UMD DATA605 - Big Data Systems

Orchestration with Airflow

Data wrangling

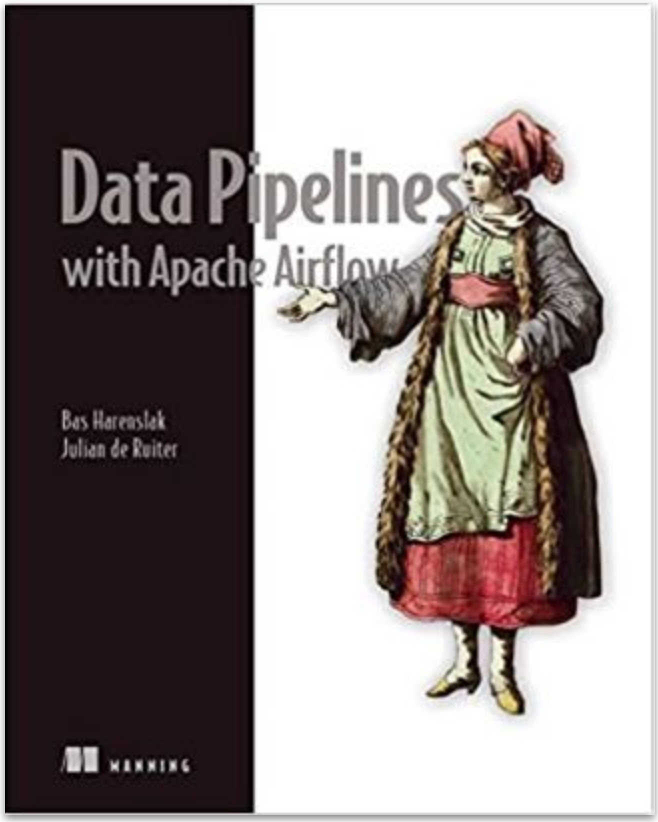
Deployment

Dr. GP Saggese

gsaggese@umd.edu

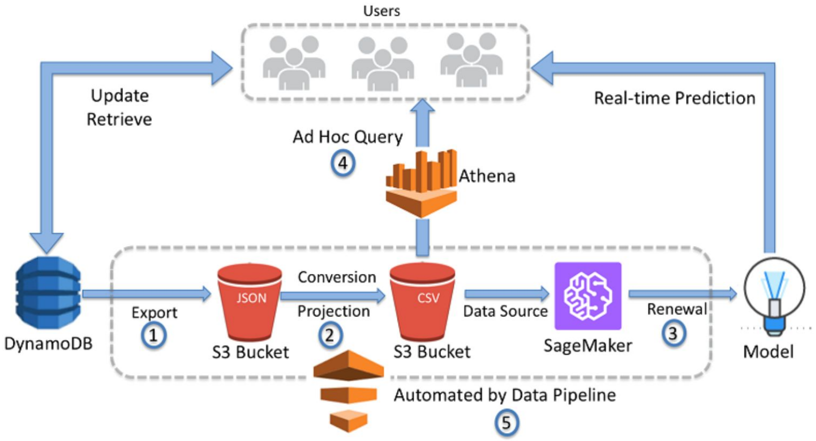
Orchestration - Resources

- Concepts in the slides
- Airflow tutorial
- Web resources
  - Documentation
  - Tutorial
- Mastery
  - Data Pipelines with Apache Airflow



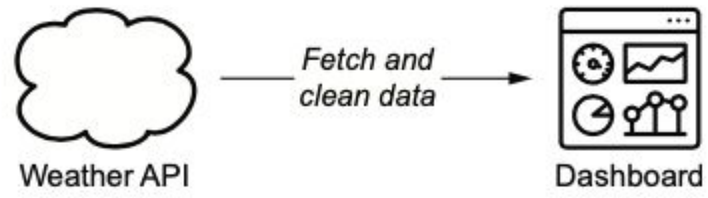
Workflow Managers

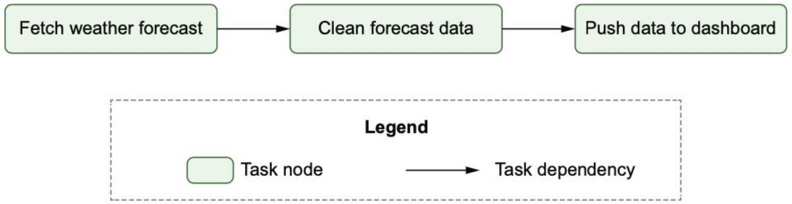
- **Data pipelines** move/transform data across data stores
- **Orchestration problem** = data pipelines require to coordinate jobs across systems
  - Run tasks on a certain schedule
  - Run tasks in a specific order (dependencies)
  - Monitor tasks
    - Notify devops if a job fails
    - Retry on failure
    - Track how long it takes to run)
  - Meet real-time constraints
  - Scale performance

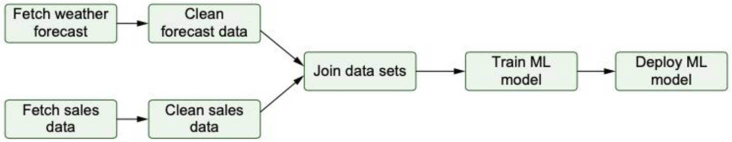


Workflow Managers

- **E.g., live weather dashboard**
  - Fetch the weather data from API
  - Clean / transform the data
  - Push data to the dashboard/ website
- Problems
  - Tasks schedule
  - Tasks dependencies
  - Monitor functionality and performance
  - Quickly one wants to add machine learning
  - Quickly the complexity increases

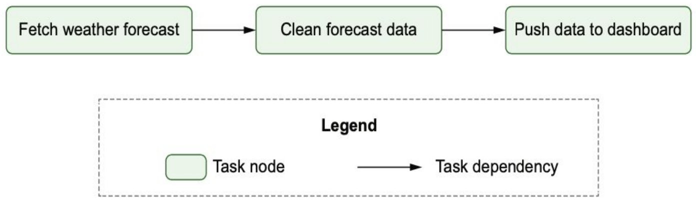


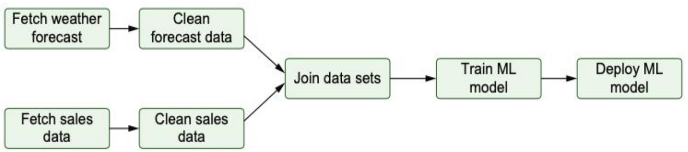




Workflow Managers

- **Workflow managers address the orchestration problem**
  - E.g., Airflow, Luigi, Metaflow, make, cron ...
- **Represent data pipelines as DAGs**
  - Nodes are tasks
  - Direct edges are dependencies
  - A task is executed only when all the ancestors have been executed
  - Independent tasks can be executed in parallel
  - Re-run failed tasks incrementally
- **How to describe data pipelines**
  - Static files (e.g., XML, YAML)
  - Workflows-as-code (e.g., Python in Airflow)
- **Provide scheduling**
  - How to describe what and when to run
- **Provide backfilling and catch-up**
  - Horizontally scalable (e.g., multiple runners)
- **Provide monitoring web interface**





(Apache) Airflow

- Developed at AirBnB in 2015
  - Open-sourced as part of Apache project
- **Batch oriented framework** for building data pipelines (not real-time)
- **Data pipelines**
  - Represented as DAGs
  - Described as Python code
- **Scheduler with rich semantics**
- Web-interface for monitoring
- Large ecosystem
  - Support many DBs
  - Many actions (e.g., emails, pager notifications)
- **Hosted and managed solution**
  - Run Airflow on your laptop (e.g., in tutorial)
  - Managed solution (e.g., AWS)

Airflow: Execution Semantics

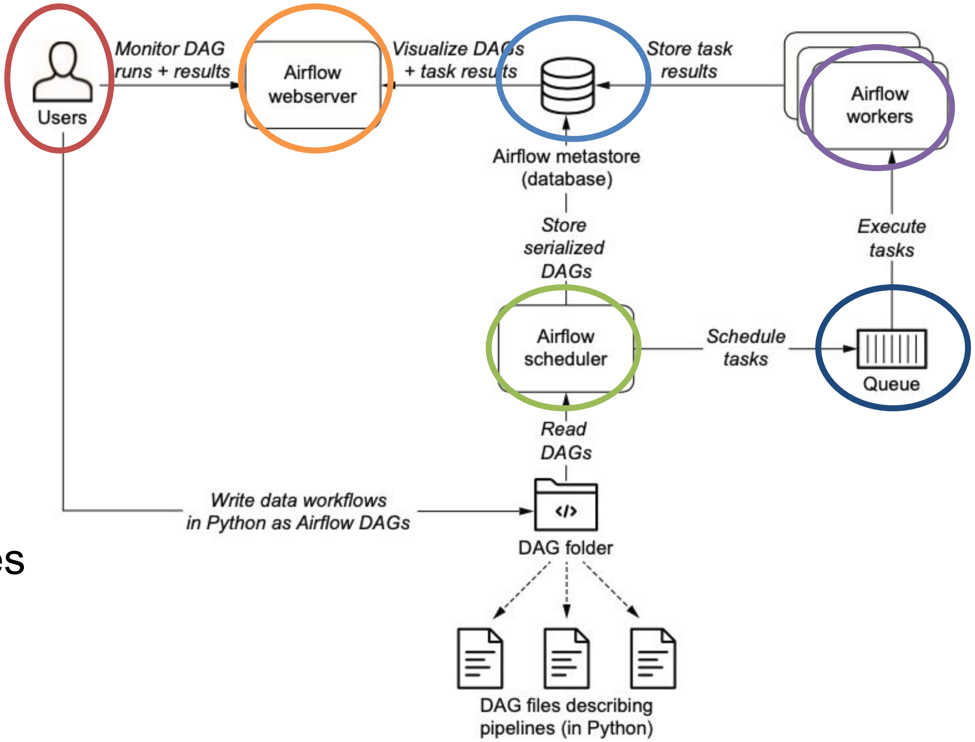
- **Scheduling semantic**
  - Describe when the next scheduling interval is
    - E.g., “every day at midnight”, “every 5 minutes on the hour”
  - Similar to `cron`
- **Retry**
  - If a task fails, it can be re-run (after a wait time) to recover from intermittent failures
- **Incremental processing**
  - Time is divided into intervals given the schedule
  - Execute DAG only for data in that interval, instead of processing the entire data set
- **Catch-up**
  - Run all the missing intervals up to now (e.g., after a downtime)
- **Backfilling**
  - Execute DAG for historical schedule intervals that occurred in the past
  - E.g., if the data pipeline has changed one needs to re-process data from scratch

Airflow: What Doesn't Do Well

- **Not great for streaming pipelines**
  - Better for recurring batch-oriented tasks
  - Time is assumed to be discrete and not continuous
    - E.g., schedule every hour, instead of process data as it comes
- **Prefer static pipelines**
  - DAGs should not change (too much) between runs
- **No data lineage**
  - No tracking of how data is transformed through the pipeline
  - Need to be implemented manually
- **No data versioning**
  - No tracking of updates to the data
  - Need to be implemented manually

Airflow: Components

- **Users (DevOps)**
- **Web-server**
  - Visualize DAGs
  - Monitor DAG runs and results
- **Metastore**
  - Keep the state of the system
  - E.g., what DAG nodes have been executed
- **Scheduler**
  - Parse DAGs
  - Keep track of completed dependencies
  - Add tasks to the execution queue
  - Schedule tasks when time comes
- **Queue**
  - Tasks ready for execution
  - Tasks picked up by a pool of Workers
- **Workers**
  - Pick up tasks from Queue
  - Execute the tasks
  - Register task outcome in Metastore



Airflow: Concepts

- Each DAG run represents a data interval, i.e., an interval between two times
  - E.g., a DAG scheduled `@daily`
  - Each data interval starts at midnight for each day, ends at midnight of next day
- DAG scheduled after data interval has ended
- Logical date
  - Simulate the scheduler running DAG / task for a specific date
  - Even if it is physically run now

Airflow: Tutorial

- From the tutorial for Airflow 2.2.2
- Follow Airflow Tutorial in README from https://github.com/sorrentum/sorrentum/tree/master/sorrentum_sandbox

Airflow: Tutorial

- The script describes the DAG structure as Python code
  - There is no computation inside the DAG code
  - It only defines the DAG structure and the metadata (e.g., about scheduling)
- The `Scheduler` executes the code to build DAG
- `BashOperator` creates a task wrapping a Bash command

airflow/example_dags/tutorial.py

```
from datetime import datetime, timedelta
from textwrap import dedent

# The DAG object; we'll need this to instantiate a DAG
from airflow import DAG

# Operators; we need this to operate!
from airflow.operators.bash import BashOperator
```

Airflow: Tutorial

- Dict with various default params to pass to the DAG constructor
  - E.g., different set-ups for dev vs prod
- Instantiate the DAG

airflow/example_dags/tutorial.py view source

```
# These args will get passed on to each operator
# You can override them on a per-task basis during operator initialization
default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'email': ['airflow@example.com'],
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
    # 'queue': 'bash_queue',
    # 'pool': 'backfill',
    # 'priority_weight': 10,
    # 'end_date': datetime(2016, 1, 1),
    # 'wait_for_downstream': False,
    # 'dag': dag,
    # 'sla': timedelta(hours=2),
    # 'execution_timeout': timedelta(seconds=300),
    # 'on_failure_callback': some_function,
    # 'on_success_callback': some_other_function,
    # 'on_retry_callback': another_function,
    # 'sla_miss_callback': yet_another_function,
    # 'trigger_rule': 'all_success'
}
```

airflow/example_dags/tutorial.py view source

```
with DAG(
    'tutorial',
    default_args=default_args,
    description='A simple tutorial DAG',
    schedule_interval=timedelta(days=1),
    start_date=datetime(2021, 1, 1),
    catchup=False,
    tags=['example'],
) as dag:
```

Airflow: Tutorial

- DAG defines tasks by instantiating `Operator` objects
  - The default params are passed to all the tasks
  - Can be overridden explicitly

airflow/example_dags/tutorial.py view source

```
t1 = BashOperator(
    task_id='print_date',
    bash_command='date',
)

t2 = BashOperator(
    task_id='sleep',
    depends_on_past=False,
    bash_command='sleep 5',
    retries=3,
)
```

- One can use a Jinja template

airflow/example_dags/tutorial.py view source

```
templated_command = dedent(
    """
{% for i in range(5) %}
    echo "{{ ds }}"
    echo "{{ macros.ds_add(ds, 7)}}"
    echo "{{ params.my_param }}"
{% endfor %}
"""
)

t3 = BashOperator(
    task_id='templated',
    depends_on_past=False,
    bash_command=templated_command,
    params={'my_param': 'Parameter I passed in'},
)
```

- Add tasks to the DAG with dependencies

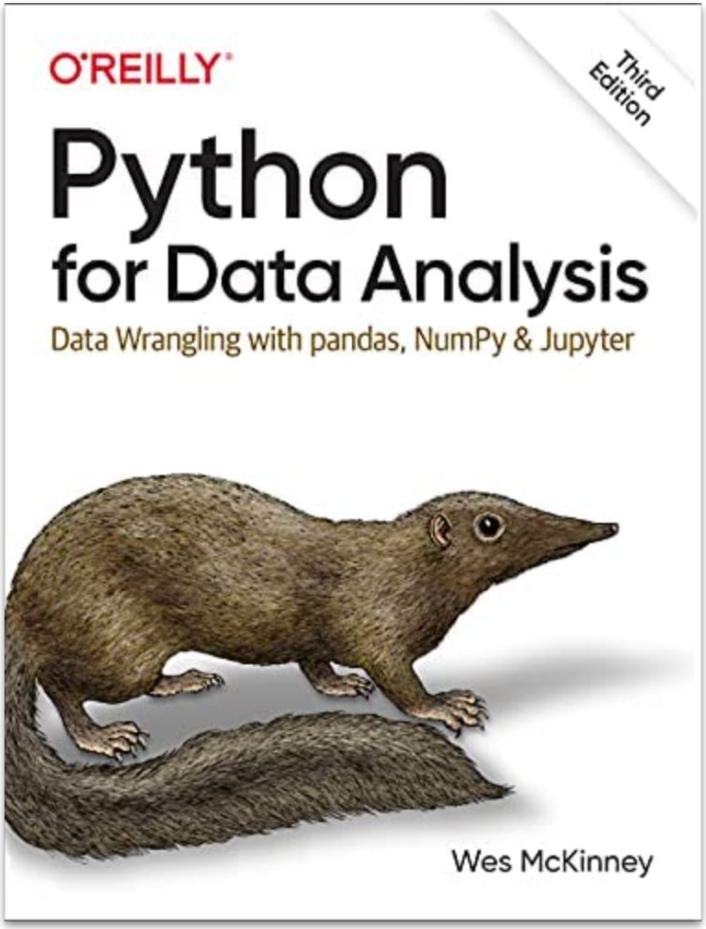
```
t1 >> [t2, t3]
```

UMD DATA605 - Big Data Systems
Orchestration with Airflow
Data wrangling (Pandas)
Deployment

Dr. GP Saggese
gsaggese@umd.edu

Resources

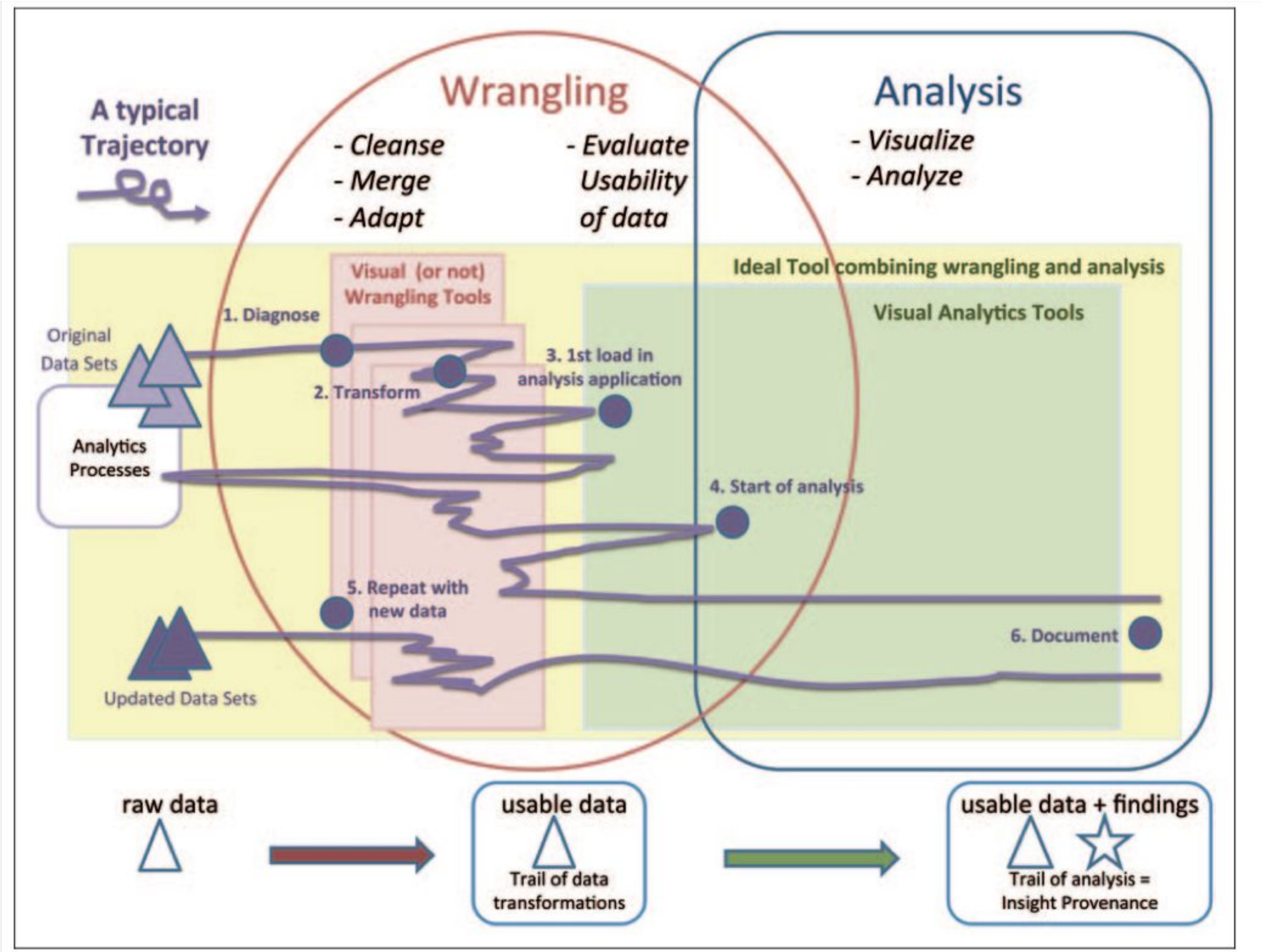
- Pandas tutorial
- Class project
- Web
  - https://pandas.pydata.org
  - Onslaught of free resources
- Mastery
  - https://wesmckinney.com/book
  - Read cover-to-cover and execute all examples 2-3x time to really *master*



Overview

- **Data wrangling**
  - Aka “data preparation”, “data munging”, “data curation”
  - Get data into a structured form suitable for analysis
  - Often it is the step where majority of time (80-90%) is spent
- **Key steps**
  - **Scraping**: extract information from sources (e.g., webpages, spreadsheets)
  - **Data cleaning**: remove inconsistencies / errors
  - **Data transformation**: get data into the right structure
  - **Data integration**: combine data from multiple sources
  - **Information extraction**: extract structured information from unstructured / text sources

Overview



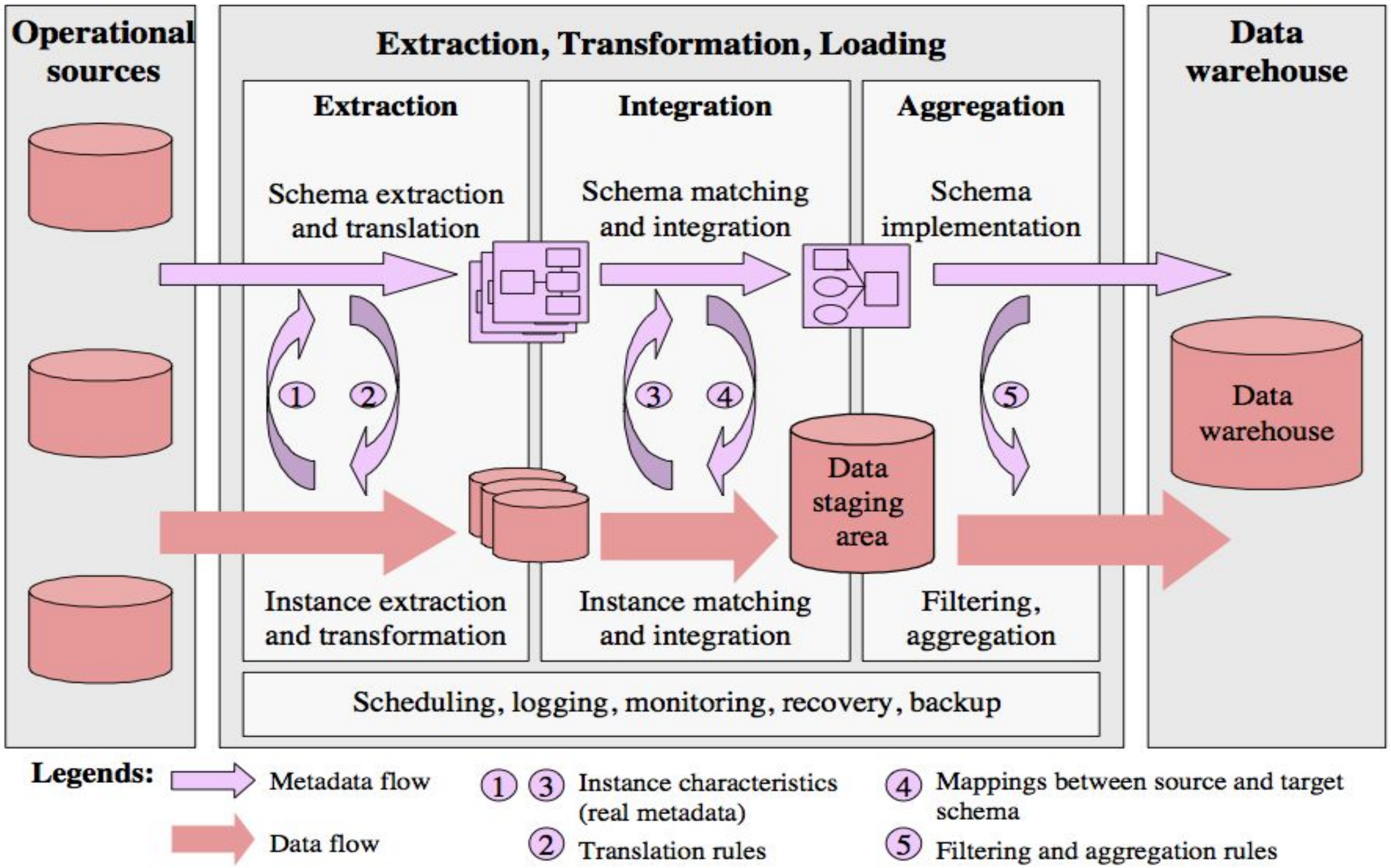
A typical Trajectory
Wrangling
- Cleanse
- Merge
- Adapt
- Evaluate Usability of data
Analysis
- Visualize
- Analyze
Ideal Tool combining wrangling and analysis
Visual (or not) Wrangling Tools
Visual Analytics Tools
Original Data Sets
Analytics Processes
1. Diagnose
2. Transform
3. 1st load in analysis application
4. Start of analysis
5. Repeat with new data
6. Document
Updated Data Sets
raw data
usable data
Trail of data transformations
usable data + findings
Trail of analysis = Insight Provenance

Overview

- Many of the data wrangling problems are not easy to formalize, and have seen little research work, e.g.,
  - Data cleaning: mainly statistics, outlier detection, imputation
  - Data transformation, i.e., put the data in the “right” structure (e.g., tidy data)
  - Information extraction: feature computation, highly domain specific
- Others aspects of have been studied in depth, e.g.,
  - Schema mapping
  - Data integration
- In an ETL process
  - Data extraction is the E step
  - Data wrangling is the T step

Overview

- From Data Cleaning: Problems and Current Approaches
- Paper somewhat old: data is mostly coming from structured sources
- Today unstructured/semi-structured are equally important



Data Scraping

- Data may reside in a wide variety of different sources
  - Files (e.g., CSV, JSON, XML)
  - Many databases
  - Spreadsheets
  - AWS S3 buckets
  - ...
  - Most analytical tools support importing data from such sources through adapters
- Web scraping
  - In some cases there may be APIs, in other cases data may have to be explicitly scraped
  - Scraping data from web sources is tough
    - Can be fragile
    - Throttling
    - It’s cat-and-mouse game between scrapers and website
  - Often pipelines are set up to do this on a periodic basis
  - Several tools out there to do this (somewhat) automatically
    - E.g., import.io, portia, ...

Tidy Data

- Tidy data, Wickham, 2014
  - Each variable forms a column
  - Each observation forms a row
- Wide vs long format

	treatmenta	treatmentb
John Smith	—	2
Jane Doe	16	11
Mary Johnson	3	1

	John Smith	Jane Doe	Mary Johnson
treatmenta	—	16	3
treatmentb	2	11	1

"Messy" data

name	trt	result
John Smith	a	—
Jane Doe	a	16
Mary Johnson	a	3
John Smith	b	2
Jane Doe	b	11
Mary Johnson	b	1

Tidy data

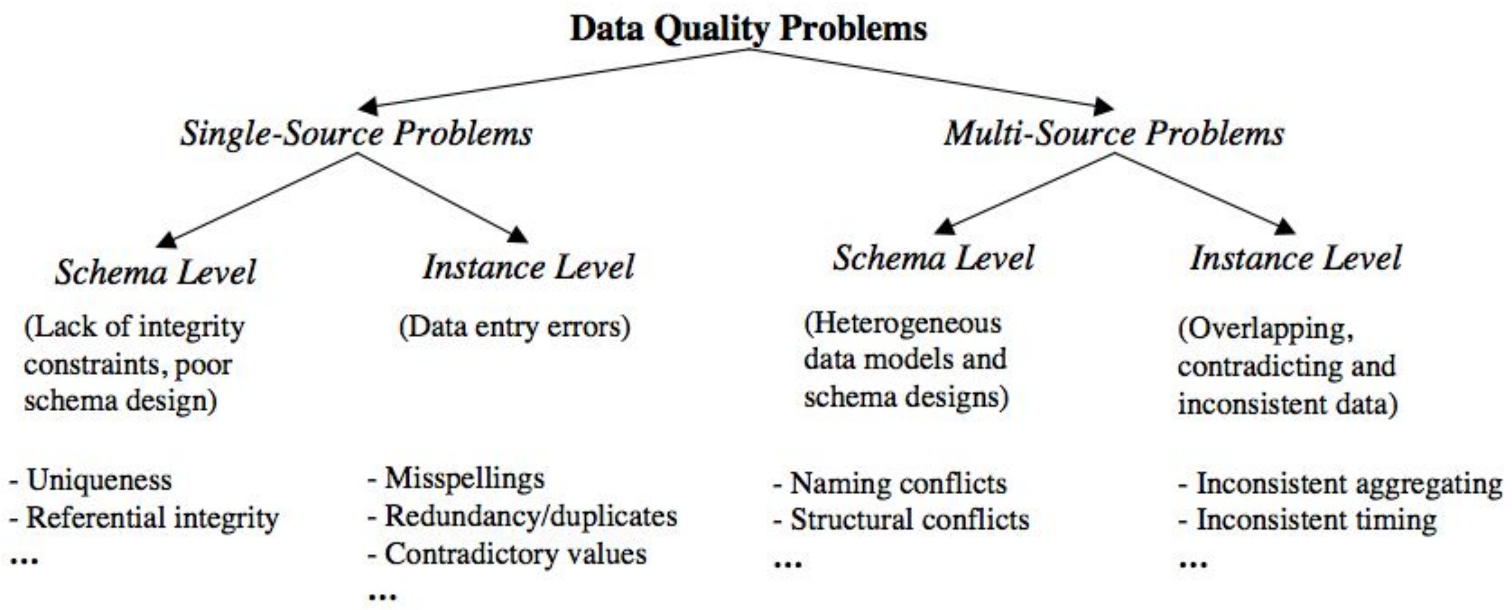
type	date	clicks	conversions	impressions
0	2020-01-01	1.0	NaN	18.0
1	2020-01-02	2.0	NaN	19.0
2	2020-01-03	1.0	1.0	14.0
3	2020-01-04	NaN	NaN	5.0
4	2020-01-05	1.0	NaN	8.0
5	2020-01-06	1.0	1.0	15.0
6	2020-01-07	2.0	NaN	8.0

Wide format

	date	type	count
0	2020-01-01	impressions	18.0
1	2020-01-02	impressions	19.0
2	2020-01-03	impressions	14.0
3	2020-01-04	impressions	5.0
4	2020-01-05	impressions	8.0
...	...	...	...
91	2020-01-28	conversions	NaN
92	2020-01-29	conversions	NaN
93	2020-01-30	conversions	NaN
94	2020-01-31	conversions	NaN
95	2020-02-01	conversions	NaN

Long format

Data Quality Problems



Single-Source Problems

- Depends largely on the source
- Databases can enforce constraints
- Data extracted from spreadsheets is often “clean”
  – At least there is a schema
- Logs are messy
- Data scraped from web-pages is much more messy
- Types of problems:
  – Ill-formatted data
  – Missing or illegal values, misspellings, use of wrong fields, extraction issues (e.g., not easy to separate out different fields)
  – Duplicated records, contradicting information, referential integrity violations
  – Unclear default/missing values
  – Evolving schemas or classification schemes (for categorical attributes)
  – Outliers

Data Quality Problems

Scope/Problem		Dirty Data	Reasons/Remarks
Attribute	Missing values	phone=9999-999999	unavailable values during data entry (dummy values or null)
	Misspellings	city="Liipzig"	usually typos, phonetic errors
	Cryptic values, Abbreviations	experience="B"; occupation="DB Prog."	
	Embedded values	name="J. Smith 12.02.70 New York"	multiple values entered in one attribute (e.g. in a free-form field)
	Misfielded values	city="Germany"	
Record	Violated attribute dependencies	city="Redmond", zip=77777	city and zip code should correspond
Record type	Word transpositions	$name_1$= "J. Smith", $name_2$="Miller P."	usually in a free-form field
	Duplicated records	emp_1=(name="John Smith",...); emp_2=(name="J. Smith",...)	same employee represented twice due to some data entry errors
	Contradicting records	emp_1=(name="John Smith", bdate=12.02.70); emp_2=(name="John Smith", bdate=12.12.70)	the same real world entity is described by different values
Source	Wrong references	emp=(name="John Smith", deptno=17)	referenced department (17) is defined but wrong

Multi-Source Problems

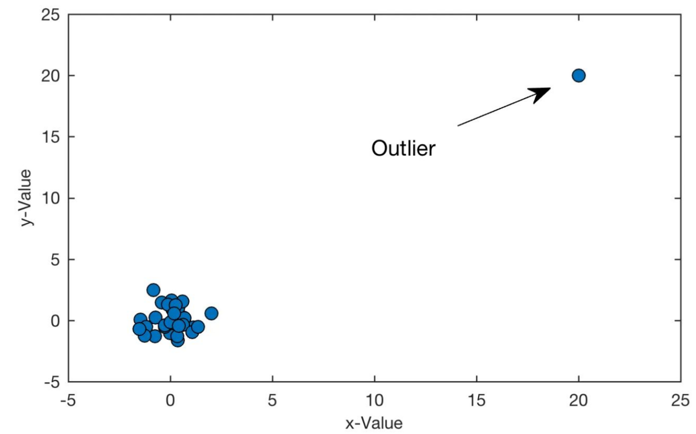
- Different data sources are:
  - Developed separately
  - Maintained by different people
  - Stored in different systems
  - ...
- Schema mapping / transformation
  - Mapping information across sources
  - Naming conflicts: same name used for different objects, different names for same objects
  - Structural conflicts: different representations across sources
- Entity resolution
  - Matching entities across sources
- Data quality issues
  - Contradicting information
  - Mismatched information
  - ...

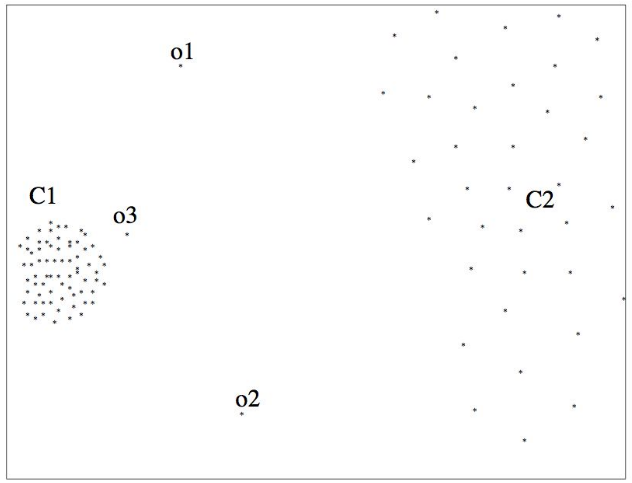
Data Cleaning: Outlier Detection

- Quantitative Data Cleaning for Large Databases, Hellerstein, 2008
  - Focuses on numerical data (i.e., integers/floats that measure some quantities of interest)
- Sources of errors in data
  - Data entry errors: users putting in arbitrary values to satisfy the form
  - Measurement errors: especially sensor data
  - Distillation errors: errors that pop up during processing and summarization
  - Data integration errors: inconsistencies across sources that are combined together

Univariate Outlier Detection

- A set of values can be characterized by metrics such as
  - Center (e.g., mean)
  - Dispersion (e.g., standard deviation)
  - Higher momenta (e.g., skew, kurtosis)
- Use statistics to identify outliers
  - Must watch out for "masking": one extreme outlier may alter the metrics sufficiently to mask other outliers
  - Robust statistics: minimize effect of corrupted data
  - Robust center metrics:
    - Median
    - k%-trimmed mean (i.e., discard lowest and highest k% values)
  - Robust dispersion:
    - Median absolute deviation (MAD)
    - Median distance of values from the median value



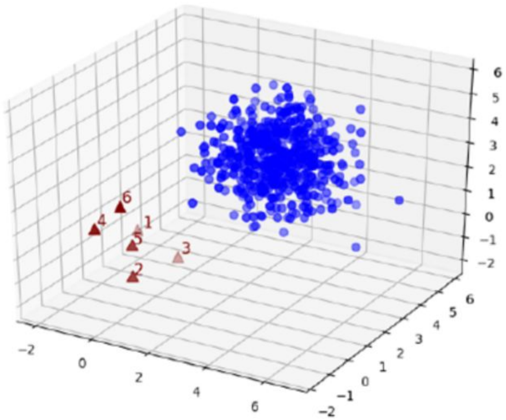


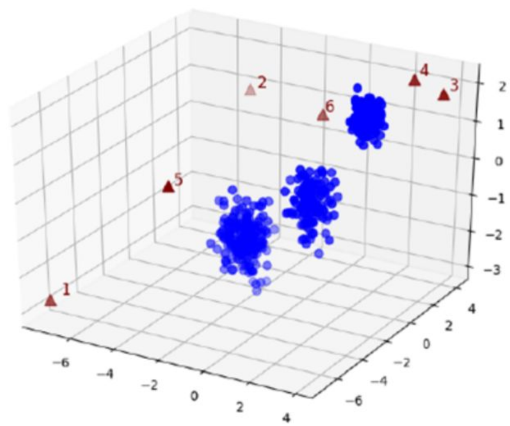
Outlier Detection

- For Gaussian data
  - Any data points 1.4826x MAD away from median
  - May need to eyeball the data (e.g., plot a histogram) to decide if this is true
- For non-Gaussian data
  - Estimate generating distribution (parametric approach)
  - Distance-based methods: look for data points that do not have many neighbors
  - Density-based methods:
    - Define *density* to be average distance to *k* nearest neighbors
    - *Relative density* = density of node/average density of its neighbors
    - Use relative density to decide if a node is an outlier
- Most of these techniques start breaking down as the dimensionality of the data increases
  - *Curse of dimensionality*
    - You need an O(e^n) points with n dimensions to estimate
    - "In high dimensional spaces, data is always is sparse"
  - Can project data into lower-dimensional space and look for outliers there
    - Not as straightforward
- [Wikipedia article on Outliers]

Multivariate Outliers

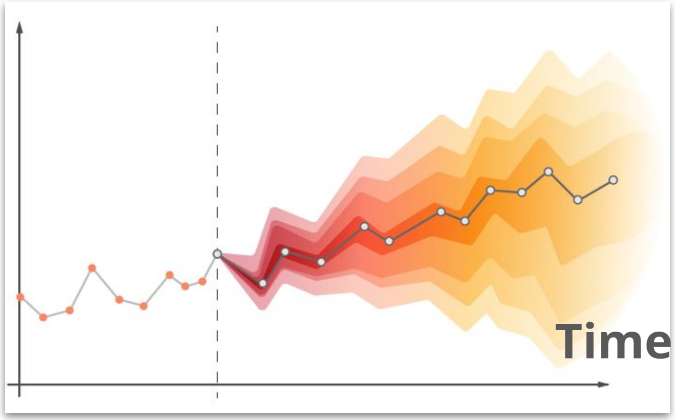
- One set of techniques *multivariate Gaussian distribution* data
  - Defined by a *mean* μ and a *covariance matrix* Σ
- Mean / covariance are not *robust* (sensitive to outliers)
- Robust statistics analogous to univariate case
- Iterative approach
  - Mahalanobis distance of a point is the square root of $(x - \mu)'\Sigma-1(x - \mu)$
  - Measures how far the point x is from a multivariate normal distribution
  - Outliers are points that are too far away according to Mahalanobis distance
  - Remove outlier points
  - Recompute the mean and covariance
- Often volume of data is too much
  - Approximation techniques often used
- Need to try different techniques based on the data





Time Series Outliers

- Often data is in the form of a time series
- A **time series** is a sequence of data points recorded at regular time intervals tracking a variable over time
  - Stock prices
  - Sales revenue
  - Website traffic
  - Inventory levels
  - Energy consumption
  - Market demand
  - Social media engagement
  - Hourly energy usage
  - Customer satisfaction ratings over time
  - Weekly retail foot traffic
  - …
- Rich literature on *forecasting* in time series data
- Can use the historical patterns in the data to flag outliers
  - Rolling MAD (median absolute variation)



Split-Apply-Combine

- The Split-Apply-Combine Strategy for Data Analysis, Wickam, 2011
- Common data analysis pattern
  - Split: break data into smaller pieces
  - Apply: operate on each piece independently
  - Combine: combine the pieces back together
- Pros
  - Code is compact
  - Easy to parallelize
- E.g.,
  - group-wise ranking
  - group vars (e.g., sums, means, counts)
  - create new models per group
- Supported by many languages
  - Pandas
  - SQL GROUP BY operator
  - Map-Reduce

```
In [94]: animals.groupby("kind").height.agg(
   ....:     min_height="min",
   ....:     max_height="max",
   ....: )
   ....:
Out[94]:
      min_height  max_height
kind
cat          9.1         9.5
dog          6.0        34.0
```

UMD DATA605 - Big Data Systems
Orchestration with Airflow
Data wrangling
Deployment

Dr. GP Saggese

gsaggese@umd.edu

Serialization Formats

- Programs need to send data to each other (on the network, on disk)
  - E.g., Remote Procedure Calls (RPCs)
  - Several recent technologies based around schemas
    - JSON, YAML, Protocol Buffer
- Serialization formats are data models

JSON

- JSON = JavaScript Object Notation
- Data is nested dictionaries and arrays
- Very similar to XML
  - More human-readable
  - Less boilerplate
  - Executable in JavaScript (and Python)

```
{
   "firstName": "John",
   "lastName": "Smith",
   "isAlive": true,
   "age": 25,
   "height_cm": 167.6,
   "address": {
      "streetAddress": "21 2nd Street",
      "city": "New York",
      "state": "NY",
      "postalCode": "10021-3100"
   },
   "phoneNumbers": [
      {
      "type": "home",
      "number": "212 555-1234”
      },
      {
      "type": "office",
      "number": "646 555-4567"
      }
   ],
   "children": [],
   "spouse": null
}
```

Protocol Buffers

- Developed by Google
- [Open-source](#)
- Represent data structures in:
  - Language agnostic
  - Platform agnostic
  - Versioning
- Schema is mostly relational
  - Optional fields
  - Types
  - Default values
  - Structures
  - Arrays
- Schema specified using a `.proto` file
- Compiled by `protoc` to produce C++, Java, or Python code to initialize, read, serialize objects

```
message Person {
  optional string name = 1;
  optional int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    optional string number = 1;
    optional PhoneType type = 2;
  }

  repeated PhoneNumber phones = 4;
}
```

```
import addressbook_pb2
person = addressbook_pb2.Person()
person.id = 1234
person.name = "John Doe"
person.email = "jdoe@example.com"
phone = person.phones.add()
phone.number = "555-4321"
  phone.type =
  addressbook_pb2.Person.HOME
```

Serialization Formats

- Avro
  - Richer data structures
  - JSON-specified schema

```
{
      "namespace": "example.avro",
      "type": "record",
      "name": "User",
      "fields": [
                {"name": "name", "type": "string"},
                {"name": "favorite_number", "type": ["int", "null"]},
                {"name": "favorite_color", "type": ["string", "null"]}
      ]
 }
```

- Thrift
  - Developed by Facebook
  - Now Apache project
  - More languages supported
  - Supports exceptions and sets

Comma Separated Values (CSV)

- CSV stores data row-wise as text without schema
  - Each line of the file is a data record
  - Each record consists of one or more fields, separated by commas
- **Pros**
  - Very portable
    - It's text
    - Supported by every tool
  - Human-friendly
- **Cons**
  - Large footprint
    - Compression
  - Parsing is CPU intensive
  - No easy random access
  - No read only a subset of columns
  - No schema / types
    - Annotate CSV files with schema
  - Mainly read-only, difficult to modify

Year	Make	Model	Description	Price
1997	Ford	E350	ac, abs, moon	3000.00
1999	Chevy	Venture "Extended Edition"		4900.00
1999	Chevy	Venture "Extended Edition, Very Large"		5000.00
1996	Jeep	Grand Cherokee	MUST SELL! air, moon roof, loaded	4799.00

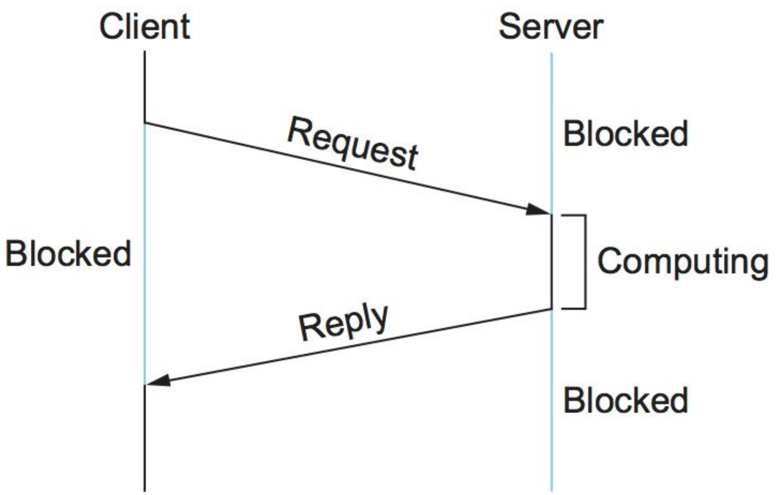
```
Year,Make,Model,Description,Price
1997,Ford,E350,"ac, abs, moon",3000.00
1999,Chevy,"Venture ""Extended Edition""","",4900.00
1999,Chevy,"Venture ""Extended Edition, Very Large""","",5000.00
1996,Jeep,Grand Cherokee,"MUST SELL!
air, moon roof, loaded",4799.00
```

(Apache) Parquet

- Parquet allows to read tiles of data
  - That's what the name comes from
- Supports multi-dimensional and nested data
  - A generalization of dataframes
- Column-storage
  - Each column is stored together, has uniform data type, and compressed (efficiently)
- Queries can be executed by IO layer
  - Only the necessary chunks of data is read from disk
- **Pros**
  - 10x smaller than CSV
  - 10x faster (with multi-threading)
  - You can read only a subset of columns and rows
- **Cons**
  - Binary, non-human friendly
  - Need ingestion step converting the inbound format to Parquet
  - Mainly read-only, difficult to modify

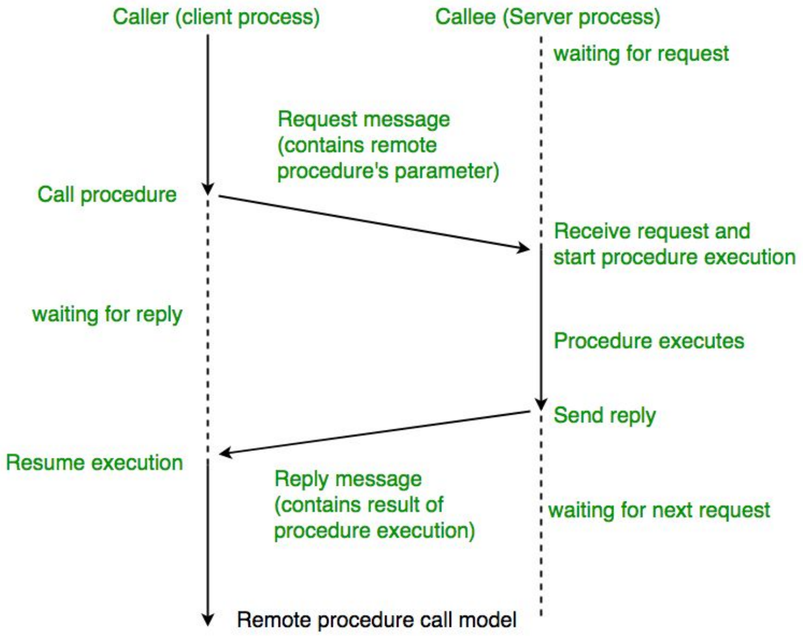
Remote Procedure Call

- **Remote Procedure Call** (RPC) is a protocol to request a service from a program located in another computer abstracting the details of the network communication
- **Goal**: similar to how procedure calls are made within a single process, without having to understand the network's details
- **Problems**
  - Can’t serialize pointers
  - Asynchronous communication
  - Failures and retry
- Used in distributed systems
  - E.g., microservices architectures, cloud services, and client-server applications
- Can be synchronous or asynchronous



RPCs: Internals

- **Client procedure call**: Client calls a stub function, providing the necessary arguments
- **Request marshalling**: Client stub serializes the procedure's arguments into a format suitable for transmission over the network
- **Server communication**: Client's RPC runtime sends the procedure request across the network to the server
- **Server-side unmarshalling**: Server's RPC runtime receives the request and deserializes the arguments
- **Procedure execution**: Server calls the actual procedure on the server-side
- **Response marshalling**: Once the procedure completes, the return values are marshaled into a response message
- **Client communication** / **response unmarshalling** / **return to client**: Return values are passed back to the client's original stub call, and execution continues as if the call were local.



Software testing

- Evaluate the functionality, reliability, performance, and security of a product and ensure it meets specified requirements
  - Software testing is a critical phase in the development process
- Adage:
  - "If it's not tested, it doesn't work"
  - "Debugging is 2x harder than writing code"
    - Corollary: if I'm doing my best to write code, how I can possibly debug it?

- **Different types of testing**
- **Unit testing**: test individual components to ensure that each part functions correctly in isolation
- **Integration testing**: ensure that components work together as expected (e.g., detect interface defects)
- **System testing**: evaluate a fully integrated system's compliance with its specified requirements

Software testing

- **Smoke/sanity testing**: A quick, non-exhaustive run-through of the functionalities to ensure that the main functions work as expected
  - E.g., decide if a new build is stable enough to use
  - E.g., the application doesn't crash upon launching
- **Regression testing**: ensure that the new changes have not adversely affected existing functionality
  - Confusing: regressing in the sense of "getting worse"
- **Acceptance testing**: final phase of testing before the software is released
  - More used in waterfall workflows than Agile development
- **Performance testing**: load testing, stress testing, and spike testing
- **Security testing**: identify vulnerabilities, threats, and risks in the software
- **Usability testing**: assess how easy it is for end-users to use the software application
  - E.g., UI/UX
- **Compatibility testing**: check if the software is compatible with different browsers, database versions, operating systems, mobile devices, ...

Unit-testing

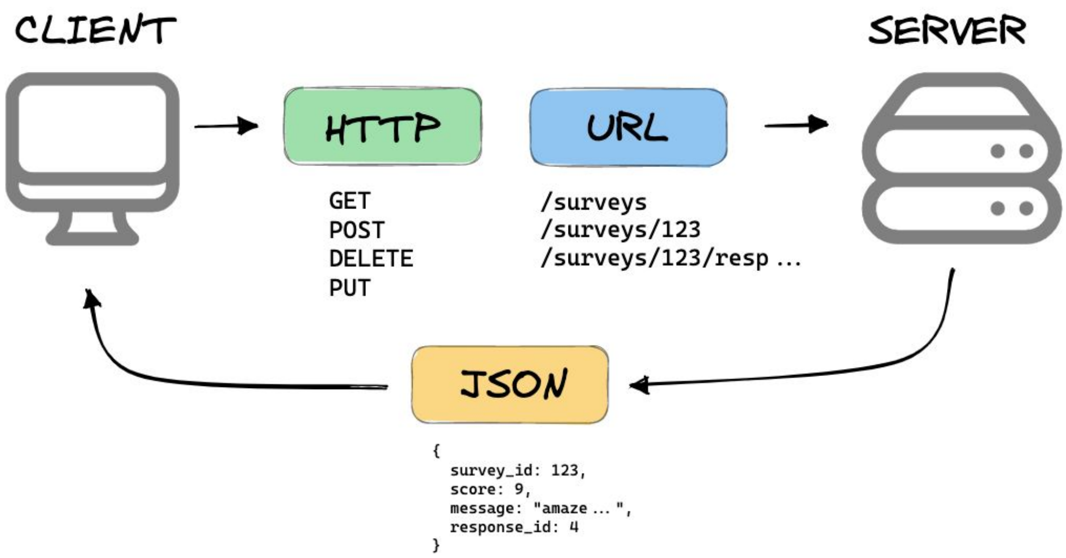
- **Continuous integration (CI)**
  - Devs merge their code changes into a central repository, often multiple times a day
  - Automated build and test code after each change
  - **Goal**: Detect and fix integration errors quickly
  - Need unit tests! Add code together with tests!
- **Continuous deployment (CD)**
  - Automatically deploy all code changes to a production environment
  - without human intervention
  - after the build and test phases pass,
  - **Goal**: new features, bug fixes, and updates are continuously delivered to users in real-time
- E.g., GitHub actions, GitLab workflows, AWS Code, Jenkins

RESTful API

- REST API
  - = Web service API conforming to the REST style
  - REST = REpresentational State Transfer
  - Style to develop distributed systems
  - Many APIs share the same logic to make it easy
- **Uniform interface**
  - Refer to resources (e.g., document, services, URI, persons)
  - Use HTTP methods (GET, POST, PUT, DELETE)
  - Naming convention, link format
  - Response (XML or JSON)
- **Stateless**
  - Each request from client to server must contain all of the information necessary
  - No shared state
  - Inspired by HTTP (modulo cookies)

RESTful API

- **Cacheable**
  - The data in a response should be labeled as cacheable or non-cacheable
  - If cacheable, the client can reuse the response later
  - Increase scalability and performance
- **Layered system**
  - Each layer cannot "see" beyond the immediate layer that they interface
  - E.g., in a tier application



Stages of deployment

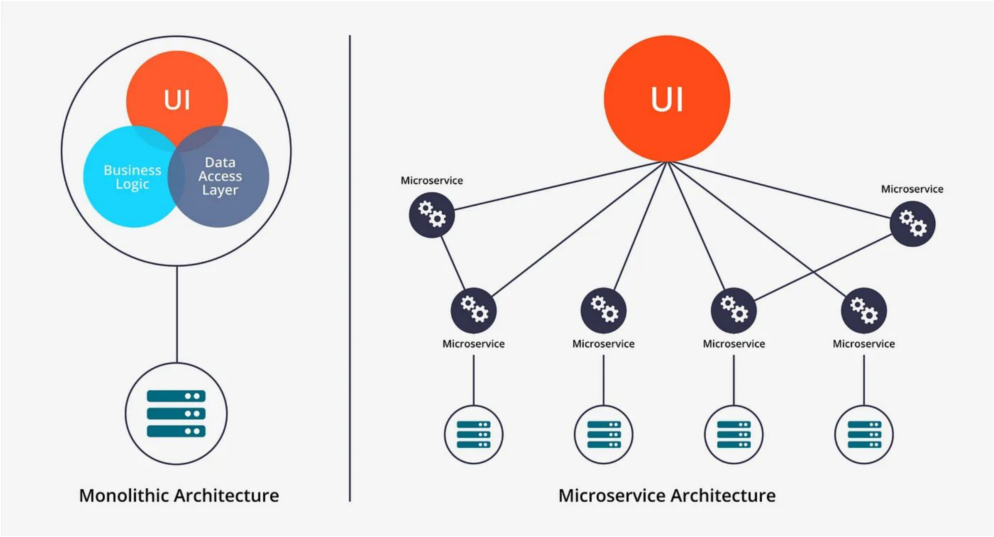
- The deployment of software progresses through several environments
  - Each environment is designed to progressively test, validate, and prepare the software for release to end user
- **Development environment (Dev)**
  - Individual for each developer or feature team
  - Goal: Where developers write and initially test the code
- **Testing or Quality Assurance (QA) environment**
  - Mirrors the production environment as closely as possible to perform under conditions similar to production
  - Goal: Dedicated to systematic testing of the software to uncover defects and ensure quality.
- **Staging/Pre-Prod environment**
  - Serves as the final testing phase before deployment to production
  - It is a replica of the production environment used for final checks and for stakeholders to review the new changes
- **Production Environment (Prod)**
  - It is the live environment where the software is available to its end users
  - Highly optimized for security, performance, and scalability
  - Focus is on uptime, user experience, and data integrity

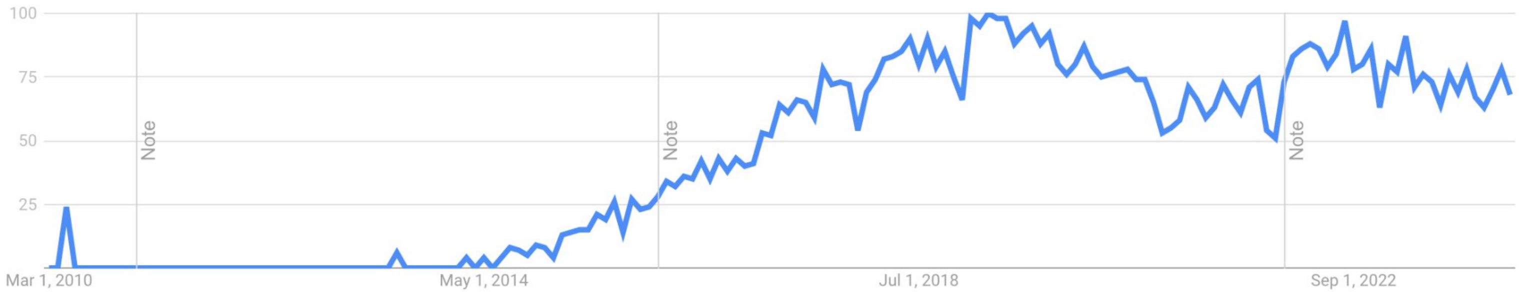
Semantic versioning

- Semantic versioning is a versioning scheme for software that aims to convey meaning about the underlying changes
  - Systematic approach
  - Understand the potential impact of updating to a new version
- Major Version (`**X**.y.z`)
  - Incremented for incompatible API changes or significant updates that may break backward compatibility
- Minor Version (`x.**Y**.z`)
  - Incremented for backward-compatible enhancements and significant new features that don't break existing functionalities
- Patch Version (`x.y.**Z**`)
  - Incremented for backward-compatible bug fixes that address incorrect behavior
- Pre-release Version:
  - Label to denote a pre-release version that might not be stable (e.g., `1.0.0-alpha`, `1.0.0-beta`)
  - These releases are for testing and feedback, not for production use
- Build Metadata
  - Optional metadata to denote build information or environment specifics
  - E.g., 1.0.0+20210313120000 or 1.0.0+f8a34b3228c

Microservices vs Monolithic Architecture

- Different styles of building complex systems
- Find the right granularity





“Microservice” interest over time (from Google Trends)

Microservice Architecture

- **Modularity**: composed of small, independently deployable services, each implementing a specific business functionality
- **Scalability**: services can be scaled independently, allowing for efficient use of resources based on demand for specific features
- **Technology diversity**: each service can be developed using the most appropriate technology stack for its functionality
- **Deployment flexibility**: allows for continuous delivery and deployment practices, enabling faster iterations and updates
- **Resilience**: failure in one service doesn’t necessarily bring down the entire system; easier to isolate and address faults
- **Cons**
  - Complexity to deploy
  - Needs tooling

Monolithic Architecture

- **Simplicity**: (initially) simpler to develop, test, deploy, and scale as a single application unit
- **Tightly coupled components**: all components run within the same process, leading to potential scalability and resilience issues as the application grows
- **Technology stack uniformity**: the entire application is developed with a single technology stack, which can limit flexibility
- **Deployment complexity**: Updates to a small part of the application require redeploying the entire application
- **Single point of failure**: Issues in any module can potentially affect the availability of the entire application