Section #12

Spring 2019

Topics

John Snow

Charles Minard

Data-Ink Ratio

Color Usage

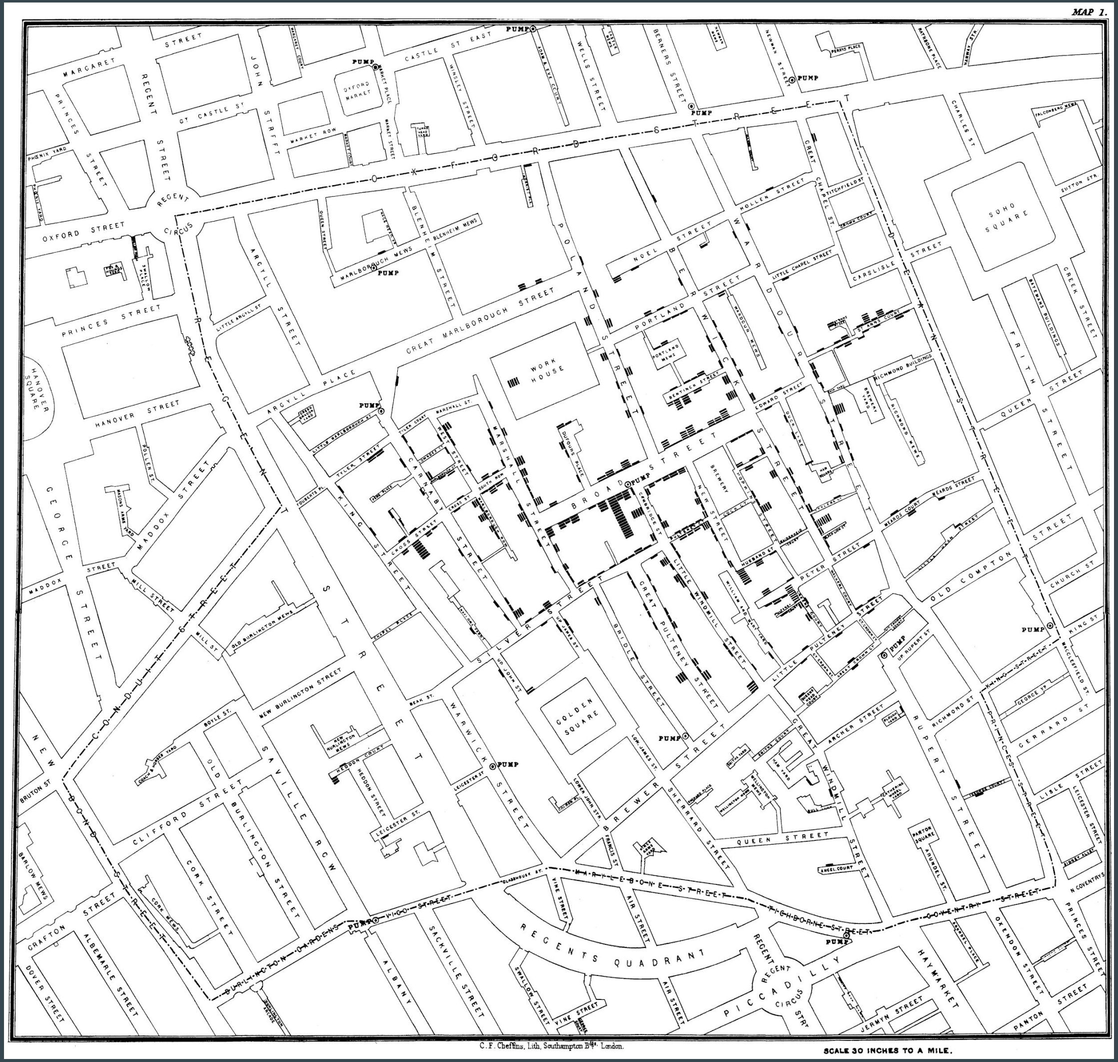
Star and Snowflake DB Patterns

Django ORM Examples

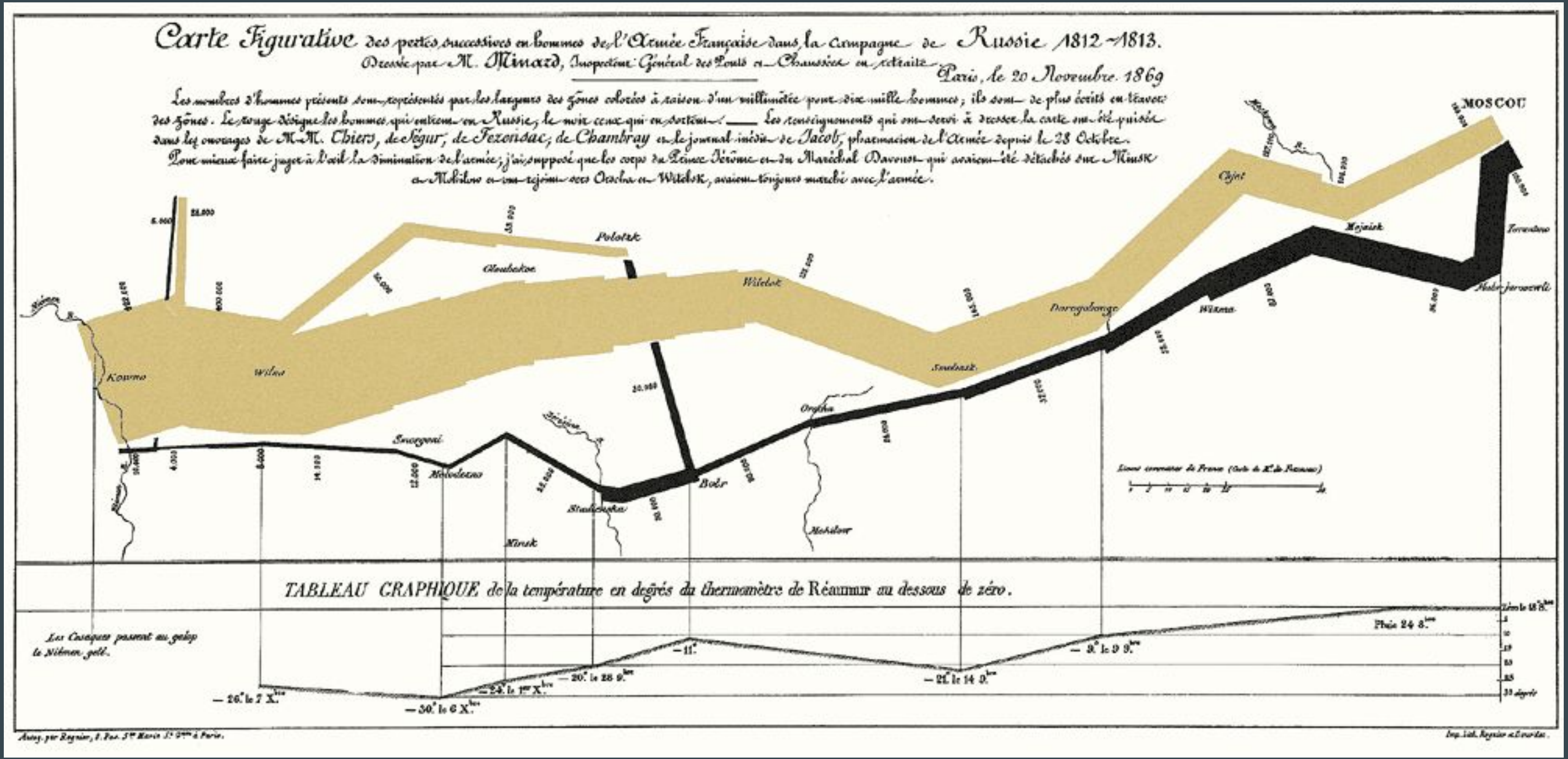
SQLAlchemy ORM Examples

John Snow

Mapping the 1854 London Cholera Outbreak



Charles Minard



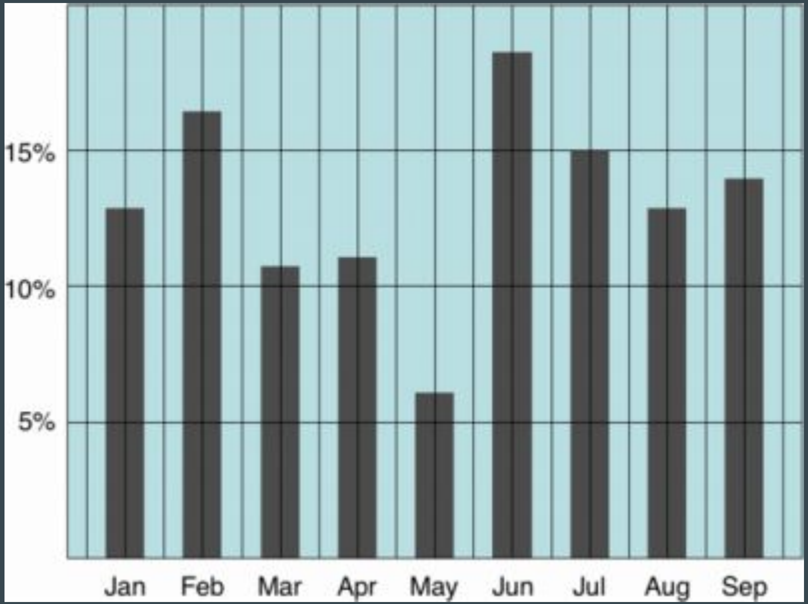
Data-Ink Ratio

$$\text{Data-ink ratio} = \frac{\text{Data-ink}}{\text{Total ink used to print the graphic}}$$

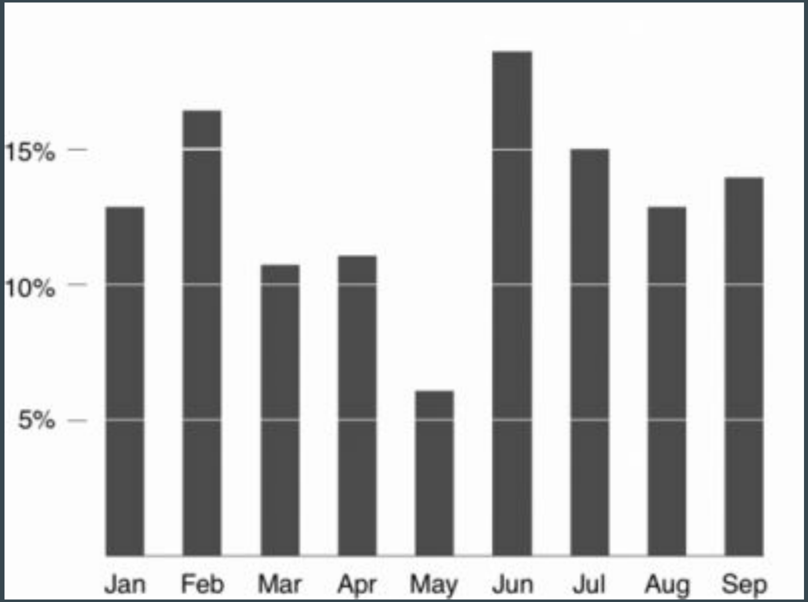
$$= \text{proportion of a graphic's ink devoted to the non-redundant display of data-information}$$

$$= 1.0 - \text{proportion of a graphic that can be erased}$$

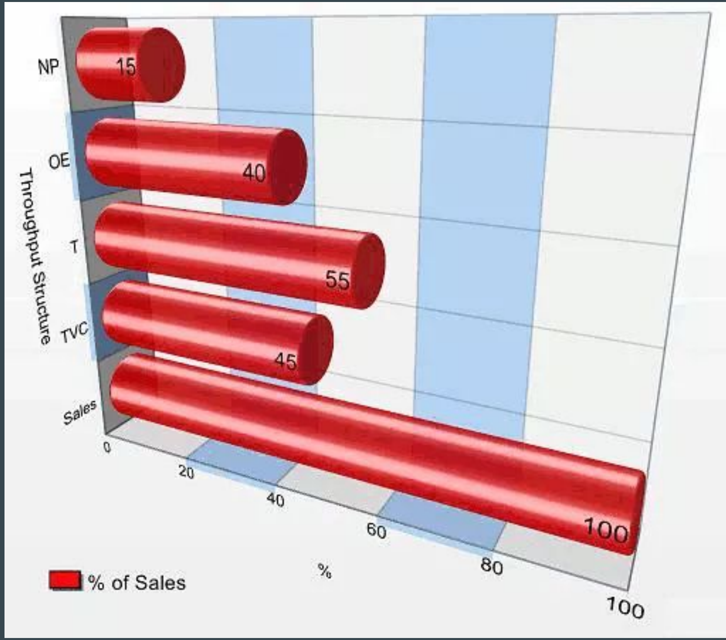
Low Data-Ink Ratio

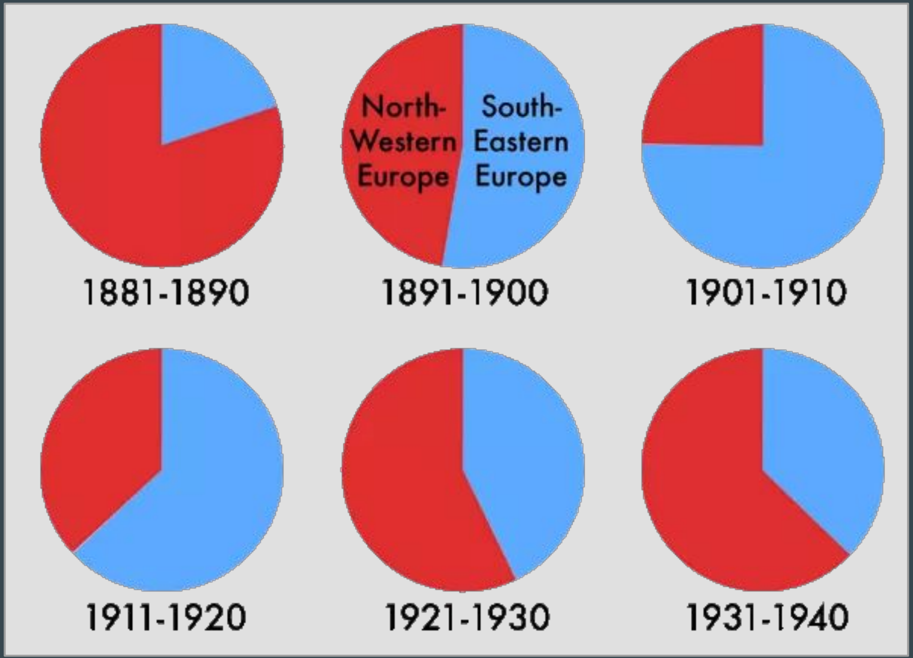


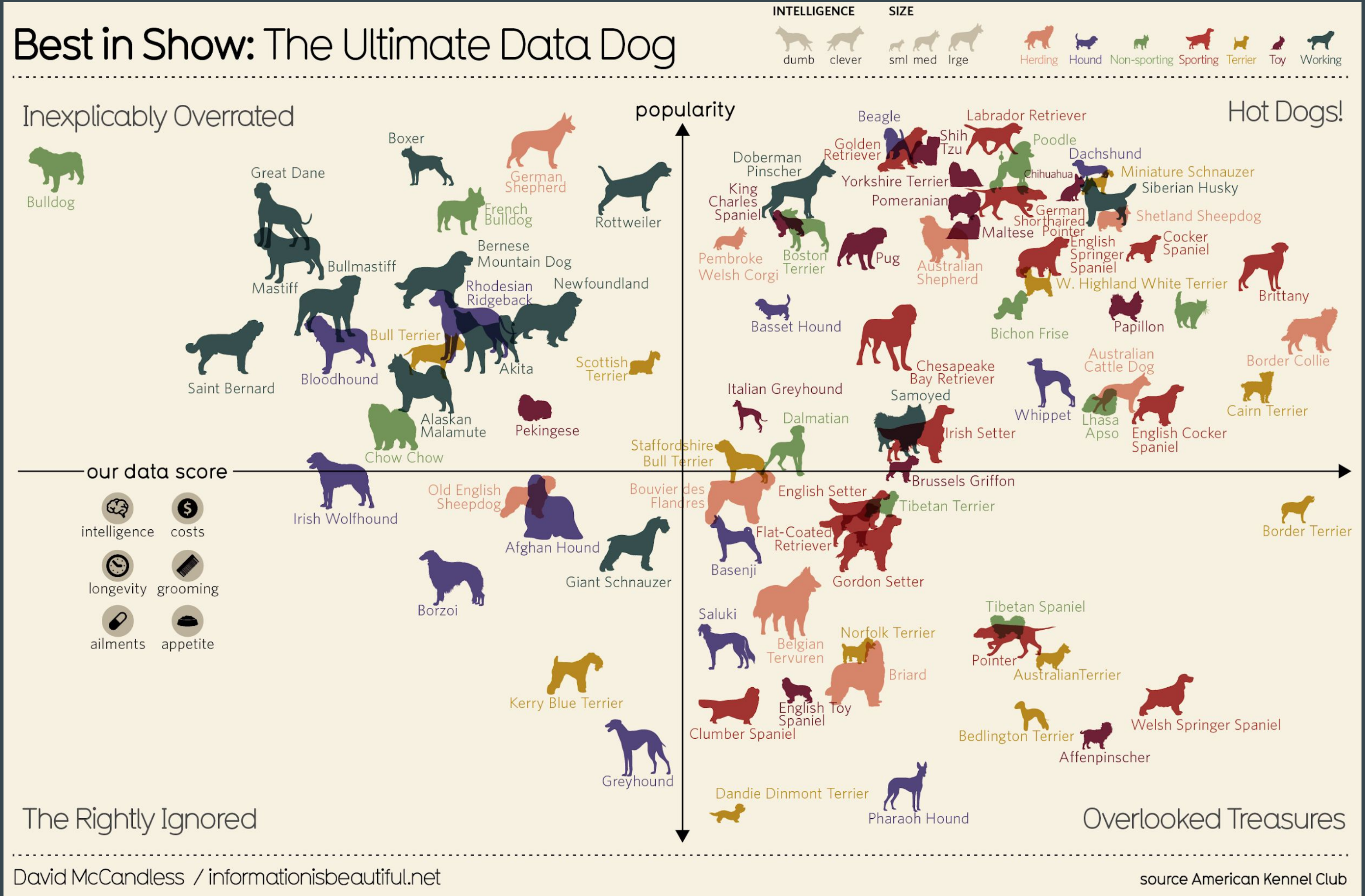
High Data-Ink Ratio



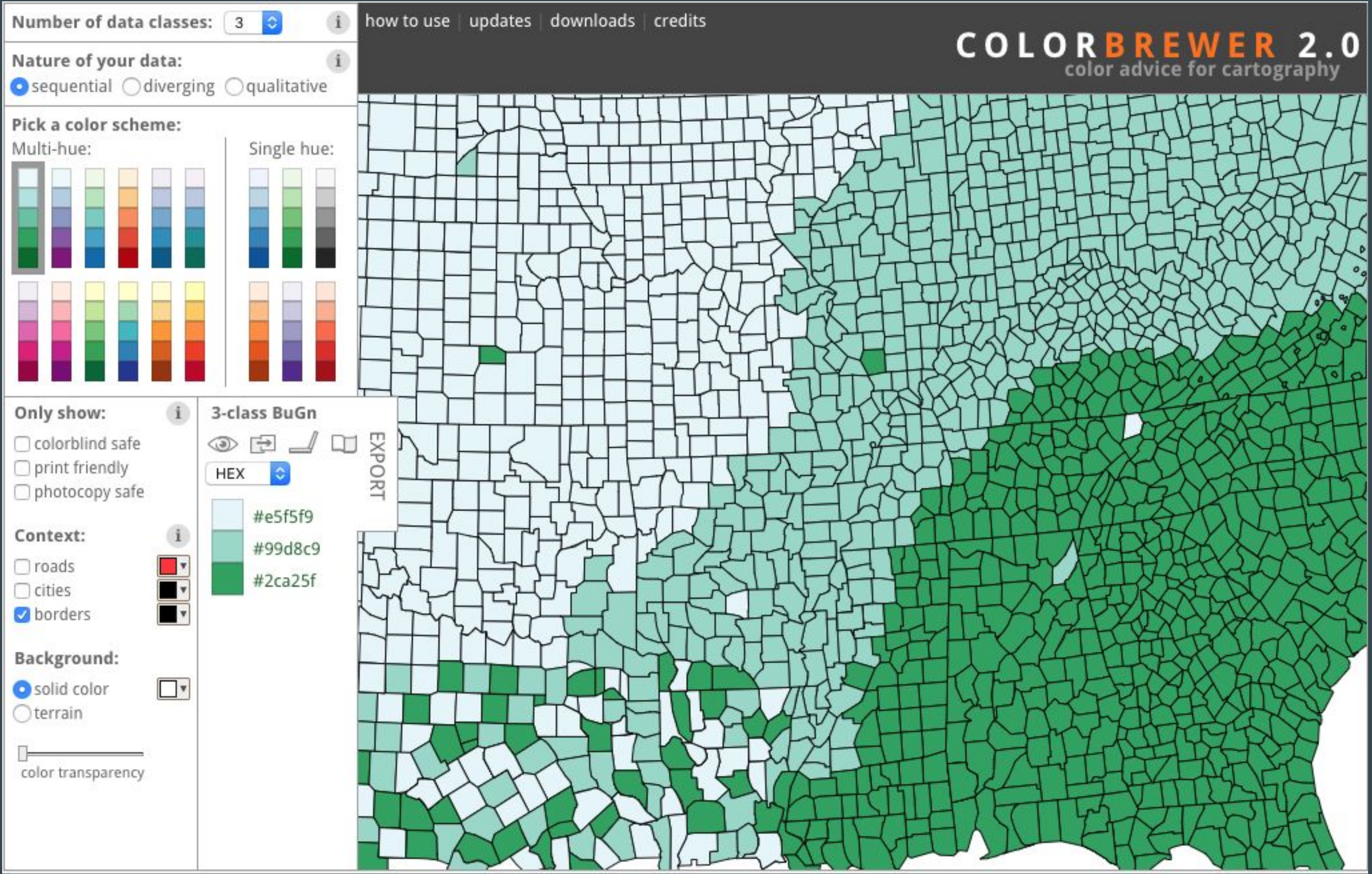
Rich Vis, Poor Vis



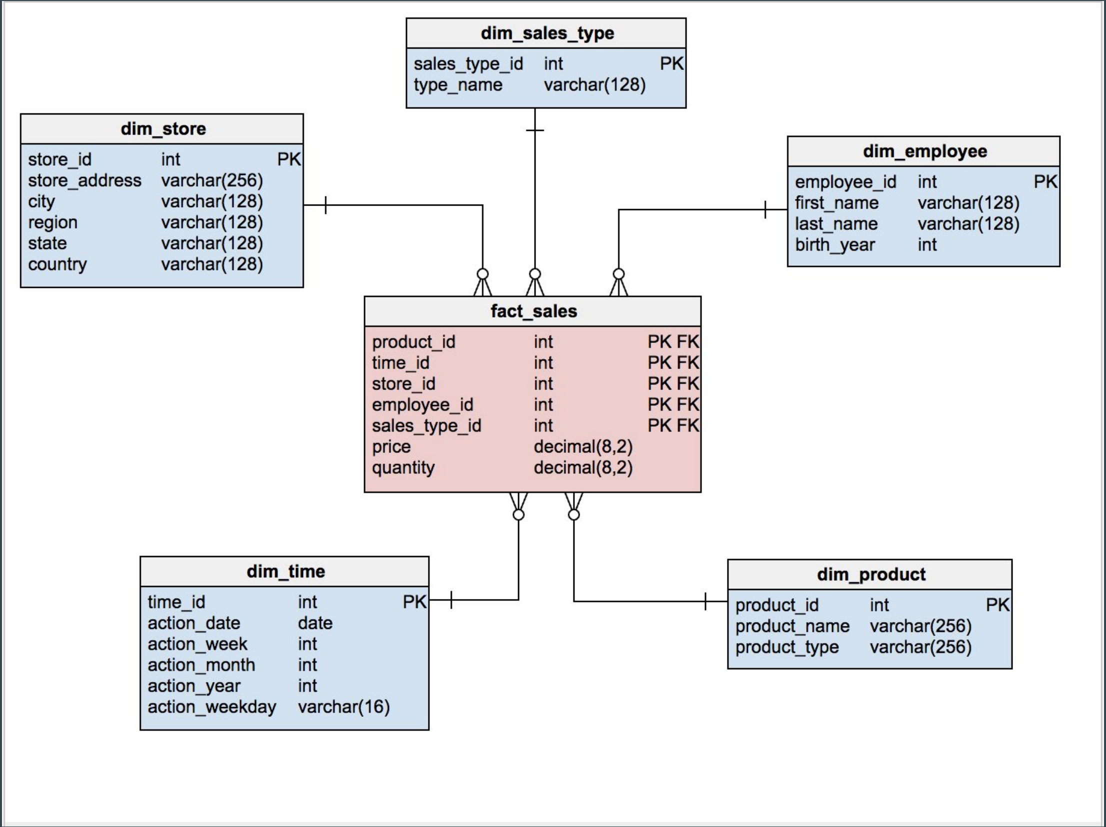




Best in Show: The Ultimate Data Dog
INTELLIGENCE
dumb
clever
SIZE
sml
med
lrge
Herding
Hound
Non-sporting
Sporting
Terrier
Toy
Working
Inexplicably Overrated
popularity
Hot Dogs!
our data score
intelligence
costs
longevity
grooming
ailments
appetite
The Rightly Ignored
Overlooked Treasures
Bulldog
Boxer
Great Dane
German Shepherd
French Bulldog
Rottweiler
Mastiff
Bullmastiff
Bernese Mountain Dog
Rhodesian Ridgeback
Newfoundland
Saint Bernard
Bloodhound
Bull Terrier
Akita
Scottish Terrier
Alaskan Malamute
Pekingese
Chow Chow
Irish Wolfhound
Old English Sheepdog
Afghan Hound
Giant Schnauzer
Borzoi
Kerry Blue Terrier
Greyhound
Staffordshire Bull Terrier
Bouvier des Flandres
Beagle
Labrador Retriever
Golden Retriever
Shih Tzu
Poodle
Doberman Pinscher
Dachshund
Chihuahua
Miniature Schnauzer
Siberian Husky
Yorkshire Terrier
King Charles Spaniel
Pomeranian
German Shorthaired Pointer
Shetland Sheepdog
Maltese
English Springer Spaniel
Cocker Spaniel
Pembroke Welsh Corgi
Boston Terrier
Pug
Australian Shepherd
W. Highland White Terrier
Brittany
Basset Hound
Bichon Frise
Papillon
Border Collie
Chesapeake Bay Retriever
Australian Cattle Dog
Italian Greyhound
Samoyed
Whippet
Cairn Terrier
Dalmatian
Irish Setter
Lhasa Apso
English Cocker Spaniel
Brussels Griffon
English Setter
Tibetan Terrier
Border Terrier
Flat-Coated Retriever
Basenji
Gordon Setter
Saluki
Belgian Tervuren
Norfolk Terrier
Tibetan Spaniel
Pointer
AustralianTerrier
Briard
English Toy Spaniel
Clumber Spaniel
Welsh Springer Spaniel
Bedlington Terrier
Affenpinscher
Dandie Dinmont Terrier
Pharaoh Hound
David McCandless / informationisbeautiful.net
source American Kennel Club


Number of data classes: 3
Nature of your data:
sequential
diverging
qualitative
Pick a color scheme:
Multi-hue:
Single hue:
Only show:
colorblind safe
print friendly
photocopy safe
Context:
roads
cities
borders
Background:
solid color
terrain
color transparency
3-class BuGn
EXPORT
HEX
#e5f5f9
#99d8c9
#2ca25f
how to use
updates
downloads
credits
COLORBREWER 2.0
color advice for cartography

Star Database



Often found in data marts/data warehousing applications

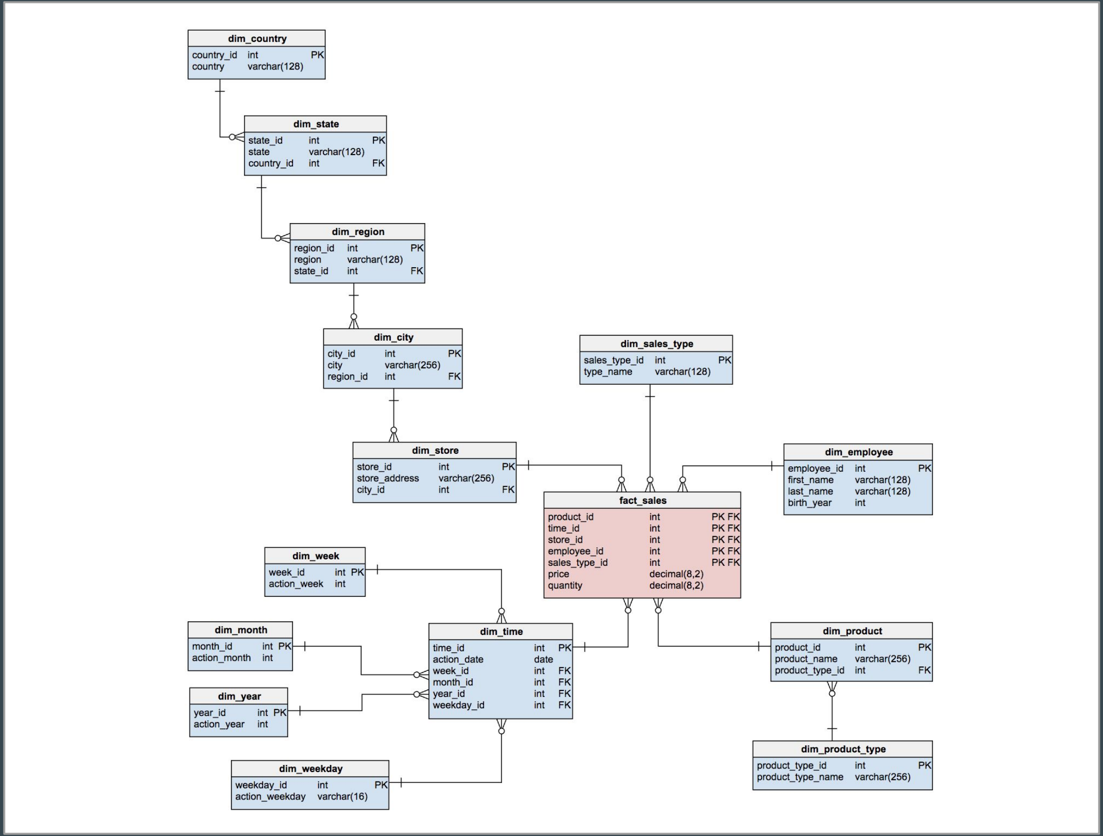
Central fact table contains detail information from surrounding dimension tables as well as foreign keys

Detail data in the fact table may be raw or aggregated from values in dimension tables

PK is often composite of all foreign key values

Flatter/simpler than snowflake

Snowflake Database



Same data as star, but normalized dimension tables create a distinctive snowflake pattern

Data Warehousing: Further Resources

Apache orc™

HOME DOCUMENTATION TALKS NEWS HELP DEVELOP

the smallest, fastest columnar storage for Hadoop workloads.

ACID Support

Includes support for ACID transactions and snapshot isolation

ACID support →

Built-in Indexes

Jump to the right row with indexes including minimum, maximum, and bloom filters for each column.

ORC's indexes →

Complex Types

Supports all of Hive's types including the compound types: structs, lists, maps, and unions

ORC's types →

Star vs Snowflake

- Snowflake schemas will use less space to store dimension tables. This is because as a rule any normalized database produces far fewer redundant records .
- Denormalized data models increase the chances of data integrity problems. These issues will complicate future modifications and maintenance as well.
- A snowflake schema query can be more complex. Because the dimension tables are normalized, we need to dig deeper to get the details. We have to add another JOIN for every new level inside the same dimension.
- In the star schema, we only join the fact table with those dimension tables we need. At most, we'll have only one JOIN per dimension table. And if we're not using a dimension table, we don't even need to bother with it. In the snowflake schema query, we don't know how deep we'll have to go to get the right dimension level, so that complicates the process of writing queries.
- Joining two tables takes time because the DBMS takes longer to process the request. Two tables may be placed in close proximity in our model, but they may not be located nowhere near each other on the disk. There is a better possibility that data will be physically closer on the disk if it lives inside the same table.

http://www.vertabelo.com/blog/technical-articles/data-warehouse-modeling-star-schema-vs-snowflake-schema

Data Warehousing: Further Resources

Amazon Redshift

Fast, simple, cost-effective data warehouse that can extend queries to your data lake

Get started with a free 2-month trial

Follow the Getting Started Guide

Amazon Redshift is a fast, scalable data warehouse that makes it simple and cost-effective to analyze all your data across your data warehouse and data lake. Redshift delivers ten times faster performance than other data warehouses by using machine learning, massively parallel query execution, and columnar storage on high-performance disk. You can setup and deploy a new data warehouse in minutes, and run queries across petabytes of data in your Redshift data warehouse, and exabytes of data in your data lake built on Amazon S3. You can start small for just $0.25 per hour and scale to $250 per terabyte per year, less than one-tenth the cost of other solutions.

To create your first Amazon Redshift data warehouse, follow our Getting Started Guide and get the most out of your experience. Contact us to request support for your proof-of-concept or evaluation.To accelerate your migration to Amazon Redshift, you can use the AWS Database Migration Service (DMS) free for six months. Learn more »

Django ORM: Joining Tables

prefetch_related(*lookups)¶

Returns a QuerySet that will automatically retrieve, in a single batch, related objects for each of the specified lookups.

This has a similar purpose to select_related, in that both are designed to stop the deluge of database queries that is caused by accessing related objects, but the strategy is quite different.

select_related works by creating an SQL join and including the fields of the related object in the SELECT statement. For this reason, select_related gets the related objects in the same database query. However, to avoid the much larger result set that would result from joining across a 'many' relationship, select_related is limited to single-valued relationships - foreign key and one-to-one.

prefetch_related, on the other hand, does a separate lookup for each relationship, and does the 'joining' in Python. This allows it to prefetch many-to-many and many-to-one objects, which cannot be done using select_related, in addition to the foreign key and one-to-one relationships that are supported by select_related.

Prefetch_related example (from Django docs)

```
from django.db import models

class Topping(models.Model):
    name = models.CharField(max_length=30)

class Pizza(models.Model):
    name = models.CharField(max_length=50)
    toppings = models.ManyToManyField(Topping)

    def __str__(self):
        return "%s (%s)" % (
            self.name,
            ", ".join(topping.name for topping in self.toppings.all()),
        )
```

Prefetch_related example (from Django docs)

```
>>> Pizza.objects.all()
["Hawaiian (ham, pineapple)", "Seafood (prawns, smoked salmon)"...
```

The problem with this is that every time **`Pizza.__str__()`** asks for **`self.toppings.all()`** it has to query the database, so**`Pizza.objects.all()`** will run a query on the Toppings table for **every** item in the Pizza **`QuerySet`**.

We can reduce to just two queries using **`prefetch_related`**:

```
>>> Pizza.objects.all().prefetch_related('toppings')
```

This implies a **`self.toppings.all()`** for each **`Pizza`**; now each time **`self.toppings.all()`** is called, instead of having to go to the database for the items, it will find them in a prefetched **`QuerySet`** cache that was populated in a single query.

SQLAlchemy ORM

```
from sqlalchemy import Integer, ForeignKey, String,
Column
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship

Base = declarative_base()

class Customer(Base):
    __tablename__ = 'customer'
    id = Column(Integer, primary_key=True)
    name = Column(String)

    billing_address_id = Column(Integer,
ForeignKey("address.id"))
    shipping_address_id = Column(Integer,
ForeignKey("address.id"))
```

```
    billing_address = relationship("Address")
    shipping_address = relationship("Address")

class Address(Base):
    __tablename__ = 'address'
    id = Column(Integer, primary_key=True)
    street = Column(String)
    city = Column(String)
    state = Column(String)
    zip = Column(String)
```

SQLAlchemy ORM: Joining on FK

```
class Customer(Base):
    __tablename__ = 'customer'
    id = Column(Integer, primary_key=True)
    name = Column(String)

    billing_address_id = Column(Integer, 
ForeignKey("address.id"))
    shipping_address_id = Column(Integer, 
ForeignKey("address.id"))

    billing_address = relationship("Address")
    shipping_address = relationship("Address")
```

```
class Customer(Base):
    __tablename__ = 'customer'
    id = Column(Integer, primary_key=True)
    name = Column(String)

    billing_address_id = Column(Integer, 
ForeignKey("address.id"))
    shipping_address_id = Column(Integer, 
ForeignKey("address.id"))

    billing_address = relationship("Address", 
foreign_keys=[billing_address_id])
    shipping_address = relationship("Address", 
foreign_keys=[shipping_address_id])
```

Q&A