Award
2417789

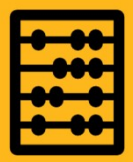
CI Pathways

Leading the Way to Effective Cyberinfrastructure Use

Your Pathway: Data Science with SQL and Pandas

Bryon Gill and John Urbanic – Pittsburgh Supercomputing Center

Roadmap



SQL

Traditional tools for data analysis

Pandas

A modern and flexible tool for data analysis

Big Data

The need for a new paradigm

Spark

A distributed, scalable tool for data analysis

Spark II

A deeper dive

Part 1: SQL

Database Language Foundations

Relational Databases

An RDBMS (Relational DataBase Management System) organizes data into tables of columns (attributes) and rows (records).

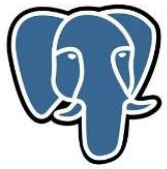
This concept has been developed and refined since 1970, and is a mature concept at this point.

Most RDMBSs use SQL as their query language. This has become an ISO standard (with many deviations).

SQL Databases?

There are a variety of SQL server and client programs. each have their own deviations from the ISO standard, as well as significant performance differences.

Most databases operate with a client/server model but we will be using the lightweight database sqlite3 for this class so that each user can have their own instance.



Setting Up

We will each make an initial copy of the example database we will be using to our home directories.

```
[bgill@dt-login03 ~]$ cp /projects/becs/bgill/sql/sample.db .
[bgill@dt-login03 ~]$ sqlite3 sample.db
SQLite version 3.41.2 2023-03-22 11:56:21
Enter ".help" for usage hints.
sqlite>
```

(example database adapted from sqlitetutorial.net)

Database Structure

The structure, or “schema” is the most important characteristic of any database. We can get a top level view by listing the tables:

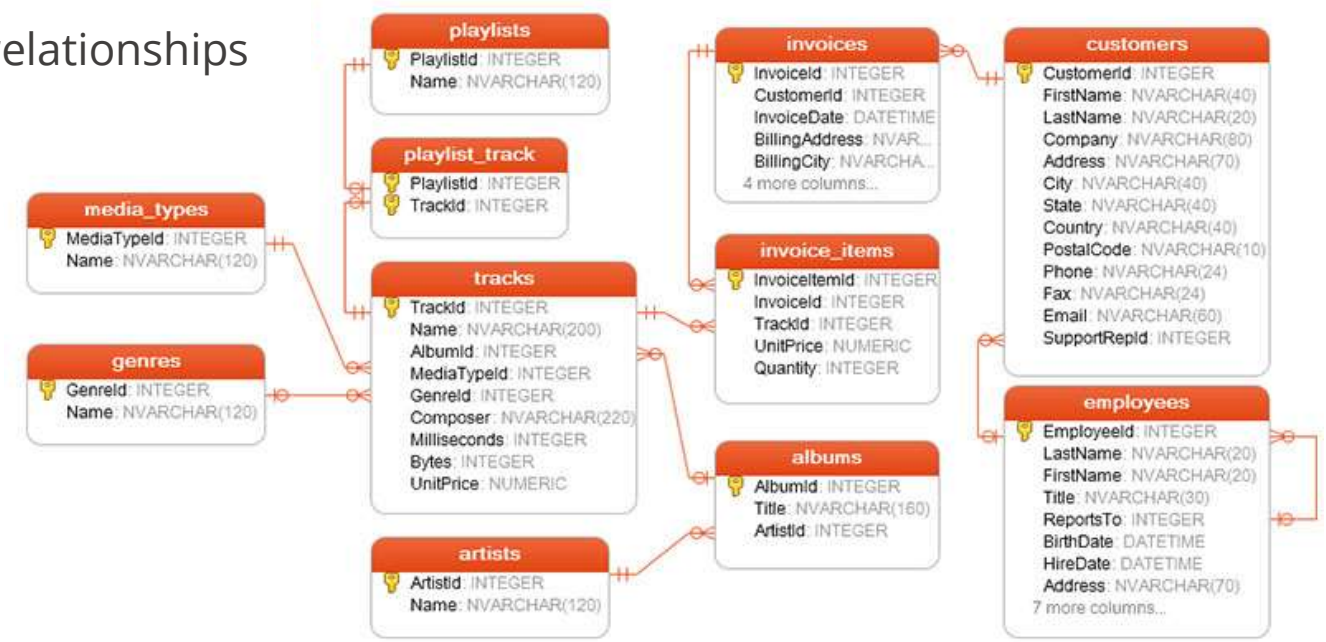
```
sqlite> .tables

albums          employees       invoices        playlists
artists         genres          media_types     tracks
customers       invoice_items   playlist_track
```

There’s much more to say about the contents of the tables and relationships between them.

SQL Entity Relationship Diagram

An Entity Relationship Diagram can be a helpful way to visualize table relationships in a database.

Our sample database represents a digital music company that sells a variety of tracks to consumers worldwide.



SELECT

The SELECT command is our most useful command in manipulating data, and we will look at some of the common variations.

```
sqlite> SELECT * FROM customers WHERE CustomerId = 1;

1|Luís|Gonçalves|Embraer - Empresa Brasileira de Aeronáutica S.A.|Av. Brigadeiro Faria
   Lima, 2170|São José dos Campos|SP|Brazil|12227-000|+55 (12) 3923-5555|+55 (12) 3923-
   5566|luisg@embraer.com.br|3

sqlite> SELECT * FROM invoices WHERE CustomerId = 1;

98|1|2010-03-11 00:00:00|Av. Brigadeiro Faria Lima, 2170|São José dos
   Campos|SP|Brazil|12227-000|3.98

121|1|2010-06-13 00:00:00|Av. Brigadeiro Faria Lima, 2170|São José dos
   Campos|SP|Brazil|12227-000|3.96

..
```

Note that “*” is our wildcard.

Output Mode

We can include headers and cell frames by changing the output mode:

```
sqlite> .mode table
sqlite> SELECT CustomerId, FirstName, LastName FROM customers WHERE CustomerId = 1;
+------------+-----------+-----------+
| CustomerId | FirstName | LastName  |
+------------+-----------+-----------+
| 1          | Luís      | Gonçalves |
+------------+-----------+-----------+
```

https://www.sqlite.org/docs.html

More Complex SELECT Statements

We can alias a column with *AS*. This alias only exists for the duration of the query. We are introducing some powerful qualifiers here. GROUP BY will group rows that have the same value into summary rows, and is usually used with aggregate functions (*COUNT(), MAX(), MIN(), SUM(), AVG()*) to group the results.

ORDER (in this case in descending order) and *LIMIT* are also very useful and common. You can choose the order DESC or ASC.

```
sqlite> SELECT CustomerId, COUNT(*) AS NumInvoices
   ...> FROM invoices
   ...> GROUP BY CustomerId
   ...> ORDER BY NumInvoices DESC
   ...> LIMIT 5;
+------------+-------------+
| CustomerId | NumInvoices |
+------------+-------------+
| 1          | 7           |
| 2          | 7           |
| 3          | 7           |
| 4          | 7           |
| 5          | 7           |
+------------+-------------+
```

Select Subqueries

We can treat that subquery as a table itself. Here we apply the count(*) to it. Next we will connect these together.

```
sqlite> SELECT CustomerId, COUNT(*) AS NumOrders FROM Invoices GROUP BY CustomerId HAVING
   NumOrders > 6;
+------------+-----------+
| CustomerId | NumOrders |
+------------+-----------+
| 2          | 7         |
| 17         | 7         |
| 38         | 7         |
| 40         | 7         |
+------------+-----------+
```

SQL requires that derived tables have a name (alias). So we must name our subquery.

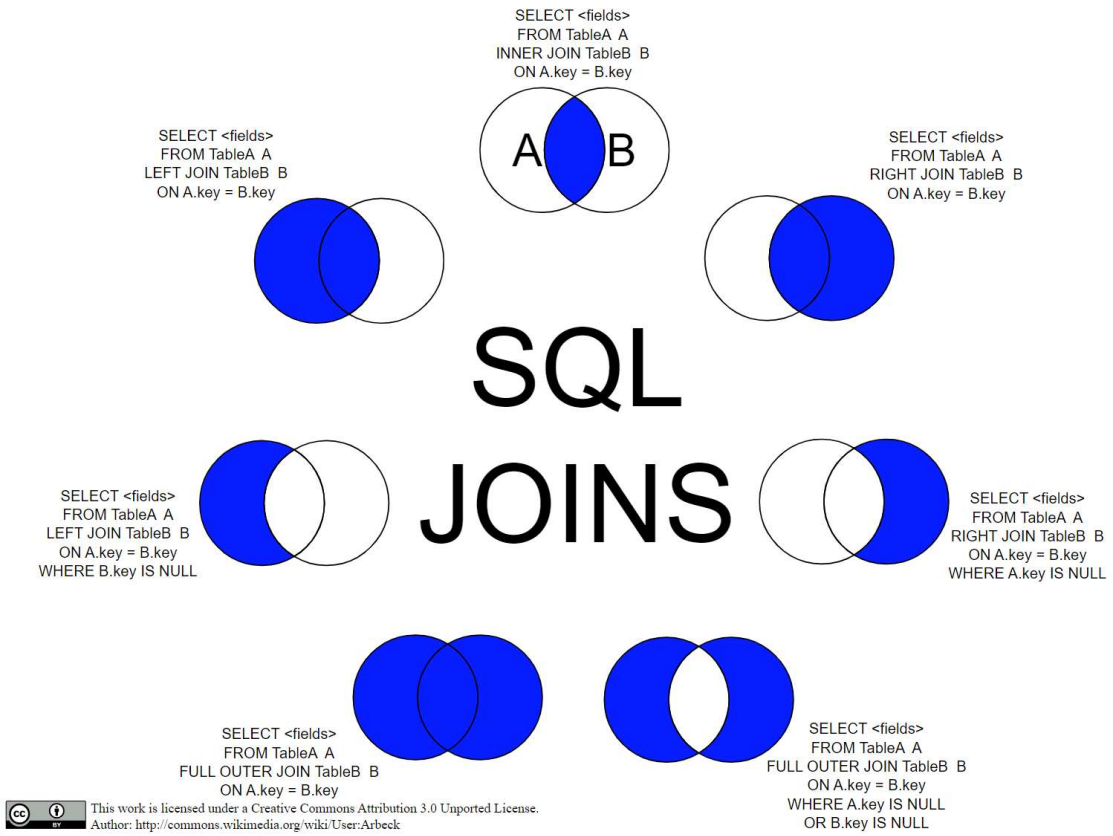
Select Subqueries

This output might be more useful with some customer information mixed in, but that data isn't found in the invoices table. We are interested in customers data:

```
sqlite> pragma table_info(customers);
+-----+-------------+-------------+---------+------------+----+
| cid |     name    |     type    | notnull | dflt_value | pk |
+-----+-------------+-------------+---------+------------+----+
| 0   | CustomerId  | INTEGER     | 1       |            | 1  |
| 1   | FirstName   | NVARCHAR(40)| 1       |            | 0  |
| 2   | LastName    | NVARCHAR(20)| 1       |            | 0  |
| 3   | Company     | NVARCHAR(80)| 0       |            | 0  |
| 4   | Address     | NVARCHAR(70)| 0       |            | 0  |
```

... (This command is not SQL standard and differs by db.)

Combining Table Data



Inner Join

This is the default “join” and most common. It collects items with matching keys from both tables. The keys are specified with an “ON” clause.

```
SELECT left_table.B, right_table.F
FROM left_table
JOIN right_table
ON   left_table.A = right_table.E;
```

Left Table

A	B	C	D
K0	B0	C0	D0
K1	B1	C1	D1
K2	B2	C2	D2
K3	B3	C3	D3

Right Table

E	F	G	H
K1	F0	G0	H0
K1	F1	G1	H1
K0	F2	G2	H2
K6	F3	G3	H3

Result

B	F
B0	F2
B1	F0
B1	F1

Inner Join Example

The table name is usually inferred from the FROM clause but in a JOIN columns must be disambiguated as table.column:

```
SELECT Customers.*, TopOrders.NumOrders FROM
(SELECT CustomerId, COUNT(*) AS NumOrders
    FROM Invoices
    GROUP BY CustomerId
    HAVING NumOrders > 6) AS TopOrders
JOIN Customers
    ON TopOrders.CustomerId = Customers.CustomerId;
```

Inner Join (abbreviated) output

CustomerId	LastName	NumOrders
2	Köhler	7
4	Hansen	6
9	Nielsen	6
11	Rocha	6
13	Ramos	6
15	Peterson	6
17	Smith	7
19	Goyer	6
21	Chase	6
23	Gordon	6

…

Inner Join Example

Let's break this down one subquery at a time:

```
SELECT Customers.*, TopOrders.NumOrders FROM
(SELECT CustomerId, COUNT(*) AS NumOrders
    FROM Invoices
    GROUP BY CustomerId
    HAVING NumOrders > 6) AS TopOrders
JOIN Customers ON TopOrders.CustomerId = Customers.CustomerId;
```

Views

As queries and subqueries get more complex it becomes cumbersome and inefficient to keep recreating them. A VIEW lets us capture them as temporary tables:

```
CREATE VIEW TopCustomers AS
SELECT Customers.* FROM (SELECT CustomerId, COUNT(*) AS NumOrders
FROM Invoices
  GROUP BY CustomerId
  HAVING NumOrders > 6) AS TopOrders
JOIN Customers
  ON TopOrders.CustomerId = Customers.CustomerId;

.tables

TopCustomers    customers       invoice_items   playlist_track
albums          employees       invoices        playlists
artists         genres          media_types     tracks
```

A Useful View

Let’s create a view containing all the Jazz tracks. Note that we can use % as a wildcard in a single quoted string:

```
CREATE VIEW JazzTracks
AS SELECT *
FROM tracks
INNER JOIN genres
ON tracks.GenreId = genres.GenreId
WHERE genres.Name LIKE 'Jaz%';
```

A Promotion

The marketing department has requested a list of all jazz track invoices. We use DISTINCT to ensure that we only get one InvoiceId even if multiple tracks were purchased.

```
CREATE VIEW JazzOrders AS
SELECT DISTINCT InvoiceId
FROM invoice_items
INNER JOIN jazztracks
ON invoice_items.TrackId = jazztracks.TrackId;
```

A Promotion (Cont'd)

The marketing department now wants a list of customers who ever purchased a Jazz track.

```
CREATE VIEW JazzCustomers AS
SELECT DISTINCT Invoices.CustomerId
FROM Invoices
INNER JOIN JazzOrders
ON Invoices.InvoiceId = JazzOrders.InvoiceId;
```

A Promotion (Cont'd)

The marketing department has one more request, they'd like to offer their promotion only to their top customers, but they'll send a different message if they have ordered a jazz track.

```
SELECT *
FROM TopCustomers
LEFT JOIN JazzCustomers
ON TopCustomers.CustomerId = JazzCustomers.CustomerId;
```

Left Joins

LEFT JOIN will include all elements from the left table and matching ones from the right table. Unmatched values will be shown as NULL.

```
SELECT left_table.B, right_table.F
FROM left_table
LEFT JOIN right_table
ON   left_table.A = right_table.E;
```

Left Table

A	B	C	D
K0	B0	C0	D0
K1	B1	C1	D1
K2	B2	C2	D2
K3	B3	C3	D3

Right Table

E	F	G	H
K1	F0	G0	H0
K1	F1	G1	H1
K0	F2	G2	H2
K6	F3	G3	H3

Result

B	F
B0	F2
B1	F0
B1	F1
B2	NULL
B3	NULL

Right Joins

As you might expect, RIGHT JOIN includes all elements from the right table and matching ones from the left. Unmatched values will be shown as NULL.

```
SELECT left_table.B, right_table.F
FROM left_table
RIGHT JOIN right_table
ON   left_table.A = right_table.E;
```

Left Table

A	B	C	D
K0	B0	C0	D0
K1	B1	C1	D1
K2	B2	C2	D2
K3	B3	C3	D3

Right Table

E	F	G	H
K1	F0	G0	H0
K1	F1	G1	H1
K0	F2	G2	H2
K6	F3	G3	H3

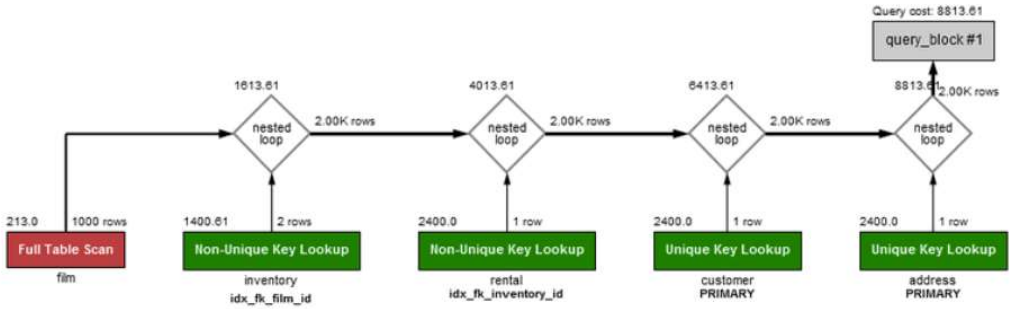
Result

B	F
B1	F0
B1	F1
B0	F2
NULL	F3

Joins Are Loops

You notice how we loop through the keys as we manually create our joins. This is what our database must do as well. Nested joins turn into nested loops. Here is a typical query from a classic film rental database (don't try to paste it today).

```
SELECT CONCAT(customer.last_name, ', ', customer.first_name)
  AS customer, address.phone, film.title FROM rental
INNER JOIN customer ON rental.customer_id = customer.customer_id
INNER JOIN address ON customer.address_id = address.address_id
INNER JOIN inventory ON rental.inventory_id = inventory.inventory_id
INNER JOIN film ON inventory.film_id = film.film_id
WHERE rental.return_date IS NULL
AND rental_date + INTERVAL film.rental_duration DAY < C
LIMIT 5;
```



Keys

If we are trying to quickly equate things from two tables, you might imagine that the organization of those tables might have a major effect on performance. Indeed, the correct selection of keys for each table is the most important consideration.

There are a variety of key types. Two are very important.

Primary Key
A column (or possibly combination of columns) with unique values.

Foreign Key
A column whose values point to a Primary Key in a different table.

There are other terms for keys that are less important to know. *Candidate Keys* are any keys that could be a Primary Key. A *Unique Key* could have a single NULL value (which is not allowed for a Primary Key). A *Composite Key* is a key created from multiple columns, etc.

Keys (Cont'd)

Primary Keys are very important as the database can use that as an index to allow us to quickly find a record. This is usually via a good *hashing algorithm*.

When we are doing a join, this allows us to quickly find any two items we are wishing to compare.

This is why we really prefer our joins to use the assigned table keys if possible.

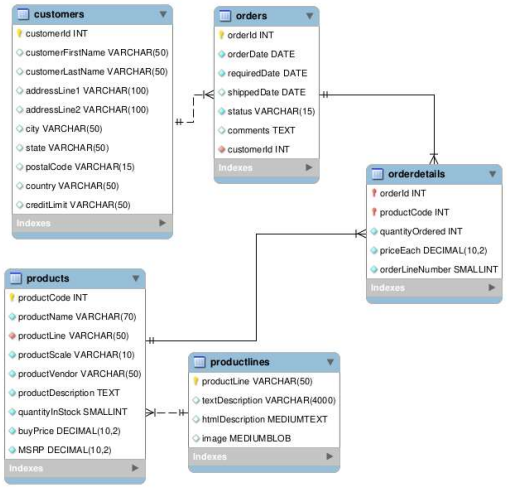
Keys (Cont'd)

Keys can also aid greatly in ensuring data integrity.

If it is the case that every record should be unique (order #s, for example), then using that as the primary key will enforce that condition.

A necessary relationship between data in different tables can be enforced with foreign keys. If an Order table uses a customer ID as a foreign key, they will ensure that a matching customer exists in a Customer data table.

A common default Primary Key is simply an integer that might be auto-incremented as each new record is added. (In *Pandas* we always have a row number.)



Hashing and Indexing

You won't get very far in data science without hearing about how hashing is used to organize important data. It is by far the most common way to index any substantial RDBMS table.

In this context, a hashing algorithm's job is to take a key and use it to generate an index into the data storage.

From the mathematical perspective, it takes some string - of possibly arbitrary length - and generates a fixed size number. In general this means that it can't guarantee the uniqueness of that number, but you hope it does a good job of distributing the indices around. And, you hope it is fast.

Create

So far we've just analyzed data, let's see how to create some things.

Creating a new database is simple in sqlite3 from the command line:

```
sqlite3 newdatabase.db
```

Creating a table:

```
CREATE TABLE vendors (vendorId INTEGER PRIMARY KEY, vendorName varchar(100)
DEFAULT NULL, addressLine1 varchar(100) DEFAULT NULL, addressLine2
varchar(100) DEFAULT NULL, city varchar(50) DEFAULT NULL, state varchar(50)
DEFAULT NULL, postalCode varchar(15) DEFAULT NULL, country varchar(50)
DEFAULT NULL);
```

Altering Existing Tables

```
ALTER TABLE vendors ADD COLUMN comment VARCHAR(200);
sqlite> pragma table_info(vendors);
+-----+-------------+-------------+---------+------------+----+
| cid |     name    |     type    | notnull | dflt_value | pk |
+-----+-------------+-------------+---------+------------+----+
| 0   | vendorId    | INTEGER     | 0       |            | 1  |
| 1   | vendorName  | varchar(100)| 0       | NULL       | 0  |
| 2   | addressLine1| varchar(100)| 0       | NULL       | 0  |
| 3   | addressLine2| varchar(100)| 0       | NULL       | 0  |
| 4   | city        | varchar(50) | 0       | NULL       | 0  |
| 5   | state       | varchar(50) | 0       | NULL       | 0  |
| 6   | postalCode  | varchar(15) | 0       | NULL       | 0  |
| 7   | country     | varchar(50) | 0       | NULL       | 0  |
| 8   | comment     | VARCHAR(200)| 0       |            | 0  |
+-----+-------------+-------------+---------+------------+----+
```

Inserting Data

```
INSERT INTO Vendors (vendorName,addressLine1,addressLine2,city,state,postalCode,country,comment)
VALUES (‘SoundBlasters','123 Imaginary Place',NULL,'Sampletown','PA','15217','USA',NULL);

SELECT * FROM vendors;
 +----------+---------------+---------------------+-------------+-----------+-------+-----------+---------+---------+
| vendorId |  vendorName   |    addressLine1     | addressLine2 |    city    | state | postalCode | country | comment |
+----------+---------------+---------------------+-------------+-----------+-------+-----------+---------+---------+
| 1        | SoundBlasters | 123 Imaginary Place |              | Sampletown | PA    | 15217      | USA     |         |
+----------+---------------+---------------------+-------------+-----------+-------+-----------+---------+---------+
```

Updating Data

```
UPDATE Vendors SET vendorname = 'Soundclashers' WHERE vendorId = 1;

SELECT * FROM vendors;
```

```
 +----------+---------------+---------------------+-------------+------------+-------+------------+---------+---------+
| vendorId |  vendorName   |    addressLine1     | addressLine2 |    city    | state | postalCode | country | comment |
+----------+---------------+---------------------+-------------+------------+-------+------------+---------+---------+
| 1        | Soundclashers | 123 Imaginary Place |              | Sampletown | PA    | 15217      | USA     |         |
+----------+---------------+---------------------+-------------+------------+-------+------------+---------+---------+
```

Deleting Data

```
sqlite> DELETE FROM vendors WHERE city = 'Sampletown';
sqlite> SELECT * FROM vendors;
```

```
sqlite> DROP TABLE vendors;
sqlite> SELECT * FROM vendors;
Parse error: no such table: vendors
```

```
sqlite> DROP VIEW TopCustomers;
```

SQL Injection Attacks

Consider a typical website, which asks the user to enter their username. It then constructs a string to use in querying the database for that user's info:

```
var statement = "SELECT * FROM users WHERE name = '" + userName + "'";
```

This seems reasonable. However, what if a nefarious user enters this as their username:

```
a';DROP TABLE users; SELECT * FROM userinfo WHERE 't' = 't
```

Then the SQL command that gets constructed is

```
SELECT * FROM users WHERE name = 'a';DROP TABLE users; SELECT * FROM userinfo
WHERE 't' = 't';
```

And we have not only exposed all user data, but also deleted our users table.

Good practices can help to mitigate this and sanitize the inputs. Be aware.

ACID

ACID is a set of properties of database transactions intended to guarantee data validity despite errors, power failures, and other mishaps. In the context of databases, a sequence of database operations that satisfies the ACID properties (which can be perceived as a single logical operation on the data) is called a transaction. For example, a transfer of funds from one bank account to another, even involving multiple changes such as debiting one account and crediting another, is a single transaction.

Acid (Cont'd)

Atomicity

An atomic system must guarantee atomicity in each and every situation, including power failures, errors, and crashes. A guarantee of atomicity prevents updates to the database from occurring only partially, which can cause greater problems than rejecting the whole series outright.

Consistency

Consistency ensures that a transaction can only bring the database from one consistent state to another, preserving database invariants: any data written to the database must be valid according to all defined rules, including constraints, cascades, triggers, and any combination thereof. This prevents database corruption by an illegal transaction.

ACID (Cont'd)

Isolation

Isolation ensures that concurrent execution of transactions leaves the database in the same state that would have been obtained if the transactions were executed sequentially.

Durability

Durability guarantees that once a transaction has been committed, it will remain committed even in the case of a system failure (e.g., power outage or crash). This usually means that completed transactions (or their effects) are recorded in non-volatile memory.

Triggers

Triggers are stored actions that take effect under certain conditions.

```
CREATE TRIGGER upd_check BEFORE UPDATE ON account
FOR EACH ROW
BEGIN
   IF NEW.amount < 0 THEN
         SET NEW.amount = 0;
      ELSEIF NEW.amount > 100 THEN
         SET NEW.amount = 100;
      END IF;
END;
```

Procedures

Procedures are simply stored collections of SQL commands.

We can create a trivial one like so:

```
CREATE PROCEDURE SelectAllCustomers
AS
SELECT * FROM Customers
GO;
EXEC SelectAllCustomers;
```

In real usage significant business logic is often stored in procedures with variables, conditionals, etc.

Part 2: Pandas

A Flexible Data Analysis Tool

Pandas

Pandas has become the standard Python way to input, manipulate, and write basic data.

It integrates well with other tools (e.g. visualization with Matplotlib).

It has limitations which is why SQL and big data techniques are still essential, but for quick and dirty/limited applications it's very efficient.

It's easy to add to an existing python environment.

Pandas setup

```
# for pandas we will load a module and copy a file

ssh login.delta.ncsa.illinois.edu
module load anaconda3_cpu
# “.” refers to your present working directory
cp /projects/becs/bgill/pandas/titanic.csv .
python
>>> import pandas as pd
>>>
```

A Modest (but Titanic!) Dataset

We will begin our exploration of Pandas using a well known dataset drawn from the infamous Titanic disaster. It has a variety of data on each of 891 passengers.

Amongst the typical demographic data is included their survival. It enables an interesting, if somewhat morbid, analysis to determine the foremost factors in survival.

Women and children first? Or, save the rich?

Dataframes

Dataframes are tables represented as Python objects.

```
df = pd.DataFrame(
       {“a” : [4,5,6],
        “b” : [7,8,9],
        “c” : [10,11,12]}
       )
```

A Few Simple Tools

Get the number of rows and columns:

df.shape

Summarize data:

sum(), min(), max(), mean(), median(), etc.

Drop rows where any column has missing/null data:

df.dropna()

There's a lot of nice cheatsheets out there, here's a good one:

https://pandas.pydata.org/Pandas_Cheat_Sheet.pdf

Merges (joins)

```
pd.merge(adf, bdf, how='left', on='column_name')
```

"how" can be left, right, inner or outer much like sql

Pandas Examples

```
>>> import pandas as pd
>>> titanic = pd.read_csv("titanic.csv")
>>> titanic
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S
..	...	...	...	...	...	...	...	...	...	...	...	...
886	887	0	2	Montvila, Rev. Juozas	male	27.0	0	0	211536	13.0000	NaN	S
887	888	1	1	Graham, Miss. Margaret Edith	female	19.0	0	0	112053	30.0000	B42	S
888	889	0	3	Johnston, Miss. Catherine Helen "Carrie"	female	NaN	1	2	W./C. 6607	23.4500	NaN	S
889	890	1	1	Behr, Mr. Karl Howell	male	26.0	0	0	111369	30.0000	C148	C
890	891	0	3	Dooley, Mr. Patrick	male	32.0	0	0	370376	7.7500	NaN	Q

```
[891 rows x 12 columns]
```

Input Formats

csv and JSON are very common, but Pandas can handle a lot of types of input.

Format Type	Data Description	Reader	Writer
text	CSV	read_csv	to_csv
text	JSON	read_json	to_json
text	HTML	read_html	to_html
text	Local clipboard	read_clipboard	to_clipboard
binary	MS Excel	read_excel	to_excel
binary	OpenDocument	read_excel	
binary	HDF5 Format	read_hdf	to_hdf
binary	Feather Format	read_feather	to_feather
binary	Parquet Format	read_parquet	to_parquet
binary	Msgpack	read_msgpack	to_msgpack
binary	Stata	read_stata	to_stata
binary	SAS	read_sas	
binary	Python Pickle Format	read_pickle	to_pickle
SQL	SQL	read_sql	to_sql
SQL	Google Big Query	read_gbq	to_gbq

http://www.datasciencelovers.com/python-for-data-science/pandas-data-input-and-output/

Columns

Survived	Survival (0 = No; 1 = Yes)
Pclass	Passenger Class (1 = 1st; 2 = 2nd; 3 = 3rd)
Name	Name
Sex	
Age	
SibSp	Number of Siblings/Spouses Aboard
Parch	Number of Parents/Children Aboard
Ticket	Ticket Number
Fare	Fare (British pound)
Cabin	Cabin number
Embarked	Port of Embarkation (C = Cherbourg; Q = Queenstown; S = Southampton)

DataFrame Queries

```
>>> titanic["Name"]
0                                Braund, Mr. Owen Harris
1      Cumings, Mrs. John Bradley (Florence Briggs Th...
2                                 Heikkinen, Miss. Laina
3           Futrelle, Mrs. Jacques Heath (Lily May Peel)
4                               Allen, Mr. William Henry
                             ...
886                               Montvila, Rev. Juozas
887                        Graham, Miss. Margaret Edith
888            Johnston, Miss. Catherine Helen "Carrie"
889                               Behr, Mr. Karl Howell
890                                 Dooley, Mr. Patrick
Name: Name, Length: 891, dtype: object
```

DataFrame Queries

```
>>> titanic[["Age","Sex"]]
      Age     Sex
0    22.0    male
1    38.0  female
2    26.0  female
3    35.0  female
4    35.0    male
..    ...     ...
886  27.0    male
887  19.0  female
888   NaN  female
889  26.0    male
890  32.0    male
[891 rows x 2 columns]
```

Dataframe Conditional Queries

```
>>> titanic[titanic["Age"]>30]
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S
6	7	0	1	McCarthy, Mr. Timothy J	male	54.0	0	0	17463	51.8625	E46	S
11	12	1	1	Bonnell, Miss. Elizabeth	female	58.0	0	0	113783	26.5500	C103	S
..	...	...	...	...	...	...	...	...	...	...	...	...
873	874	0	3	Vander Cruyssen, Mr. Victor	male	47.0	0	0	345765	9.0000	NaN	S
879	880	1	1	Potter, Mrs. Thomas Jr (Lily Alexenia Wilson)	female	56.0	0	1	11767	83.1583	C50	C
881	882	0	3	Markun, Mr. Johann	male	33.0	0	0	349257	7.8958	NaN	S
885	886	0	3	Rice, Mrs. William (Margaret Norton)	female	39.0	0	5	382652	29.1250	NaN	Q
890	891	0	3	Dooley, Mr. Patrick	male	32.0	0	0	370376	7.7500	NaN	Q

```
[305 rows x 12 columns]
```

DataFrame Sorting

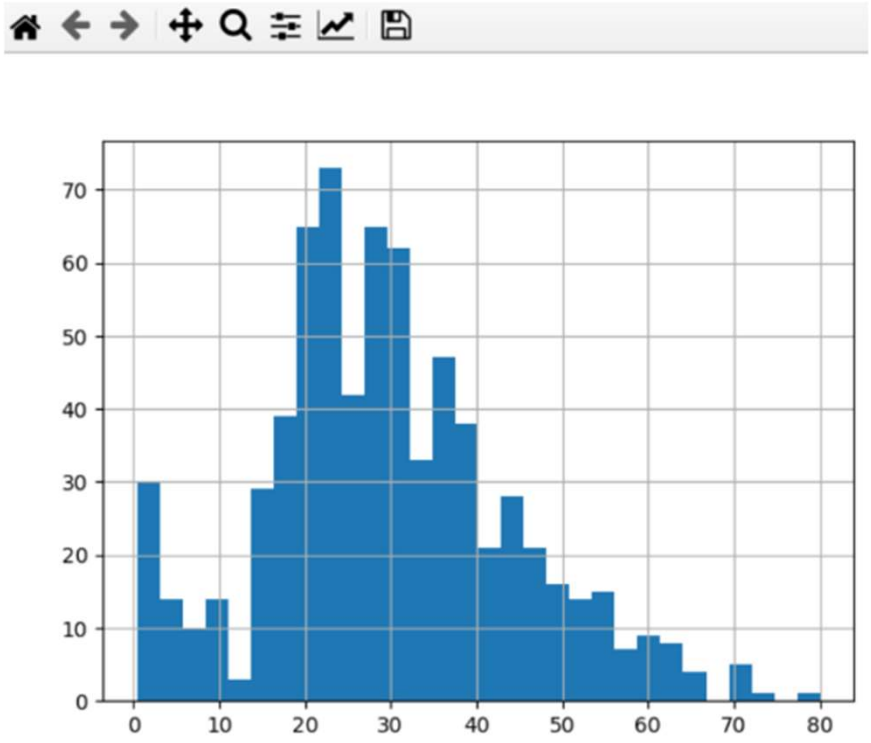
```
>>> titanic.sort_values(by="Age")[["Name","Age"]]
                                      Name   Age
803           Thomas, Master. Assad Alexander  0.42
755                 Hamalainen, Master. Viljo  0.67
644                    Baclini, Miss. Eugenie  0.75
469              Baclini, Miss. Helene Barbara  0.75
78               Caldwell, Master. Alden Gates  0.83
..                                         ...   ...
859                           Razi, Mr. Raihed   NaN
863          Sage, Miss. Dorothy Edith "Dolly"   NaN
868                van Melkebeke, Mr. Philemon   NaN
878                           Laleff, Mr. Kristo   NaN
888  Johnston, Miss. Catherine Helen "Carrie"   NaN
[891 rows x 2 columns]
```

```
>>> titanic.sort_values(by="Age")[["Name","Age"]][0:10]
                                    Name   Age
803  Thomas, Master. Assad Alexander  0.42
755        Hamalainen, Master. Viljo  0.67
644           Baclini, Miss. Eugenie  0.75
469    Baclini, Miss. Helene Barbara  0.75
78     Caldwell, Master. Alden Gates  0.83
831  Richards, Master. George Sibley  0.83
305   Allison, Master. Hudson Trevor  0.92
827            Mallet, Master. Andre  1.00
381      Nakid, Miss. Maria ("Mary")  1.00
164     Panula, Master. Eino Viljami  1.00
```

Plotting

```
import matplotlib.pyplot as plt
titanic["Age"].hist(bins=30)
plt.show()
```

Note that this probably won't work in your shell here, but it's trivial to set up on your laptop.



Pandas Assignment

Can you find a significant factor in the data which could be used to predict survival rates?

I will suggest you focus on one variable at a time.

Note that there are many possible answers. Going from a simple hypothesis ("Maybe people from Cherbourg are unlucky?") to a more complex formula incorporating multiple variables - with the goal of a more accurate prediction - is the path of data analysis. This is our first step on that journey

SQL Assignment

Times are tough. Corporate has tasked you with identifying the three least popular genres of music ranked by number of tracks purchased.

Submit the answer along with the queries used to arrive at your solution. It's fine to use interstitial views but be sure to include the queries that created those as well (you can use the command ".schema TopCustomers" to see it if you've already scrolled past it).