CS 217 Data Management and Information Processing

Defining Databases and Adding Data

Course Project Topics

- A lot of variety!

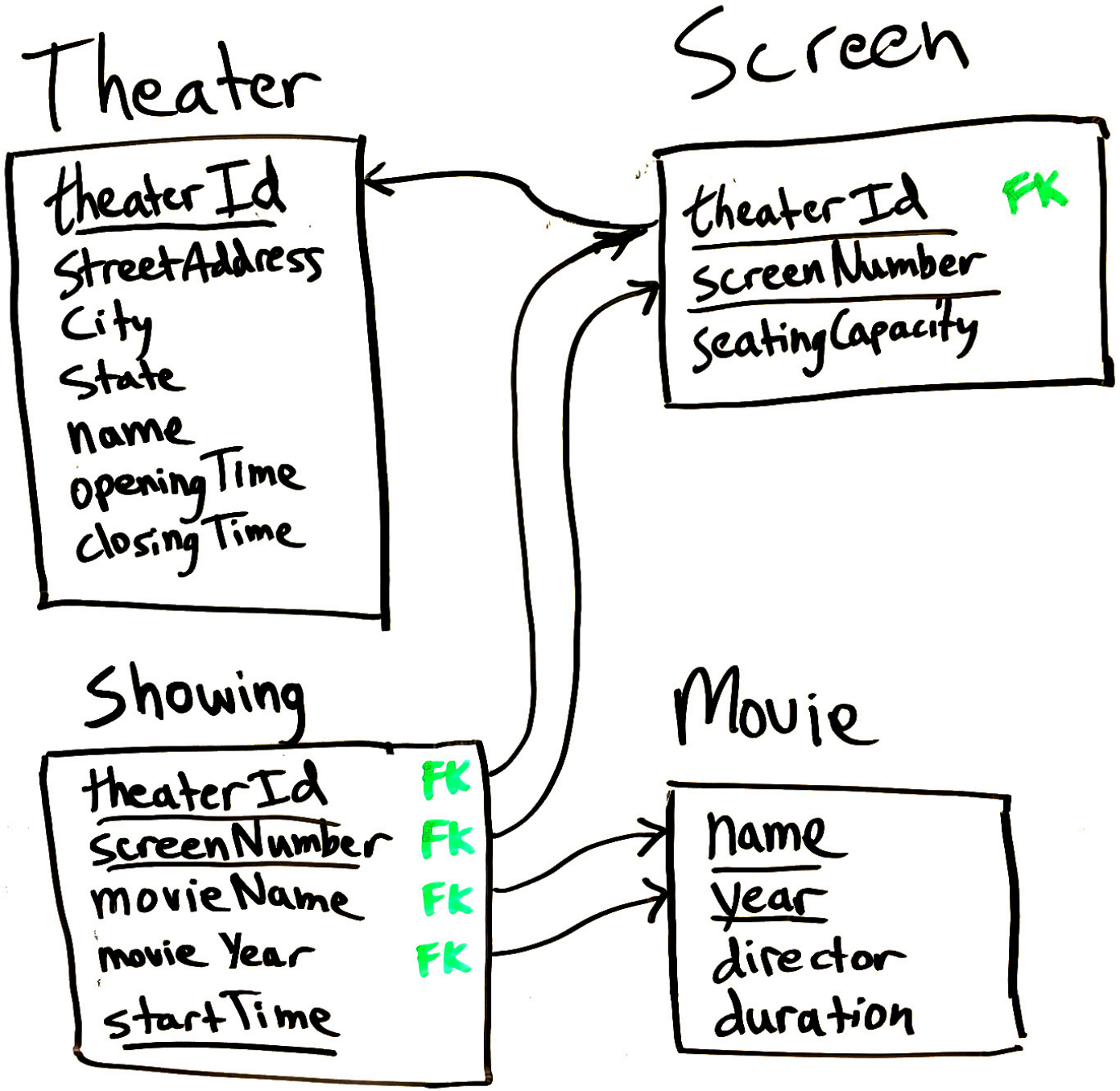
- Sports: NBA, MLB, FIFA, NFL
- Entertainment: Spotify, Movie, Netflix, Games
- Business: startups, restaurants, housing market, Energy, Airbnb
- Environment: air quality, climate change
- COVID-19 related: nutrition, hospital, food safety

Common Problems

- The dataset you use may not answer your question
  - E.g., Yelp dataset may not update often, so may not be used for COVID-19 related research. (Redfin update more regularly)
- Problems to study are too ambitious
  - 4 to 6 problems is sufficient
- Not picking the best tool.
  - SQL or Pandas if you are given a CSV file?

Database Schema Refinement

Movie Theater

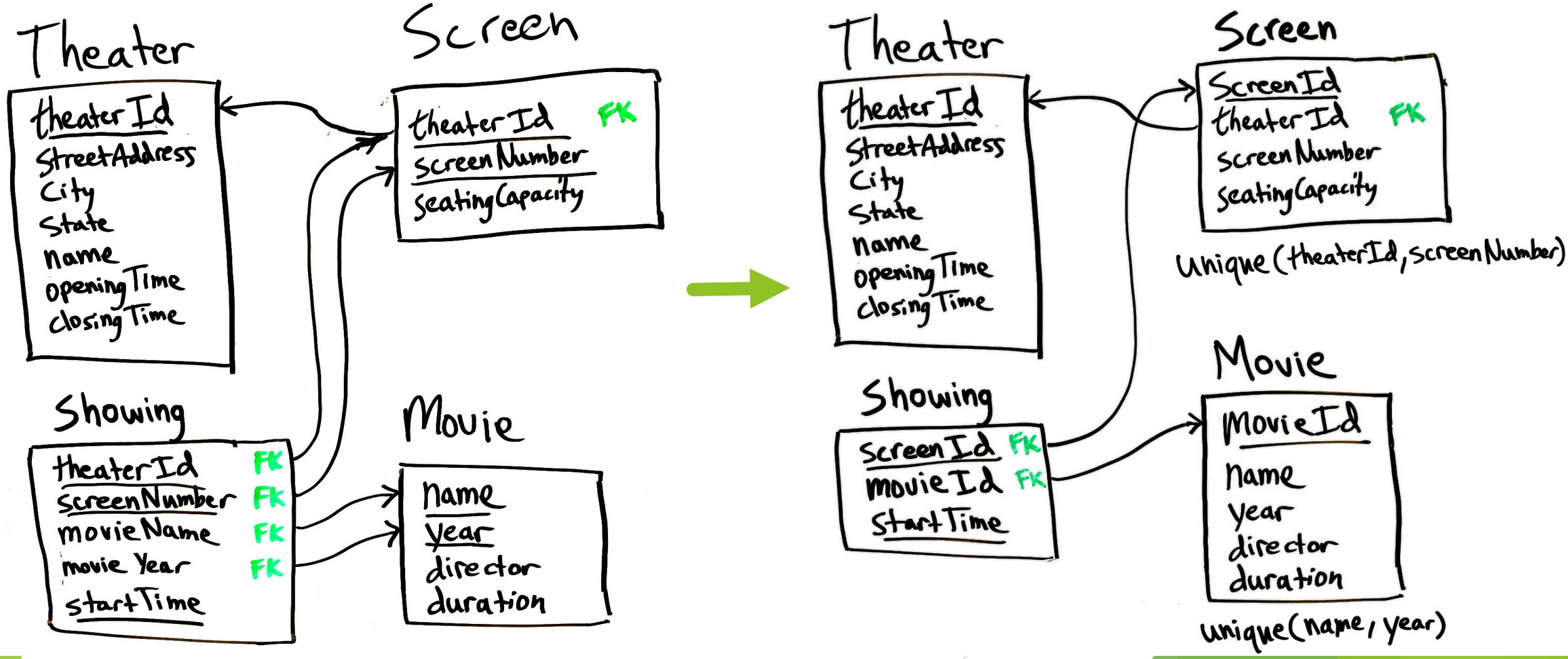


Composite Primary Keys

- Primary Keys uniquely identify rows
  - Used as *indexes* to find a row of interest
  - Prevent duplication
- Often we need more than one column to uniquely identify rows
  - E.g., a Screen is uniquely identified by **theaterId** *and* **screenNumber**.
  - **theaterId** alone cannot be a primary key because it's OK for multiple screens to exist at the same theater, as long as they have different **screenNumber**.
  - **screenNumber** alone cannot be a primary key because different theaters can use the same screen numbers (1, 2, 3 …).
- However, composite primary keys make foreign keys and parent-child relationships messy.

Adding a **ScreenId** and **MovieId** simplifies the schema.

Showing table becomes smaller and JOINs are simpler



Non-primary/Unique Keys

- When a table is a parent, it is common to create a meaningless "ID" column for the primary key, then add a non-primary composite key to enforce the integrity constraint.
- For example, in the movie theater example:
  - movieId is meaningless, but it is a convenient way for other tables to refer to movies in foreign keys.
  - add a *unique key* on (name, year) to prevent two instances of the same movie
  - Showing table can have just a single column movieId as a foreign key instead of two columns (name, year).

Updating Database

Modifying SQL databases

- Define tables
- Add rows to tables
- Delete rows from tables
- Update columns in a row
- Alter tables by adding or removing:
  - Columns
  - Indexes
  - Foreign keys
- … and much more

- `CREATE TABLE …`
- `INSERT INTO …`
- `DELETE FROM …`
- `UPDATE …`
- `ALTER TABLE …`

Look up the detailed syntax online:

https://sqlite.org/lang.html

Deleting rows

- **DELETE** command deletes rows in a table matching some criterion.
- Very similar to the `SELECT` statements you're familiar with.
- Just replace `SELECT` with `DELETE` and don't specify any columns
- This deletes the specified row in the Faculty table:

```
DELETE FROM Faculty WHERE StaffID=12;
```

- If you don't include a WHERE clause, all the rows in that tables will be deleted:

```
DELETE FROM Faculty;
```

- To be safe, run a `SELECT` query first to see what will be deleted:

```
SELECT * FROM Faculty WHERE StaffID=12;
```

Foreign Keys affect deletions

- In the SchoolScheduling database, there is a foreign key in the *Faculty_Classes* table which refers to the *Faculty* table.
  - What happens if we try to delete a faculty that has several associated classes?
- If you try to delete a row that is a parent to another row there are several possible results, depending on the particular foreign key settings:
  - `RESTRICT` is the default behavior, it would block the deletion
    - You would have to delete the classes first, then the faculty
  - `CASCADE` causes the child rows to be deleted as well
    - Classes would be deleted
  - `SET NULL` causes the child rows to have the column set to null
    - Classes would remain, but with a NULL StaffID
  - https://www.sqlite.org/foreignkeys.html

Updating rows

- **UPDATE** command is used to change one or more columns in rows matching some criterion.

```
UPDATE Departments SET DeptName="Social Studies"
  WHERE DeptName="History";
```

- Just like DELETE, a single UPDATE command can affect many rows and it can use subqueries:

```
UPDATE Students SET StudMajor=
  (SELECT MajorID FROM Majors WHERE
Major="English");
```

- Can also refer to existing column values and use math functions:

```
UPDATE Student_Schedules SET Grade=Grade+5
WHERE ClassID=1500;
```

Updating multiple columns

- Use a comma-separated list to update multiple columns at once:

```
UPDATE my_table
  SET column1=value1,
      column2=value2,
      column3=value3
  WHERE id=123;
```

Inserting new rows

- `INSERT` command creates one row with the column values specified.
- List the column values in same order that the columns were defined:

```
INSERT INTO Buildings VALUES ("FD", "Ford", 5, 1,
0);
```

- Or, explicitly list the columns being set (this is more clear):

```
INSERT INTO Buildings (BuildingName, BuildingCode,
     NumberOfFloors, ElevatorAccess,
SiteParkingAvailable)
  VALUES ("Ford", "FD", 5, 1, 0);
```

- Unspecified columns will get the default value specified when the table was created (more on this later).

Bulk loading data

Three options for inserting lots of rows:

1. Write code in a programming language like R or Python to read the source data and run lots of `INSERT` statements or one really big `INSERT` statement:

```
INSERT INTO animals VALUES (1, "cat", 5), (2, "dog", 2),
   (3, "mouse", 9), (4, "rat", 3) …
```

2. Import a CSV file:
   - CSV (**C**omma **S**eparated **V**alues) is a very simple, standard spreadsheet format.
   - Exact import steps are different for each DBMS.
   - In DB Browser for SQLite use *File* → *Import* → *Table from CSV file*
3. Use an ETL software package (Extract, Transform, Load)

Creating tables

- `CREATE TABLE` command defines:
  - Table name
  - Column names
  - Column types (int, float, text, etc.)
  - Whether columns are optional or required (NOT NULL)
  - Primary key
  - Foreign keys
  - Unique keys
  - Indexes (non-unique keys)
- In other words, everything that we drew in the data model diagrams

CREATE TABLE Syntax examples from SchoolScheduling.sqlite

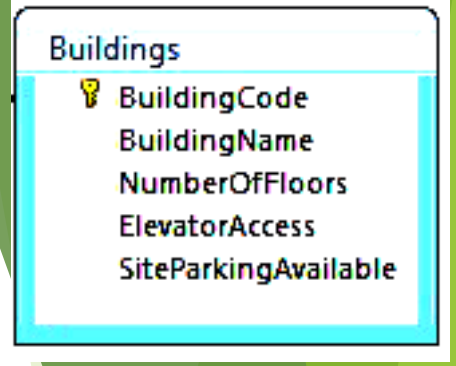


Table name

Required column, not optional

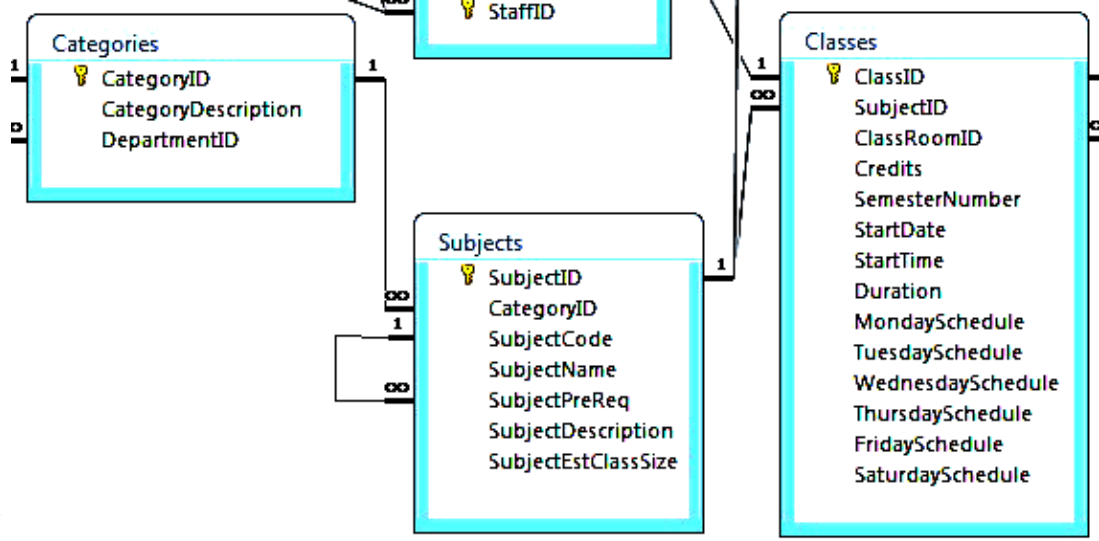
```
CREATE TABLE Buildings (
    BuildingCode nvarchar(3) NOT NULL,
    BuildingName nvarchar(25),
    NumberOfFloors smallint,
    ElevatorAccess bit NOT NULL DEFAULT 0,
    SiteParkingAvailable bit NOT NULL DEFAULT 0,
    PRIMARY KEY (BuildingCode)
);
```

Columns

Text with at most 25 characters

Column cannot be NULL, but it will take a value of zero if none is specified.

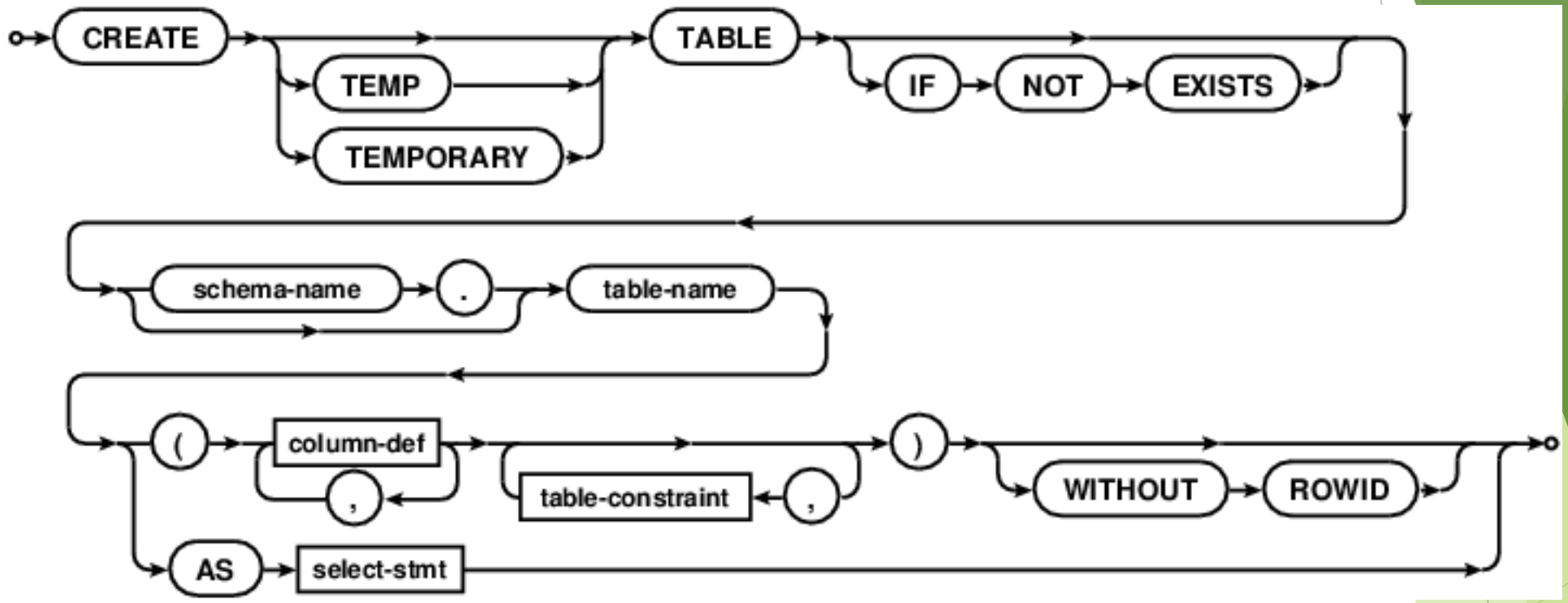
Each column has a data type, like nvarchar(3) *or* smallint




```
CREATE TABLE Subjects (
  SubjectID int NOT NULL DEFAULT 0 ,
  CategoryID nvarchar (10) NULL
      REFERENCES Categories(CategoryID),
  SubjectCode nvarchar (8) NULL ,
  SubjectName nvarchar (50) NULL ,
  SubjectPreReq nvarchar (8) NULL DEFAULT NULL
      REFERENCES Subjects(SubjectCode),
  SubjectDescription text NULL ,
  SubjectEstClassSize smallint NOT NULL DEFAULT 0,
  PRIMARY KEY (SubjectID),
  UNIQUE (SubjectCode)
);
```

Foreign keys

CREATE TABLE syntax diagram



column-def:

column-constraint:

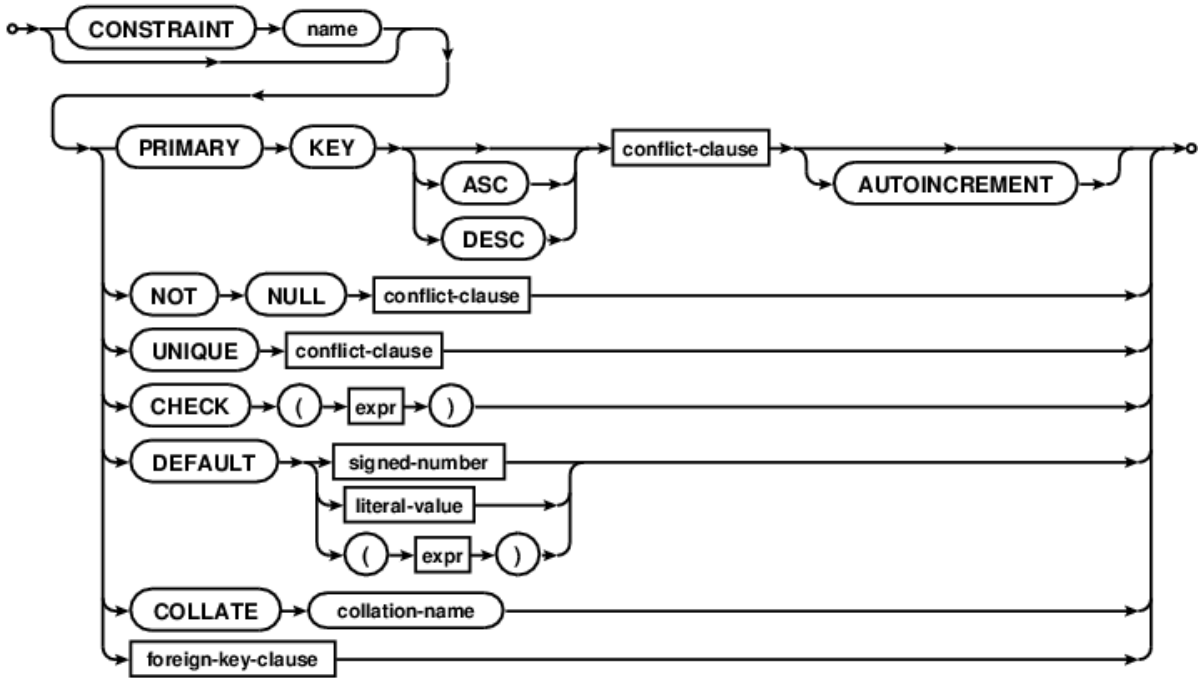
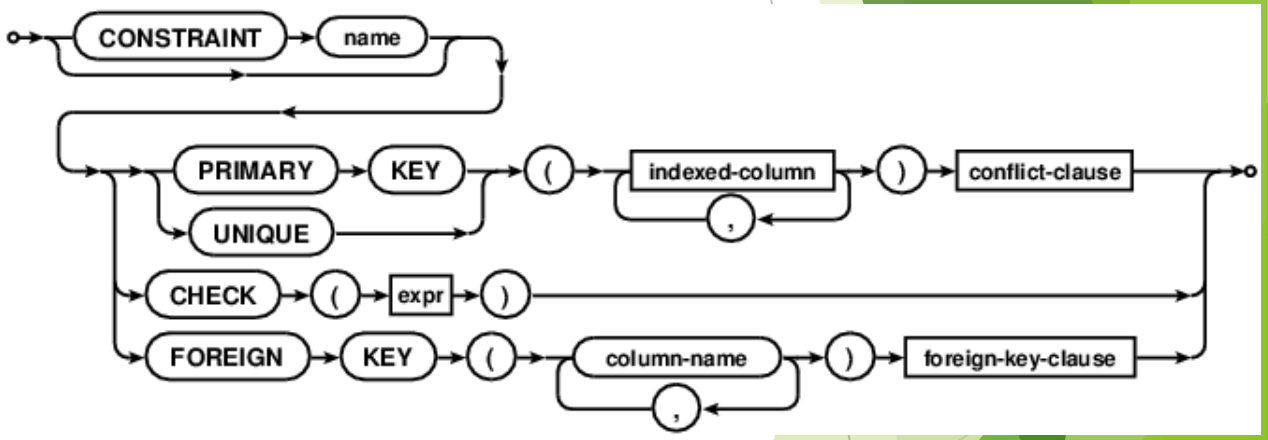


table-constraint:



Debugging a data import

- If data fails to import completely, try loading it into a *temporary text table*
  - Don't enforce key constraints and use large text types for every column
- Query the text table to look for unexpected values in the source data

This table has strict constraints on what kind of data can be inserted:

```
CREATE TABLE person (
  SSN int NOT NULL,
  firstName varchar(30) NOT NULL,
  lastName varchar(30) NOT NULL,
  birthDate char(10) NOT NULL,
  PRIMARY KEY (SSN)
);
```

This **temporary table** relaxes those constraints:

```
CREATE TABLE _import_person (
  SSN varchar(1000) NOT NULL,
  firstName varchar(1000) NOT NULL,
  lastName varchar(1000) NOT NULL,
  birthDate varchar(1000) NOT NULL,
);
```

Using queries to fill tables

- You can transfer data from the temporary to permanent tables by putting a SELECT in an INSERT query. For example:
  - ```
    INSERT INTO orders (col1, col2)
    SELECT col1, col2 FROM tmp_orders;
    ```
- Above query copies data from `tmp_orders` to `orders` table.

- Note that DB Browser to sqlite does not always work well with very large CSV files. You may have to be import big files using the commandline version of sqlite.
    ```