Can also do very complicated things
  - e.g. insert rows into other tables to log failure situations
  - e.g. properly handle an overdrawn account

Total Account Balance – MySQL

- Declared as a function – returns a value

```
CREATE FUNCTION acct_total(cust_name VARCHAR(20))
RETURNS NUMERIC(12,2)
BEGIN
    -- Variables to accumulate into
    DECLARE bal NUMERIC(12,2);
    DECLARE total NUMERIC(12,2) DEFAULT 0;

    -- Cursor, and flag for when fetching is done
    DECLARE done INT DEFAULT 0;
    DECLARE cur CURSOR FOR
        SELECT balance
        FROM account NATURAL JOIN depositor AS d
        WHERE d.customer_name = cust_name;
```

Total Account Balance (2)


```
    -- When fetch is complete, handler sets flag
    -- 02000 is MySQL error for "zero rows fetched"
    DECLARE CONTINUE HANDLER FOR SQLSTATE '02000'
        SET done = 1;

    OPEN cur;
    REPEAT
        FETCH cur INTO bal;
        IF NOT done THEN
            SET total = total + bal;
        END IF;
    UNTIL done END REPEAT;
    CLOSE cur;
    RETURN total;
END
```

Using Our Stored Procedure

- Can compute total balances now:

```
SELECT customer_name,
       acct_total(customer_name) AS total
FROM customer;
```

  - Result:

```
+---------------+---------+
| customer_name | total   |
+---------------+---------+
| Adams         |    0.00 |
| Brooks        |    0.00 |
| Curry         |    0.00 |
| Glenn         |    0.00 |
| Green         |    0.00 |
| Hayes         |  900.00 |
| Jackson       |    0.00 |
| Johnson       | 1400.00 |
| Jones         |  750.00 |
| Lindsay       |  700.00 |
| Majeris       |  850.00 |
| McBride       |    0.00 |
| Smith         | 1325.00 |
| Turner        |  350.00 |
| Williams      |    0.00 |
+---------------+---------+
```

Stored Procedure Benefits

- Very effective for manipulating large datasets in unusual ways, within the database
  - Don’t incur communications overhead of sending commands and exchanging data
  - Database can frequently perform such tasks more efficiently than the applications can
- Often used to provide a secure interface to data
  - e.g. banks will lock down data tables, and only expose certain operations through stored procedures
- Can encapsulate business logic in procedures
  - Forbid invalid states by requiring all operations go through stored procedures

Stored Procedure Drawbacks

- Increases load on database system
  - Can reduce performance for *all* operations being performed by DBMS
  - Need to make sure the operation *really* requires a stored procedure…
    - Most projects do not need stored procedures!
- Very hard to migrate to a different DBMS
  - Different vendors' procedural languages have *many* distinct features and limitations

ADVANCED SQL DDL

CS121: Introduction to Relational Database Systems

Fall 2014 – Lecture 10

Advanced SQL DDL

- Last time, covered stored procedures and user-defined functions (UDFs)
  - Relatively simple but powerful mechanism for extending capabilities of a database
  - Most databases support these features (in different ways, of course…)
- Today, will cover three more advanced features of SQL data definition
  - Triggers
  - Materialized views (briefly)
  - Security constraints in databases

Triggers

- Triggers are procedural statements executed automatically when a database is modified
  - Usually specified in procedural SQL language, but other languages are frequently supported
- Example: an audit log for bank accounts
  - Every time a balance is changed, a trigger can update an "audit log" table, storing details of the change
    - e.g. old value, new value, who changed the balance, and why
- Why not have applications update the log directly?
  - Could easily forget to update audit log for some updates!
  - Or, a malicious developer might leave a back-door in an application, allowing them to perform unaudited operations

Triggers (2)

- If the database handles audit-log updates automatically and independently:
  - Application code doesn't become more complex by introducing audit functionality
  - Audit log will be a more trustworthy record of modifications to bank account records
- Triggers are used for many other purposes, such as:
  - Preventing invalid changes to table data
  - Automatically updating timestamp values, derived attributes, etc.
  - Executing business rules when data changes in specific ways
    - e.g. place an order for more parts when current inventory dips below a specific value
  - Replicating changes to another table, or even another database

Trigger Mechanism

- DB trigger mechanism must keep track of two things:
- When is the trigger actually executed?
  - The <u>event</u> that causes the trigger to be considered
  - The <u>condition</u> that must be satisfied before the trigger will execute
    - (Not every database requires a condition on triggers…)
- What does the trigger do when it's executed?
  - The <u>actions</u> performed when the trigger executes
- Called the <u>event-condition-action</u> model for triggers

When Triggers Execute

- Databases usually support triggering on inserts, updates, and deletes
- Can't trigger on selects
  - Implication: Can't use triggers to audit or prevent read-accesses to a database (bummer)
- Commercial databases also support triggering on many other operations
  - Data-definition operations (create/alter/drop table, etc.)
  - Login/logout of specific users
  - Database startup, shutdown, errors, etc.
- For simplicity, will limit discussion to DML triggers only

When Triggers Execute

- Can typically execute the trigger <u>before</u> or <u>after</u> the triggering DML event
  - Usually, DDL/user/database triggering events only run the trigger *after* the event (pretty obvious)
  - "Before" triggers can abort the DML operation, if necessary
- Some DBs also support "instead of" triggers
  - Execute trigger instead of performing the triggering operation
- Triggers are row-level triggers or statement-level triggers
  - A <u>row-level trigger</u> is executed for every single row that is modified by the statement
    - *(…as long as the row satisfies the trigger condition, if specified…)*
  - A <u>statement-level trigger</u> is executed once for the entire statement

Trigger Data

- Row-level triggers can access the old and new version of the row data, when available:
  - Insert triggers only get the new row data
  - Update triggers get both the old and new row data
  - Delete triggers only get the old row data
- Triggers can also access and modify other tables
  - e.g. to look up or record values during execution

Trigger Syntax

- SQL:1999 specifies a syntax for triggers
  - Discussed in the textbook, section 5.3
- Again, <u>wide</u> variation from vendor to vendor
  - Oracle and DB2 are similar to SQL99, but not identical
    - (triggers always seem to involve vendor-specific features)
  - SQLServer, Postgres, MySQL all have different features
  - Constraints on what triggers can do also vary widely from vendor to vendor
- Will focus on MySQL trigger syntax, functionality

Trigger Example: Bank Overdrafts

- Want to handle overdrafts on bank accounts
- If an update causes a balance to go negative:
  - Create a new loan with same ID as the account number
  - Set the loan balance to the negative account balance
    - *(…the account balance went negative…)*
  - Need to update **`borrower`** table as well!
- Needs to be a row-level trigger, executed before updates to the **`account`** table
  - If database supports trigger conditions, only trigger on updates when account balance < 0

SQL99/Oracle Trigger Syntax

- Book uses SQL:1999 syntax, similar to Oracle/DB2

```
CREATE TRIGGER trg_overdraft AFTER UPDATE ON account
REFERENCING NEW ROW AS nrow
FOR EACH ROW WHEN nrow.balance < 0
BEGIN ATOMIC
    INSERT INTO loan VALUES (nrow.account_number,
                             nrow.branch_name,
                             -nrow.balance);

    INSERT INTO borrower
        (SELECT customer_name, account_number
         FROM depositor AS d
         WHERE nrow.account_number = d.account_number);

    UPDATE account AS a SET balance = 0
        WHERE a.account_number = nrow.account_number;
END
```

MySQL Trigger Syntax

- MySQL has more limited trigger capabilities
  - Trigger execution is only governed by events, not conditions
    - Workaround: Enforce the condition within the trigger body
  - Old and new rows have fixed names: **`OLD`**, **`NEW`**
- Change the overdraft example slightly:
  - Also apply an overdraft fee! *"Kick 'em while they're down!"*
  - What if the account is already overdrawn?
    - Loan table would already contain a record for the overdrawn account…
    - Borrower table would already contain records for the loan, too!
    - Previous version of trigger would cause a duplicate key error!

MySQL **INSERT** Enhancements

- MySQL has several enhancement to the **INSERT** command
  - (Most databases provide similar capabilities)
- Try to insert a row, but if key attributes are same as another row, simply don't perform the insert:

```
INSERT IGNORE INTO tbl ...;
```

- Try to insert a row, but if key attributes are same as another row, update the existing row:

```
INSERT INTO tbl ... ON DUPLICATE KEY
    UPDATE attr1 = value1, ...;
```

- Try to insert a row, but if key attributes are same as another row, replace the old row with the new row
  - If key is not same as another row, perform a normal **INSERT**

```
REPLACE INTO tbl ...;
```

MySQL Trigger Syntax (2)


```
CREATE TRIGGER trg_overdraft BEFORE UPDATE ON account FOR EACH ROW
BEGIN
    DECLARE overdraft_fee NUMERIC(12, 2) DEFAULT 30;
    DECLARE overdraft_amt NUMERIC(12, 2);

    -- If an overdraft occurred then handle by creating/updating a loan.
    IF NEW.balance < 0 THEN
        -- Remember that NEW.balance is negative.
        SET overdraft_amt = overdraft_fee - NEW.balance;

        INSERT INTO loan (loan_number, branch_name, amount)
            VALUES (NEW.account_number, NEW.branch_name, overdraft_amt)
        ON DUPLICATE KEY UPDATE amount = amount + overdraft_amt;

        INSERT IGNORE INTO borrower (customer_name, loan_number)
            SELECT customer_name, account_number FROM depositor
            WHERE depositor.account_number = NEW.account_number;

        SET NEW.balance = 0;
    END IF;
END;
```

Trigger Pitfalls

- Triggers may *or may not* execute when you expect…
  - e.g. MySQL insert-triggers fire when data is bulk-loaded into the DB from a backup file
    - Databases usually allow you to temporarily disable triggers
  - e.g. truncating a table usually <u>does not</u> fire delete-triggers
- If a trigger for a commonly performed task runs slowly, it will <u>kill</u> DB performance
- If a trigger has a bug in it, it may abort changes to tables at unexpected times
  - The *actual* cause of the issue may be difficult to discern
- Triggers can write to other tables, which may also have triggers on them…
  - Not hard to create an infinite chain of triggering events

Alternatives to Triggers

- Triggers can be used to implement *many* complex tasks
- Example: Can implement referential integrity with triggers!
  - On all inserts and updates to referencing table, ensure that foreign-key column value appears in referenced table
    - If not, abort the operation!
  - On all updates and deletes to referenced table, ensure that value doesn't appear in referencing table
    - If it does, can abort the operation, or cascade changes to the referencing relation, etc.
- This is definitely slower than the standard mechanism ☺

Alternatives to Triggers (2)

- Can you use stored procedures instead?
  - Stored procedures usually have fewer limitations than triggers
    - Stored procs can take more detailed arguments, return values to indicate success/failure, have out-params, etc.
    - Can perform more sophisticated transaction processing
  - Trigger support is also very vendor-specific, so either implementation choice will have this limitation
- Typically, triggers are used in very limited ways
  - Update "row version" or "last modified timestamp" values in modified rows
  - Simple operations that don't require a great deal of logic
  - Database replication (sometimes)

Triggers and Summary Tables

- Triggers are sometimes used to compute summary results when detail records are changed
- Example: a table of branch summary values
  - e.g. (*<u>branch_name</u>*, *total_balances*, *total_loans*)
- Motivation:
  - If these values are used frequently in queries, want to avoid overhead of recomputing them all the time
- Idea: update this summary table with triggers
  - Anytime changes are made to **`account`** or **`loan`**, update the summary table based on the changes

Materialized Views

- Some databases provide materialized views, which implement such functionality
- Simple views usually treated as named SQL queries
  - i.e. a derived relation with the specified definition
- When a query refers to a simple view, database substitutes view's definition directly into the query
  - Benefit: allows optimization of the entire query
  - Drawback: if many queries reference a simple view, the same values will be computed again and again…

Materialized Views (2)

- Materialized views actually create a new table, populated by the results of the view definition
  - Queries can use values in the materialized view over and over, without recomputing
  - Database can perform optimized lookups against the materialized view, e.g. by using indexes
- Just one little problem:
  - What if the tables referenced by the view change?
  - Need to recompute contents of the materialized view!
  - Called <u>view maintenance</u>

Materialized View Maintenance

- If a database doesn't support materialized views:
  - Can perform view maintenance with triggers on the referenced tables
  - A very manual approach, but definitely an option for databases that don't support materialized views
    - e.g. Postgres, MySQL
- Databases with materialized views will perform view maintenance automatically
  - …*much* simpler than creating a bunch of triggers!
  - Typically provide many options, such as:
    - Immediate view maintenance – update contents after any change
    - Deferred view maintenance – update view on a periodic schedule

Materialized View Maintenance (2)

- A simple approach for updating materialized views:
  - Recompute entire view from scratch after every change!
  - Very expensive approach, especially if backing tables are changed frequently
- A better approach: <u>incremental</u> view maintenance
  - Using the view definition and the specific data changes applied to the backing tables, only update those parts of the view that are actually affected
- Again, DBs with materialized views will do this for you
- Can also do incremental view maintenance manually with triggers, but it can be complicated…

Authentication and Authorization

- Security systems must provide two major features
- Authentication (aka “A1”, “AuthN”, “Au”):
  - “I am who I say I am.”
- Authorization (aka “A2”, “AuthZ”, “Az”):
  - “I am allowed to do what I want to do.”
- Each component is useless without the other

User Authorization

- SQL databases perform authentication of users
  - Must specify username and password when connecting
  - Most DBMSes provide secure connections (e.g. SSL), etc.
- SQL provides an authorization mechanism for various operations
  - Different operations require different privileges in the database
  - Users can be granted privileges to perform necessary operations
  - Privileges can also be revoked, to limit available user operations

Basic SQL Privileges

- Most fundamental set of privileges:
  - **SELECT, INSERT, UPDATE, DELETE**
  - Allows (or disallows) user to perform specified action
  - User is granted access to perform specified operations on particular relations
- Simple syntax:

```
GRANT SELECT ON account TO banker;
```

  - User "banker" is allowed to issue queries against the *account* relation

Granting Privileges

- Can grant multiple privileges to multiple users

```
GRANT SELECT, UPDATE ON account
  TO banker, manager;
GRANT INSERT, DELETE ON account
  TO manager;
```

  - Bankers can view and modify account balances
  - Only managers can create or remove accounts
  - Must specify each table individually

All Users, All Privileges

- Can specify **PUBLIC** to grant privileges to all users
  - Also includes users added to DBMS in future

```
GRANT SELECT ON promotions TO PUBLIC;
```

- Can specify **ALL PRIVILEGES** to grant all privileges to a user

```
GRANT ALL PRIVILEGES ON account
  TO admin_lackey;
```

Column-Level Privileges

- For **INSERT** and **UPDATE** privileges, can constrain to specific *columns* of relations
  - **UPDATE**: can only update specified columns
  - **INSERT**: can only insert into specified columns
- Example: *employee* relation
  - Employees can only modify their contact info
  - Allow HR to manipulate *all* aspects of employees

```
GRANT UPDATE (home_phone, email) ON employee
  TO emp_user;
GRANT INSERT, UPDATE ON employee TO hr_user;
```

Revoking Privileges

- Can revoke privileges just as easily:

```
REVOKE priv1, ... ON relation
  FROM user1, ...;
```

  - Can specify a list of privileges, and a list of users
- With **INSERT** and **UPDATE**, can also revoke privileges on individual columns

Privileges and Views

- Users can be granted privileges on views
  - May differ from privileges on underlying tables
- When accessing a view:
  - Privileges on the *view* are checked, not the privileges on underlying tables
- Example: *employee* relation
  - Only HR can view all employee data
  - Employees can only view contact details

Example View Privileges

- SQL commands:

```
-- Start by disallowing all access to employee
REVOKE ALL PRIVILEGES ON employee TO PUBLIC;

-- Only allow hr_user to access employee relation
GRANT ALL PRIVILEGES ON employee TO hr_user;

-- View for "normal" employees to access
CREATE VIEW directory AS
    SELECT emp_name, email, office_phone
    FROM employee;
GRANT SELECT ON directory TO emp_user;
```

- When employees issue queries against *directory*, DB only checks *directory* privileges

View Processing

- As stated before, databases usually treat views as named SQL queries
  - Database substitutes view's definition directly into queries that reference the view
- SQL engine performs authorization *before* this process occurs
  - DB verifies access permissions on referenced views, and then substitutes view definitions into the query plan
  - Allows DB to support different access constraints on views, vs. their underlying tables

Other Privileges

- Many other privileges in SQL
  - **EXECUTE** grants privilege to execute a function or stored procedure
  - **CREATE** grants privilege to create tables, views, other schema objects
  - **REFERENCES** grants privilege to create foreign key or **CHECK** constraints
  - Most DBMSes provide several others, too
    - PostgreSQL has 11 permissions; MySQL has 27
    - Oracle has nearly 200 different permissions!

REFERENCES Privilege

- Foreign key constraints limit what users can do
  - Rows in referencing relation limit update and delete operations in referenced relation
  - A user adding a foreign key constraint can disallow these operations for all users!
- Must have the **REFERENCES** privilege to create foreign keys
- **REFERENCES** requires both a relation and some attributes to be specified
  - May create foreign keys involving those attributes

Passing On Privileges

- Users can't automatically grant their own privileges to other users
- Must explicitly allow this:

```
GRANT SELECT ON directory TO emp_user
    WITH GRANT OPTION;
```

  - **WITH GRANT OPTION** clause allows privileges to be passed on
- Can lead to confusing situations:
  - If **alex** grants a privilege to **bob**, then **alex** has that privilege revoked, should it affect **bob**?
  - If **alex** and **bob** both grant a privilege to **carl**, then **alex** revokes that privilege, does **carl** still have the privilege?
- Typically, databases implement simple solutions to these kinds of problems

Authorization Notes

- SQL authorization mechanism is very rich
- Still has a number of shortcomings
  - Can't grant/revoke privileges on per-tuple basis
    - e.g. "I can see only the rows in the *account* relation corresponding only to my bank accounts."
    - (If there were **`SELECT`** triggers, we could implement this…)
    - (Or, you could emulate this with table-returning functions…)
  - Significant variations in security models implemented by various databases

Authorization Notes (2)

- Most applications don't rely heavily on DB authorization
  - Application can implement a broad range of authorization schemes, but implementation complexity increases
  - Web applications are primary example of this
  - Database access layer typically has only one user, with full access and modification privileges
- Application performs authentication/authorization itself
  - Access-checks are sprinkled throughout application code; easy to introduce security holes! (e.g. PHP applications)
  - App-servers with declarative security specifications greatly mitigate this problem (e.g. JavaEE platform security)

Authorization Notes (3)

- Best to employ SQL auth mechanism in *some* way…
  - Declarative security specifications
  - Database simply won't allow access to privileged data, or unauthorized changes to schema
- For large, important database apps, definitely want to explore using SQL authorization features
  - At the least, create a DBMS user for each user-role that application supports
  - An "admin" user for administrators in the application, with fewer restrictions
  - A very restricted "common user" for end-users
  - <u>Greatly</u> reduces the dangers of SQL-based attacks

Next Time

- Last major topic for SQL data definition: indexes
  - Used to facilitate *much* faster database lookups
- Will also briefly discuss DB storage mechanisms, and how this affects query performance

DATABASE PERFORMANCE AND INDEXES

CS121: Introduction to Relational Database Systems

Fall 2014 – Lecture 11

Database Performance

- Many situations where query performance needs to be improved
  - e.g. as data size grows, query performance degrades and tuning needs to be performed
  - Extreme cases: data warehouses with millions or billions of rows to aggregate and summarize
- To optimize queries effectively, we must understand what the database is doing under the hood
  - e.g. "Why are correlated subqueries slow to evaluate?"
    - Because an inner query must be evaluated *for each row* considered by the outer query. Thus, a good idea to avoid!

Database Performance (2)

- Next two lectures will explore how most databases evaluate queries
  - Specifically, how are relational algebra operations implemented, and what optimizations do they employ?
  - As usual, there are always exceptions! (e.g. MySQL)
    - Important to be aware of, so you understand each DBMS' limitations
- Today, will concentrate more on data storage and access methodologies
- Next time, explore relational algebra implementations
  - These are built on top of topics covered today

Disk Access!

- First rule of database performance:
  Disk access is <u>the</u> most expensive thing databases do!
- Accessing data in memory can be 10-100ns
- Accessing data on disk can be up to 10s of ms
  - *That's 5-6 orders of magnitude difference!*
  - Even solid-state drives are 10s-100s of μs (1000x slower)
- Unfortunately, disk IO is usually unavoidable
  - Usually the data simply doesn't fit into memory…
  - Plus, the data needs to be persistent for when the DB is shut down, or when the server crashes, etc.
- DBs work very hard to minimize the amount of disk IO

Planning and Optimization

- When the query planner/optimizer gets your query:
  - It explores many equivalent plans, estimating their cost (primarily IO cost), and chooses the least expensive one
  - Considers many options in evaluating your query:
    - What access paths does it have to the data you want?
    - What algorithms can it use for selects, joins, sorting, etc?
    - What is the nature of the data itself?
      - i.e. statistics generated by the database, directly from your data
- The planner will do the best it can… ☺
  - Sometimes it can't find a fast way to run your query
  - Also depends on sophistication of the planner itself
    - e.g. if planner doesn't know how to optimize certain queries, or if executor doesn't implement very advanced algorithms

Table Data Storage

- Databases usually store each table in its own file
- File IO is performed in fixed-size blocks or pages
  - Common page size is 4KB or 8KB; can often tune this value
  - Disks can read/write entire pages faster than small amounts of bytes or individual records
  - Also makes it *much* easier for the database to manage pages of data in memory
    - The buffer manager takes care of this very complicated task
- Each block in the file contains some number of records
- Frequently, individual records can vary in size…
  - (due to variable-size types: **`VARCHAR`**, **`NUMERIC`**, etc.)

Table Data Storage (2)

- Individual blocks have internal structure, to manage:
  - Records that vary in size
  - Records that are deleted
  - Where and how to add a new record to the block, if there is space for it
- The table file itself also has internal structure:
  - Want to make sure common operations are fast!
    - "I want to insert a new row. Which block has space for it, or do I have to allocate a new block at the end of the file?"

Record Organization

- Should table records be organized in a specific way?
- Example: records are kept in sorted order, using a key
  - Called a sequential file organization
  - Would be much faster to find records based on the key
  - Would be much faster to do range queries as well
  - *Definitely* complicates the storage of records!
    - Can't predict order records will be added or deleted
    - Requires periodic reorganization to ensure that records remain physically sorted on the disk
- Could also hash records based on some key
  - Called a hashing file organization
  - Again, speeds up access based on specific values
  - Similar organizational challenges arise over time…

Record Organization (2)

- More advanced commercial DBs support tables with sequential or hashing file organizations…
  - A few even support very advanced storage layouts, such as <u>multitable clustering file organization</u>
    - If two tables will be joined a lot, interleave their records together in a single file
    - Records that would be equijoined are stored next to each other
- By far, the most common file organization is random! ☺
  - Called a <u>heap file organization</u>
  - Every record can be placed anywhere in the table file, wherever there is space for the record
  - Just about all databases provide heap file organization
  - Usually perfectly sufficient, except for most demanding tasks

Heap Files and Queries

- Given that DBs normally use heap file organization, how does the DB evaluate a query like:

```
SELECT * FROM account
WHERE account_id = 'A-591';
```

- A simple approach:
  - Search through the entire table file, looking for all rows where value of *account_id* is A-591
  - This is called a file scan, for obvious reasons
- This will be slow, but it's all we can do so far…
- Need a way to optimize accesses like this

Table Indexes

- Most queries use a small number of rows from a table
  - Need a faster way to look up those values, besides scanning through entire data file
- Approach: build an index on the table
  - Each index is associated with a specific column or set of columns in the table, called the search key for the index
  - Queries involving those columns can often be made *much* faster by using the index on those columns
  - (Queries not using those columns will still use a file scan ☹)
- Index is always structured in some way, for fast lookups
- Index is much smaller than the actual table itself
  - Much faster to search within the index (fewer IO operations)

Index Characteristics

- Many different varieties of indexes, with different access characteristics
  - What kind of lookup is most efficient for the kind of index?
  - How costly is it to find a particular item, or a set of items?
    - e.g. a query retrieving records with a range of values
- Indexes do impose both a time and space overhead
  - **Indexes must be kept up to date!** Frequently, they *slow down* update operations, while making selects faster.
- Different kinds of indexes impose different overheads:
  - How much time to add a new item to the index?
  - How much time to delete an item from the index?
  - How much additional space does the index take up?

Index Characteristics (2)

- Two major categories of indexes:
  - Ordered indexes keep values in a sorted order
  - Hash indexes divide values into bins, using a hash function
- Many variations within these two categories!
- Example: dense vs. sparse indexes
  - A dense index includes every single value from the source column(s). Faster lookups, but a larger space overhead.
  - A sparse index only includes some of the values. Lookups require searching more records, but index is smaller.
- The indexes we are covering today are dense indexes
  - Heap files are in random order, so an index won't help us very much unless it includes every value from the table

Index Implementations

- Indexes are usually stored in files separate from the actual table data
  - Indexes are also read/written as blocks
    - (Same reasons as before…)
- Indexes use <u>record pointers</u> to reference specific records in the table file
  - Simply consists of the block number the record is in, and the offset of the record within that block
- <u>Index records</u> contain values (or hashes), and one or more pointers to table records with those values

Index Implementations (2)

- Virtually all databases provide ordered indexes, using some kind of balanced tree structure
  - B^+-tree and B-tree indexes, typically referred to as "btree" indexes
- Some databases also provide hash indexes
  - More complex to manage than ordered indexes, so not very common in open-source databases
- Several other kinds of indexes as well:
  - Bitmap indexes – to speed up queries on multiple keys
    - Also less common in open-source databases
  - R-tree indexes – to make spatial queries very fast
    - With ubiquity of geospatial data, quite common these days

B^+-Tree Indexes

- A *very* widely used ordered index storage format
- Manages a balanced tree structure
  - Every path from root to leaf is the same length
  - Generally remains efficient for selects, even with inserts and deletes occurring
- Can consume significant space, since individual nodes can be up to half empty!
- Index updates for insert and delete can be slow…
  - Tree structure must be updated properly
- Performance benefits on queries more than outweigh these costs!

B+-Tree Indexes (2)

- Each tree node has up to *n* children
  - Simplification: *n* is fixed for the entire tree
- Each node stores *n* pointers and *n* – 1 values

P_1	K_1	P_2	K_2	P_3	…	P_{n-1}	K_{n-1}	P_n

  - K_i are search-key values, P_i are record pointers
  - Values are kept in sorted order: if $i < j$ then $K_i < K_j$
  - All nodes (except root) must be at least half full
- Size of *n* depends on block size, search-key size, and record pointer size, but it is usually <u>large</u>!
  - Example: 4KB blocks, 4B record pointers, 4B integer keys
  - *n* will be >500! B+-tree indexes are shallow, broad trees.

B^+-Tree Leaf Nodes

- For leaf nodes:

P_1	K_1	P_2	K_2	P_3	…	P_{n-1}	K_{n-1}	P_n

P_i points to record(s) with key value K_i

P_n points to next leaf in sequence (i.e. leaf whose first value is K_n)

  - Pointer P_i refers to record(s) with search-key value K_i
  - If search key is a candidate key, P_i points to the record with key value K_i
  - If search key isn't a candidate key, P_i points to a collection of pointers to all records with key value K_i
- No two leaves have overlapping ranges
  - Leaves can be arranged in sequential order
  - Pointer P_n points to the next leaf in sequential order

B^+-Tree Non-Leaf Nodes

- For non-leaf nodes:

P_1	K_1	P_2	K_2	P_3	…	P_{n-1}	K_{n-1}	P_n

P_1 is subtree with values $< K_1$

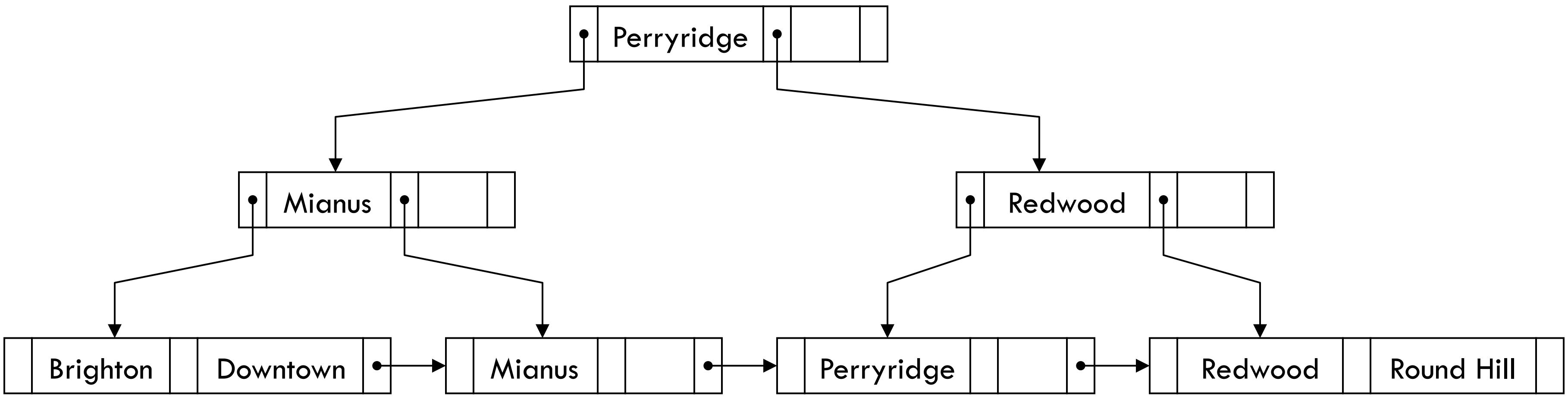
P_i is subtree with key values $K_{i-1} \leq K < K_i$

P_n is subtree with values $\geq K_{n-1}$

  - All pointers P_i refer to other B^+-tree nodes
- For $1 < i < n$:
  - Pointer P_i points to subtree containing search-key values of at least K_{i-1}, but less than K_i
- For $i = 1$ or $i = n$:
  - Pointer P_1 points to subtree containing search-key values less than K_1
  - Pointer P_n points to subtree containing search-key values at least K_{n-1}

Example B$^+$-Tree

- A simple B$^+$-tree, with $n = 3$



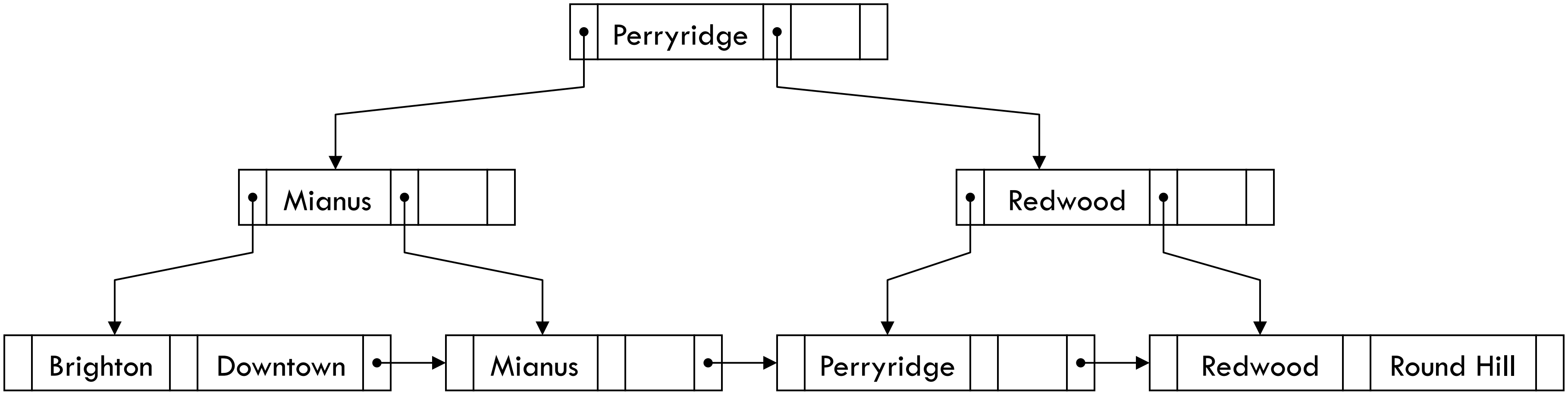
- Queries are straightforward
- Inserts may require a node to be split
- Deletes may require nodes to be merged

B$^+$-Trees and String Keys

- String columns are problematic for indexing
  - Frequently specified to have large/variable-size values
  - Large keys reduce branching factor of each node, increasing tree depth and access cost
  - Large keys can also interfere with tree restructuring
- Simple solution: don't use the entire string! ☺
  - Can use prefix compression technique
  - Non-leaf nodes only store a prefix of the search string
  - Size of prefix must be large enough to distinguish reasonably well between values in each subtree
    - Otherwise, can't effectively narrow down records to consider

B⁺-Trees and B-Trees

- In B^+-trees, key values appear in multiple nodes



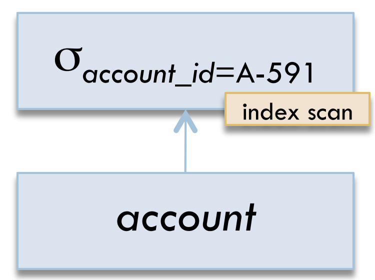
- B-tree indexes have a slightly different structure
  - Each key value only appears once in the hierarchy
  - Non-leaf nodes must also refer to records with each key value, as well as to subtrees
  - Slightly more complex structure, but saves space

Indexes and Queries

- Indexes provide an alternate <u>access path</u> to specific records in a table
  - If looking for a specific value or range of values, use the index to find where to start looking in the table file
- Query planner looks for indexes on relevant columns when optimizing your query
- Query from before:

```
SELECT * FROM account
WHERE account_id='A-591';
```

Execution Plan:



- If there is an index on *account_id* column, planner can use an index scan instead of a file scan
  - Execution plan is annotated with these kinds of details

Keys and Indexes

- Databases create many indexes automatically
  - DB will create an index on the primary key columns, and sometimes on foreign key columns too
  - Makes it much faster for DB to enforce key and referential integrity constraints
- Many of your queries already use these indexes!
  - Lookups on primary keys, and joins on primary/foreign key columns
- Sometimes queries use columns that don't have indexes
  - e.g. `SELECT * FROM account WHERE balance >= 3000;`
- How do we tell what indexes the DB uses for a query?
- How do we create additional indexes on our tables?

EXPLAIN Yourself

- Most databases have an **EXPLAIN**-type command
  - Performs query planning and optimization phases, then outputs details about the execution plan
  - Reports, among other things, what indexes are used
- MySQL **EXPLAIN** command:

```
EXPLAIN SELECT * FROM account
WHERE account_id = 'A-591';
```

```
+----+-------------+---------+-------+---------------+---------+---------+-------+------+-------+
| id | select_type | table   | type  | possible_keys | key     | key_len | ref   | rows | Extra |
+----+-------------+---------+-------+---------------+---------+---------+-------+------+-------+
|  1 | SIMPLE      | account | const | PRIMARY       | PRIMARY | 17      | const |    1 |       |
+----+-------------+---------+-------+---------------+---------+---------+-------+------+-------+
```

  - This query uses primary key index to look up the record
  - MySQL knows that the result will be one row, or no rows

MySQL **EXPLAIN** (2)

- More interesting result with a different account ID:
  EXPLAIN SELECT * FROM account
  WHERE account_id = 'A-000';

```
+----+-------------+-------+-----+-----------------------------------------------------+
| id | select_type | table | ... | Extra                                               |
+----+-------------+-------+-----+-----------------------------------------------------+
|  1 | SIMPLE      | NULL  | ... | Impossible WHERE noticed after reading const tables |
+----+-------------+-------+-----+-----------------------------------------------------+
```

  - MySQL planner uses the primary key index to discern that the specified ID doesn't appear in the *account* table!

- Another query against *account*:
  EXPLAIN SELECT * FROM account
  WHERE balance >= 3000;

```
+----+-------------+---------+------+---------------+------+---------+------+------+-------------+
| id | select_type | table   | type | possible_keys | key  | key_len | ref  | rows | Extra       |
+----+-------------+---------+------+---------------+------+---------+------+------+-------------+
|  1 | SIMPLE      | account | ALL  | NULL          | NULL | NULL    | NULL |   60 | Using where |
+----+-------------+---------+------+---------------+------+---------+------+------+-------------+
```

  - No index available to use for this column ☹

Adding Indexes to Tables

- If many queries reference columns that don't have indexes, and performance becomes an issue:
  - Create additional indexes on a table to help the DB
- Usually specified with **`CREATE INDEX`** commands
- To speed up queries on account balances:

```
CREATE INDEX idx_balance ON account (balance);
```

  - Database will create the index file and populate it from the current contents of the *account* relation
    - *(this could take some time for really large tables…)*
- Can also create multi-column indexes
- Can specify many options, such as the index type
  - Virtually all databases create **`BTREE`** indexes by default

Adding Indexes to Tables (2)

- MySQL allows you to specify indexes in the **`CREATE TABLE`** command itself…
  - *…not many other DBs support this, so it's not portable.*
- Any drawbacks to putting an index on account balances?
  - It's a bank. Account balances change all the time.
  - Will definitely incur a performance penalty on updates *(but, it probably won't be terribly substantial…)*

Verifying Index Usage

- Very important to verify that your new index is actually being used!
  - If your query doesn't use the index, best to get rid of it!

```
EXPLAIN SELECT * FROM account
  WHERE balance >= 3000;
+----+-------------+---------+------+---------------+------+---------+------+------+-------------+
| id | select_type | table   | type | possible_keys | key  | key_len | ref  | rows | Extra       |
+----+-------------+---------+------+---------------+------+---------+------+------+-------------+
|  1 | SIMPLE      | account | ALL  | idx_balance   | NULL | NULL    | NULL |   60 | Using where |
+----+-------------+---------+------+---------------+------+---------+------+------+-------------+
```

- Hmm, MySQL doesn't use the index for this query. ☹
  - If other expensive queries use it, makes sense to keep it (e.g. the rank query would use this index)
  - Otherwise, just get rid of it and keep your updates fast

Indexes on Large Values

- Large keys seriously degrade index performance
- Example: B-trees and B^+-trees
  - Biggest benefit is very large branching factor of each node
  - Large key-values will dramatically reduce the branching factor, deepening the tree and increasing IO costs
- Can specify indexes on only the first *N* characters/bytes of a string/LOB value

```
CREATE INDEX idx_name ON customer (cust_name(5));
```

  - Only uses first five characters for customer-name index
  - If most values differ in first *N* bytes, index will be much smaller and faster for both updates and queries
  - If values don't differ much, index won't do much good

Indexes and Performance Tuning

- Adding indexes to a schema is a common task in many database projects
- As a performance-tuning task, usually occurs after DB contains some data, and queries are slow
  - **<u>Always</u> avoid premature optimization!**
  - **<u>Always</u> find out what the DB is doing first!**
- Indexes impose an overhead in both space and time
  - Speeds up selects, but slows down all modifications
- Always need to verify that a new index is actually being used by the database. *If not, get rid of it!*

Administrivia

- Next time: SQL Query Evaluation II
  - Overview of how most relational algebra operators are implemented, including common-case optimizations

- Midterm time is a-comin'…
  - Next Monday, October 27, is midterm review
  - Come to class, watch the video, get the slides, whatever.
  - Midterm will be available towards end of next week
  - No assignment due the week of the midterm

SQL QUERY EVALUATION

CS121: Introduction to Relational Database Systems

Fall 2014 – Lecture 12

Query Evaluation

- Last time:
  - Began looking at database implementation details
  - How data is stored and accessed by the database
  - Using indexes to dramatically speed up certain kinds of lookups
- Today: What happens when we issue a query?
  - …and how can we make it faster?
- To optimize database queries, must understand what the database does to compute a result

Query Evaluation (2)

- Today:
  - Will look at higher-level query evaluation details
  - How relational algebra operations are implemented
    - Common-case optimizations employed in implementations
  - More details on how the database uses these details to plan and optimize your queries
- There are always exceptions…
  - e.g. MySQL's join processor is very different from others
  - Every DBMS has documentation about query evaluation and query optimization, for that specific database

SQL Query Processing

- Databases go through three basic steps:
  - Parse SQL into an internal representation of a plan
  - Transform this into an optimized execution plan
  - Evaluate the optimized execution plan
- Execution plans are generally based on the extended relational algebra
  - Includes generalized projection, grouping, etc.
  - Also some other features, like sorting results, nested queries, LIMIT/OFFSET, etc.

Query Evaluation Example

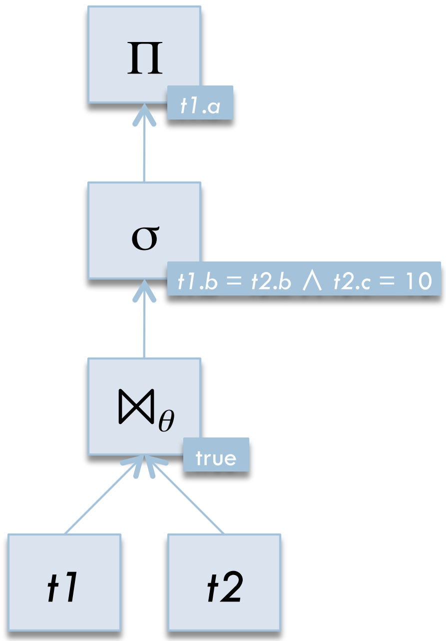
- A simple query:

```
SELECT t1.a FROM t1, t2
WHERE t1.b = t2.b AND t2.c = 10;
```

- Translating directly into the relational algebra:

$$\Pi_{t1.a}(\sigma_{t1.b\ =\ t2.b\ \wedge\ t2.c\ =\ 10}(t1 \times t2))$$

- Database might create this structure:
  - DBs usually implement common join operations with theta-join plan nodes
  - Can be evaluated using a push- or a pull-based approach
  - Evaluation loop retrieves results from top-level Π operation



Query Optimization

- Are there alternate formulations of our query?

  $\Pi_{t1.a}(\sigma_{t1.b\ =\ t2.b\ \wedge\ t2.c\ =\ 10}(t1 \times t2))$

  $\Pi_{t1.a}(t1 \bowtie_{t1.b\ =\ t2.b} (\sigma_{t2.c\ =\ 10}(t2)))$

  $\Pi_{t1.a}(\sigma_{t2.c\ =\ 10}(t1 \bowtie_{t1.b\ =\ t2.b}\ t2))$

  - *Which one is fastest?*
- The query optimizer generates many equivalent plans using a set of equivalence rules
  - <u>Cost-based optimizers</u> assign each plan a cost, and then the lowest-cost plan is chosen for execution
  - <u>Heuristic optimizers</u> just follow a set of rules for optimizing a query plan

Query Evaluation Costs

- A variety of costs in query evaluation
- Primary expense is reading data from disk
  - Usually, data being processed won't fit entirely into memory
  - Try to minimize disk seeks, reads and writes!
- CPU and memory requirements are secondary
  - Some ways of computing a result require more CPU and memory resources than others
  - Becomes especially important in concurrent usage scenarios
- Can be other costs as well
  - In distributed database systems, network bandwidth must be managed by query optimizer

Query Optimization (2)

- Several questions the optimizer has to consider:
  - How is a relation's data stored on the disk?
    - …and what access paths are available to the data?
  - What implementations of the relational algebra operations are available to use?
    - Will one implementation of a particular operation be much better or worse than another?
  - How does the database decide which query execution plan is best?
- Given the answers to these questions, what can <u>we</u> do to make the database go faster?

Select Operation

- How to implement σ_P operation?
- Easy solution from last time: scan the entire data file
  - Called a <u>file scan</u>
  - Test selection predicate against each tuple in the data file
  - Will be slow, since every disk block must be read
- This is a *general* solution, but not a *fast* one.
- What is the selection predicate P?
  - Depending on the characteristics of P, might be able to choose a more optimal evaluation strategy
  - If we can't, just stick with the file scan

Select Operation (2)

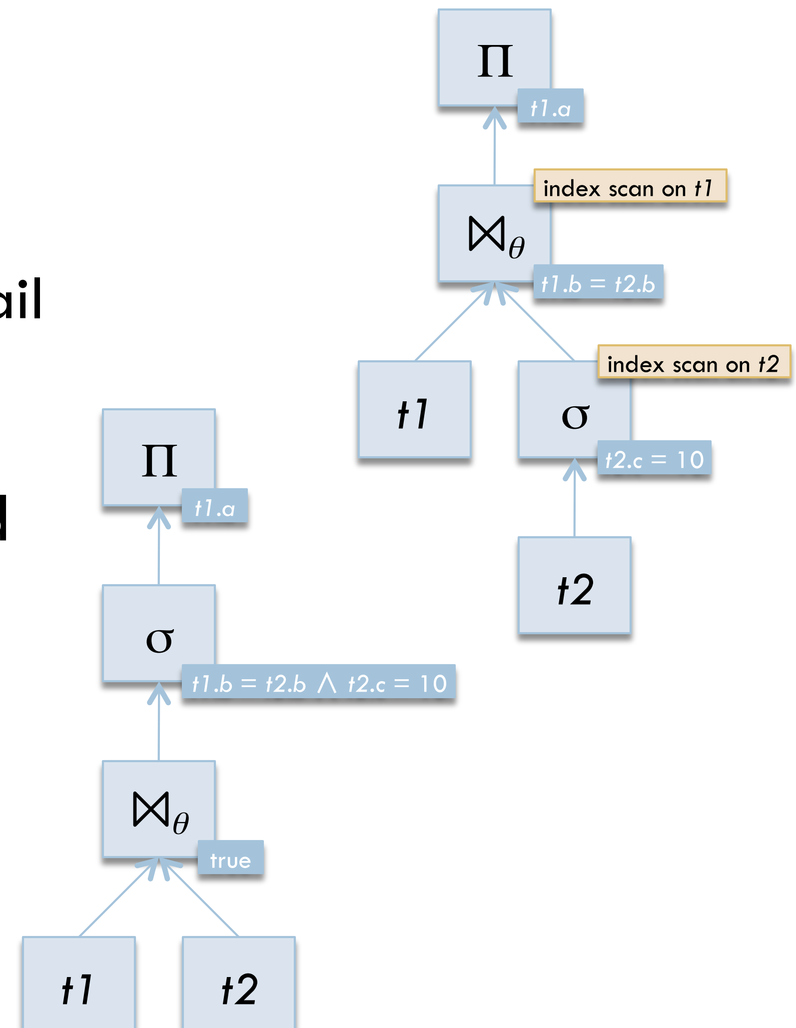
- Most select predicates involve a binary comparison
  - "Is an attribute equal to some value?"
  - "Is an attribute less than some value?"
- If data file was ordered, could use a binary search…
  - Would substantially reduce number of blocks read
  - Maintaining the logical record ordering becomes very costly if data changes frequently
- Solution:
  - Continue using heap file organization for table data
  - For important attributes, build <u>indexes</u> against the data file
    - Index provides a faster way to find specific values in the data file

Select Operation

- Query planner/optimizer looks at all access paths available for a given attribute
- For select operations:
  - If select predicate is an equality test and an index is available for that attribute, can use an <u>index scan</u>
  - Can also use index scan for comparison/range tests if an ordered index is available for the attribute
- For more complicated tests, or if no index is available for attributes being used:
  - Use the simple <u>file scan</u> approach

Query Optimization Using Indexes

- Database query optimizer looks for available indexes
  - If a select/lookup operation can use an index, execution plan is annotated with this detail
  - Overall plan cost is computed including these optimizations
- Indexes can only be exploited in certain circumstances
  - Typically, only by plan nodes that directly access the table
  - e.g. original plan can't really exploit indexes at all ☹



Project Operation

- Project operation is simple to implement
  - For each input tuple, create a new tuple with only the specified attributes
  - May also involve computed values
- Which would be faster, in general?

  $\Pi_{balance}(\sigma_{balance\ <\ 2500}(account))$

  $\sigma_{balance\ <\ 2500}(\Pi_{balance}(account))$
  - Want to project as few rows as possible, to minimize CPU and memory usage
    - Do select first: $\Pi_{balance}(\sigma_{balance\ <\ 2500}(account))$
  - Good heuristic example: "Do projects as late as possible."

Sorting

- SQL allows results to be ordered
- Databases must provide sorting capabilities in execution plans
  - Data being sorted may be much larger than memory!
- For tables that fit in memory, traditional sorting techniques are used (e.g. quick-sort)
- For tables that are larger than memory, must use an <u>external-memory sorting</u> technique
  - Table is divided into <u>runs</u> to be sorted in memory
  - Each run is sorted, then written to a temporary file
  - All runs are merged using an N-way merge sort

Sorting (2)

- In general, sorting should be applied as late as possible
  - Ideally, rows being sorted will fit into memory
- Some other operations can also use sorted inputs to improve performance
  - Join operations
  - Grouping and aggregation
  - Usually occurs when sorted results are already available
- Could also perform sorting with an ordered index
  - Scan index, and retrieve each tuple from table file in order
  - With magnetic disks, seek-time usually makes this prohibitive
    - (solid-state disks don't have this issue!)

Join Operations

- Join operations are *very* common in SQL queries
  - ...especially when using normalized schemas
- Could also potentially be a very costly operation!
  - $r \bowtie s$ defined as $\sigma_{r.A\,=\,s.A}(r \times s)$
- A simple strategy for $r \bowtie_\theta s$:

  for each tuple t_r **in** r **do begin**
      for each tuple t_s **in** s **do begin**
          if t_r , t_s satisfy condition θ **then**
              add $t_r \cdot t_s$ to result
      end
  end
- $t_r \cdot t_s$ denotes the concatenation of t_r with t_s

Nested-Loop Join

- Called the <u>nested-loop join</u> algorithm:

  for each tuple t_r **in** r **do begin**
      for each tuple t_s **in** s **do begin**
          if t_r , t_s satisfy condition θ **then**
              add $t_r \cdot t_s$ to result
      end
  end

- A very slow join implementation
  - Scans *r* once, and *s* once for each row in *r* !
  - Not so horrible if *s* fits entirely in memory
- But, it can handle arbitrary conditions
  - For some queries, the only option is a nested-loop join!

Indexed Nested-Loop Join

- Most join conditions involve equalities
  - Called <u>equijoins</u>
- Indexes can speed up table lookups…
- Modify nested-loop join to use indexes in inner loop:

for each tuple t_r **in** r **do begin**
    use index on s to retrieve tuple t_s
    if t_r , t_s satisfy condition θ **then**
        add $t_r \cdot t_s$ to result
end

- Only an option for equijoins, where an index exists for the join attributes

MySQL Join Processor

- MySQL join processor is based on nested-loop join algorithm
  - Instead of joining two tables, can join N tables at once

```
for each tuple t_r in r do begin
    for each tuple t_s in s do begin
        for each tuple t_t in t do begin
            if t_r , t_s , t_t , … satisfy condition θ then
                add t_r · t_s · t_t · … to result
        end
    end
end
```

- Employs <u>many</u> optimizations
  - When possible, outer table is processed in blocks, to reduce number of iterations over inner tables
  - Indexes are exploited *heavily* for finding tuples in inner tables.
  - If a subquery can be resolved into a constant, it is.

MySQL Join Processor (2)

- Since MySQL join processor relies so heavily on indexes, what kinds of queries is it bad at?
  - Queries against tables without indexes… (duh)
  - **Queries involving joins against derived relations (ugh!)**
  - MySQL isn't smart enough to save the derived relation into a temporary table, then build an index against it
    - A common technique for optimizing complex queries in MySQL
- For more sophisticated queries, really would like more advanced join algorithms…
  - Most DBs include several other very powerful join algorithms
  - (Can't add to MySQL easily, since it doesn't use relational algebra as a query-plan representation…)

Sort-Merge Join

- If tables are already ordered by join attributes, can use a merge-sort technique
  - Must be an equijoin!
- Simple high-level description:
  - Two pointers to traverse tables in order:
    - p_r starts at first tuple in r
    - p_s starts at first tuple in s
  - If one pointer's tuple has join-attribute values less than the other pointer, advance that pointer
  - When pointers have the same value of the join attribute, generate joins using those rows
    - If p_r or p_s points to a run of records with the same value, must include all of these records in the join result

Sort-Merge Join (2)

- Much better performance than nested-loop join
  - Dramatically reduces disk accesses
  - Unfortunately, relations aren't usually ordered
- Can also enhance sort-merge joins when at least one relation has an index on the join attributes
  - e.g. one relation is sorted, and the unsorted relation has an index on the join attributes
  - Traverse unsorted relation's index in order
  - When rows match, use index to pull those tuples from disk
  - Disk seek cost must be managed carefully with this technique
    - e.g. can sort record pointers before reading the tuples from disk, to minimize the overall seek time

Hash Join

- Another join technique for equijoins
- For tables *r* and *s* :
  - Use a hash function on the join attributes to divide rows of *r* and *s* into partitions
    - Use same hash function on both *r* and *s*, of course
    - Partitions are saved to disk as they are generated
    - Aim for each partition to fit in memory
    - *r* partitions: $H_{r1}, H_{r2}, \ldots, H_{rn}$
    - *s* partitions: $H_{s1}, H_{s2}, \ldots, H_{sn}$
  - Rows in H_{ri} will only join with rows in H_{si}

Hash Join (2)

- After partitioning:

```
for i = 1 to n do
    build a hash index on H_si                    (using a second hash function!)
    for each row t_r in H_ri
        probe hash index for matching rows in H_si
        for each matching tuple t_s in H_si
            add t_r · t_s to result
        end
    end
end
```

- Very fast and efficient equijoin strategy
  - Very good for joining against derived relations!
  - Can perform badly when rows can't be hashed into partitions that fit into memory

Outer Joins

- Join algorithms can be modified to generate left outer joins reasonably efficiently
  - Right outer join can be restated as left outer join
  - Will still impact overall query performance if many rows are generated
- Full outer joins can be significantly harder to implement
  - Sort-merge join can compute full outer join easily
  - Nested loop and hash join are much harder to extend
  - Full outer joins can also impact query performance heavily

Other Operations

- Set operations require duplicate elimination
  - Duplicate elimination can be performed with sorting or with hashing
- Grouping and aggregation can be implemented in several ways
  - Can sort results on the grouping attributes, then compute aggregates over the sorted values
    - All rows in a given group are adjacent to each other, so uses memory very efficiently (at least, after the sorting step…)
    - MySQL uses this approach by default
  - Can also use hashing to perform grouping and aggregation
    - Hash tuples on the grouping attributes, and compute each group's aggregate values incrementally

Optimizing Query Performance

- To improve query performance, you must know how the database actually runs your query
- Discussed the "explain" statement last time
  - Runs planner and optimizer on your query, then outputs the plan and corresponding cost estimates
- Using this information, you can:
  - Create indexes on tables, where appropriate
  - Restate the query to help the DB pick a better plan
- Harder cases may require multiple steps:
  - Generate intermediate results more well-suited for the desired query
  - Then, use intermediate results to generate final results

Query Execution Example

- For each assignment, finds the average size of the <u>last</u> submission from students for that assignment:

```
SELECT shortname,
       AVG(last_submission_size) AS
           avg_last_submission_size
FROM assignment NATURAL JOIN
     submission NATURAL JOIN
     (SELECT sub_id,
             total_size AS last_submission_size
      FROM fileset NATURAL JOIN
           (SELECT sub_id, MAX(sub_date) AS sub_date
            FROM fileset GROUP BY sub_id
           ) AS last_sub_dates
      ) AS last_sub_sizes
GROUP BY shortname;
```

Find the date of the last fileset submitted for each student's submission. Name the result columns to allow a natural join against the fileset table.

Query Execution Example (2)

- For each assignment, finds the average size of the <u>last</u> submission from students for that assignment:

```
SELECT shortname,
       AVG(last_submission_size) AS
           avg_last_submission_size
FROM assignment NATURAL JOIN
     submission NATURAL JOIN
     (SELECT sub_id,
             total_size AS last_submission_size
      FROM fileset NATURAL JOIN
           (SELECT sub_id, MAX(sub_date) AS sub_date
            FROM fileset GROUP BY sub_id
           ) AS last_sub_dates
      ) AS last_sub_sizes
GROUP BY shortname;
```

Join the derived result against fileset so we can retrieve the total size of the submitted files.

Query Execution Example (3)

- For each assignment, finds the average size of the <u>last</u> submission from students for that assignment:

```
SELECT shortname,
       AVG(last_submission_size) AS
           avg_last_submission_size
FROM assignment NATURAL JOIN
     submission NATURAL JOIN
     (SELECT sub_id,
             total_size AS last_submission_size
      FROM fileset NATURAL JOIN
           (SELECT sub_id, MAX(sub_date) AS sub_date
            FROM fileset GROUP BY sub_id
           ) AS last_sub_dates
      ) AS last_sub_sizes
GROUP BY shortname;
```

Outermost query finds the averages of these last submissions, and also incorporates the short-name of each assignment.

MySQL Execution and Analysis

- MySQL executes this query rather slowly*
  - About 3 sec on a server with 8GB RAM, RAID1 mirroring
  - Intuitively makes sense…
    - Joins against derived relations, non-index columns, etc.
    - All the stuff that MySQL isn't so good at handling
- **`EXPLAIN`** output:

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	<derived2>	ALL	NULL	NULL	NULL	NULL	1506	Using temporary; Using filesort
1	PRIMARY	submission	eq_ref	PRIMARY	PRIMARY	4	last_sub_sizes.sub_id	1	
1	PRIMARY	assignment	eq_ref	PRIMARY	PRIMARY	4	donnie_db.submission.asn_id	1	
2	DERIVED	<derived3>	ALL	NULL	NULL	NULL	NULL	1506	
2	DERIVED	fileset	ALL	NULL	NULL	NULL	NULL	2799	Using where; Using join buffer
2	DERIVED	submission	eq_ref	PRIMARY	PRIMARY	4	last_sub_dates.sub_id	1	Using index
3	DERIVED	fileset	ALL	NULL	NULL	NULL	NULL	2799	Using temporary; Using filesort

- Confirms our suspicions
- Can optimize by storing innermost results in a temp table, and creating indexes on (sub_id, sub_date)

* Test was performed with MySQL 5.1; MariaDB 5.5 executes this query extremely quickly.

PostgreSQL Execution/Analysis (1)

- Postgres executes this query instantaneously. On a laptop.
  - Fundamental difference: more sophisticated join algorithms
    - Specifically hash join, which is very good at joining relations on non-indexed attributes
- **EXPLAIN** output:

```
HashAggregate  (cost=221.38..221.39 rows=1 width=8)
  ->  Nested Loop  (cost=144.28..221.37 rows=1 width=8)
        ->  Nested Loop  (cost=144.28..213.09 rows=1 width=20)
              ->  Nested Loop  (cost=144.28..212.81 rows=1 width=20)
                    ->  Hash Join  (cost=144.28..204.53 rows=1 width=12)
                          Hash Cond: ((fileset.sub_id = fileset.sub_id) AND ((max(fileset.sub_date)) = fileset.sub_date))
                          ->  HashAggregate  (cost=58.35..77.18 rows=1506 width=12)
                                ->  Seq Scan on fileset  (cost=0.00..44.57 rows=2757 width=12)
                          ->  Hash  (cost=44.57..44.57 rows=2757 width=16)
                                ->  Seq Scan on fileset  (cost=0.00..44.57 rows=2757 width=16)
                    ->  Index Scan using submission_pkey on submission  (cost=0.00..8.27 rows=1 width=8)
                          Index Cond: (submission.sub_id = fileset.sub_id)
              ->  Index Scan using assignment_pkey on assignment  (cost=0.00..0.27 rows=1 width=8)
                    Index Cond: (assignment.asn_id = submission.asn_id)
        ->  Index Scan using submission_pkey on submission  (cost=0.00..8.27 rows=1 width=4)
              Index Cond: (submission.sub_id = fileset.sub_id)
```

- As expected, Postgres uses a hash join to join the derived relation against `fileset` table on non-index columns

PostgreSQL Execution/Analysis (2)

- Can disable various join algorithms in Postgres ☺
  - **`SET enable_hashjoin = off;`**
- **`EXPLAIN`** output:

```
HashAggregate  (cost=422.68..422.69 rows=1 width=8)
  ->  Nested Loop  (cost=373.85..422.67 rows=1 width=8)
        ->  Nested Loop  (cost=373.85..414.39 rows=1 width=20)
              ->  Nested Loop  (cost=373.85..414.11 rows=1 width=20)
                    ->  Merge Join  (cost=373.85..405.83 rows=1 width=12)
                          Merge Cond: ((fileset.sub_id = fileset.sub_id) AND (fileset.sub_date = (max(fileset.sub_date))))
                          ->  Sort  (cost=202.12..209.01 rows=2757 width=16)
                                Sort Key: fileset.sub_id, fileset.sub_date
                                ->  Seq Scan on fileset  (cost=0.00..44.57 rows=2757 width=16)
                          ->  Sort  (cost=171.73..175.50 rows=1506 width=12)
                                Sort Key: fileset.sub_id, (max(fileset.sub_date))
                                ->  HashAggregate  (cost=58.35..77.18 rows=1506 width=12)
                                      ->  Seq Scan on fileset  (cost=0.00..44.57 rows=2757 width=12)
                    ->  Index Scan using submission_pkey on submission  (cost=0.00..8.27 rows=1 width=8)
                          Index Cond: (submission.sub_id = fileset.sub_id)
              ->  Index Scan using assignment_pkey on assignment  (cost=0.00..0.27 rows=1 width=8)
                    Index Cond: (assignment.asn_id = submission.asn_id)
        ->  Index Scan using submission_pkey on submission  (cost=0.00..8.27 rows=1 width=4)
              Index Cond: (submission.sub_id = fileset.sub_id)
```

- Sort + sort-merge join is still faster than nested loops!!

PostgreSQL Execution/Analysis (3)

- Now, disable sort-merge joins too:
  - **`SET enable_mergejoin = off;`**
- Finally, Postgres performance is closer to MySQL
- **`EXPLAIN`** output:

```
HashAggregate  (cost=103956.21..103956.23 rows=1 width=8)
  -> Nested Loop  (cost=93.75..103956.21 rows=1 width=8)
        -> Nested Loop  (cost=93.75..103947.93 rows=1 width=20)
              -> Nested Loop  (cost=93.75..103947.65 rows=1 width=20)
                    -> Nested Loop  (cost=93.75..103939.37 rows=1 width=12)
                          Join Filter: ((fileset.sub_id = fileset.sub_id) AND (fileset.sub_date = (max(fileset.sub_date))))
                          -> Seq Scan on fileset  (cost=0.00..44.57 rows=2757 width=16)
                          -> Materialize  (cost=93.75..108.81 rows=1506 width=12)
                                -> HashAggregate  (cost=58.35..77.18 rows=1506 width=12)
                                      -> Seq Scan on fileset  (cost=0.00..44.57 rows=2757 width=12)
                    -> Index Scan using submission_pkey on submission  (cost=0.00..8.27 rows=1 width=8)
                          Index Cond: (submission.sub_id = fileset.sub_id)
              -> Index Scan using assignment_pkey on assignment  (cost=0.00..0.27 rows=1 width=8)
                    Index Cond: (assignment.asn_id = submission.asn_id)
        -> Index Scan using submission_pkey on submission  (cost=0.00..8.27 rows=1 width=4)
              Index Cond: (submission.sub_id = fileset.sub_id)
```

Query Estimates

- Query planner/optimizer must make *estimates* about the cost of each stage
- Database maintains statistics for each table, to facilitate planning and optimization
- Different levels of detail:
  - Some DBs only track min/max/count of values in each column. Estimates are very basic.
  - Some DBs generate and store histograms of values in important columns. Estimates are much more accurate.
- Different levels of accuracy:
  - Statistics are expensive to maintain! Databases update these statistics relatively infrequently.
  - If a table's contents change substantially, must recompute statistics

Table Statistics Analysis

- Databases also frequently provide a command to compute table statistics
- MySQL command:

  ```
  ANALYZE TABLE assignment, submission, fileset;
  ```

- PostgreSQL command:

  ```
  VACUUM ANALYZE;
  ```

  - for all tables in database

  ```
  VACUUM ANALYZE tablename;
  ```

  - for a specific table
- These commands are expensive!
  - Perform a full table-scan
  - Also, typically lock the table(s) for exclusive access

Review

- Discussed general details of how most databases evaluate SQL queries
- Some relational algebra operations have several ways to evaluate them
  - Optimizations for very common special cases, e.g. equijoins
- Can give the database some guidance
  - Create indexes on tables where appropriate
  - Rewrite queries to be more efficient
  - Make sure statistics are up-to-date, so that planner has best chance of generating a good plan

CS121 MIDTERM REVIEW

CS121: Introduction to Relational Database Systems

Fall 2014 – Lecture 13

Before We Start…

Midterm Overview

- 6 hours, multiple sittings
- Open book, open notes, open lecture slides
- No collaboration
- Possible Topics:
  - Basically, everything you've seen on homework assignments to this point
  - Relational model
    - relations, keys, relational algebra operations (queries, modifications)
  - SQL DDL commands
    - **`CREATE TABLE`**, **`CREATE VIEW`**, integrity constraints, etc.
    - Altering existing database schemas
    - Indexes

Midterm Overview (2)

- Possible Topics (cont):
  - SQL DML commands
    - **`SELECT`**, **`INSERT`**, **`UPDATE`**, **`DELETE`**
    - Grouping and aggregation, subqueries, etc.
    - Aggregates of aggregates ☺
    - Translation to relational algebra, performance considerations, etc.
  - Procedural SQL
    - User-defined functions (UDFs)
    - Stored procedures
    - Triggers
    - Cursors

Midterm Overview (2)

- You can use a MySQL database, if you want to
  - e.g. make sure your DDL syntax is correct, check schema-alteration steps, verify that UDFs work
- WARNING: Don't let it become a time-sink!
  - I won't necessarily give you actual data for problems
  - Don't waste time making up data just to test your SQL

Assignments and Solution Sets

- Some assignments may not be graded in time for the midterm (e.g. HW4)
- HW1-HW4 solution sets will be on Moodle by the time of the midterm

Relational Model

- Be familiar with the relational model:
  - What's a relation? What's a relation schema? What's a tuple? etc.
- Remember, relations are different from SQL tables in a very important way:
  - Relations are <u>sets</u> of tuples. SQL tables are <u>multisets</u> of tuples.

Keys in the Relational Model

- Be familiar with the different kinds of keys
  - Keys uniquely identify tuples within a relation
- Superkey
  - Any set of attributes that uniquely identifies a tuple
  - If a set of attributes *K* is a superkey, then so is any superset of *K*
- Candidate key
  - A <u>minimal</u> superkey
  - If any attribute is removed, no longer a superkey
- Primary key
  - A particular candidate key, chosen as the <u>primary</u> means of referring to tuples

Keys and Constraints

- Keys constrain the set of tuples that can appear in a relation
  - In a relation *r* with a candidate key *K*, no two tuples can have the same values for *K*

- Can also have foreign keys
  - One relation contains the key attributes of another relation
  - Referencing relation has a foreign key
  - Referenced relation has a primary (or candidate) key
  - Referencing relation can only contain values of foreign key that also appear in referenced relation
  - Called <u>referential integrity</u>

Foreign Key Example

- Bank example:

  account(*<u>account_number</u>*, *branch_name*, *balance*)

  depositor(*customer_name*, *account_number*)

- *depositor* is the referencing relation
  - *account_number* is a foreign-key to *account*
- *account* is the referenced relation

A Note on Notation

- Depositor relation:
  - *depositor*(*customer_name*, *account_number*)
- In the relational model:
  - Every (*customer_name*, *account_number*) pair in *depositor* is unique
- When translating to SQL:
  - **`depositor`** table <u>could</u> be a multiset…
  - Need to ensure that SQL table is actually a <u>set</u>, not a multiset
  - **`PRIMARY KEY (customer_name, account_number)`** after all columns are declared

Referential Integrity in Relational Model

- In the relational model, you must pay attention to referential integrity constraints
  - Make sure to perform modifications in an order that maintains referential integrity
- Example: Remove customer "Jones" from bank
  - Customer name appears in *customer*, *depositor*, and *borrower* relations
  - Which relations reference which?
    - *depositor* references *customer*
    - *borrower* references *customer*
  - Remove Jones records from *depositor* and *borrower* first
  - Then remove Jones records from *customer*

Relational Algebra Operations

- Six fundamental operations:

σ	select operation
Π	project operation
$\cup$	set-union operation
$-$	set-difference operation
$\times$	Cartesian product operation
ρ	rename operation

  - Operations take one or two relations as input
  - Each produces another relation as output

Additional Relational Operations

- Several additional operations, defined in terms of fundamental operations:

$\cap$	set-intersection
$\bowtie$	natural join (also theta-join $\bowtie_\theta$)
$\div$	division
$\leftarrow$	assignment

- Extended relational operations:

Π	*generalized* project operation
$\mathcal{G}$	grouping and aggregation
⟕ ⟖ ⟗	left outer join, right outer join, full outer join

Join Operations

- Be familiar with different join operations in relational algebra
- Cartesian product $r \times s$ generates every possible pair of rows from r and s
- Summary of other join operations:

$r =$

attr1	attr2
a	r1
b	r2
c	r3

$s =$

attr1	attr3
b	s2
c	s3
d	s4

$r \bowtie s$

attr1	attr2	attr3
b	r2	s2
c	r3	s3

r ⟕ s

attr1	attr2	attr3
a	r1	*null*
b	r2	s2
c	r3	s3

r ⟖ s

attr1	attr2	attr3
b	r2	s2
c	r3	s3
d	*null*	s4

r ⟗ s

attr1	attr2	attr3
a	r1	*null*
b	r2	s2
c	r3	s3
d	*null*	s4

Rename Operation

- Mainly used when joining a relation to itself
  - Need to rename one instance of the relation to avoid ambiguities
- Remember you can specify names with both Π and $\mathcal{G}$
  - Can rename attributes
  - Can assign a name to computed results
  - Naming computed results in Π or $\mathcal{G}$ is shorter than including an extra ρ operation
- Use ρ when you are <u>only</u> renaming things
  - Don't use Π or $\mathcal{G}$ just to rename something
  - Also, ρ doesn't create a new relation-variable! Assignment $\leftarrow$ does this.

Examples

- Schema for an auto insurance database:

  car(*<u>license</u>*, *vin*, *make*, *model*, *year*)
  - *vin* is also a candidate key, but not the primary key

  customer(*<u>driver_id</u>*, *name*, *street*, *city*)

  owner(*<u>license</u>*, *driver_id*)

  claim(*<u>driver_id</u>*, *<u>license</u>*, *<u>date</u>*, *description*, *amount*)

- Find names of all customers living in Los Angeles or New York.

  $\Pi_{name}(\sigma_{city=\text{“Los Angeles”} \lor\ city=\text{“New York”}}(customer))$
  - Select predicate can refer to attributes, constants, or arithmetic expressions using attributes
  - Conditions combined with $\land$ and $\lor$

Examples (2)

- Schema:

  car(<u>*license*</u>, *vin*, *make*, *model*, *year*)

  customer(<u>*driver_id*</u>, *name*, *street*, *city*)

  owner(<u>*license*</u>, *driver_id*)

  claim(<u>*driver_id*</u>, <u>*license*</u>, <u>*date*</u>, *description*, *amount*)

- Find customer name, street, and city of all Toyota owners
  - Need to join *customer*, *owner*, *car* relations
  - Could use Cartesian product, select, etc.
  - Or, use natural join operation:

    $\Pi_{name,street,city}(\sigma_{make=\text{“Toyota”}}(customer \bowtie owner \bowtie car))$

Examples (3)

- Schema:

  car(<u>*license*</u>, *vin*, *make*, *model*, *year*)

  customer(<u>*driver_id*</u>, *name*, *street*, *city*)

  owner(<u>*license*</u>, *driver_id*)

  claim(<u>*driver_id*</u>, <u>*license*</u>, <u>*date*</u>, *description*, *amount*)

- Find how many claims each customer has
  - Don't include customers with no claims…
  - Simple grouping and aggregation operation

    $_{driver\_id}\mathcal{G}_{\mathbf{count}(license)\ \mathbf{as}\ num\_claims}(claim)$

    - The specific attribute that is counted is irrelevant here…
  - Aggregate operations work on <u>multisets</u> by default
  - Schema of result?

    (*driver_id*, *num_claims*)

Examples (4)

- Now, include customers with no claims
  - They should have 0 in their values
  - Requires outer join between *customer*, *claim*
  - "Outer" part of join symbol is towards relation whose rows should be null-padded
  - Want all customers, and claim records if they are there, so "outer" part is towards *customer*

$$_{driver\_id}\mathcal{G}_{\mathbf{count}(license)\ \mathbf{as}\ num\_claims}(customer ⟕ claim)$$

  - Aggregate functions ignore *null* values

Selecting on Aggregate Values

- Grouping/aggregation op produces a relation, not an individual scalar value

  You cannot use aggregate functions in select predicates!!!

- To select rows based on an aggregate value:
  - Create a grouping/aggregation query to generate the aggregate results
    - This is a relation, so…
  - Use Cartesian product (or another appropriate join operation) to combine rows with the relation containing aggregated results
  - Select out the rows that satisfy the desired constraints

Selecting on Aggregate Values (2)

- General form of grouping/aggregation:
  - $_{G_1, G_2, \ldots}\mathcal{G}_{F(A_1), F(A_2), \ldots}(\ldots)$
- Results of aggregate functions are unnamed!
- This query is <u>wrong</u>:
  - $\sigma_{F(A_1) = \ldots}(_{G_1, G_2, \ldots}\mathcal{G}_{F(A_1), F(A_2), \ldots}(\ldots))$
  - Attribute in result does <u>not</u> have name $F(A_1)$!
- Must *assign* a name to the aggregate result
  - $_{G_1, G_2, \ldots}\mathcal{G}_{F(A_1)\ \textbf{as}\ V_1, F(A_2)\ \textbf{as}\ V_2, \ldots}(\ldots)$
- Then, can properly select against the result:
  - $\sigma_{V_1 = \ldots}(_{G_1, G_2, \ldots}\mathcal{G}_{F(A_1)\ \textbf{as}\ V_1, F(A_2)\ \textbf{as}\ V_2, \ldots}(\ldots))$

An Aggregate Example

- Schema: *car*(*license*, *vin*, *make*, *model*, *year*)
  customer(*driver_id*, *name*, *street*, *city*)
  owner(*license*, *driver_id*)
  claim(*driver_id*, *license*, *date*, *description*, *amount*)
- Find the claim(s) with the largest amount
  - Claims are identified by (*driver_id*, *license*, *date*), so just return all attributes of the claim
  - Use aggregation to find the maximum claim amount:

    $\mathcal{G}_{\textbf{max}(amount)\ \textbf{as}\ max\_amt}(claim)$
  - This generates a relation! Use Cartesian product to select the row(s) with this value.

$$\Pi_{driver\_id,license,date,description,amount}(\sigma_{amount=max\_amt}(claim \times \mathcal{G}_{\textbf{max}(amount)\ \textbf{as}\ max\_amt}(claim)))$$

Another Aggregate Example

- Schema: *car*(<u>*license*</u>, *vin*, *make*, *model*, *year*)
  customer(<u>*driver_id*</u>, *name*, *street*, *city*)
  owner(<u>*license*</u>, *driver_id*)
  claim(<u>*driver_id*</u>, <u>*license*</u>, <u>*date*</u>, *description*, *amount*)
- Find the customer with the most insurance claims, along with the number of claims
- This involves two levels of aggregation
  - Step 1: generate a count of each customer's claims
  - Step 2: compute the maximum count from this set of results
- Once you have result of step 2, can reuse the result of step 1 to find the final result
- Common subquery: computation of how many claims each customer has

Another Aggregate Example (2)

- Use assignment operation to store temporary result

  $claim\_counts \leftarrow {}_{driver\_id}\mathcal{G}_{\mathbf{count}(license)\ \mathbf{as}\ num\_claims}(claim)$

  $max\_count \leftarrow \mathcal{G}_{\mathbf{max}(num\_claims)\ \mathbf{as}\ max\_claims}(claim\_counts)$

- Schemas of *claim_counts* and *max_count* ?

  $claim\_counts(driver\_id, num\_claims)$

  $max\_count(max\_claims)$

- Finally, select row from *claim_counts* with the maximum count value
  - Obvious here that a Cartesian product is necessary

    $\Pi_{driver\_id, num\_claims}($
    $\sigma_{num\_claims = max\_claims}(claim\_counts \times max\_count))$

Modifying Relations

- Can add rows to a relation

  $r \leftarrow r \cup \{ (\ldots), (\ldots) \}$

  - $\{ (\ldots), (\ldots) \}$ is called a <u>constant relation</u>
  - Individual tuple literals enclosed by parentheses ( )
  - Set of tuples enclosed with curly braces { }
- Can delete rows from a relation

  $r \leftarrow r - \sigma_P(r)$
- Can modify rows in a relation

  $r \leftarrow \Pi(r)$

  - Uses generalized project operation

Modifying Relations (2)

- **Remember to include unmodified rows!**

  $r \leftarrow \Pi(\sigma_P(r)) \;\cup\; \sigma_{\neg P}(r)$

- Relational algebra is <u>not</u> like SQL for updates!
  - Must <u>explicitly</u> include unaffected rows
- Example:

  Transfer $10,000 in assets to all Horseneck branches.

  $branch \leftarrow \Pi_{branch\_name, branch\_city, assets+10000}(\sigma_{branch\_city=\text{“Horseneck”}}(branch))$

  Wrong: This version *throws out* all branches not in Horseneck!

  $branch \leftarrow \Pi_{branch\_name, branch\_city, assets+10000}(\sigma_{branch\_city=\text{“Horseneck”}}(branch)) \;\cup$
  $\sigma_{branch\_city \neq \text{“Horseneck”}}(branch)$

  Correct. Non-Horseneck branches are included, unmodified.

Structured Query Language

- Some major differences between SQL and relational algebra!
- Tables are like relations, but are multisets
- Most queries generate multisets
  - **`SELECT`** queries produce multisets, unless they specify **`SELECT DISTINCT ...`**
- Some operations <u>do</u> eliminate duplicates!
  - Set operations: **`UNION`**, **`INTERSECT`**, **`EXCEPT`**
    - Duplicates are eliminated automatically, unless you specify **`UNION ALL`**, **`INTERSECT ALL`**, **`EXCEPT ALL`**

SQL Statements

- **`SELECT`** is most ubiquitous

  `SELECT` A_1, A_2, `...` `FROM` r_1, r_2, `...`
  `WHERE P;`

  - Equivalent to: $\Pi_{A_1,\, A_2,\, \ldots}(\sigma_P(r_1 \times r_2 \times \ldots))$
- **`INSERT`**, **`UPDATE`**, **`DELETE`** all have common aspects of **`SELECT`**
  - All support **`WHERE`** clause, subqueries, etc.
  - Also **`INSERT ... SELECT`** statement

Join Alternatives

- **`FROM r1, r2`**
  - Cartesian product
  - Can specify join conditions in **`WHERE`** clause
- **`FROM r1 JOIN r2 ON (r1.a = r2.a)`**
  - Most like theta-join operator: $r \bowtie_\theta s = \sigma_\theta(r \times s)$
  - Doesn't eliminate any columns!
- **`FROM r1 JOIN r2 USING (a)`**
  - Eliminates duplicate column **`a`**
- **`FROM r1 NATURAL JOIN r2`**
  - Uses <u>all</u> common attributes to join **`r1`** and **`r2`**
  - Also eliminates <u>all</u> duplicate columns in result

Join Alternatives (2)

- Can specify inner/outer joins with **`JOIN`** syntax
  - **`r INNER JOIN s ...`**
  - **`r LEFT OUTER JOIN s ...`**
  - **`r RIGHT OUTER JOIN s ...`**
  - **`r FULL OUTER JOIN s ...`**
- Can also specify **`r CROSS JOIN s`**
  - Cartesian product of *r* with *s*
  - Can't specify **`ON`** condition, **`USING`**, or **`NATURAL`**
- Can actually leave out **`INNER`** or **`OUTER`**
  - **`OUTER`** is implied by **`LEFT`**/**`RIGHT`**/**`FULL`**
  - If you just say **`JOIN`**, this is an **`INNER`** join

Self-Joins

- Sometimes helpful to do a self-join
  - A join of a table with itself
- Example: employees

  employee(*emp_id*, *emp_name*, *salary*, *manager_id*)
- Tables can contain foreign-key references to themselves
  - *manager_id* is a foreign-key reference to *employee* table's *emp_id* attribute
- Example:
  - Write a query to retrieve the name of each employee, and the name of each employee's boss.

```
SELECT e.emp_name, b.emp_name AS boss_name
  FROM employee AS e JOIN employee AS b
       ON (e.manager_id = b.emp_id);
```

Subqueries

- Can include subqueries in **`FROM`** clause
  - Called a derived relation
  - Nested **`SELECT`** statement in **`FROM`** clause, given a name and a set of attribute names
- Can also use subqueries in **`WHERE`** clause
  - Can compare an attribute to a scalar subquery
    - This is different from the relational algebra!
  - Can also use set-comparison operations to test against a subquery
    - **`IN`**, **`NOT IN`** – set membership tests
    - **`EXISTS`**, **`NOT EXISTS`** – empty-set tests
    - **`ANY`**, **`SOME`**, **`ALL`** – comparison against a set of values

Scalar Subqueries

- Find name and city of branch with the least assets
  - Need to generate the "least assets" value, then use this to select the specific branch records
- Query:

```
SELECT branch_name, branch_city FROM branch
WHERE assets = (SELECT MIN(assets) FROM branch);
```

  - This is a <u>scalar subquery</u>: one row, one column
  - Don't need to name **`MIN(assets)`** since it doesn't appear in final result, and we don't refer to it
- Don't do this:

```
WHERE assets=ALL (SELECT MIN(assets) FROM branch)
```

  - **`ANY`**, **`SOME`**, **`ALL`** are for comparing a value to a <u>set</u> of values
  - Don't need these when comparing to a scalar subquery

Subqueries vs. Views

- Don't create views unnecessarily
  - Views are part of a database's schema
  - Every database user sees the views that are defined
- Views should generally expose "final results," not intermediate results in a larger computation
  - Don't use views to compute intermediate results
- If you *really* want functionality like this, read about the **`WITH`** clause (Book, 6th ed: §3.8.6, pg. 97)
  - MySQL doesn't support **`WITH`**, so unfortunately you can't use it in CS121 ☹

WHERE Clause

- **WHERE** clause specifies selection predicate
  - Can use **AND**, **OR**, **NOT** to combine conditions
  - **NULL** values affect comparisons!
    - Can't use `= NULL` or `<> NULL`
      - <u>Never</u> evaluates to true, regardless of other value
    - Must use `IS NULL` or `IS NOT NULL`
  - Can use **BETWEEN** to simplify range checks
    - `a >= v1 AND a <= v2`
    - `a BETWEEN v1 AND v2`

Grouping and Aggregation

- SQL supports grouping and aggregation
- **GROUP BY** specifies attributes to group on
  - Apply aggregate functions to non-grouping columns in **SELECT** clause
  - Can filter results of grouping operation using **HAVING** clause
    - **HAVING** clause can refer to aggregate values too
- Difference between **WHERE** and **HAVING** ?
  - **WHERE** is applied <u>before</u> grouping;
    HAVING is applied <u>after</u> grouping
  - **HAVING** can refer to aggregate results, too
    - Unlike relational algebra, can use aggregate functions in **HAVING** clause

Grouping: SQL, Relational Algebra

- Another difference between relational algebra notation and SQL syntax
- Relational algebra syntax:

$$_{G_1, G_2, \ldots, G_n}\mathcal{G}_{F_1(A_1), F_2(A_2), \ldots, F_m(A_m)}(E)$$

  - Grouping attributes appear only on left of $\mathcal{G}$
  - Schema of result: $(G_1, G_2, \ldots, F_1, F_2, \ldots)$
    - (Remember, F_i generate unnamed results.)
- SQL syntax:

```
SELECT G1,G2,..., F1(A1),F2(A2),...
  FROM r1,r2,... WHERE P
  GROUP BY G1,G2,...
```

  - To include group-by values in result, specify grouping attributes in **SELECT** clause and in **GROUP BY** clause

Grouping and Distinct Results

- SQL grouping syntax:

```
SELECT G1,G2,..., F1(A1),F2(A2),...
  FROM r1,r2,... WHERE P
  GROUP BY G1,G2,...
```

- If <u>all</u> grouping attributes are in **SELECT** clause:
  - Are all rows in the results distinct?
  - Yes! $(G_1, G_2, \ldots)$ is a superkey on the results.
  - Each group in result has a unique set of values for the set of grouping attributes
  - Don't need to specify **SELECT <u>DISTINCT</u>** $G_1, G_2, \ldots$ if all grouping attributes are listed

Grouping and Results

- Another example:

```
SELECT G3, F1(A1), F2(A2)
  FROM r1,r2,... WHERE P
  GROUP BY G1, G2, G3
```

- You can specify only a subset of the grouping attributes in the **SELECT** clause (or even none of the grouping attributes)
  - Results no longer guaranteed to be distinct, of course
- Main constraint (approximately):
  - Can't specify *non-grouping* attributes in **SELECT** clause unless they are arguments to an aggregate function
  - Default MySQL configuration allows you to violate this rule, but the results are not well-defined!
    - This has been turned off all term, hopefully this has helped… ☺

SQL Query Example

- Schema:

  car(*<u>license</u>*, *vin*, *make*, *model*, *year*)

  customer(*<u>driver_id</u>*, *name*, *street*, *city*)

  owner(*<u>license</u>*, *driver_id*)

  claim(*<u>driver_id</u>*, *<u>license</u>*, *<u>date</u>*, *description*, *amount*)
- Find customers with more claims than the average number of claims per customer
- This is an aggregate of another aggregate
- Each **`SELECT`** can only compute <u>one level</u> of aggregation
  - **`AVG(COUNT(*))`** is **not allowed** in SQL (or in relational algebra, so no big surprise)

Aggregates of Aggregates

- Two steps to find average number of claims
- Step 1:
  - Must compute a count of claims for each customer

```
SELECT COUNT(*) AS num_claims
  FROM claim GROUP BY driver_id
```

  - Then, compute the average in a second **SELECT**:

```
SELECT AVG(num_claims)
  FROM (SELECT COUNT(*) AS num_claims
        FROM claim GROUP BY driver_id) AS c
```

- This generates a single result
  - Can use it as a scalar subquery if we want.

Aggregates of Aggregates (2)

- Finally, can compute the full result:

```
SELECT driver_id, COUNT(*) AS num_claims
  FROM claim GROUP BY driver_id
HAVING num_claims >=
     (SELECT AVG(num_claims)
      FROM (SELECT COUNT(*) AS num_claims
                 FROM claim GROUP BY driver_id) AS c);
```

  - Comparison must be in **HAVING** clause
- This won't work:

```
SELECT driver_id, COUNT(*) AS num_claims
  FROM claim GROUP BY driver_id
HAVING num_claims = AVG(num_claims);
```

  - Tries to do two levels of aggregation in one **SELECT**

Alternative 1: Make a View

- Knowing each customer's total number of claims *could* be generally useful…
- Define a view for it:

```
CREATE VIEW claim_counts AS
  SELECT driver_id, COUNT(*) AS num_claims
  FROM claim GROUP BY driver_id;
```

  - Then the query becomes:

```
SELECT * FROM claim_counts
WHERE num_claims >
      (SELECT AVG(num_claims) FROM
  claim_counts)
```

  - View hides one level of aggregation

Alternative 2: Use **WITH** Clause

- **WITH** is like defining a view for a single statement
- Using **WITH**:

```
WITH claim_counts (driver_id, num_claims) AS (
     SELECT driver_id, COUNT(*)
     FROM claim GROUP BY name)
SELECT * FROM claim_counts
WHERE num_claims > (SELECT AVG(num_claims)
                    FROM claim_counts);
```

  - **WITH** doesn't pollute the database schema with a bunch of random views
  - Can specify multiple **WITH** clauses, too (see book)
  - (Unfortunately, MySQL doesn't support **WITH**...)

SQL Data Definition

- Specify table schemas using **`CREATE TABLE`**
  - Specify each column's name and domain
  - Can specify domain constraint: **`NOT NULL`**
  - Can specify key constraints
    - **`PRIMARY KEY`**
    - **`UNIQUE`** (candidate keys)
    - **`REFERENCES table (column)`** (foreign keys)
  - Key constraints can go in column declaration
  - Can also specify keys after all column decls.
- Be familiar with common SQL data types
  - **`INTEGER`**, **`CHAR`**, **`VARCHAR`**, date/time types, etc.

DDL Example

- Relation schema:

  car(*<u>license</u>*, *vin*, *make*, *model*, *year*)

  - *vin* is also a candidate key

- **CREATE TABLE** statement:

```
CREATE TABLE car (
  license CHAR(10)     PRIMARY KEY,
  vin     CHAR(30)     NOT NULL UNIQUE,
  make    VARCHAR(20)  NOT NULL,
  model   VARCHAR(20)  NOT NULL,
  year    INTEGER      NOT NULL
);
```

DDL Example (2)

- Relation schema:

  claim(*driver_id*, *license*, *date*, *description*, *amount*)

- **CREATE TABLE** statement:

```
CREATE TABLE claim (
  driver_id   CHAR(12),
  license     CHAR(10),
  date        TIMESTAMP,
  description VARCHAR(4000) NOT NULL,
  amount      NUMERIC(8,2),

  PRIMARY KEY (driver_id, license, date),
  FOREIGN KEY driver_id REFERENCES customer,
  FOREIGN KEY license REFERENCES car
);
```

Key Constraints and **NULL**

- Some key constraints automatically include **NOT NULL** constraints, but not all do.
- **PRIMARY KEY** constraints
  - Disallows **NULL** values
- **UNIQUE** constraints
  - Allows **NULL** values, unless you specify **NOT NULL**
- **FOREIGN KEY** constraints
  - Allows **NULL** values , unless you specify **NOT NULL**
- Understand how **NULL** values affect **UNIQUE** and **FOREIGN KEY** constraints that allow **NULL**s

Referential Integrity Constraints

- Unlike relational algebra, SQL DBs automatically enforce referential integrity constraints for you
  - You still need to perform operations in the correct order, though
- Same example as before:
  - Remove customer "Jones" from the bank database
  - DBMS will ensure that referential integrity is enforced, but you still have to delete rows from **`depositor`** and **`borrower`** tables first!

```
DELETE FROM depositor WHERE customer_name = 'Jones'
DELETE FROM borrower WHERE customer_name = 'Jones'
DELETE FROM customer WHERE customer_name = 'Jones'
```

Midterm Details

- Midterm posted online around Thursday, October 30
- Due Thursday, November 6 at 2:00AM
  (the usual time)

- No homework to do next week

- Good luck! ☺