W4111 – Introduction to Databases
Section 002, Spring 2021

Lecture 6: Finishing SQL and ER; Project Discussion

COLUMBIA | ENGINEERING
The Fu Foundation School of Engineering and Applied Science

W4111 – Introduction to Databases
Section 002, Spring 2021

Lecture 6: Finishing SQL and ER; Project Discussion

We will start in a couple of minutes.

Contents

Contents

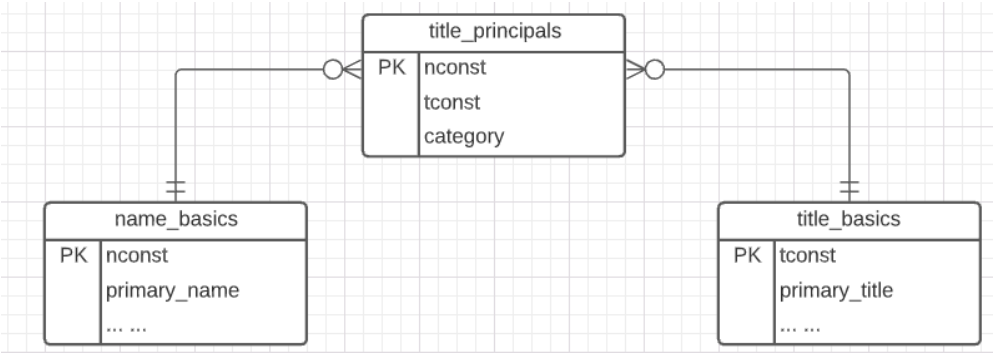
- Final SQL Concepts:
  - Two simple index examples.
  - Completing:
    - Functions.
    - Triggers.
    - Procedures.
  - Security

- Some Advanced ER → Relational Mapping
  - Inheritance (Continued)
  - Composite Attributes
  - Multivalued attributes.
  - N-ary relationships, for N > 2.

- Midterm walkthrough. (There will be recitations)

Two Simple Index Examples

Consider a Subset of the IMDB Data Model



- Three core entities:
  - name_basics: Core information about actors, writers, ... (10M rows)
  - title_basics: Core information about a movie, TV series, episode, ... (7M rows)
  - title_principals: Associative entity implement many-to-many relationship with properties. (42M rows)
- Associative entities have foreign keys to the entities they link.
- Let's look at some queries and scenarios:
  - Simple lookups.
  - JOINs
  - Foreign Keys

Switch to notebook.

Attribute

- The set of allowed values for each attribute is called the **domain** of the attribute
- Attribute values are (normally) required to be **atomic**; that is, indivisible
- The special value ***null*** is a member of every domain. Indicated that the value is “unknown”
- The null value causes complications in the definition of many operations

DFF Comments:

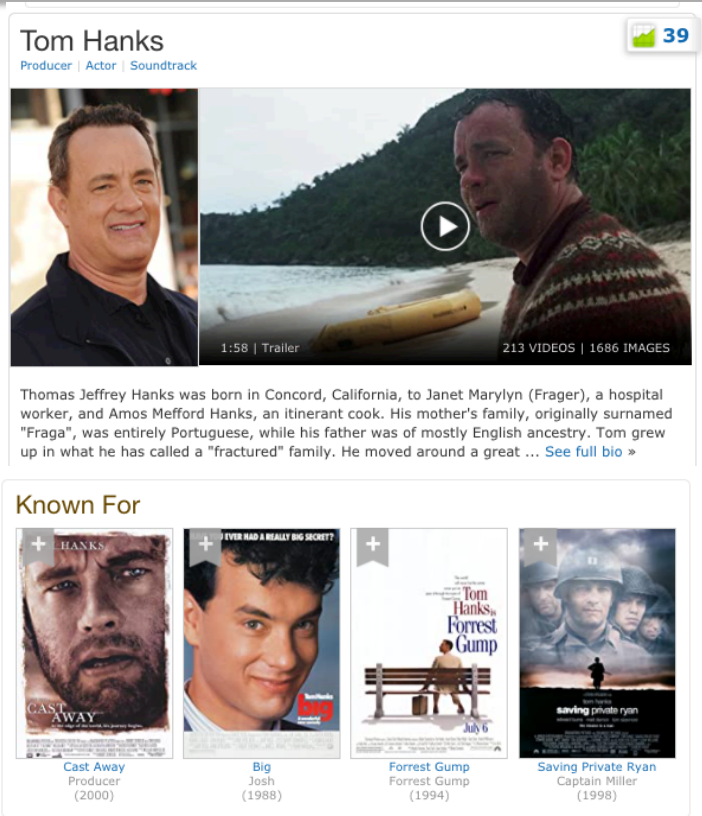
- Atomic and use of Null is important?
- I will explain the importance of atomic attributes and null in examples.

- Remember this comment from lecture 2? You will now see one reason why.

Complex Attributes

- Attribute types:
  - **Simple** and **composite** attributes.
  - **Single-valued** and **multivalued** attributes
    - Example: multivalued attribute: *phone_numbers*
  - **Derived** attributes
    - Can be computed from other attributes
    - Example: age, given date_of_birth
- **Domain** – the set of permitted values for each attribute

IMDB Known For



- Raw data when loaded into DB is:

Result Grid | Filter Rows: Search | Edit: | Export/Import:

nconst	primary_name	birth_year	death_year	primary_profession	known_for_titles
nm0000158	Tom Hanks	1956	NULL	producer,actor,soundtrack	tt0162222,tt0120815,tt0109830,tt0094737

- Known_for_titles is a *multivalued* attribute. It is 4 values in a set drawn from a domain.
- We saw some interesting index issues for a composite attribute.
- Multivalued attributes also introduce some interesting issues.
- Switch to notebook.

Multivalued and Composite Attributes

- Multivalued and composite attributes
  - Can result in some very strange SQL.
  - And are impossible to index properly.
  - Which results in awkward and slow queries.
- Multivalued and composite attributes can also compromise integrity.
  - Foreign keys are an example.
  - I cannot define a foreign key that ensures that:
    - Each element in “tt0162222,tt0120815,tt0109830,tt0094737”
    - Has a corresponding row in in another table.
- How do you handle these situations?
  - Decompose composite attributes into columns or a separate table.
  - Multi-valued attributes are a many-to-many pattern.

Functions, Procedures, Triggers

Concepts (Reminder)

Functions and Procedures

- Functions and procedures allow "business logic" to be stored in the database and executed from SQL statements.
- These can be defined either by the procedural component of SQL or by an external programming language such as Java, C, or C++.
- The syntax we present here is defined by the SQL standard.
  - Most databases implement nonstandard versions of this syntax.

Note:

- The programming language, runtime and tools for functions, procedures and triggers are not easy to use.
- My view is that calling external functions is an anti-pattern (bad idea).
  - External code degrades the reliability, security and performance of the database.
  - Databases are often mission critical and the heart of environments.

- We covered the language and functions in the last lecture.
  We will cover triggers in this lecture.

Triggers

Triggers

- A **trigger** is a statement that is executed automatically by the system as a side effect of a modification to the database.
- To design a trigger mechanism, we must:
  - Specify the conditions under which the trigger is to be executed.
  - Specify the actions to be taken when the trigger executes.
- Triggers introduced to SQL standard in SQL:1999, but supported even earlier using non-standard syntax by most databases.
  - Syntax illustrated here may not work exactly on your database system; check the system manuals

Triggering Events and Actions in SQL

- Triggering event can be **insert**, **delete** or **update**
- Triggers on update can be restricted to specific attributes
  - For example, **after update of** *takes* **on** *grade*
- Values of attributes before and after an update can be referenced
  - **referencing old row as** : for deletes and updates
  - **referencing new row as** : for inserts and updates
- Triggers can be activated before an event, which can serve as extra constraints. For example, convert blank grades to null.

```
create trigger setnull_trigger before update of takes
referencing new row as nrow
for each row
    when (nrow.grade = ' ')
begin atomic
        set nrow.grade = null;
end;
```

Trigger to Maintain credits_earned value

- **create trigger** *credits_earned* **after update of** *takes* **on** (*grade*)
  referencing new row as *nrow*
  referencing old row as *orow*
  for each row
  when *nrow.grade* <> 'F' **and** *nrow.grade* **is not null**
  and (*orow.grade* = 'F' **or** *orow.grade* **is null**)
  begin atomic
  update *student*
  set *tot_cred*= *tot_cred* +
  (**select** *credits*
  from *course*
  where *course.course_id*= *nrow.course_id*)
  where *student.id* = *nrow.id*;
  end;

Statement Level Triggers

- Instead of executing a separate action for each affected row, a single action can be executed for all rows affected by a transaction
  - Use **for each statement** instead of **for each row**
  - Use **referencing old table** or **referencing new table** to refer to temporary tables (called ***transition tables***) containing the affected rows
  - Can be more efficient when dealing with SQL statements that update a large number of rows

When Not To Use Triggers

- Triggers were used earlier for tasks such as
  - Maintaining summary data (e.g., total salary of each department)
  - Replicating databases by recording changes to special relations (called **change** or **delta** relations) and having a separate process that applies the changes over to a replica
- There are better ways of doing these now:
  - Databases today provide built in materialized view facilities to maintain summary data
  - Databases provide built-in support for replication
- Encapsulation facilities can be used instead of triggers in many cases
  - Define methods to update fields
  - Carry out actions as part of the update methods instead of through a trigger

When Not To Use Triggers (Cont.)

- Risk of unintended execution of triggers, for example, when
  - Loading data from a backup copy
  - Replicating updates at a remote site
  - Trigger execution can be disabled before such actions.
- Other risks with triggers:
  - Error leading to failure of critical transactions that set off the trigger
  - Cascading execution

Materialized Views

- Certain database systems allow view relations to be physically stored.
  - Physical copy created when the view is defined.
  - Such views are called **Materialized view**:
- If relations used in the query are updated, the materialized view result becomes out of date
  - Need to **maintain** the view, by updating the view whenever the underlying relations are updated.

Trigger Example – Violating when not to Use

- Materialized view support, especially update, is inconsistent.
- We are going to use triggers to keep a materialized view (copy table) up to date.

REFRESHING MATERIALIZED VIEWS

Materialized Views can be refreshed in different kinds. They can be refreshed:

- never (only once in the beginning, for static data only)
- on demand (for example once a day, for example after nightly load)
- immediately (after each statement)

A refresh can be done in the following ways:

- completely (slow, full from scratch)
- deferred (fast, by a log table)

By storing the change information in a log table. Also some snapshots or time delayed states can be produced:

- refresh up to date
- refresh full

Back to the Notebook.
Write a Trigger.

Summary

Comparison

comparing triggers, functions, and procedures

	triggers	functions	stored procedures
change data	yes	no	yes
return value	never	always	sometimes
how they are called	reaction	in a statement	exec

lynda.com

Comparison – Some Details

Sr.No.	User Defined Function	Stored Procedure
1	Function must return a value.	Stored Procedure may or not return values.
2	Will allow only Select statements, it will not allow us to use DML statements.	Can have select statements as well as DML statements such as insert, update, delete and so on
3	It will allow only input parameters, doesn't support output parameters.	It can have both input and output parameters.
4	It will not allow us to use try-catch blocks.	For exception handling we can use try catch blocks.
5	Transactions are not allowed within functions.	Can use transactions within Stored Procedures.
6	We can use only table variables, it will not allow using temporary tables.	Can use both table variables as well as temporary table in it.
7	Stored Procedures can't be called from a function.	Stored Procedures can call functions.
8	Functions can be called from a select statement.	Procedures can't be called from Select/Where/Having and so on statements. Execute/Exec statement can be used to call/execute Stored Procedure.
9	A UDF can be used in join clause as a result set.	Procedures can't be used in Join clause

A *trigger* has capabilities like a procedure, except ...

- You do not call it. The DB engine calls it before or after an INSERT, UPDATE, DELETE.
- The inputs are the list of incoming new, modified rows.
- The outputs are the modified versions of the new or modified rows.

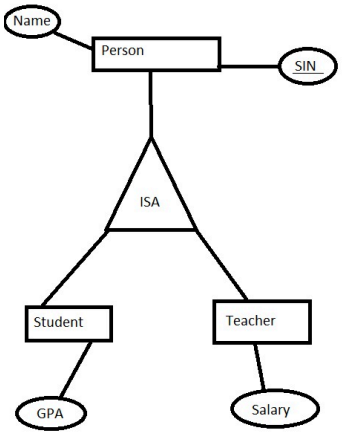
Complex Examples
Columbia Courses, Sections, Enrollments
(Part II – Students and Faculty)

IsA, Inheritance, Generalization, Specialization

Faculty and Students

- We modelled *course.*
- To *enroll* students in courses, we need ...
  - Sections
  - Faculty
  - Students
- Let's focus on the "people aspect." We have
  - People
  - A *Faculty* **is a** *People.*
  - A *Student* **is a** *People.*

ER Notation



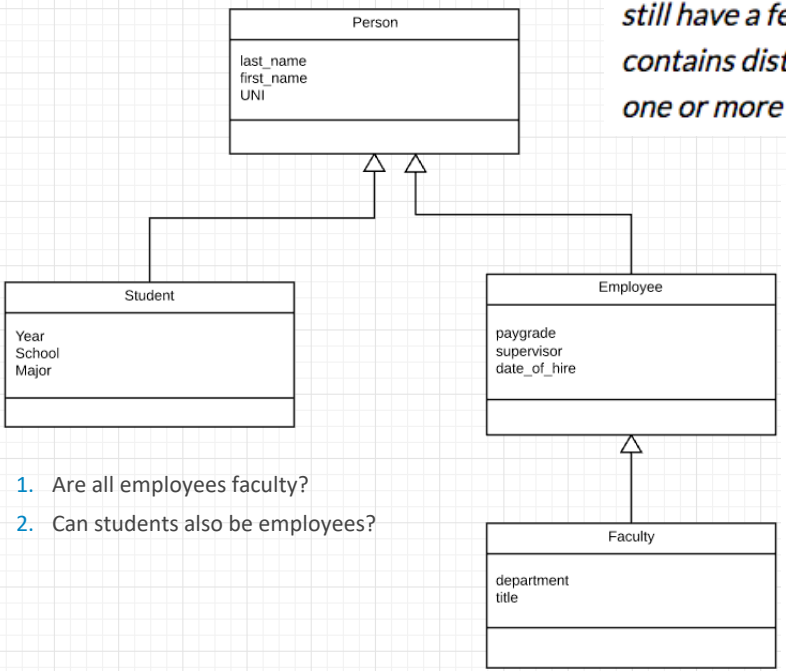
- "In object-oriented programming, inheritance is the mechanism of basing an object or class upon another object (prototype-based inheritance) or class (class-based inheritance), retaining similar implementation. Also defined as deriving new classes (sub classes) from existing ones such as super class or base class and then forming them into a hierarchy of classes."
- This is a new type of relationship in ER modeling.

Specialization

- Top-down design process; we designate sub-groupings within an entity set that are distinctive from other entities in the set.
- These sub-groupings become lower-level entity sets that have attributes or participate in relationships that do not apply to the higher-level entity set.
- Depicted by a *triangle* component labeled ISA (e.g., *instructor* "is a" *person*).
- **Attribute inheritance** – a lower-level entity set inherits all the attributes and relationship participation of the higher-level entity set to which it is linked.

Inheritance, IsA, Specialization

In the process of designing our entity relationship diagram for a database, we may find that attributes of two or more entities overlap, meaning that these entities seem very similar but still have a few differences. In this case, we may create a subtype of the parent entity that contains distinct attributes. A parent entity becomes a supertype that has a relationship with one or more subtypes.



1. Are all employees faculty?
2. Can students also be employees?

The subclass association line is labeled with specialization constraints. Constraints are described along two dimensions:

1 incomplete/complete

- In an **incomplete** specialization only some instances of the parent class are specialized (have unique attributes). Other instances of the parent class have only the common attributes.
- In a **complete** specialization, every instance of the parent class has one or more unique attributes that are not common to the parent class.

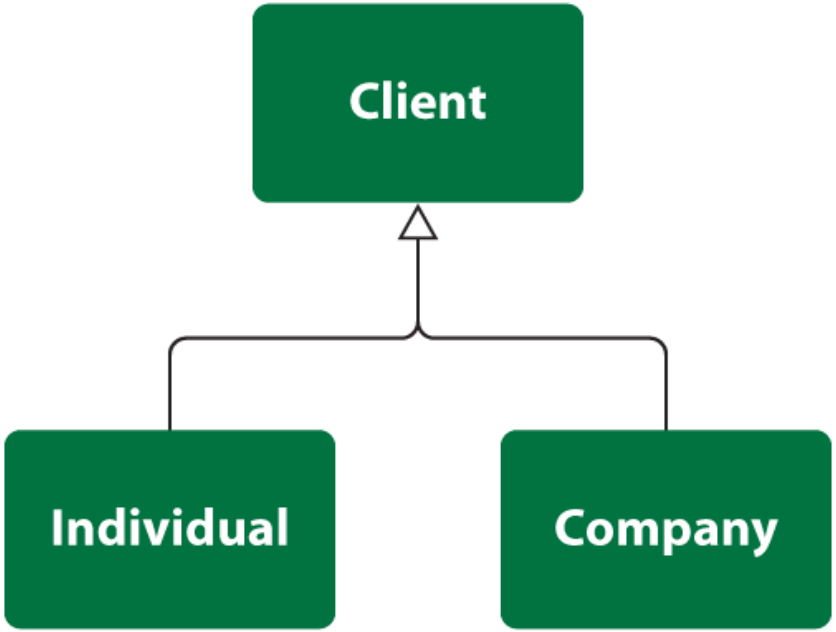
2 disjoint/overlapping

- In a **disjoint** specialization, an object could be a member of only one specialized subclass.
- In an **overlapping** specialization, an object could be a member of more than one specialized subclass.

http://www.vertabelo.com/blog/technical-articles/inheritance-in-a-relational-database

Simpler Example

In class Client we distinguish two subtypes: Individual and Company. This specialization is disjoint (client can be an individual or a company) and complete (these are all possible subtypes for supertype).

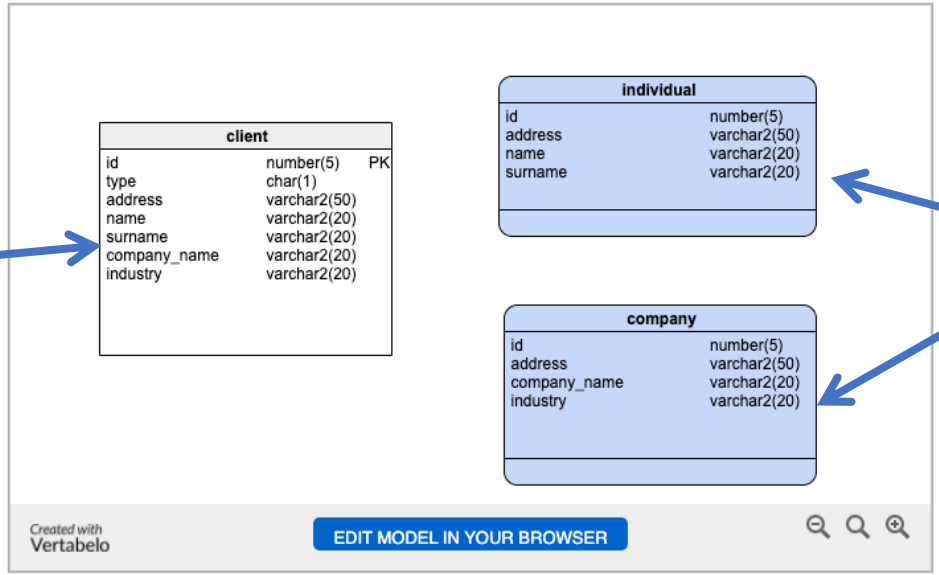


One Table Implementation

One table implementation

In a one table implementation, table `client` has attributes of both types.

The diagram below shows the table `client` and two views: `individual` and `company`:



Tables

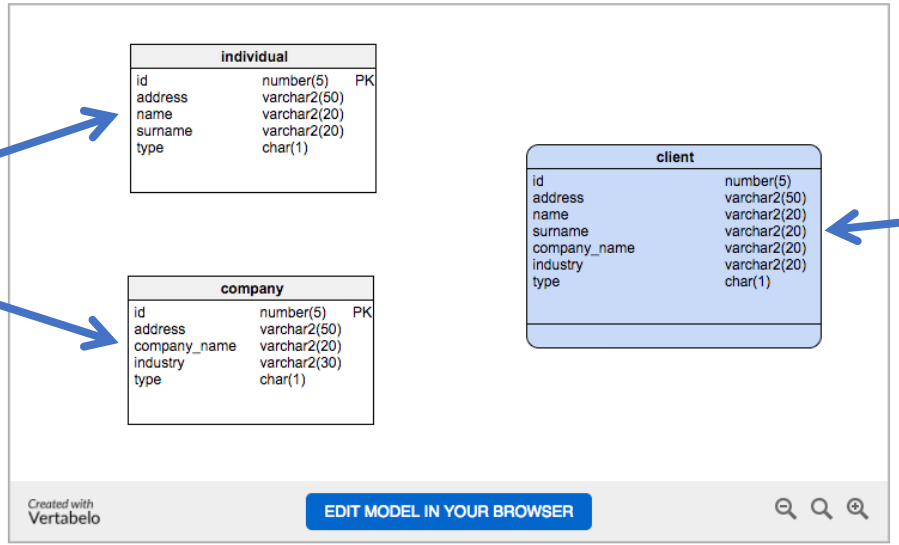
Views

Two Table Implementation

Two-table implementation

In a two-table implementation, we create a table for each of the subtypes. Each table gets a column for all attributes of the supertype and also a column for each attribute belonging to the subtype. Access to information in this situation is limited, that's why it is important to create a view that is the union of the tables. We can add an additional attribute called 'type' that describes the subtype.

The diagram below presents two tables, `individual` and `company`, and a view (the blue one) called `client`.

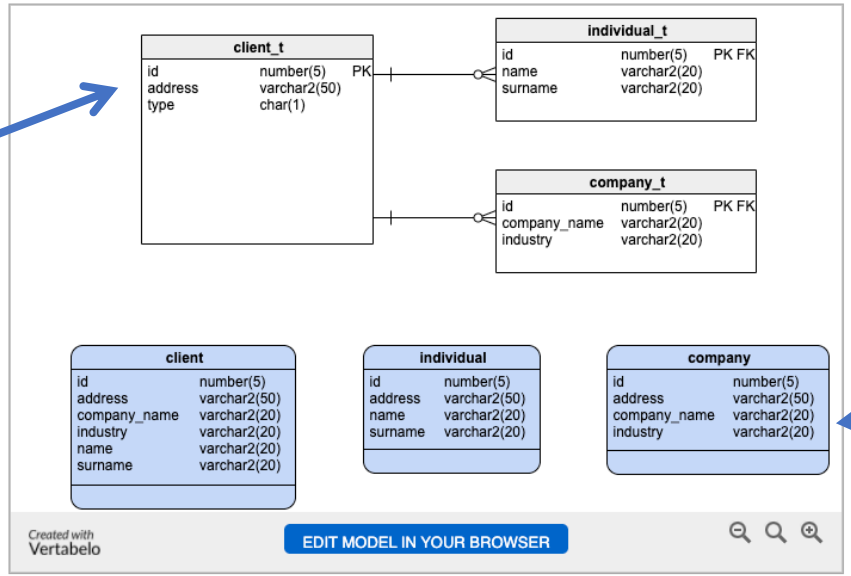


Two Table Implementation

Three-table implementation

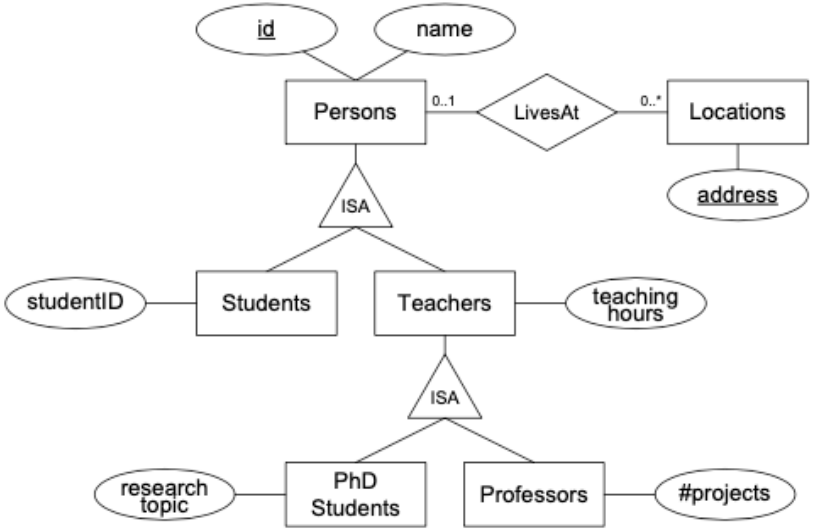
In a third solution we create a single table `client_t` for the parent table, containing common attributes for all subtypes, and tables for each subtype (`individual_t` and `company_t`) where the primary key in `client_t` (base table) determines foreign keys in dependent tables. There are three views: `client`, `individual` and `company`.

Tables

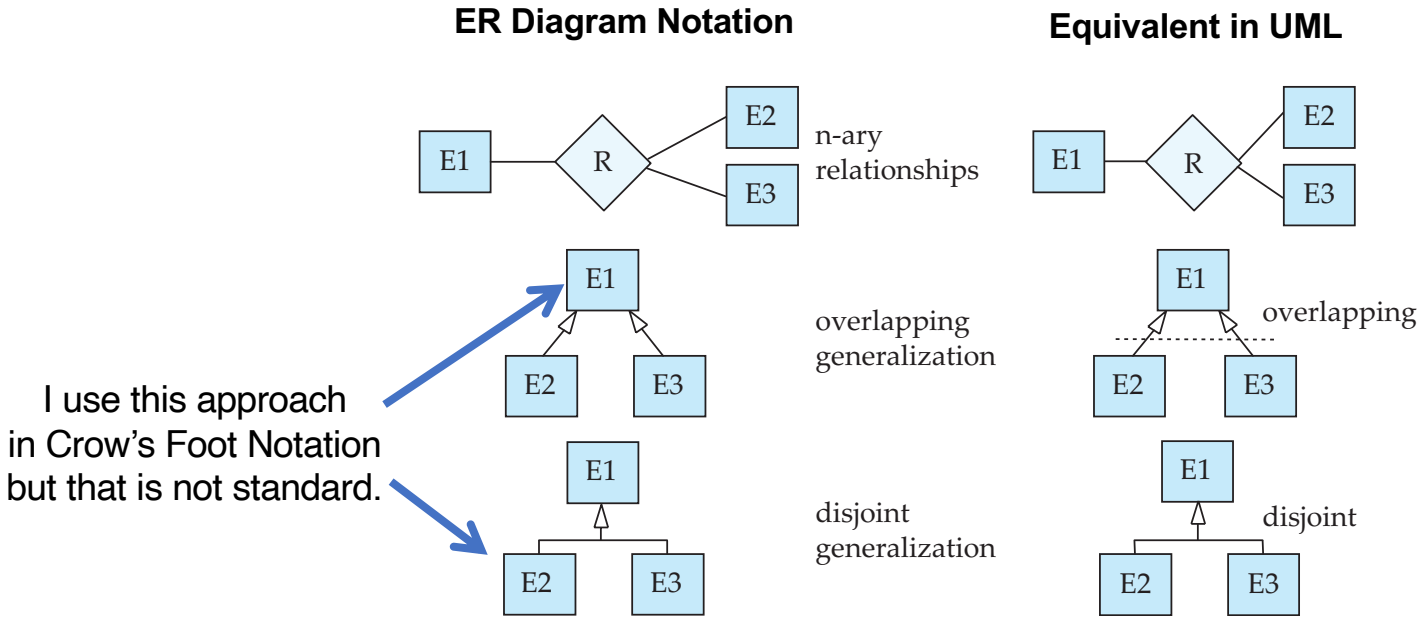


Views

ISA Relationship



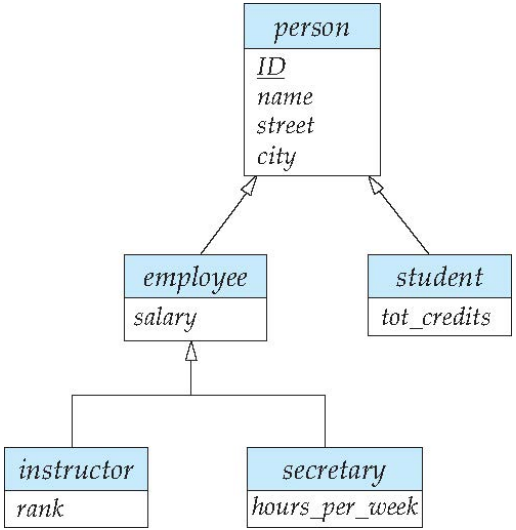
ER vs. UML Class Diagrams



* Generalization can use merged or separate arrows independent of disjoint/overlapping

Specialization Example

- **Overlapping** – *employee* and *student*
- **Disjoint** – *instructor* and *secretary*
- Total and partial



Generalization

- **A bottom-up design process** – combine a number of entity sets that share the same features into a higher-level entity set.
- Specialization and generalization are simple inversions of each other; they are represented in an E-R diagram in the same way.
- The terms specialization and generalization are used interchangeably.

Completeness constraint

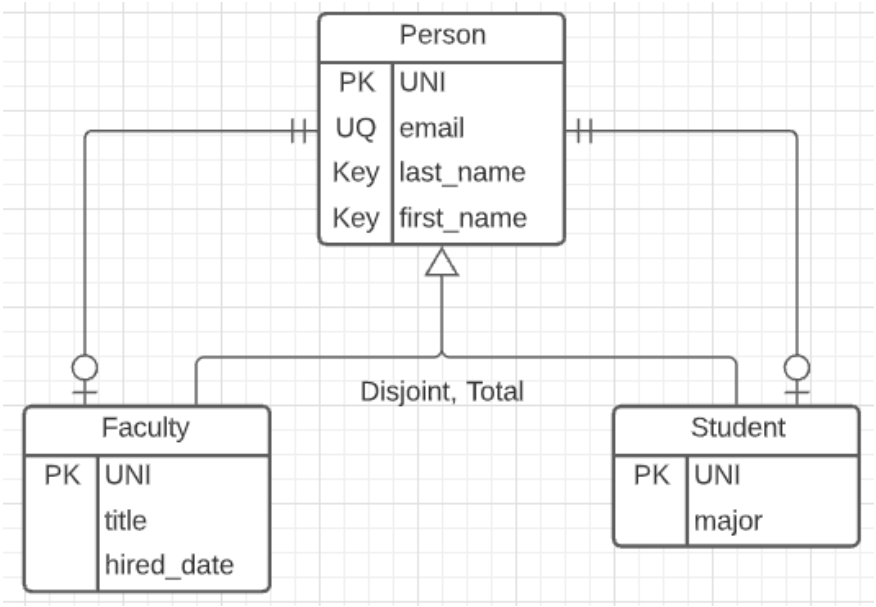
- **Completeness constraint** -- specifies whether or not an entity in the higher-level entity set must belong to at least one of the lower-level entity sets within a generalization.
  - **total**: an entity must belong to one of the lower-level entity sets
  - **partial**: an entity need not belong to one of the lower-level entity sets

Completeness constraint (Cont.)

- Partial generalization is the default.
- We can specify total generalization in an ER diagram by adding the keyword **total** in the diagram and drawing a dashed line from the keyword to the corresponding hollow arrow-head to which it applies (for a total generalization), or to the set of hollow arrow-heads to which it applies (for an overlapping generalization).
- The *student* generalization is total: All student entities must be either graduate or undergraduate. Because the higher-level entity set arrived at through generalization is generally composed of only those entities in the lower-level entity sets, the completeness constraint for a generalized higher-level entity set is usually total

Worked Example

Person, Faculty, Student



- Model
  - Complete:
    - Every entity is a Faculty or a Student.
    - There are no entities that are just Person.
  - Disjoint:
    - Either a Faculty or a Student
    - But not both.
  - 3-Table solution.
- Relationships:
  - Faculty-Person is one-to-one.
  - Student-Person is one-on-one.
  - Person is exactly 1.
  - Faculty or Student is 0-1.
- There is no easy way to handle the fact that one of Faculty or Student must exist.

The UNI

- The UNI creates some interesting challenges:
  - A given UNI is in two tables: Person, Faculty or Person, Student.
  - A given UNI can only be in the Faculty table or the Student table.
  - My algorithm for generating the UNI is:
    - Prefix: Concatenate
      - First two characters of first name.
      - First two characters of the last name
    - Suffix: Add 1 to the number of existing UNIs with the same prefix.
    - Concatenate the prefix and suffix.
  - Immutable: Cannot change.
- Implementing the semantics in applications accessing the data is error prone.
  - The more places you implement something, the more places you can mess up.
  - You have to find all the apps and change them if the logic changes.
  - So, we are going to implement using a function and trigger.

Table Definitions

```
CREATE TABLE `person_three` (
  `first_name` varchar(128) NOT NULL,
  `last_name` varchar(128) NOT NULL,
  `preferred_email` varchar(256) DEFAULT NULL,
  `uni` varchar(12) NOT NULL,
  `uni_email` varchar(256) GENERATED ALWAYS AS (concat(`uni`,_utf8mb4'@columbia.edu')) STORED,
  `preferred_name` varchar(128) DEFAULT NULL,
  PRIMARY KEY (`uni`),
  UNIQUE KEY `uni_email_UNIQUE` (`uni_email`),
  CONSTRAINT `person_three_chk_2` CHECK (((`preferred_email` is null) or
      ((`preferred_email` like _utf8mb4'%@%') and (not((`preferred_email` like _utf8mb4'%@%@%'))))))
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

CREATE TABLE `faculty_three` (
  `uni` varchar(12) NOT NULL,
  `hire_date` year NOT NULL,
  `title` enum('Adjunct Professor','Associate Professor','Assistant Professor','Professor','Professor of Practice','Lecturer','Senior Lecturer') NOT NULL,
  PRIMARY KEY (`uni`),
  CONSTRAINT `faculty_person` FOREIGN KEY (`uni`) REFERENCES `person_three` (`uni`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

CREATE TABLE `student_three` (
  `uni` varchar(12) NOT NULL,
  `major` varchar(4) NOT NULL,
  PRIMARY KEY (`uni`),
  CONSTRAINT `student_person` FOREIGN KEY (`uni`) REFERENCES `person_three` (`uni`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
```

Function

Switch to notebook.

```
CREATE DEFINER=`root`@`localhost` FUNCTION `generate_uni`(first_name VARCHAR(128),
    last_name VARCHAR(128)) RETURNS varchar(12) CHARSET utf8mb4
  DETERMINISTIC
BEGIN

  DECLARE result      VARCHAR(12);
  DECLARE fn_prefix   VARCHAR(2);
  DECLARE ln_prefix   VARCHAR(2);
  DECLARE uni_prefix  VARCHAR(5);
  DECLARE uni_count   INT;

  SET fn_prefix = SUBSTRING(first_name, 1, 2);
  SET ln_prefix = SUBSTRING(last_name, 1, 2);
  SET fn_prefix = LOWER(fn_prefix);
  SET ln_prefix = LOWER(ln_prefix);

  SET uni_prefix = CONCAT(fn_prefix, ln_prefix, '%');

  SET uni_count = (SELECT count(*) FROM person_three WHERE uni LIKE uni_prefix);

  SET result = CONCAT(fn_prefix, ln_prefix, uni_count);

RETURN result;
END
```

Design Pattern for Inheritance Integrity

- So far, we have created:
  - A three-table solution to a simple inheritance design problem.
  - Core elements are:
    - Function for computing UNIs.
    - A view to query the concrete class without exposing inheritance.
    - A procedure to create one of the concrete classes and the base classes.
- But, We would still need to implement the update and delete stored procedures to ensure encapsulation and integrity. This prevents users from corrupting the data by removing the need to understand the model and write SQL scripts.
- But, what if someone mistakenly uses the tables directly with SQL statements. That would still cause integrity problems.
- We can use identity, roles and permissions.

Hands On – Write a Stored Procedure

- Create Student, Create Faculty

Security

Security Concepts (Terms from Wikipedia)

- Definitions:
  - "A (digital) identity is information on an entity used by computer systems to represent an external agent. That agent may be a person, organization, application, or device."
  - "Authentication is the act of proving an assertion, such as the identity of a computer system user. In contrast with identification, the act of indicating a person or thing's identity, authentication is the process of verifying that identity."
  - "Authorization is the function of specifying access rights/privileges to resources, ... More formally, "to authorize" is to define an access policy. ... During operation, the system uses the access control rules to decide whether access requests from (authenticated) consumers shall be approved (granted) or disapproved.
  - "Within an organization, roles are created for various job functions. The permissions to perform certain operations are assigned to specific roles. Members or staff (or other system users) are assigned particular roles, and through those role assignments acquire the permissions needed to perform particular system functions."
  - "In computing, privilege is defined as the delegation of authority to perform security-relevant functions on a computer system. A privilege allows a user to perform an action with security consequences. Examples of various privileges include the ability to create a new user, install software, or change kernel functions."

- SQL and relational database management systems implementing security by:
  - Creating identities and authentication policies.
  - Creating roles and assigning identities to roles.
  - Granting and revoking privileges to/from roles and identities.

Authorization

- We may assign a user several forms of authorizations on parts of the database.
  - **Read** - allows reading, but not modification of data.
  - **Insert** - allows insertion of new data, but not modification of existing data.
  - **Update** - allows modification, but not deletion of data.
  - **Delete** - allows deletion of data.
- Each of these types of authorizations is called a **privilege**. We may authorize the user all, none, or a combination of these types of privileges on specified parts of a database, such as a relation or a view.

Authorization (Cont.)

- Forms of authorization to modify the database schema
  - **Index** - allows creation and deletion of indices.
  - **Resources** - allows creation of new relations.
  - **Alteration** - allows addition or deletion of attributes in a relation.
  - **Drop** - allows deletion of relations.

Authorization Specification in SQL

- The **grant** statement is used to confer authorization

  grant <privilege list> **on** <relation or view > **to** <user list>
- <user list> is:
  - a user-id
  - **public**, which allows all valid users the privilege granted
  - A role (more on this later)
- Example:
  - **grant select on** *department* **to** Amit, Satoshi
- Granting a privilege on a view does not imply granting any privileges on the underlying relations.
- The grantor of the privilege must already hold the privilege on the specified item (or be the database administrator).

Privileges in SQL

- **select**: allows read access to relation, or the ability to query using the view
  - Example: grant users U_1, U_2, and U_3 **select** authorization on the *instructor* relation:

    grant select on *instructor* **to** U_1, U_2, U_3
- **insert**: the ability to insert tuples
- **update**: the ability to update using the SQL update statement
- **delete**: the ability to delete tuples.
- **all privileges**: used as a short form for all the allowable privileges

Revoking Authorization in SQL

- The **revoke** statement is used to revoke authorization.

  revoke <privilege list> **on** <relation or view> **from** <user list>

- Example:

  revoke select on *student* **from** U_1, U_2, U_3

- <privilege-list> may be **all** to revoke all privileges the revokee may hold.
- If <revokee-list> includes **public,** all users lose the privilege except those granted it explicitly.
- If the same privilege was granted twice to the same user by different grantees, the user may retain the privilege after the revocation.
- All privileges that depend on the privilege being revoked are also revoked.

Roles

- A **role** is a way to distinguish among various users as far as what these users can access/update in the database.
- To create a role we use:

  create a role <name>
- Example:
  - **create role** instructor
- Once a role is created we can assign “users” to the role using:
  - **grant** <role> **to** <users>

Roles Example

- **create role** instructor;
- **grant** *instructor* **to** Amit**;**
- Privileges can be granted to roles:
  - **grant select on** *takes* **to** *instructor*;
- Roles can be granted to users, as well as to other roles
  - **create role** *teaching_assistant*
  - **grant** *teaching_assistant* **to** *instructor*;
    - *Instructor* inherits all privileges of *teaching_assistant*
- Chain of roles
  - **create role** *dean*;
  - **grant** *instructor* **to** *dean*;
  - **grant** *dean* **to** Satoshi;

Authorization on Views

- **create view** *geo_instructor* **as**
  (**select** *
  from *instructor*
  where *dept_name* = 'Geology');
- **grant select on** *geo_instructor* **to** *geo_staff*
- Suppose that a *geo_staff* member issues
  - **select** *
    from *geo_instructor*;
- What if
  - *geo_staff* does not have permissions on *instructor?*
  - Creator of view did not have some permissions on *instructor?*

Other Authorization Features

- **references** privilege to create foreign key
  - **grant reference** (*dept_name*) **on** *department* **to** Mariano;
  - Why is this required?
- transfer of privileges
  - **grant select on** *department* **to** Amit **with grant option**;
  - **revoke select on** *department* **from** Amit, Satoshi **cascade**;
  - **revoke select on** *department* **from** Amit, Satoshi **restrict**;
  - And more!

Note:

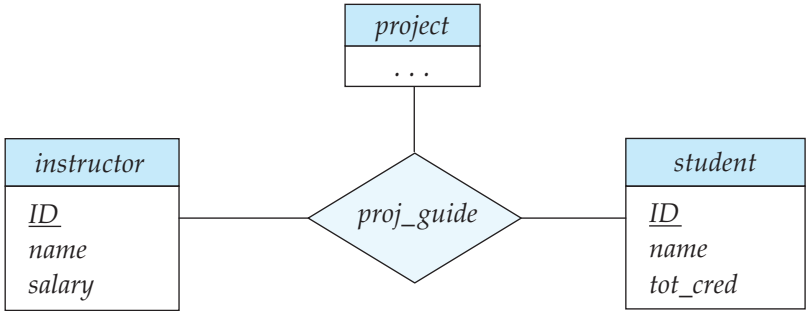
- Like in many other cases, SQL DBMS have product specific variations.

Switch to notebook.

More Complex ER Mapping

Non-binary Relationship Sets

- Most relationship sets are binary
- There are occasions when it is more convenient to represent relationships as non-binary.
- E-R Diagram with a Ternary Relationship



Let's do this.

1. Crow's Foot Diagram
2. SQL Schema
3. Indexes

Weak Entity Sets

- Consider a *section* entity, which is uniquely identified by a *course_id*, *semester, year*, and *sec_id*.
- Clearly, section entities are related to course entities. Suppose we create a relationship set *sec_course* between entity sets *section* and *course*.
- Note that the information in *sec_course* is redundant, since *section* already has an attribute *course_id*, which identifies the course with which the section is related.
- One option to deal with this redundancy is to get rid of the relationship s*ec_course*; however, by doing so the relationship between *section* and *course* becomes implicit in an attribute, which is not desirable.

Let's look at *classicmodels.*

Weak Entity Sets (Cont.)

- An alternative way to deal with this redundancy is to not store the attribute *course_id* in the *section* entity and to only store the remaining attributes *section_id*, *year*, and *semester.*
  - However, the entity set *section* then does not have enough attributes to identify a particular *section* entity uniquely
- To deal with this problem, we treat the relationship *sec_course* as a special relationship that provides extra information, in this case, the *course_id*, required to identify *section* entities uniquely.
- A **weak entity set** is one whose existence is dependent on another entity, called its **identifying entity**
- Instead of associating a primary key with a weak entity, we use the identifying entity, along with extra attributes called **discriminator** to uniquely identify a weak entity.

Weak Entity Sets (Cont.)

- An entity set that is not a weak entity set is termed a **strong entity set**.
- Every weak entity must be associated with an identifying entity; that is, the weak entity set is said to be **existence dependent** on the identifying entity set.
- The identifying entity set is said to **own** the weak entity set that it identifies.
- The relationship associating the weak entity set with the identifying entity set is called the **identifying relationship**.
- Note that the relational schema we eventually create from the entity set *section* does have the attribute *course_id*, for reasons that will become clear later, even though we have dropped the attribute *course_id* from the entity set *section.*

Some Data Cleanup Tasks

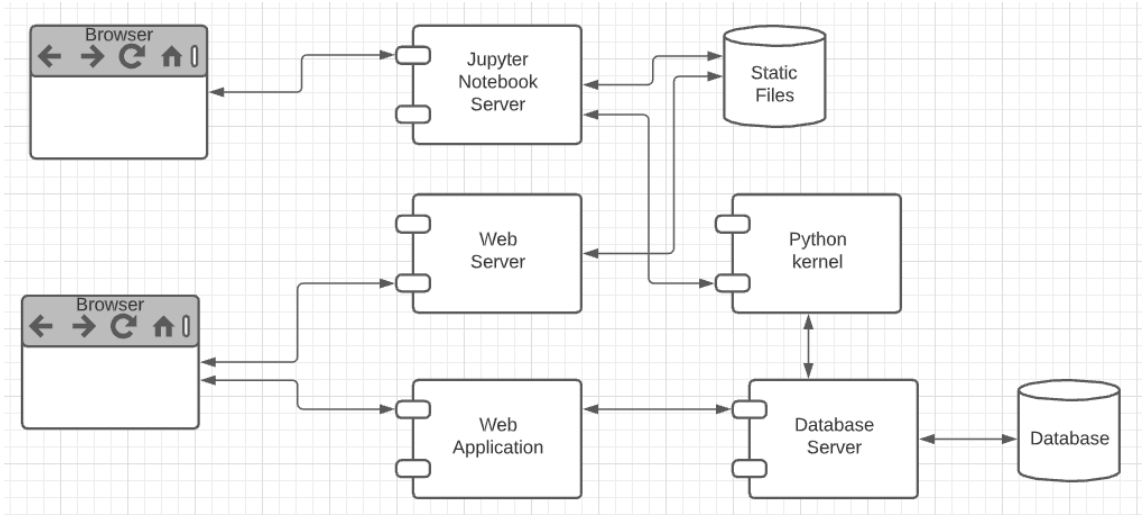
Some Worked Data Cleanup Tasks

- Country, City, State in Lahmans People.
- Harry Potter data.

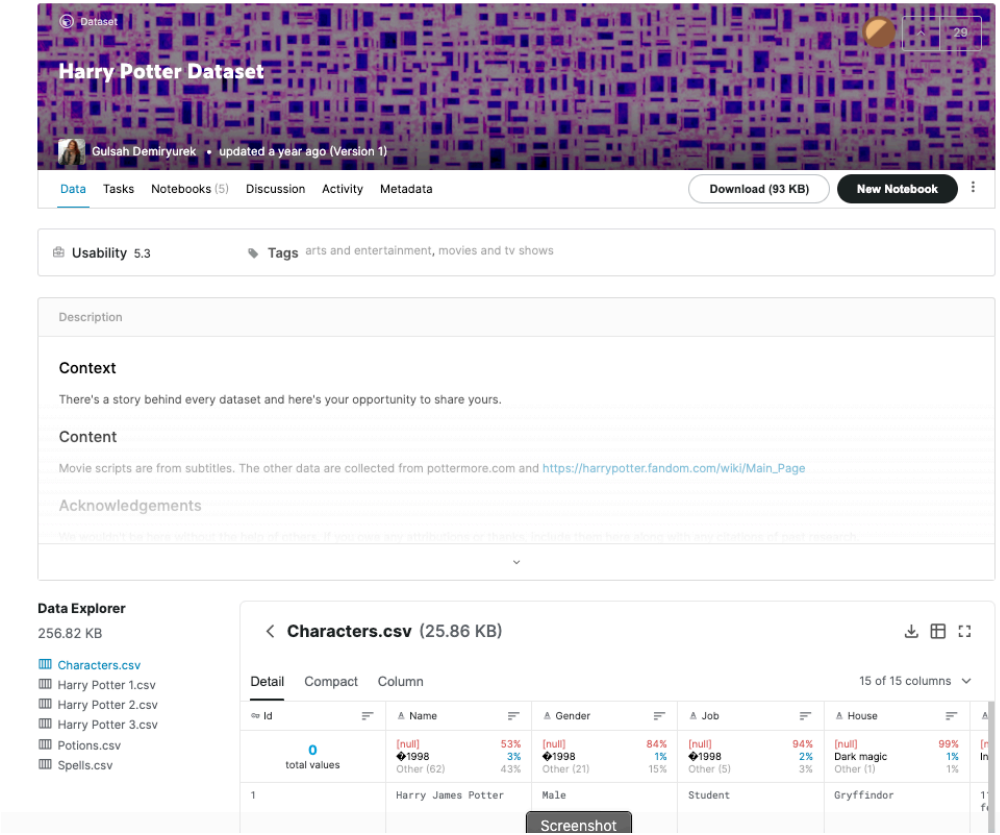
Harry Potter
(Slides and Notebook from Lecture 2)

Harry Potter Character Information Application

- Dataset:
  - https://www.kaggle.com/gulsahdemiryurek/harry-potter-dataset
  - Six “comma separated value (CSV)” files. We will start with *Characters.csv.*
- Tasks: (Both tasks start with *importing and engineering the data.)*
  - Build an interactive web application for *create-retrieve-update-delete* entries. The application must enforce *semantic integrity.*
  - Build a Jupyter Notebook that does some interesting visualization and analysis.
- Elements:
  - Content/applications in a browser.
  - Jupyter Notebook Server is both:
    - A web server
    - A (specialized) web application server.
  - Web server delivers HTML, images, ... to the browser.
  - Web application server executes an
    - Application server framework (Flask)
    - The Python code implementing the app.
  - The Jupyter Notebook Server runs code cells in a Python kernel.
  - I am mostly going to ignore the browser and content.



The Data – Characters.csv



15 columns:

- Id
- Name
- Gender
- Job
- House
- Wand
- Patronus
- Species
- Blood status
- Hair colour
- Eye colour
- Loyalty
- Skills
- Birth
- Death

- Downloaded the file.
- The columns are separated by ‘;’
- Despite being “Comma” Separated Values, other separators are common.

- Loaded using DataGrip into MySQL
  - Connect
  - Create Schema
  - Use Schema
  - Use the data load tool.

But First, Let's Look at the Data

There are some weird characters, no pun intended.

- 9º" Chestnut dragon heartstring
- Dumbledore's Army |Hogwarts School of Witchcraft and Wizardry
- 9Ω" Fir dragon heartstring
- Professor†of†Transfiguration†| Head of Gryffindor

- How did this happen?
- What do we do?

	Female	Professor†of†Transfiguration†| Head of Gryffindor	Gryffindor	9Ω" Fir dragon heartstring
	Female		Gryffindor	Unknown
	Male	Head of the†Misuse of Muggle Artefacts Office	Gryffindor	Unknown
	Male	Defence Against the Dark Arts(1991-1992)	Ravenclaw	9" Alder unicorn hair bendy
	Female	Student	Ravenclaw	Unknown
	Female	Student	Ravenclaw	Unknown
	Male	Defence Against the Dark Arts(1992-1993)	Ravenclaw	9" Cherry dragon heartstring
	Male	Professor of Charms	Head of Ravenclaw	Ravenclaw	Unknown
	Female	Professor†of†Divination	Ravenclaw	9 Ωî hazel unicorn hair core
	Male	Wandmaker	Ravenclaw	12æ" Hornbeam dragon heartstring
tle)	Female	Student	Ravenclaw	Unknown
	Female	Student	Ravenclaw	Unknown
	Male	Student	Ravenclaw	Unknown
	Female	Student	Ravenclaw	Unknown
	Male	Student	Ravenclaw	Unknown
	Male	Student	Ravenclaw	Unknown
	Male	Professor of Potions	Head of Slytherin	Slytherin	Unknown
	Male	Student	Slytherin	10" Hawthorn unicorn hair
	Male	Student	Slytherin	Unknown
	Male	Student	Slytherin	Unknown
	Female		Slytherin	12æ" Walnut dragon heartstring
	Female	Professor†of†Defence Against the Dark Arts†| Department of Magical Law Enforcement	Slytherin	8" Birch dragon heartstring
		Professor of Potions	Slytherin	10º" Cedar dragon heartstring fairly flexible
	Male	School Governor	Slytherin	Elm and dragon heartstring
	Female		Slytherin	Unknown
	Male		Slytherin	Unknown
	Female	Student	Slytherin	Unknown

Analysis (I)

- Some of the issues are character encoding/set issues:
  - "In computing, data storage, and data transmission, character encoding is used to represent a repertoire of characters by some kind of encoding system that assigns a number to each character for digital representation." (https://en.wikipedia.org/wiki/Character_encoding)
  - "A character set is a collection of characters that might be used by multiple languages." (https://en.wikipedia.org/wiki/Character_encoding)
- This will be a common issue when dealing with string data that you import to build your database.
- This is especially true if the data comes from "data scraping".
  - "Data scraping is a technique in which a computer program extracts data from human-readable output coming from another program." (https://en.wikipedia.org/wiki/Data_scraping)
  - There are several types of scraping: screen scraping, web scraping, ...
- That is what happened here. The data comes from web scraping a wiki page.

Content

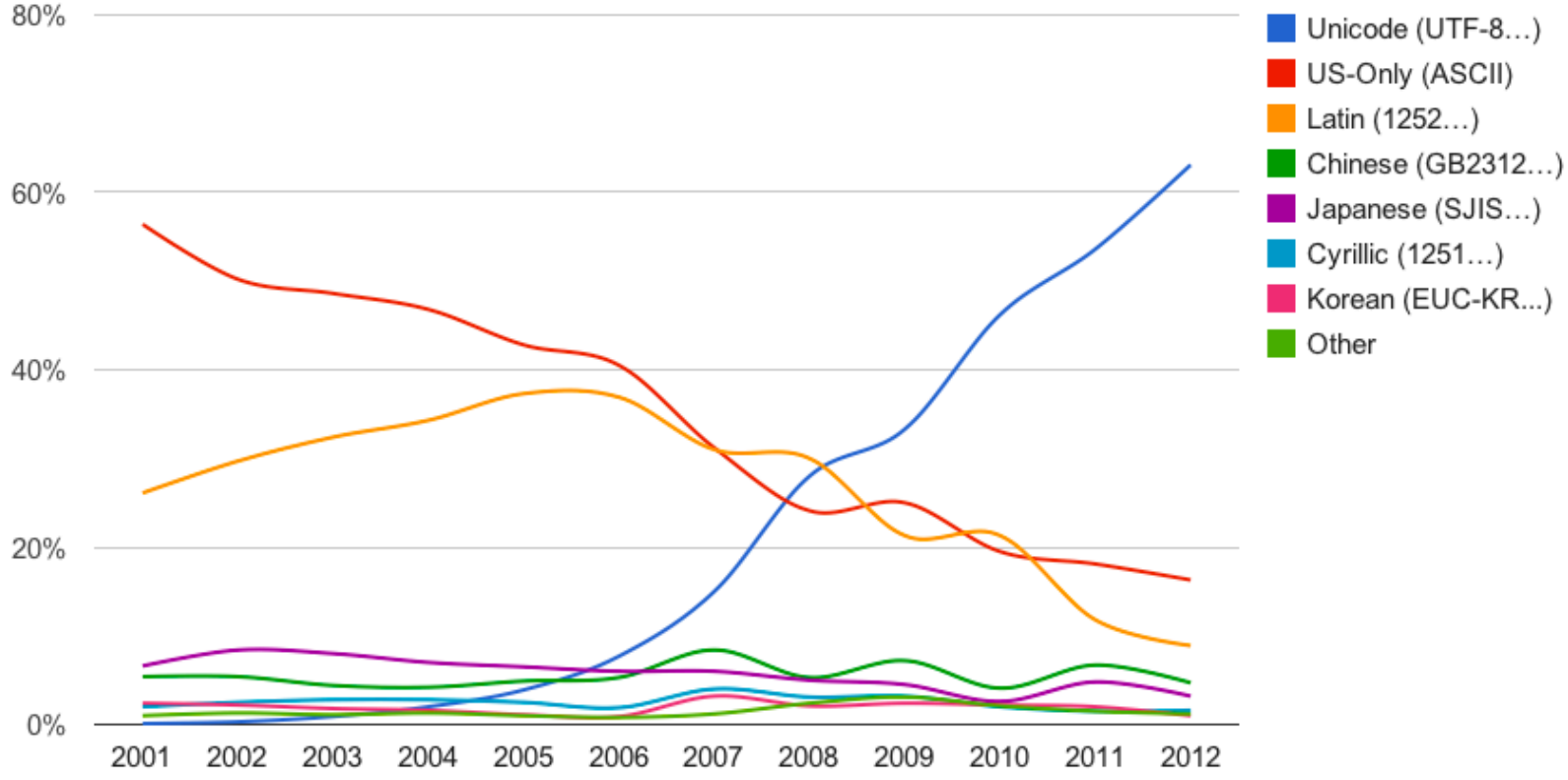
Movie scripts are from subtitles. The other data are collected from pottermore.com and https://harrypotter.fandom.com/wiki/Main_Page

Character Encodings/Character Sets

Common character encodings [edit]

- ISO 646
  - ASCII
- EBCDIC
- ISO 8859:
  - ISO 8859-1 Western Europe
  - ISO 8859-2 Western and Central Europe
  - ISO 8859-3 Western Europe and South European (Turkish, Maltese plus Esperanto)
  - ISO 8859-4 Western Europe and Baltic countries (Lithuania, Estonia, Latvia and Lapp)
  - ISO 8859-5 Cyrillic alphabet
  - ISO 8859-6 Arabic
  - ISO 8859-7 Greek
  - ISO 8859-8 Hebrew
  - ISO 8859-9 Western Europe with amended Turkish character set
  - ISO 8859-10 Western Europe with rationalised character set for Nordic languages, including complete Icelandic set
  - ISO 8859-11 Thai
  - ISO 8859-13 Baltic languages plus Polish
  - ISO 8859-14 Celtic languages (Irish Gaelic, Scottish, Welsh)
  - ISO 8859-15 Added the Euro sign and other rationalisations to ISO 8859-1
  - ISO 8859-16 Central, Eastern and Southern European languages (Albanian, Bosnian, Croatian, Hungarian, Polish, Romanian, Serbian and Slovenian, but also French, German, Italian and Irish Gaelic)
- CP437, CP720, CP737, CP850, CP852, CP855, CP857, CP858, CP860, CP861, CP862, CP863, CP865, CP866, CP869, CP872
- MS-Windows character sets:
  - Windows-1250 for Central European languages that use Latin script, (Polish, Czech, Slovak, Hungarian, Slovene, Serbian, Croatian, Bosnian, Romanian and Albanian)
  - Windows-1251 for Cyrillic alphabets
  - Windows-1252 for Western languages
  - Windows-1253 for Greek
  - Windows-1254 for Turkish
  - Windows-1255 for Hebrew
  - Windows-1256 for Arabic
  - Windows-1257 for Baltic languages
  - Windows-1258 for Vietnamese
- Mac OS Roman
- KOI8-R, KOI8-U, KOI7
- MIK
- ISCII
- TSCII
- VISCII
- JIS X 0208 is a widely deployed standard for Japanese character encoding that has several encoding forms.
  - Shift JIS (Microsoft Code page 932 is a dialect of Shift_JIS)
  - EUC-JP
  - ISO-2022-JP
- JIS X 0213 is an extended version of JIS X 0208.
  - Shift_JIS-2004
  - EUC-JIS-2004
  - ISO-2022-JP-2004
- Chinese Guobiao
  - GB 2312
  - GBK (Microsoft Code page 936)
  - GB 18030
- Taiwan Big5 (a more famous variant is Microsoft Code page 950)
  - Hong Kong HKSCS
- Korean
  - KS X 1001 is a Korean double-byte character encoding standard
  - EUC-KR
  - ISO-2022-KR
- Unicode (and subsets thereof, such as the 16-bit 'Basic Multilingual Plane')
  - UTF-8
  - UTF-16
  - UTF-32
- ANSEL or ISO/IEC 6937

Character Set Usage over Time



https://www.w3.org/International/questions/qa-who-uses-unicode

But, … …

- Consider **Job** column entry: “Professor†of†Transfiguration†| Head of Gryffindor”
- There are two odd characters: “†” and “|”
- How should I correct/interpret the strings? My interpretation is:
  - Replace “†” with “ “.
  - The “|” means that the person has two jobs:
    - Professor of Transfiguration
    - Head of Gryffindor
  - The **Job** column is a **multivalued** attribute.
  - The character “|” indicates that a text string is one string containing multiple values for an attribute.
- Can I always replace non-|, odd characters with “ “?
  - ‘9Ω" Fir dragon heartstring’ and ‘9º" Chestnut dragon heartstring’ would wind up having an extra space. I want 9”, not 9 “.
  - The cleanup is getting complicated.

The Cleanup is Rule-Based

- In this dataset, I can use a set of rules for data clean up:
  - Two characters separated by † maps to two characters separated by " " (space).
  - A number and " separated by an odd character maps the the number and "
  - Strings of the form s1 | s2 | s3 maps to [s1, s2, s3]
  - etc.
- Well, I could write a custom function for each rule,
  - But what happens when I find I need other rules.
  - It would be nice if there were some kind of tools that helps.
- A **regular expression** is a sequence of characters that define a search pattern. Usually, such patterns are used by string-searching algorithms for "find" or "find and replace" operations on strings, or for input validation. It is a technique developed in theoretical computer science and formal language theory. (https://en.wikipedia.org/wiki/Regular_expression)

Multivalued Attributes

- How about strings of the form s1 | s2 | s3 maps to [s1, s2, s3].
  Is this cool?
- All the data entity sets we have seen have had simple types/value attributes.
  Depends on the database model, e.g. relational, document, ...

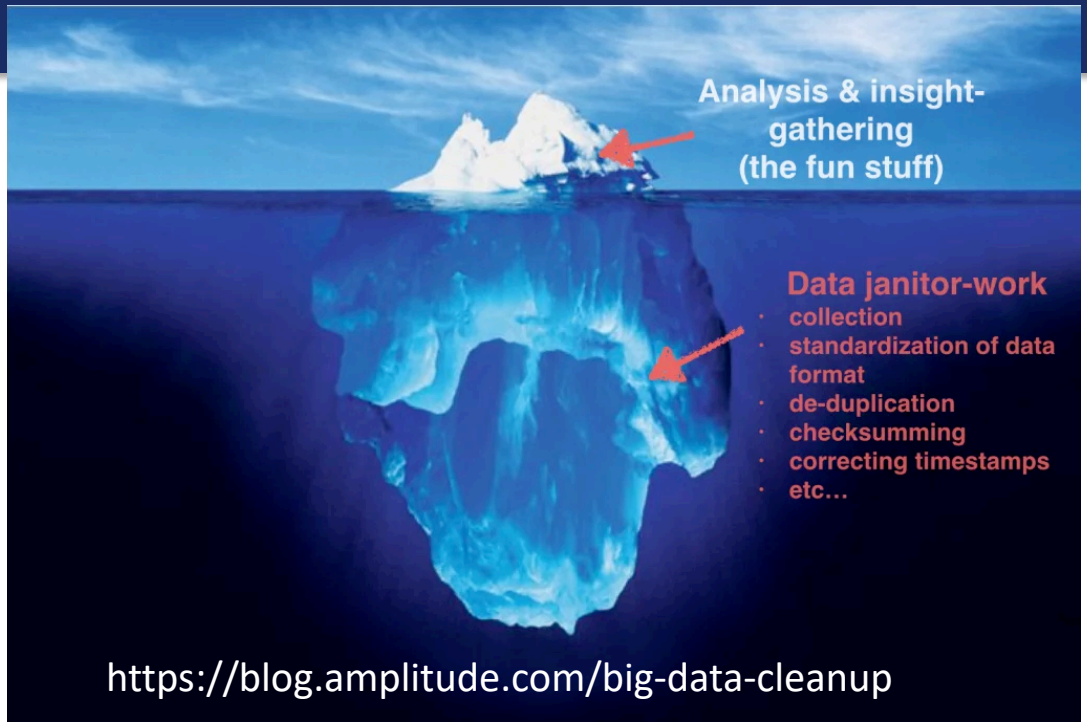
ID	*name*	*dept_name*	*salary*
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000

(a) The *instructor* table

course_id	*title*	*dept_name*	*credits*
BIO-101	Intro. to Biology	Biology	4
BIO-301	Genetics	Biology	4
BIO-399	Computational Biology	Biology	3
CS-101	Intro. to Computer Science	Comp. Sci.	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3
CS-319	Image Processing	Comp. Sci.	3
CS-347	Database System Concepts	Comp. Sci.	3
EE-181	Intro. to Digital Systems	Elec. Eng.	3
FIN-201	Investment Banking	Finance	3
HIS-351	World History	History	3
MU-199	Music Video Production	Music	3
PHY-101	Physical Principles	Physics	4

Figure 2.2 The *course* relation.

Data Cleansing



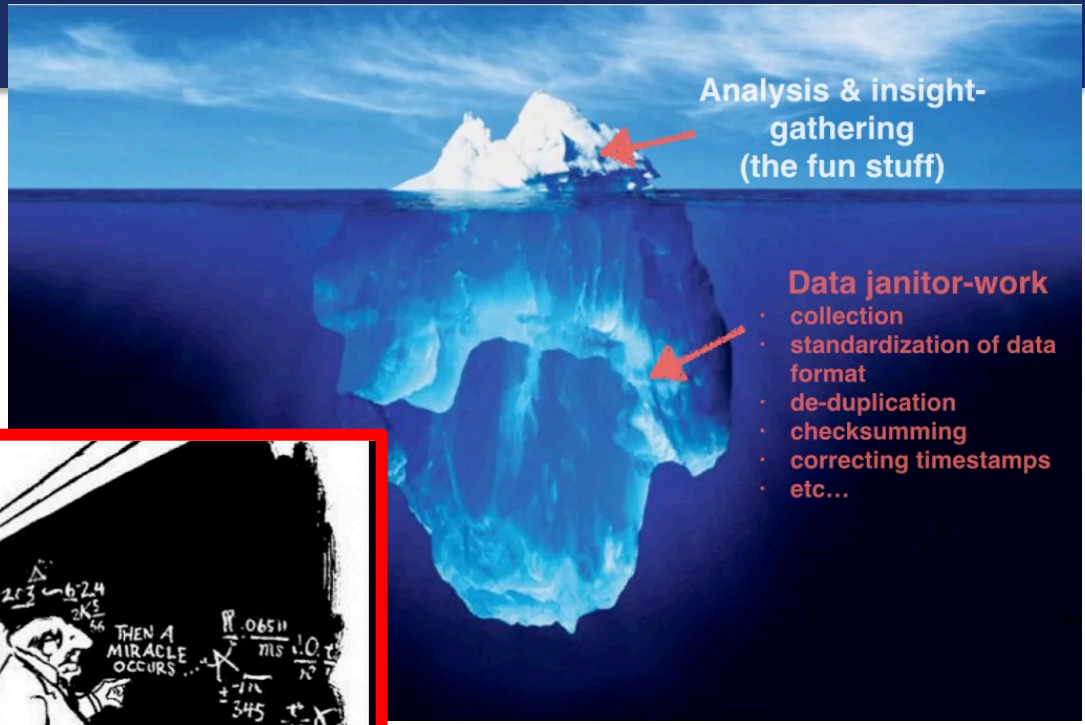
Database and data science classes **love to teach the fun stuff**
queries, data modeling, machine learning, spooky math, algorithms, ...

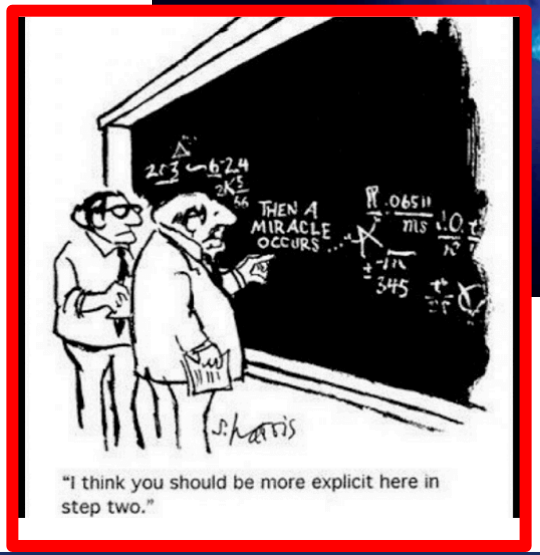
Data Cleansing

Syllabus Topics

Recommended Syllabus

- Relational Foundations
  - Overview (1 lecture)
  - ER Model (2 lectures)
  - Relational Model (4 lectures)
  - Relational Algebra (2 lectures)
  - SQL (5 lectures)
  - Application Programming and Database APIs (1 lecture)
  - Security (2 lectures)
  - Normalization (2 lectures)
  - Overview of Storage and Indexes (1 lecture)
  - Overview of Query Optimization (1 lecture)
  - Overview of Transaction Processing (1 lecture)
- Beyond Relational Foundations
  - NoSQL (1 lecture)
  - Data Preparation and Cleaning (1 lecture)
  - Graphs (1 lecture)
  - Object-Relational Databases (2 lectures)
  - Cloud Databases (1 lecture)

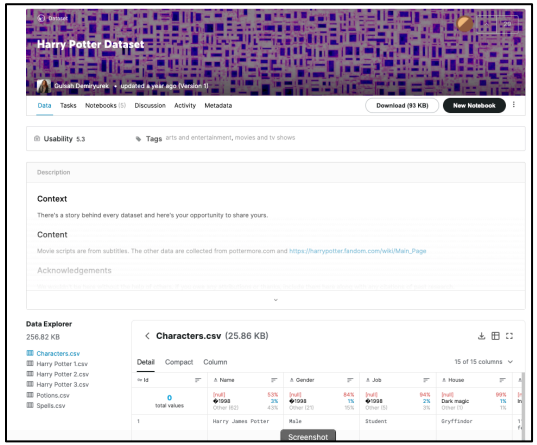


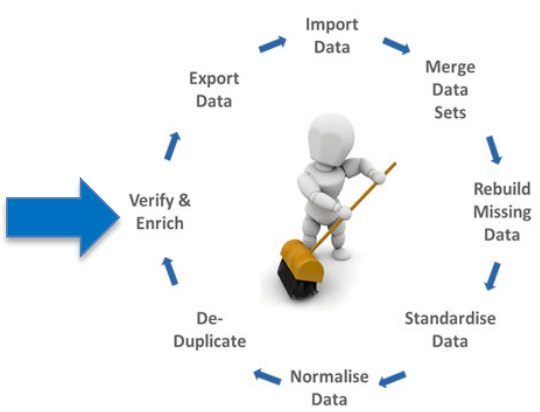


I place more emphasis on data cleansing and refactoring than other sections of W4111.

Next Steps in the Plan

Source Data





Relational Data

attributes (or columns)

tuples (or rows)

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

- We are starting the semester with the *relational datamodel/databases.*
- Before we get too far in our data janitor work,
  we should understand the relational model a little more.

But, Before We Leave ... ...

- What are these regular expressions whereout you speak?
- Many environments come with regular expressions functions for searching and transforming strings.
  - Python: Lib/re.py (https://docs.python.org/3/library/re.html)
  - Java: java.util.regex (https://www.w3schools.com/java/java_regex.asp)
  - "SNOBOL ("StriNg Oriented and symBOlic Language") is a series of programming languages ... ... having patterns as a first-class data type." (https://en.wikipedia.org/wiki/SNOBOL)
  - Most SQL databases have some from of regular expression library, but the libraries differ from product to product.

Table 12.14 Regular Expression Functions and Operators

Name	Description
NOT REGEXP	Negation of REGEXP
REGEXP	Whether string matches regular expression
REGEXP_INSTR()	Starting index of substring matching regular expression
REGEXP_LIKE()	Whether string matches regular expression
REGEXP_REPLACE()	Replace substrings matching regular expression
REGEXP_SUBSTR()	Return substring matching regular expression
RLIKE	Whether string matches regular expression

- This was FYI.
- Not a core part of W4111.
- We will play with it a little.

```
In [40]:
1  s = '12æ" Hornbeam dragon heartstring'
2  x = re.sub(r'([0-9]).(")', r'\1\2', s)
3  x

Out[40]: '12" Hornbeam dragon heartstring'
```

Some More Set-Like Concepts

Set Comparison – “some” Clause

- Find names of instructors with salary greater than that of some (at least one) instructor in the Biology department.

```
select distinct T.name
from instructor as T, instructor as S
where T.salary > S.salary and S.dept name = 'Biology';
```

- Same query using > **some** clause

```
select name
from instructor
where salary > some (select salary
                     from instructor
                     where dept name = 'Biology');
```

Definition of "some" Clause

- F <comp> **some** $r \Leftrightarrow \exists\ t \in r$ such that (F <comp> t)
  Where <comp> can be: $<,\ \leq,\ >,\ =,\ \neq$

(5 < **some** | 0 / 5 / 6 |) = true (read: 5 < some tuple in the relation)

(5 < **some** | 0 / 5 |) = false

(5 = **some** | 0 / 5 |) = true

(5 ≠ **some** | 0 / 5 |) = true (since 0 ≠ 5)

(= **some**) ≡ **in**
However, (≠ **some**) ≢ **not in**

Set Comparison – “all” Clause

- Find the names of all instructors whose salary is greater than the salary of all instructors in the Biology department.

```
select name
from instructor
where salary > all (select salary
                    from instructor
                    where dept name = 'Biology');
```

Definition of "all" Clause

- F <comp> **all** $r \Leftrightarrow \forall\ t \in r$ (F <comp> *t)*

(5 < **all** | 0, 5, 6 |) = false

(5 < **all** | 6, 10 |) = true

(5 = **all** | 4, 5 |) = false

(5 ≠ **all** | 4, 6 |) = true (since 5 ≠ 4 and 5 ≠ 6)

(≠ **all**) ≡ **not in**
However, (= **all**) ≠ **in**

Test for Empty Relations

- The **exists** construct returns the value **true** if the argument subquery is nonempty.
- **exists** $r \Leftrightarrow r \neq \emptyset$
- **not exists** $r \Leftrightarrow r = \emptyset$

Use of "exists" Clause

- Yet another way of specifying the query "Find all courses taught in both the Fall 2017 semester and in the Spring 2018 semester"

```
select course_id
from section as S
where semester = 'Fall' and year = 2017 and
        exists  (select *
                 from section as T
                 where semester = 'Spring' and year= 2018
                         and S.course_id = T.course_id);
```

- **Correlation name** – variable S in the outer query
- **Correlated subquery** – the inner query

Use of "not exists" Clause

- Find all students who have taken all courses offered in the Biology department.

```
select distinct S.ID, S.name
from student as S
where not exists ( (select course_id
                    from course
                    where dept_name = 'Biology')
                   except
                    (select T.course_id
                     from takes as T
                     where S.ID = T.ID));
```

  - First nested query lists all courses offered in Biology
  - Second nested query lists all courses a particular student took
- Note that $X - Y = \emptyset \Leftrightarrow X \subseteq Y$
- Note: Cannot write this query using = all and its variants

Test for Absence of Duplicate Tuples

- The **unique** construct tests whether a subquery has any duplicate tuples in its result.
- The **unique** construct evaluates to “true” if a given subquery contains no duplicates .
- Find all courses that were offered at most once in 2017

```
select T.course_id
 from course as T
 where unique ( select R.course_id
                from section as R
                where T.course_id= R.course_id
                      and R.year = 2017);
```

Completing Subqueries

- A subquery is a query inside a query.
- Logically, the engine calls the subquery when evaluating every row in processing.
- Correlation means properties from the outer row being evaluated are inputs to the inner query.
- Subqueries:
    - Return tables if in the FROM clause.
    - Scalars in the SELECT clause.
    - Tables or scalars depending on the comparison operator in WHERE.

Subqueries in the From Clause

- SQL allows a subquery expression to be used in the **from** clause
- Find the average instructors' salaries of those departments where the average salary is greater than $42,000."

```
select dept_name, avg_salary
from ( select dept_name, avg (salary) as avg_salary
        from instructor
        group by dept_name)
where avg_salary > 42000;
```

- Note that we do not need to use the **having** clause
- Another way to write above query

```
select dept_name, avg_salary
from ( select dept_name, avg (salary)
        from instructor
        group by dept_name)
        as dept_avg (dept_name, avg_salary)
where avg_salary > 42000;
```

Scalar Subquery

- Scalar subquery is one which is used where a single value is expected
- List all departments along with the number of instructors in each department

```
select dept_name,
        ( select count(*)
          from instructor
          where department.dept_name = instructor.dept_name)
        as num_instructors
from department;
```

- Runtime error if subquery returns more than one result tuple

Switch to Notebook

Backup