Analyzing Massive Data Sets

Summer Semester 2019

Prof. Dr. Peter Fischer

Institut für Informatik

Lehrstuhl für Datenbanken und Informationssysteme

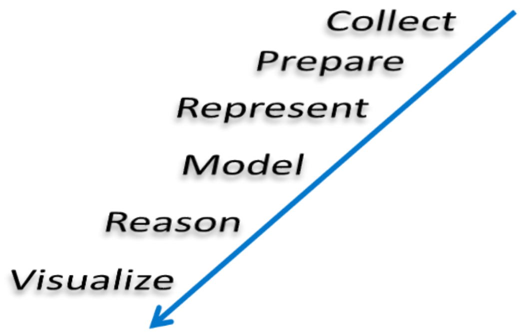
Chapter 1: Basic Tools

Plan for today

- Recap necessary steps
- Mini-Tutorial on Python for data preparation and simple analyses
- Present foundations and fundamental tradeoffs to build tools

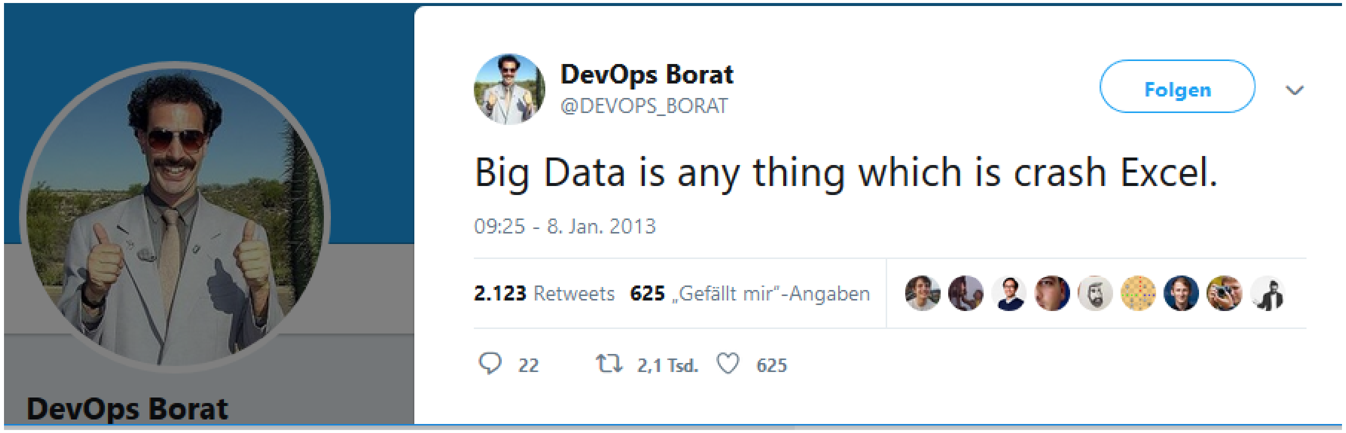
Support for Data Analysis

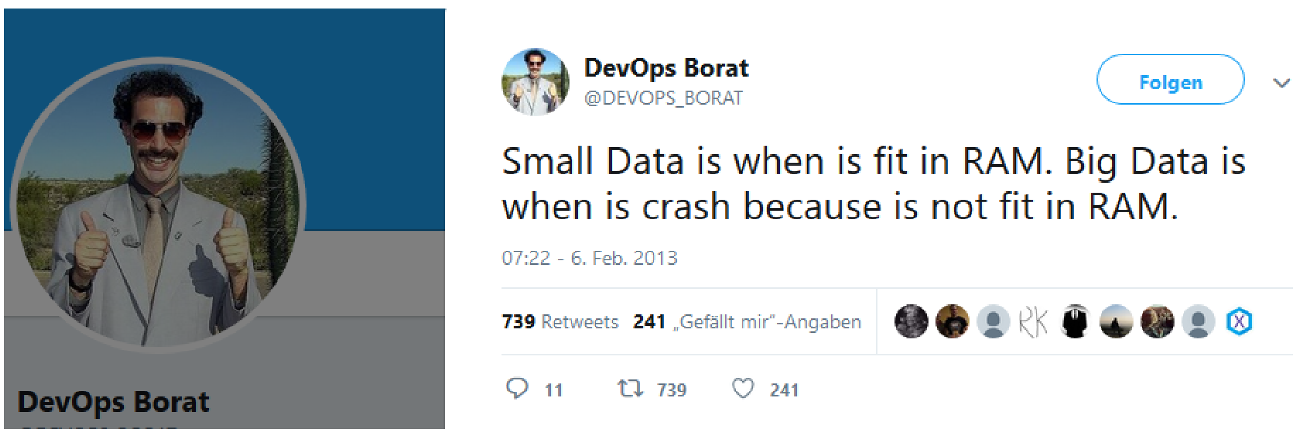
- Remember the steps from last time:



- Each of these has its own challenges and tools to support
- Typically most of the attention goes to "Represent" onwards (research, teaching, …)
- Yet, in practice the first two steps often take a lot more time and effort
- Standalone/basic tools often are most feasible
  - Get quick overview on data: size, structure, quality
  - Decide on the most appropriate models, tools and hardware
  - Quick turnaround needed (Read-Eval-Print-Loop)

Expert opinion on big data and tools





https://gist.github.com/textarcana/676ef78b2912d42dbf355a2f728a0ca1

Python Tutorial for Data Science

Introduction to Python

- High-Level, general purpose programming language
- Interpreted
- Dynamic and strong typing
- Supports multiple programming styles
  - Imperative
  - Object-oriented
  - Functional
- Occupies space between
  - Scripting tools (Shell, Perl)
  - Programming languages (Java, C#, C++)
  - Data Analysis Frameworks (R, Matlab, SPSS)
- Somewhat unusual syntax
  - no {} to delimit blocks (loops, conditions, functions)
  - Instead use
    indentation
  - Consistent formatting, but sometimes irritating errors

Python Ecosystem

- Available on all major operating systems
- Implementations in C, Java, …
- Raw performance not central focus
  - No general purpose JIT, slower that e.g., Lua or JS
  - Inherently single-threaded
- Interactive mode, no compilation needed
- Two slightly incompatible major versions:
  - Python 2: no longer actively developed, many packages
  - Python 3: current development, still acceptance issues

- Broad Level of packages both in standard distribution ("batteries included") and the overall ecosystem (PyPI)

Short example: Quicksort

Key Idea

- Get a sequence of sortable data
- Pick a splitter element: "pivot"
- Split into sequences for smaller, equal, larger
- Recursively sort smaller, larger
- Stop recursion when length of sequence <= 1
- Concatenate

- Typical Divide-and-conquer approach
- Average Case optimal (O(n log n)), wrong pivot could lead to quadratic complexity

```python
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quicksort(left) + middle + quicksort(right)

print(quicksort([3,6,8,10,1,2,1]))
```

Collection/Container Types

- Python provides standard types
- List ("vector", "array"): ordered, resizable, different types
  - Can be used as stacks, queues, …
  - Can be nested
  - xs = [3, 1, [2,4], "Hello"]
  - **print**(xs[-1], xs[1:3])
- Tuple: like List, but fixed size (3, 1, 2)

Set: unordered, no duplicates 'fish' **in** {'cat', 'dog'}

Dictionary: ("Map", "Hashtable"): from key to value

d = {'cat': 'cute', 'dog': 'furry'}

print(d['cat'])

Slicing

```
nums = list(range(5))
print(nums)
print(nums[2:4])        # Get a slice from index 2 to 4 (exclusive);
print(nums[2:])         # Get a slice from index 2 to the end; prints "[2, 3, 4]"
print(nums[:2])         # Get a slice from the start to index 2 (exclusive);
print(nums[:])          # Get a slice of the whole list; prints "[0, 1, 2, 3, 4]"
print(nums[:-1])        # Slice indices can be negative; prints "[0, 1, 2, 3]"
nums[2:4] = [8, 9]      # Assign a new sublist to a slice
print(nums)             # Prints "[0, 1, 8, 9, 4]"

A = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print('Point', A[1][1])
print('Range', A[1:3][1:3]) # Does not work, stay tuned!
```

Container Iteration

- For loops possible over all containers.
- Sets and dictionaries do not have a defined order, results may vary

```
animals = ['cat', 'dog', 'monkey'] # works also for set
for animal in reversed(sorted(animals)):
    print(animal)

d = {'person': 2, 'cat': 4, 'spider': 8}
for animal, legs in d.items():
    print('A %s has %d legs' % (animal, legs))
```

Container Comprehensions

New containers can easily by constructed using comprehensions

```
nums = [0, 1, 2, 3, 4]
even_squares = [x ** 2 for x in nums if x % 2 == 0]
print(even_squares)

d = {'person': 2, 'cat': 4, 'spider': 8}
feet_pairs = {a:b//2 for a,b in d.items()}

print(feet_pairs)
```

Python for Data Science

- Numpy: native and highly optimized vector/matrix/array operations
- Pandas: Data ingestion, filtering, …
- Scipy: broad set of general-purpose operations
- Scikit-learn: Machine Learning
- Matplotlib, Seaborn: Visualization
- networkX: Graphs and Social Networks
- NLTK: Natural Language Toolkit
- Deep Learning/GPUs
- Scikit-image
- …

Numpy – Efficient Arrays

- "Standard" Python is flexible and readable, but
  - Slow
  - Memory-inefficient
  - No dedicated operations for analytics
- Numpy introduces foundation for analytical Python: efficient arrays
- Uniform type (not only numbers), dense
- Fixed dimensions: 1 (vector), 2 (matrix), 3 (tensor)
- Mapped to highly efficient native code (BLAS, MKL)
- Typical computations
- Supporting operations:
  - Generating Vector/Matrices/Tensors of specific shape
  - Array/Matrix Transformations
  - Broadcasting: Rank matching
  - Linear equation solver

Numpy – Introduction

```
import numpy as np
a = np.array([1, 2, 3])   # Create a rank 1 array
#a = np.array(['1', '2', '3'])   # Strings work as well
print(type(a))           # Prints "<class 'numpy.ndarray'>"
print(a.shape)            # Prints "(3,)"
print(a[0], a[1], a[2], type(a[1]))
a[0] = 5                 # Change an element of the array
print(a)                  # Prints "[5, 2, 3]"

b = np.array([[1,2,3],[4,5,6]])    # Create a rank 2 array
print(b.shape)                     # Prints "(2, 3)"
print(b[0, 0], b[0, 1], b[1, 0])   # Prints "1 2 4"
```

Array Slicing

Remember how slicing of nested lists failed?

```
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
# first 2 rows and columns 1 and 2; b
# is the following array of shape (2, 2):
b = a[:2, 1:3] # [[2 3]
print (b)      #  [6 7]]
```

Slices are referenced – changes in a are reflected in b

```
a[0][1] = 13
print (b[0][0])
```

More addressing possible, e.g. compose by picking individual rows/columns

Mathematical Operations - Elementwise

```
x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)
 # Elementwise sum; both produce the array – same with - subtract
print(x + y)          # [[ 6.0  8.0]
print(np.add(x, y)) #  [10.0 12.0]]
# Elementwise product; both produce the array – same with / divide
print(x * y)                  # [[ 5.0 12.0]
print(np.multiply(x, y)) #  [21.0 32.0]]
# Elementwise square root; produces the array
# [[ 1.          1.41421356]
#  [ 1.73205081  2.        ]]
print(np.sqrt(x))
```

Mathematical Operations – "Dot"

```
x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])
v = np.array([9,10])
w = np.array([11, 12])

# Inner product of vectors; both produce 219
print(v.dot(w))
print(np.dot(v, w))
# Matrix / vector product; both produce the rank 1 array [29 67]
print(x.dot(v))
print(np.dot(x, v))
# Matrix / matrix product; both produce the rank 2 array
# [[19 22]
#  [43 50]]
print(x.dot(y))
print(np.dot(x, y))
```

Filters and Masks

- Filter operations on arrays can be turned boolean arrays:
- x = np.array([[5, 0, 3, 3], [7, 9, 3, 5], [2, 4, 7, 6]])
- x < 6
- Operations on Boolean arrays
- Masks as selections:
- x[x<6]
- x[y]

Pandas

- Data Management and Analysis
- Extension of NumPy
  - Flexible labels instead of numbers: "index"
- Series: 1D array
  - Represent a column in a spreadsheet/database
  - Labels for data points (e.g., dates)
- Dataframe: 2D array
  - Different data types in same array
  - Maps well to spreadsheets, CSV data and relations.
  - Labels for rows/columns
  - Modeling of missing data (~ NULL in SQL)
- Functions to load/store data from/to common formats

Pandas: Reading/Writing Data

- Serialized Python objects: "pickle"
- Flat Files: delimited tables/CSV, fixed-width
- Excel
- JSON
- HTML Tables
- Parquet (-> Data Lakes, next chapter)
- SQL, Google BigQuery (-> Databases)
- Feather (R<->Python), SAS, Stata, HDF
- Common aspects:
  - Read also from HTTP in newer versions of Pandas
  - Select and name columns/add header
  - Convert/Parse data types
  - Identify/treat unparseable values

Analyzing Data in Pandas

- Statistics/Aggregates on series or axis
- Applying functions on rows, columns, …
- (Vectorized) string operations
- Filtering
- Sorting
- Hierarchical Indexing (e.g. location-year)
- Grouping

Basic Access, Filtering, Sorting

```
import seaborn as sns
titanic = sns.load_dataset('titanic') # sample dataset

print (titanic.shape) #(891, 15)
print (titanic.columns) # ['survived','pclass','sex','age',...,'class',...]
print (titanic.dtypes)

print(titanic.head(3))
print (titanic[2:6])

print (titanic[['sex', 'age', 'class']])

print (titanic['embark_town'].unique())

print (titanic[titanic['age']>70].sort_values(by=['class','fare']))

print(titanic.loc[titanic['embark_town'].isnull()])
```

Hierarchical Indexes

Naive solution if multiple, orthogonal data descriptions exist

```
index = [('California', 2000), ('California', 2010),
('New York', 2000), ('New York', 2010),
('Texas', 2000), ('Texas', 2010)]
populations = [33871648, 37253956,
18976457, 19378102,
20851820, 25145561]

pop = pd.Series(populations, index=index)

print (pop[('California', 2010):('Texas', 2000)]) # point addressing

index = pd.MultiIndex.from_tuples(index) # proper multiple dimensions
pop = pop.reindex(index)

print(pop[:, 2010]) # filter, slice on each dimension
print (pop.unstack()) # ma
```

Grouping and Pivots

```
titanic = sns.load_dataset('titanic')

print (titanic['age'].min(),titanic['age'].mean(),
titanic['age'].median(),titanic['age'].max())

print(titanic.groupby('sex')[['survived']].mean())

print(titanic.groupby('sex').agg({'fare': np.mean, 'pclass': np.mean}))

print(titanic.groupby(['sex', 'class'])['survived'].aggregate('mean').unstack())
print(titanic.pivot_table('survived', index='sex', columns='class'))

age = pd.cut(titanic['age'], [0, 18, 80]) # create bins for histogram
fare = pd.qcut(titanic['fare'], 2) # bins on quantiles
print (titanic.pivot_table('survived', ['sex', age], [fare, 'class']))
```

Joins: Merge (1)

```
df1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
'group': ['Accounting', 'Engineering', 'Engineering', 'HR']})
df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],
'hire_date': [2004, 2008, 2012, 2014]})
df3 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
'salary': [70000, 80000, 120000, 90000]})

print(pd.merge(df1, df2)) # same key attribute, equality, inner join
print(pd.merge(df1, df3, left_on="employee", right_on="name"))

df4 = pd.DataFrame({'group': ['Accounting', 'Engineering', 'HR'],
'supervisor': ['Carly', 'Guido', 'Steve']})

print(pd.merge(df1, df4))
```

Joins: Merge (2)

```
df6 = pd.DataFrame({'name': ['Peter', 'Paul', 'Mary'],
'food': ['fish', 'beans', 'bread']},
columns=['name', 'food'])
df7 = pd.DataFrame({'name': ['Mary', 'Joseph'],
'drink': ['wine', 'beer']},
columns=['name', 'drink'])

print (pd.merge(df6, df7, how='inner'))

print(pd.merge(df6, df7, how='left')) # also right

print(pd.merge(df6, df7, how='outer'))
```

SciPy

- Clustering
- FFT
- Linear Algebra
- Sparse Matrices
  - Sparse Linear Algebra
  - Sparse Graphs+Graph Algorithms
- Integration
- Interpolation
- Image Processing
- Statistics
- Spatial Algorithms (e.g., distance functions)
- …

Scikit-Learn

Machine Learning

- Classification: training -> identifying
- Regression: predict variable values
- Clustering: group similar items
- Dimensionality Reduction: reduce variable dimensions
- Model selection and evaluation
- Preprocessing: extracting features

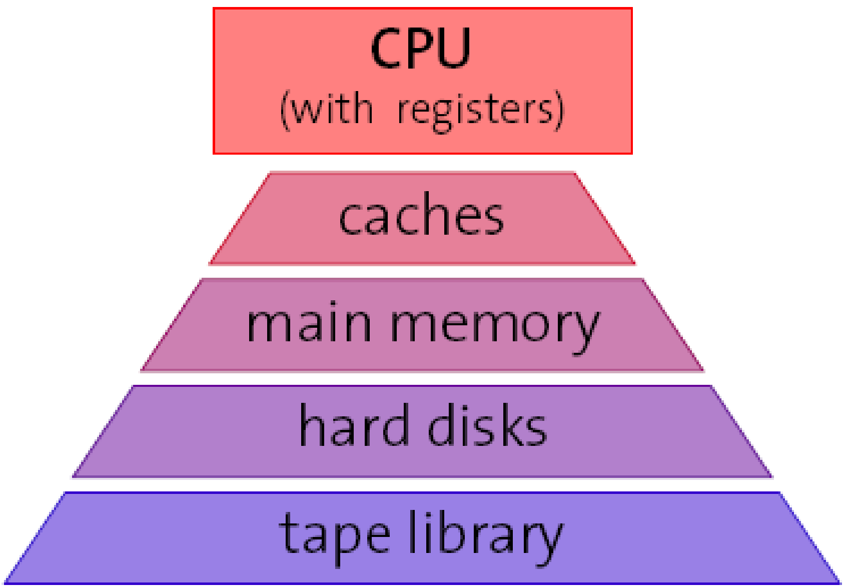
Considerations on (hardware) platforms

Understanding Performance

- In order to plan for the right infrastructure, we need to understand the **constraints** and **capabilities**
- What kind of **performance** can we expect from the a **single system** and its components?
- What are the **theoretical limits**?
- Which benefits come from **tool choice** and modeling?
- When will a **cluster** help with performance?

=> Short recap of Systemnahe Informatik and DBS1, applied to our problem settings

Recap: Memory Hierarchy

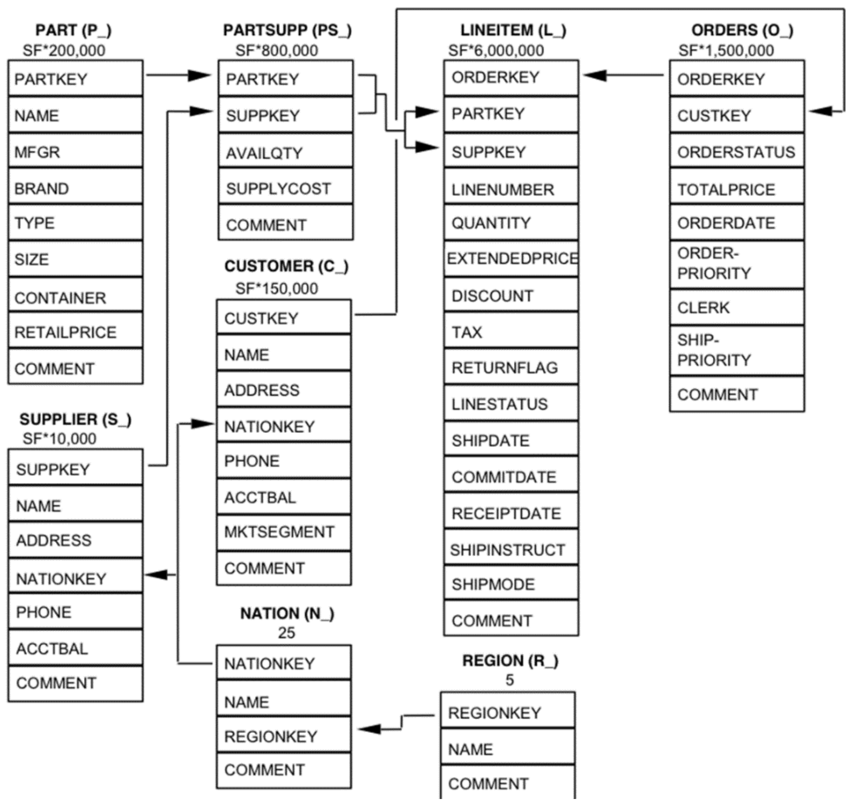


capacity	latency
bytes	< 1 ns
kilo-/megabytes	< 10 ns
gigabytes	70–100 ns
terabytes	3–10 ms
petabytes	varies

- Fast, but expensive and small memory close to CPU
- Larger, slower memory at the periphery
- For analytical workloads, throughput and latency matter

A "toy problem" (by Thomas Neumann)

- TPC-H
  - Part of TPC family of relational benchmarks
  - Analytical queries on sales data



A "toy problem" (by Thomas Neumann)

Query: Sum up quantity (5th column) in relation lineitem

1|155190|7706|1|17|21168.23|0.04|0.02|N|O|1996-03-13|...
1|67310|7311|2|36|45983.16|0.09|0.06|N|O|1996-04-12|...
1|63700|3701|3|8|13309.60|0.10|0.02|N|O|1996-01-29|...
1|2132|4633|4|28|28955.64|0.09|0.06|N|O|1996-04-21|...
...

- 725MB * ScaleFactor (total data set roughly 1GB * SF)
- 6 million * SF lines
- Text input for now (typical for first stage in many scenarios)
- Full benchmark consists of many, more complex queries
- Simple query
  - Can be analyzed easily
  - Surprisingly expensive

Performance limits

- What are the limits for that query (SF1, 725MB)?
- First aspect: **cost of moving data**
  - in practice often most important bottleneck

	Bandwidth	Execution time
1GB ethernet	100MB/s	7.3s
rotating disk	200MB/s	3.6s
SATA SSD	500MB/s	1.6s
10GB ethernet	1GB/s	0.73s
PCIe SSD	2GB/s	0.36s
DRAM	20GB/s	0.04s

- Second aspect: **CPU cost** (parsing, evaluating, …)
- Practical upper limit - how close can we get?

AWK

Unix command line utility

```
awk -F ’|’ ’BEGIN { x=0 } { x=x+ $ 5 } END { print x } ’ lineitem
```

- first execution: 4.5s
- second execution: 3.6s
- first execution was waiting for disk
- second execution was CPU bound
  (much slower than DRAM speed)
- main memory/caching has a huge effect

Python

```
sum=0
with open(sys.argv[1]) as f:
        for line in f:
                sum=sum+float(line.split('|')[4])
print (sum)
```

- first execution: 6.2s
- second execution: 6.2s
- Python is always CPU bound!
- Cannot keep up even with a rotating disk

C++

```
unsigned sum=0;
while (getline(in,s)) {
        unsigned v=0;
        for (auto iter=s.begin(),limit=s.end();iter!=limit;++iter)
        { ... }
        /* extract 5th column into v */
        sum+=v;
} cerr << sum << endl;
```

- first execution: 3.5s, second execution: 0.8s
- first execution is I/O bound, second is CPU bound
- much faster than the others, but still far from DRAM speed
- code is more complex but also more efficient
- With parallelism: 4.3s / 0.13 s
- warm cache case parallelizes well, but cold cache slower (random disk access)

Further improvements

- What else could we do?
  1. Change **data representation**
  2. **Invest** some **computation ahead of time**
- **Avoid parsing** and store data in **fitting data type**
- Switch to **column-oriented storage** –
  load just relevant column
- Utilize **vectorized execution**
  (modern compilers perform this reasonably well)
- Rough estimate (on my old desktop system):
  - 24 MB (4 byte integer *6m) -> 0.0012s transfer (RAM)
  - 6m / (3.6 Ghz * 4-way vector (AVX))
    -> 0.000412s computation
- Anything else?

Assessing the performance

- Cold cache: sequential disk speed+seek times
- Warm cache:

	Time	Throughput
Awk	3.6s	201 MB/s
Python	6.2s	116 MB/s
C++	0.8s	906 MB/s
+parallel	0.13s	5576 MB/s
Different Representation	0.0012s	N/A

- Implementation and representation matter
- Even a single desktop-class machine with standard hardware can evaluate gigabytes per second
- Preprocessing helps, but it not always feasible
- Bottleneck depends on problem setting

Outlook: Scale-Out vs Scale-Up

- We need to consider an important trade off:
  - Use a single machine with many resources (scale-up)
  - Distribute over many, less powerful computers (scale-out)
- For many workloads, utilizing a large set of (distributed) machines has become very popular
  - (Almost) unlimited, yet flexible/elastic scaling
  - Redundancy (design software for resilience)
  - Low response times possible (data locality to user)
- For Google, Amazon, Facebook – scale-out is the right option
- But you are not Google!

Why scale-up may make sense

- Single machines can provide a significant amount of resources (Scale-up):
  - Hundreds of CPU cores
  - Multiple TBs of RAM (across many memory controllers)
  - Dozens of GPUs/FPGAs
  - Dozens to hundreds of disks/SSDs
- Important workload consideration:
  - data shipping: move data to query, compute in one location
  - query shipping: move query to data, return results
- Guidelines
  - Network bandwidth << memory bandwidth
  - Network bandwidth <~ disk bandwith
  - If results size < data size, distribute – but latency matters
- In either case: on demand provisioning in cloud services (Amazon AWS, Microsoft Azure, Google, …)

Wrap-Up

- Significant part of data analysis goes into
  - Collecting
  - Preparing
  - Initial probing
- Python provides rich ecosystem for fast exploration
  - High-level language for "gluing" parts
  - Broad set of highly optimized libraries for specific tasks
  - Numpy: efficient vector/matrix/tensor operations
  - Pandas: "relational" operations on series and data frames
- Standalone, single-machine tools often a good fit
  - Quick turnaround, often interactive
  - Significant means for data "massaging"
  - Fast enough

Additional Python Examples

Control flow: Loops

```
a, b = 0, 1
while b < 10:
    print(b)
    a, b = b, a+b

for i in range(5):
    print (i)
words = ['cat', 'window', 'defenestrate']
for w in words:
     print(w, len(w))
```

Control Flow: If/Elif/Else

```python
x= 4711
if x < 0:
    x = 0
    print('Negative changed to zero')
elif x == 0:
    print('Zero')
elif x == 1:
    print('One')
else:
    print('More')
```

- no switch/case!

Data Types and Operations: Numbers

```
x = 3
print(type(x)) # Prints "<class 'int'>"
print(x + 1)
print(x * 2)   # Multiplication; prints "6"
print(x ** 2)  # Exponentiation; prints "9"
x += 1 # no x++ or x--
print(x)
x *= 2
print(x)
y = 2.5
print(type(y))
print(x / 3) # division with fraction
print(x // 2) # truncated division
```

Also support for complex numbers and arbitrary precision

Booleans and Logic

```
t = True
f = False
print(type(t)) # Prints "<class 'bool'>"
print(t and f) # Logical AND; prints "False"
print(t or f)  # Logical OR; prints "True"
print(not t)   # Logical NOT; prints "False"
print(t != f)  # Logical XOR; prints "True"
```

The following values are considered false in tests:

- False
- None
- Zero of any number
- Empty sequences
- Empty Maps

Strings (1)

Immutable sequences of Unicode "code points" (Python 3)

Ascii strings are byte arrays (mutable)! (was string in Python 2)

```
hello = 'hello'
world = "world"
print(hello)
print(len(hello))
hw = hello + ' ' + world
print(hw)
numberval = 42
print ("The answer is "+ str(numberval)) #
hw12 = '%s %s %d' % (hello, world, 12)
print(hw12)
print (hw[0:4])
```

Strings (2)

```
s = "hello"
print(s.capitalize())
print(s.upper())
print(s.rjust(7))
print(s.center(7))
print (s.find("ll"))
print(s.replace('l', '(ell)'))
print('  world '.strip())
```

Lists

```
xs = [3, 1, 2]    # Create a list
print(xs, xs[2])  # Prints "[3, 1, 2] 2"
print(xs[-1])     # Negative indices count from the end of the list;
xs[2] = 'foo'     # Lists can contain elements of different types
print(xs)         # Prints "[3, 1, 'foo']"
xs.append('bar')  # Add a new element to the end of the list
print(xs)         # Prints "[3, 1, 'foo', 'bar']"
x = xs.pop()      # Remove and return the last element of the list
print(x, xs)      # Prints "bar [3, 1, 'foo']"
del xs[1:2] # Remove index range
print(xs)
xs.append(2)
xs.remove('foo') # Remove first occurrence
```

Sets

```
animals = {'cat', 'dog'}
print('cat' in animals)   # Check if an element is in a set;"True"
print('fish' in animals)  # prints "False"
animals.add('fish')       # Add an element to a set
print('fish' in animals)  # Prints "True"
print(len(animals))       # Number of elements in a set; "3"
animals.add('cat')        # Adding an element that is already in
                          #        the set does nothing
print(len(animals))       # Prints "3"
animals.remove('cat')     # Remove an element from a set
print(len(animals))       # Prints "2"
```

Dictionaries

```
d = {'cat': 'cute', 'dog': 'furry'}  # New dictionary with data
print(d['cat'])       # Get an entry; prints "cute"
print('cat' in d)     # Check if a dictionary has a key;
d['fish'] = 'wet'     # Set an entry in a dictionary
print(d['fish'])      # Prints "wet"
print(d['monkey'])  # KeyError: 'monkey' not a key of d
print(d.get('monkey', 'N/A'))#Get an element  w/ default;
prints "N/A"
print(d.get('fish', 'N/A'))  # Get an element with a default;
prints "wet"
```