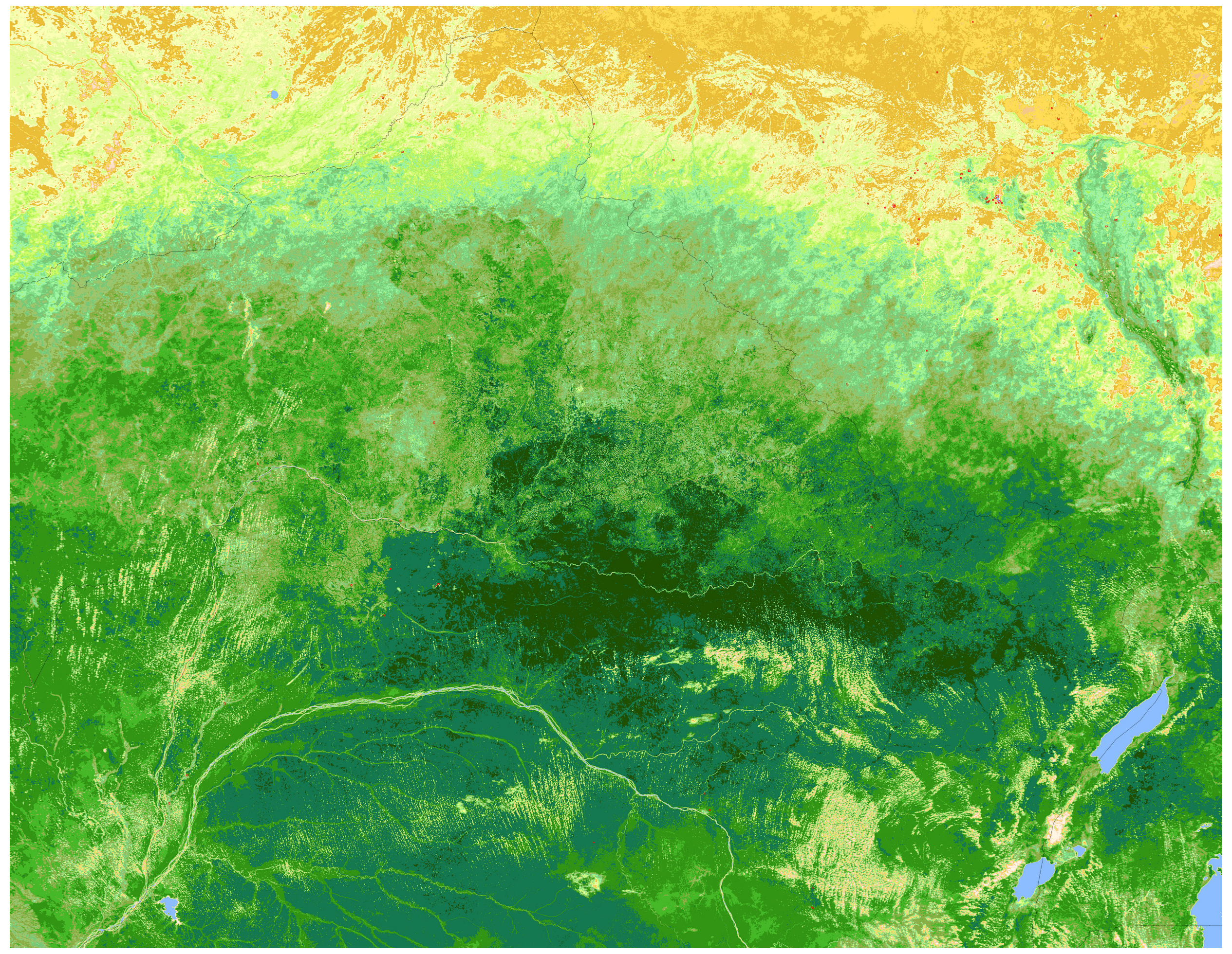
Advanced Data Management (CSCI 680/490)

Time Series Data

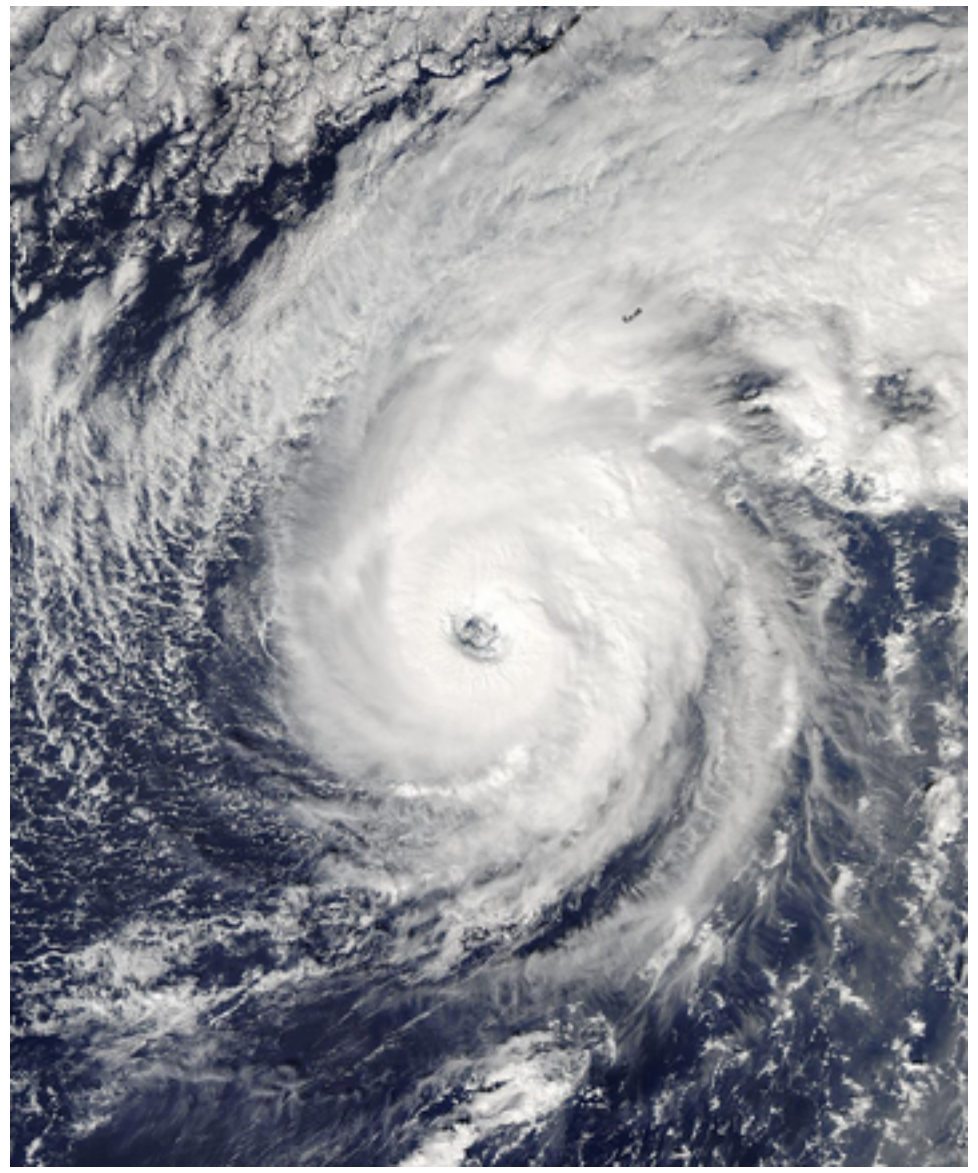
Dr. David Koop

Spatial Data

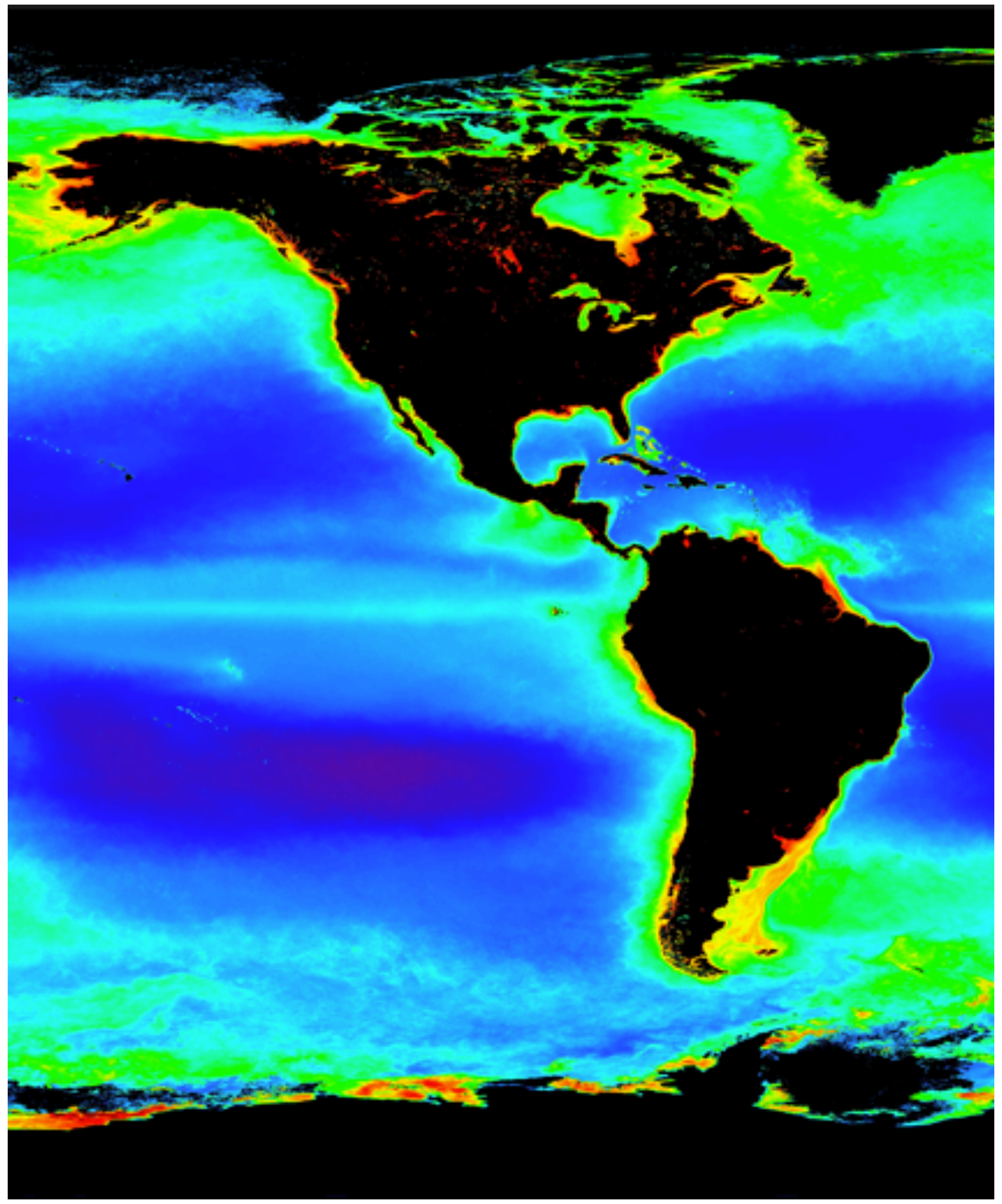
Measure vegetation density



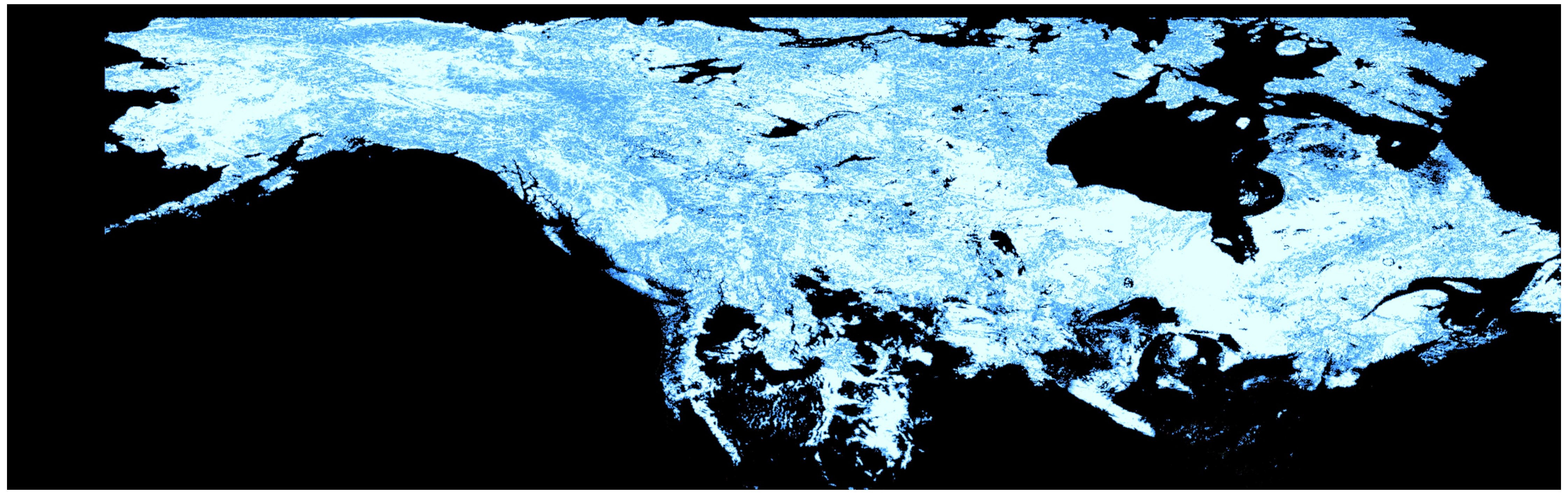
Track hurricanes



Track phytoplankton populations

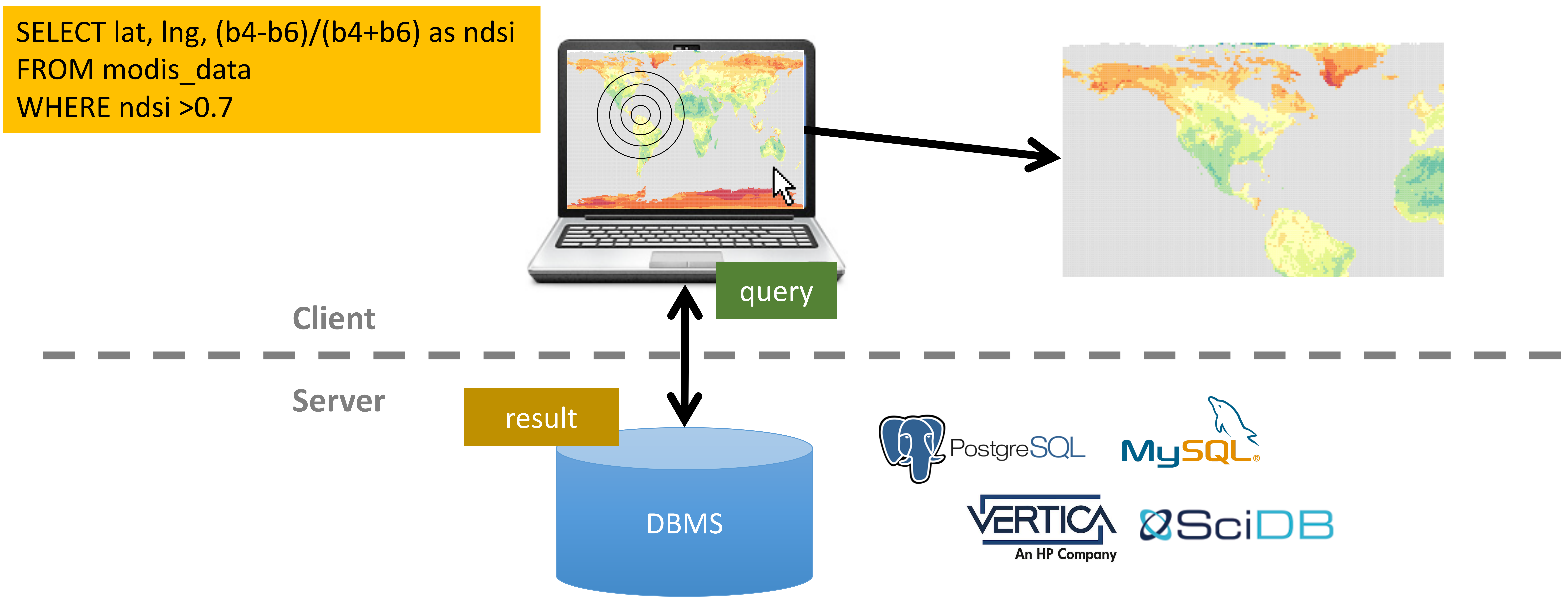


Measure snow melt



[L. Battle, 2017]

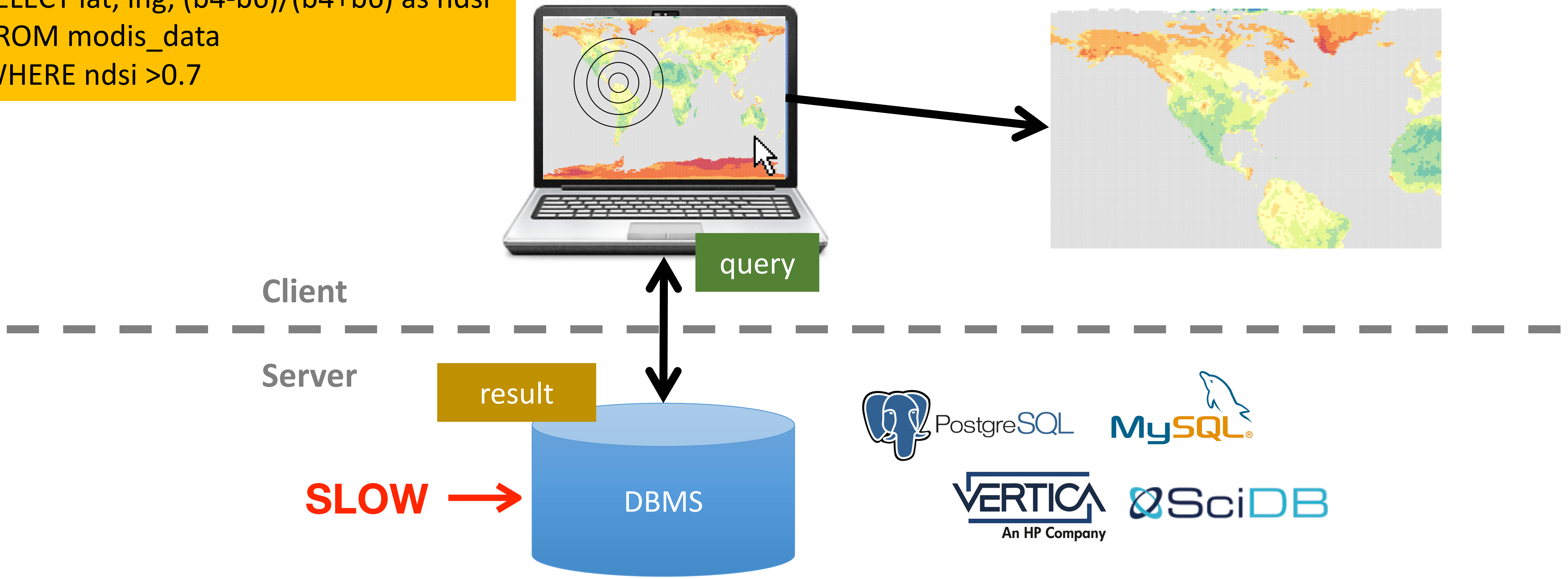
Interactive Exploration of Spatial Data



[L. Battle, 2017]

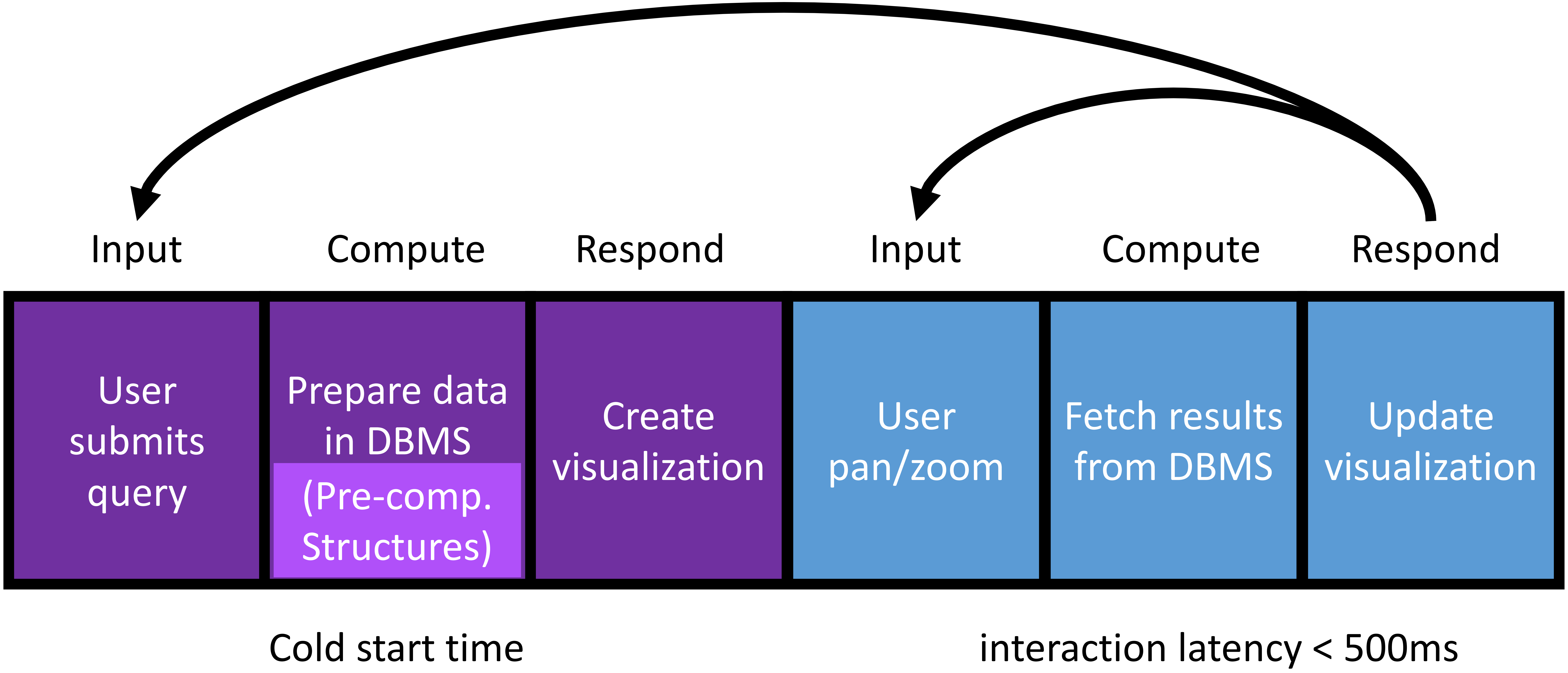
Interactive Exploration of Spatial Data

```
SELECT lat, lng, (b4-b6)/(b4+b6) as ndsi
FROM modis_data
WHERE ndsi >0.7
```



[L. Battle, 2017]

Two Inputs to Exploratory Browsing



[L. Battle, 2017]

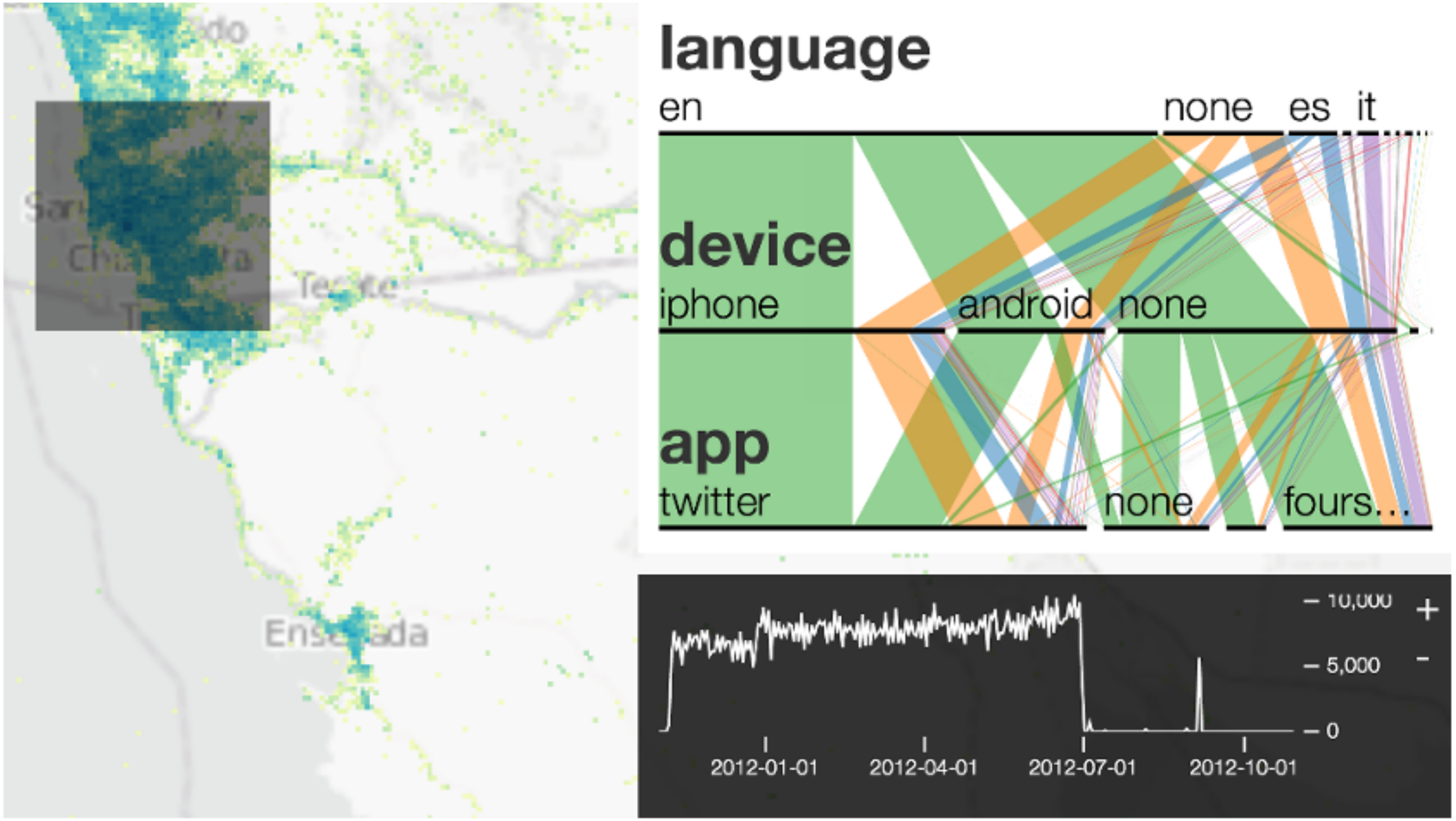
Systems for Interactive Exploration

Time

Output format	(Offline) **Pre-computed structures**	(Before interaction) **Predictive**	(After interaction) **Progressive/Incremental**
Sampling		DICE (ICDE 2014) A-WARE (HILDA 2016)	SampleAction (CHI 2012) Vizdom (VLDB 2015) DICE (ICDE 2014) A-WARE (HILDA 2016)
Aggregation	Nanocubes (Infovis 2013) imMens (Eurovis 2013) ForeCache	ATLAS (VAST 2008) Xmdv Tool (*DASFAA* 2003) ForeCache	

[L. Battle, 2017]

Nanocubes



Linked view of tweets in San Diego, US

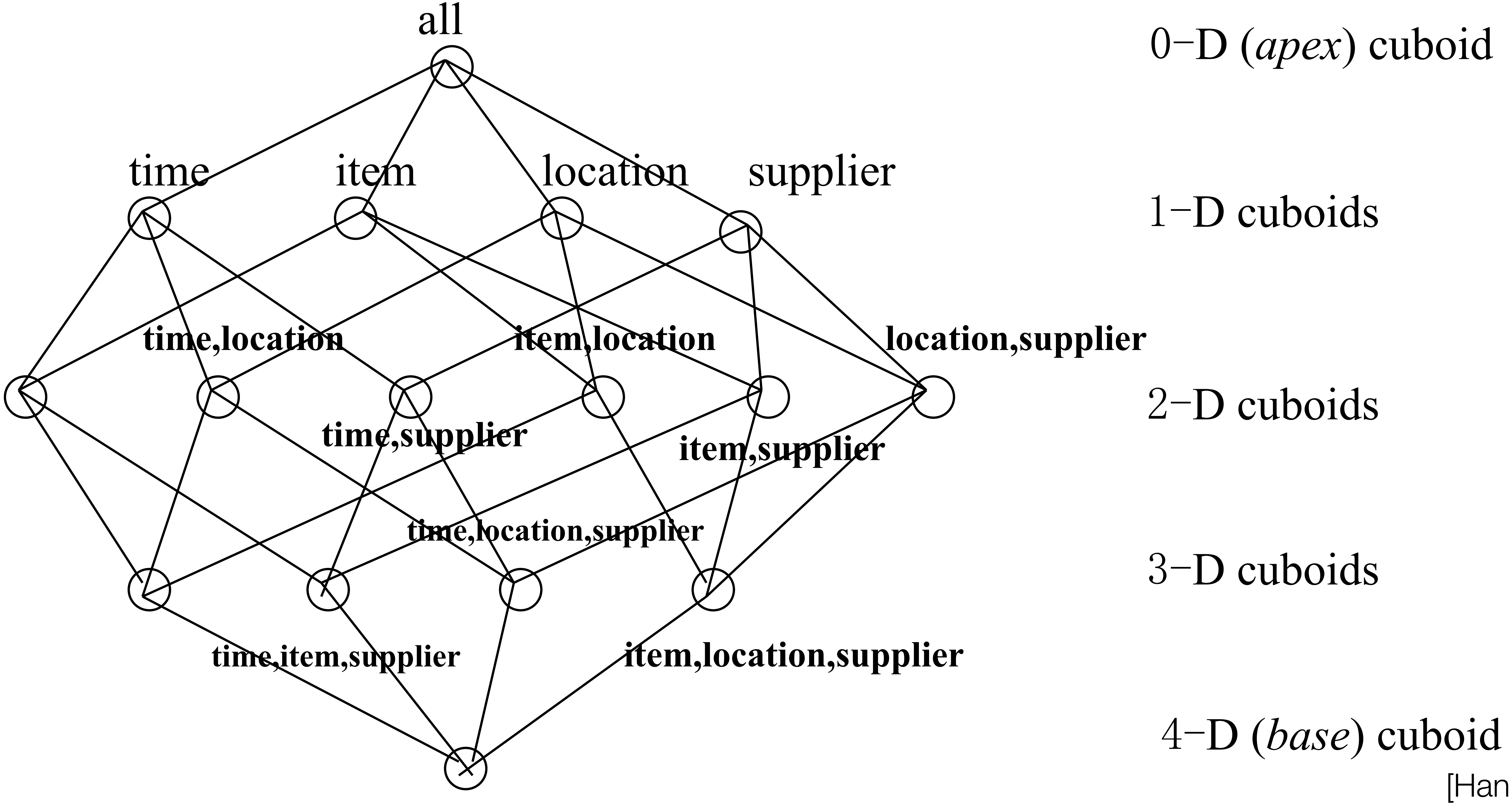
[Lins et. al, 2013]

From Tables and Spreadsheets to Data Cubes

- A **data warehouse** is based on a multidimensional data model which views data in the form of a data cube
- A **data cube**, such as sales, allows data to be modeled and viewed in multiple dimensions
  - **Dimension tables**, such as item (item_name, brand, type), or time(day, week, month, quarter, year)
  - **Fact table** contains **measures** (such as dollars_sold) and keys to each of the related dimension tables
- In data warehousing literature, an n-D base cube is called a **base cuboid**. The top most 0-D cuboid, which holds the highest-level of summarization, is called the **apex cuboid**. The lattice of cuboids forms a **data cube**.

[Han et al., 2011]

Data Cube: A Lattice of Cuboids



[Han et al., 2011]

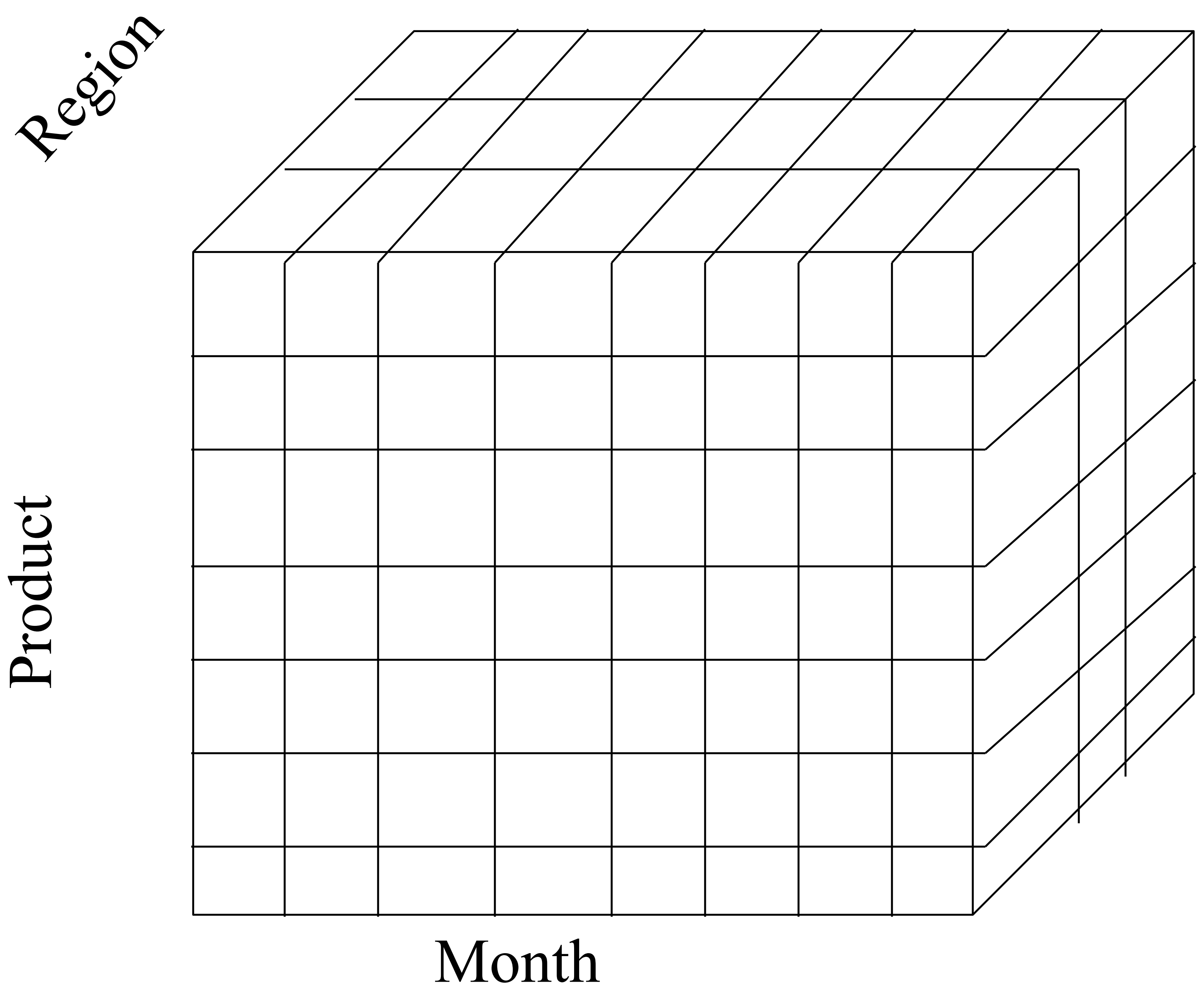
Data Cube Measures: Three Categories

- **Distributive**: if the result derived by applying the function to n aggregate values is the same as that derived by applying the function on all the data without partitioning
    - E.g., `count()`, `sum()`, `min()`, `max()`
- **Algebraic**: if it can be computed by an algebraic function with M arguments (where M is a bounded integer), each of which is obtained by applying a distributive aggregate function
    - E.g., `avg()`, `min_N()`, `standard_deviation()`
- **Holistic**: if there is no constant bound on the storage size needed to describe a subaggregate.
    - E.g., `median()`, `mode()`, `rank()`

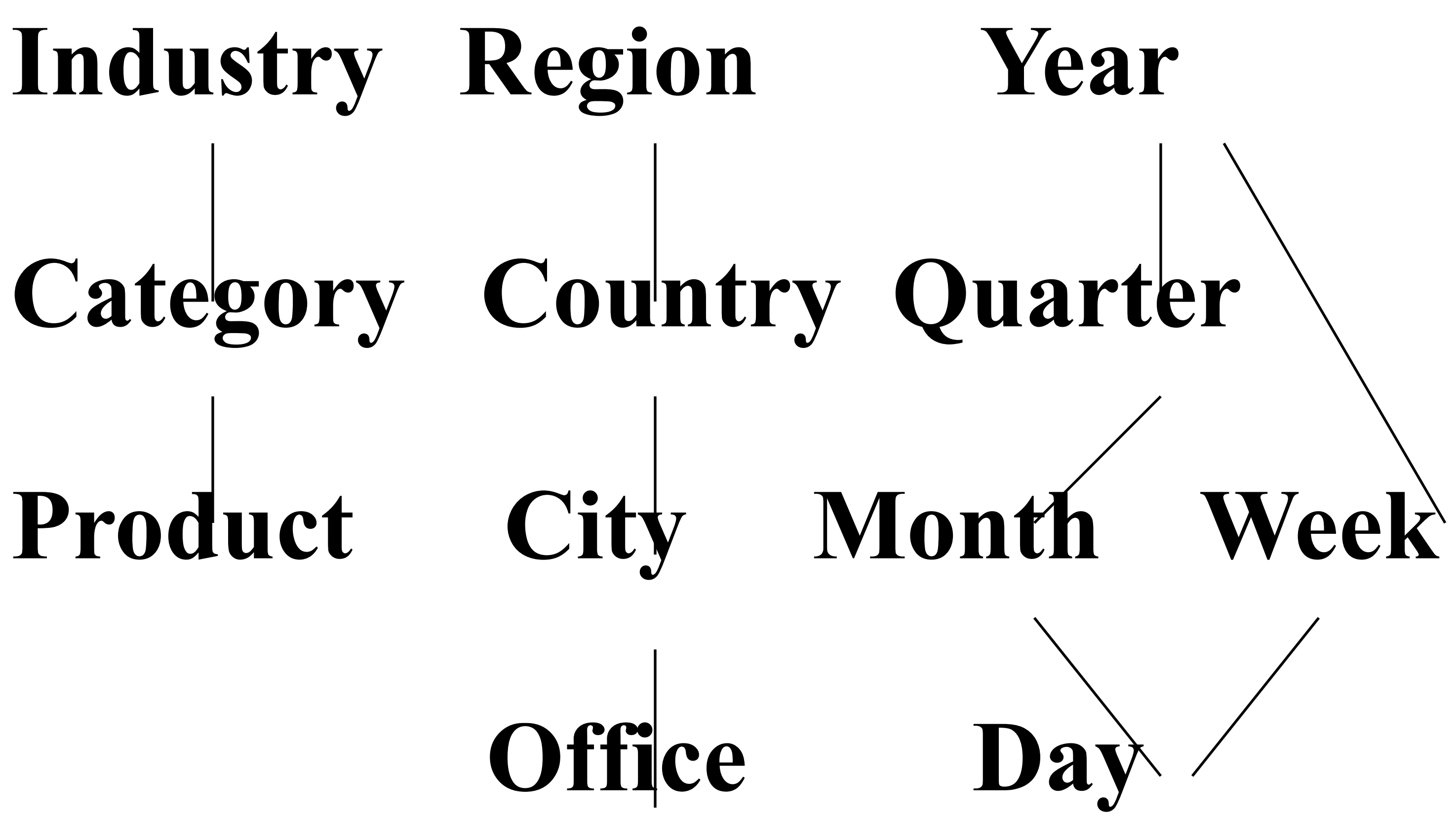
[Han et al., 2011]

Multidimensional Data

- Sales volume as a function of product, month, and region

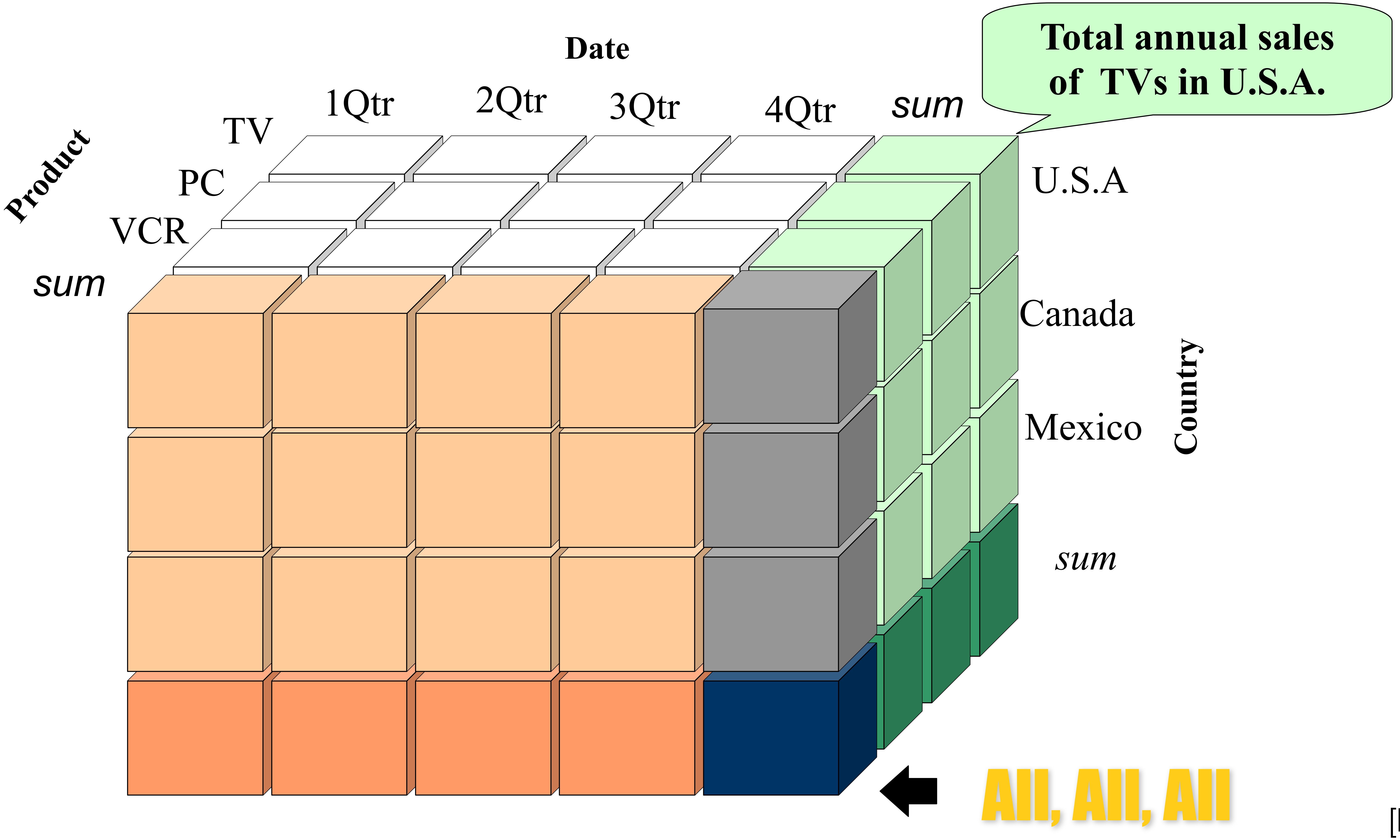


Dimensions: *Product, Location, Time*
Hierarchical summarization paths



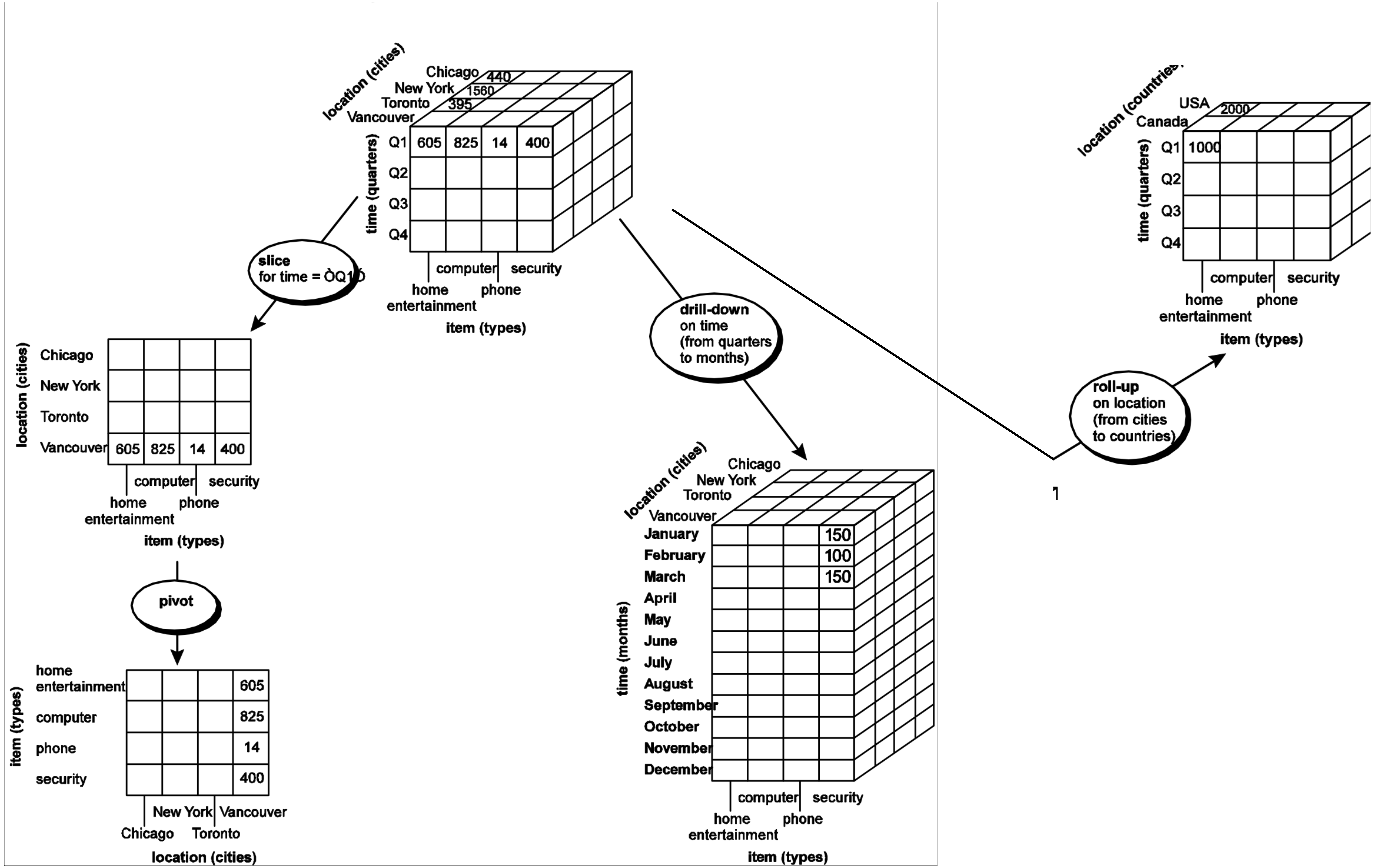
[Han et al., 2011]

A Sample Data Cube



[Han et al., 2011]

OLAP Operations



[Han et al., 2011]

Efficient Processing of OLAP Queries

- Determine which operations should be performed on the available cuboids
  - Transform drill, roll, etc. into corresponding SQL and/or OLAP operations, e.g., dice = selection + projection
- Determine which materialized cuboid(s) for OLAP operation:
  - Query: `{brand, province_or_state}` with `"year = 2004"`
  - 4 materialized cuboids available:
    1. `{year, item_name, city}`
    2. `{year, brand, country}`
    3. `{year, brand, province_or_state}`
    4. `{item_name, province_or_state}` where `year = 2004`
  - Which should be selected to process the query?

[Han et al., 2011]

Data Cube Aggregations

Relation A

Country	*Device*	*Language*
US	Android	en
US	iPhone	ru
South Africa	iPhone	en
India	Android	en
Australia	iPhone	en

Aggregation B

Country	*Device*	*Language*	*Count*
All	*All*	*All*	5

Group By on *Device, Language* C

Country	*Device*	*Language*	*Count*
All	Android	en	2
All	iPhone	en	2
All	iPhone	ru	1

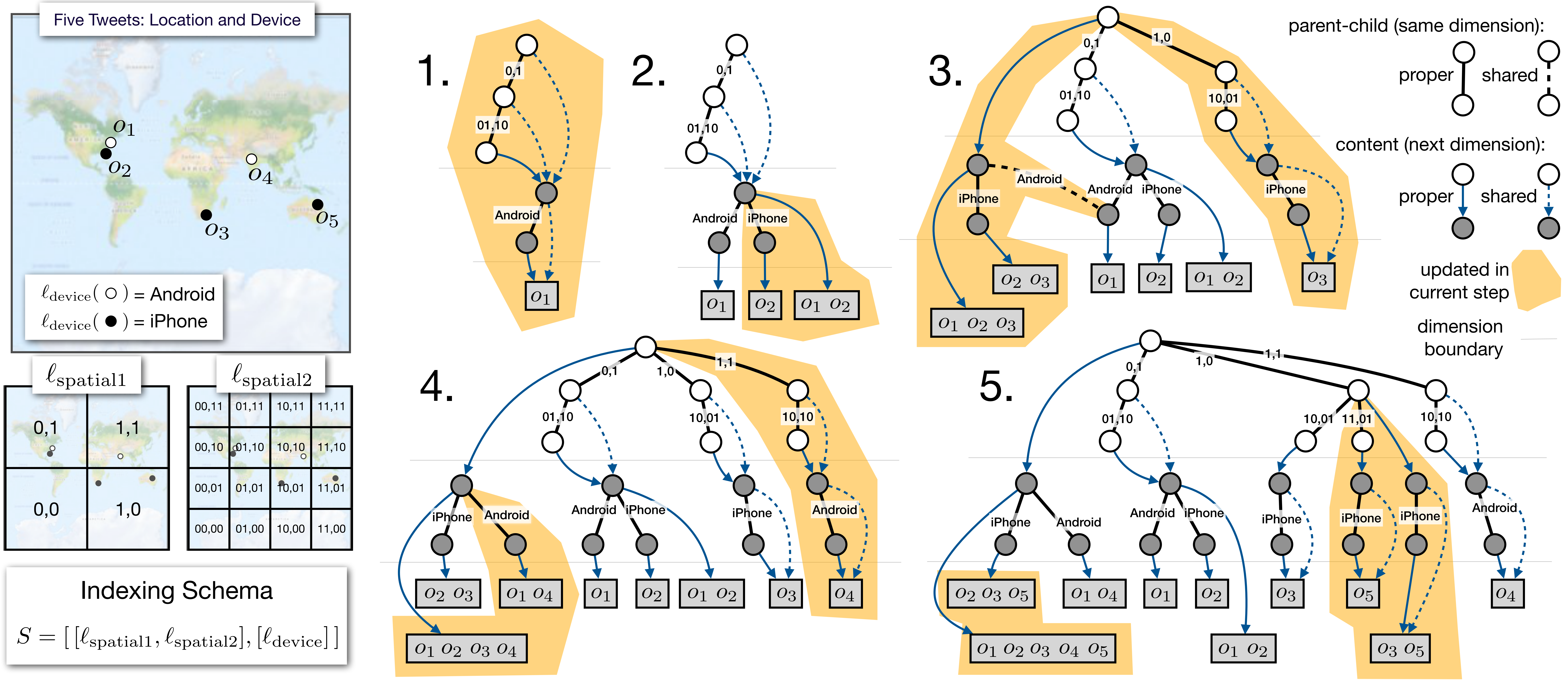
Cube on *Device, Language* D

Country	*Device*	*Language*	*Count*
All	*All*	*All*	5
All	Android	*All*	2
All	iPhone	*All*	3
All	*All*	en	4
All	*All*	ru	1
All	iPhone	ru	1
All	Android	en	2
All	iPhone	en	2

Equivalent to Group By on all possible subsets of {*Device, Language*}

[Lins et. al, 2013]

Building a Nanocube



[Lins et. al, 2013]

Assignment 5

- Chicago Bike Sharing Data
  - Spatial Analysis
  - Temporal Analysis
  - Graph Database (neo4j)

Teaching Evaluations

- This Wednesday (April 20) in class

TopKube: A Rank-Aware Data Cube for Real-Time Exploration of Spatiotemporal Data

F. Miranda, L. Lins, J. T. Klosowski, and C. T. Silva

TopKube: What about Top-k and Rankings?

- Aggregates are interesting
- Also, often interested in **top-k** answers given particular criteria
- …or **rankings**
- Search over time and space but find specific examples
- TopKube is a rank-aware data structure that computes top-k queries with low latency so interactive exploration is possible

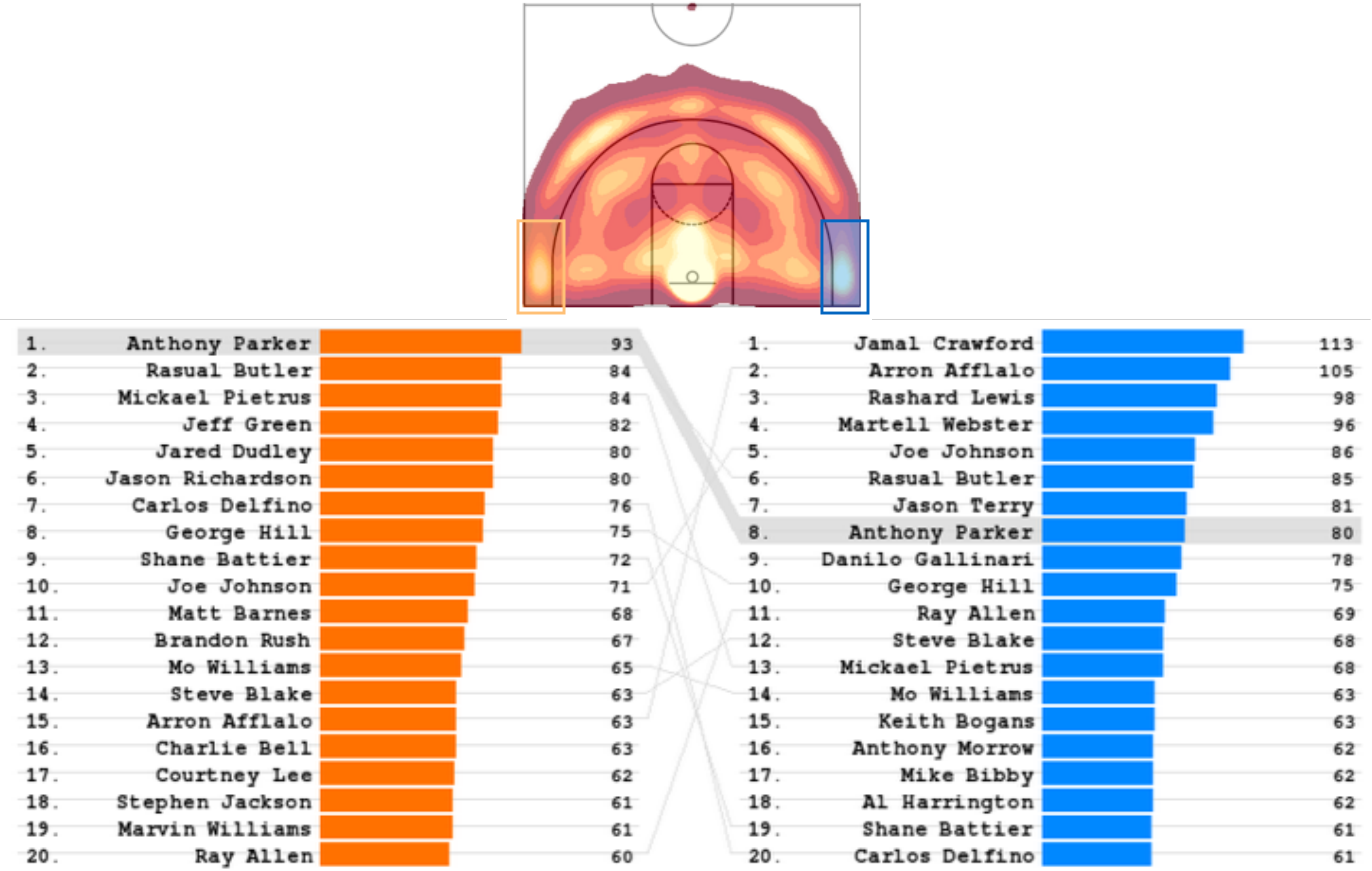
Example: Basketball

- Shots by time, number of points scored, and location on the court

team	player	time	pts	x	y
CLE	L. James	5	0	13	28
BOS	R. Rondo	5	2	38	26
CLE	L. James	7	3	42	35

- Query: Ranked list of the 50 players who took the most shots
  - `SELECT player,count(*) AS shots FROM table GROUP BY player ORDER BY shots DESC LIMIT 50`
- Query: Rank the top 50 players by points made:
  - `SELECT player,sum(pts) AS points FROM table GROUP BY player ORDER BY points DESC LIMIT 50`

Ranking by Shot Location



[F. Miranda et al., 2017]

TopKube vs. Nanocubes

- Product bin: the combination of selections from dimensions
- Nanocubes maps each product bin ((01,10), iPhone) to a **time series**

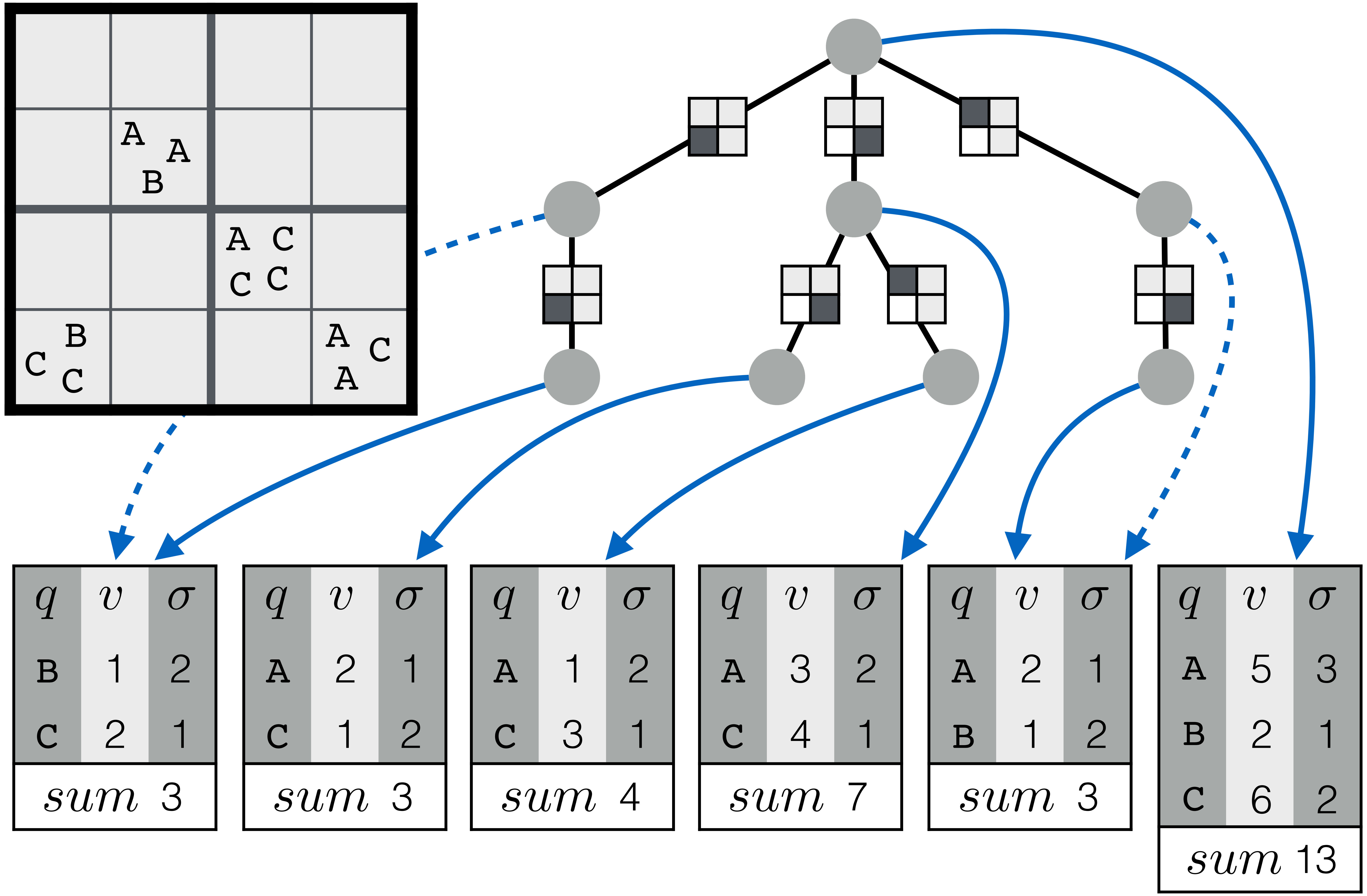
$$\beta \mapsto ((t_1, v_1), (t_2, v_1 + v_2), \ldots, (t_m, v_1 + \ldots + v_m))$$

- TopKube maps each product bin to **rank-aware multi-set**

$$\beta \mapsto \left\{ \text{lst} = ((q_1, v_1, \sigma_1), \ldots, (q_j, v_j, \sigma_j)), \text{sum} = \sum_{i=1}^{j} v_i \right\}$$

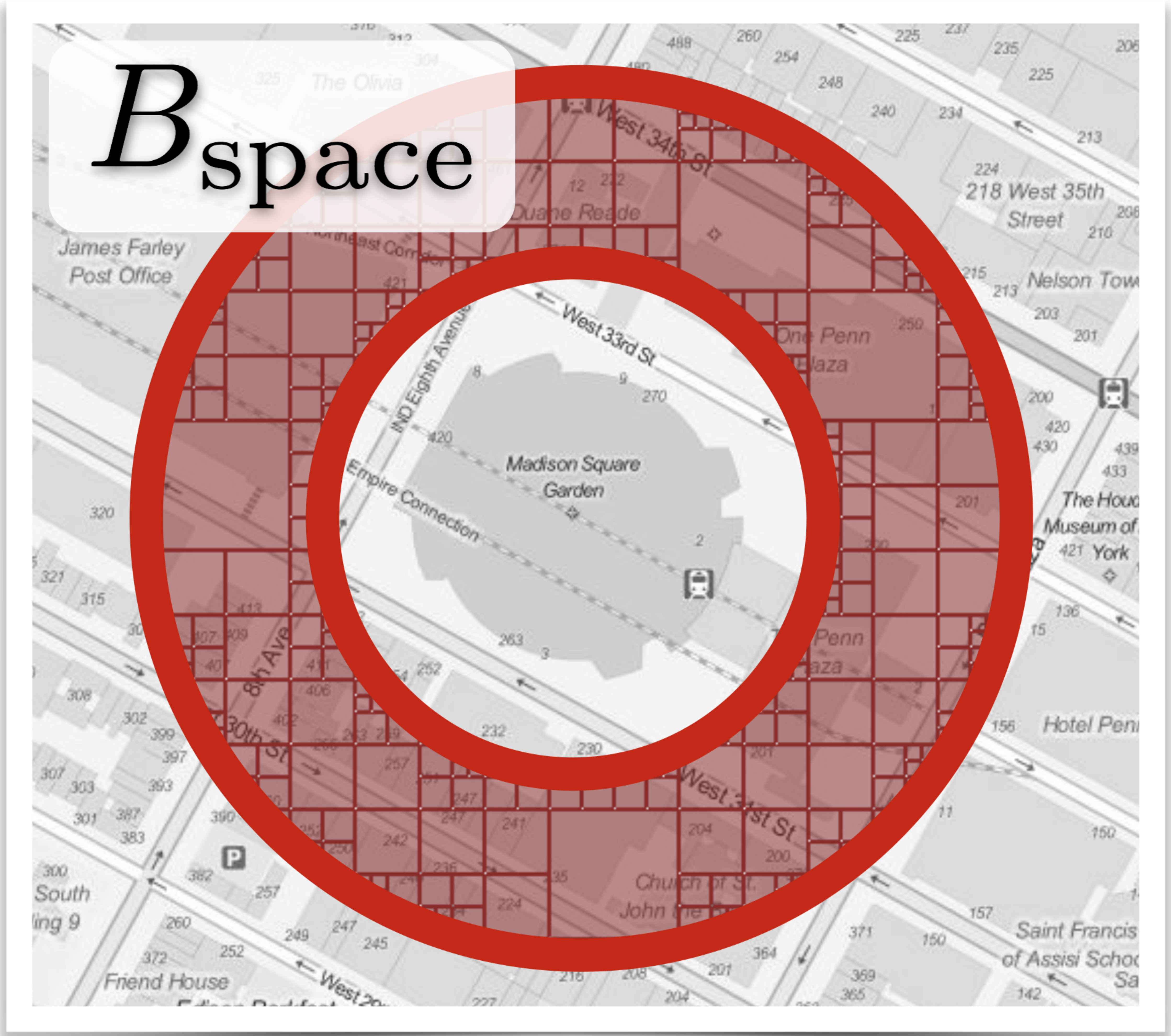
- q_i is the ith smallest key that appears in product bin
- v_i is the value of the measure for key q_i in the product bin
- σ_i is the index of the key with its largest value

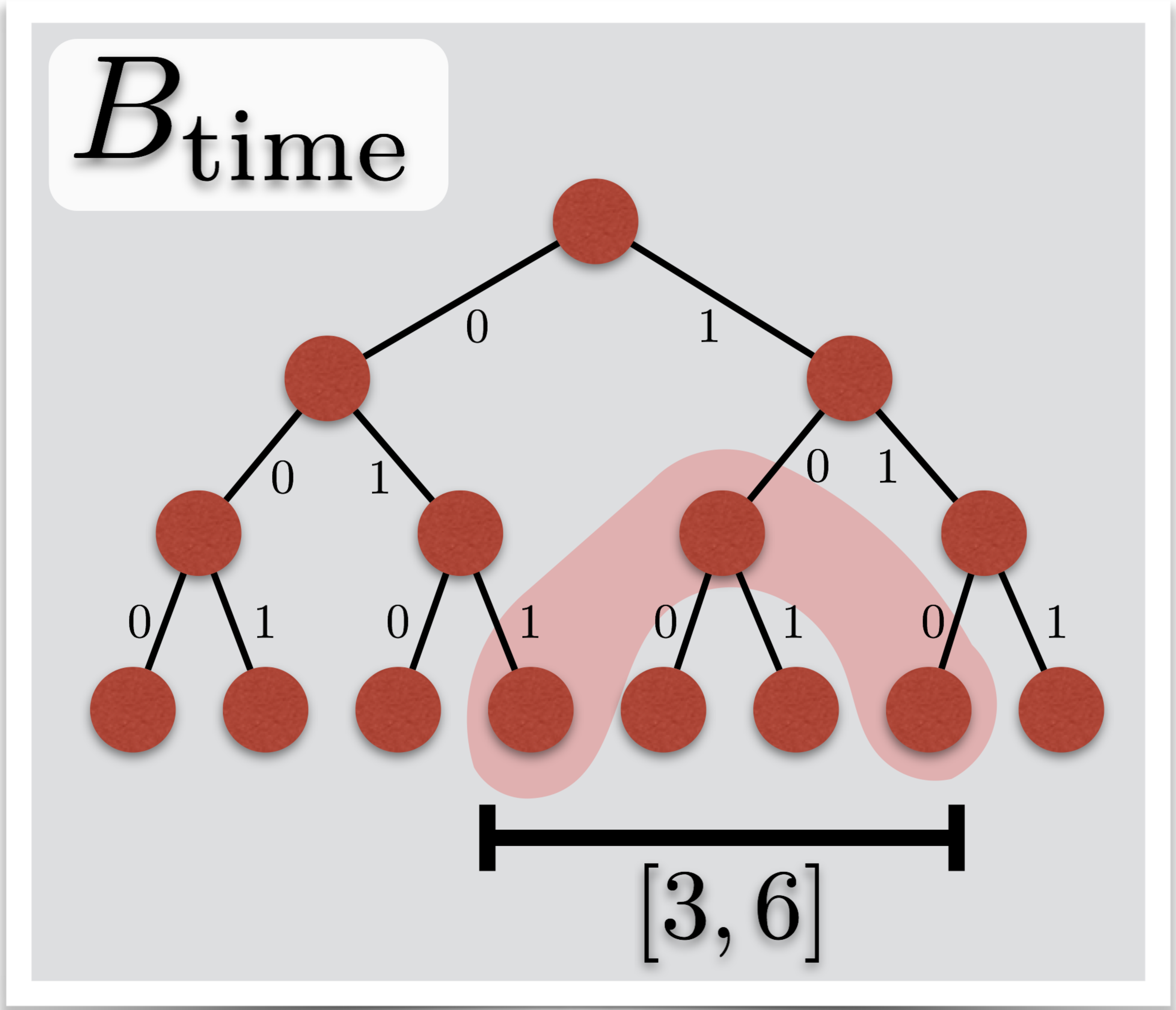
Example: One Spatial Dim. and A,B,C events



[F. Miranda et al., 2017]

Problem: Lots of Bins!



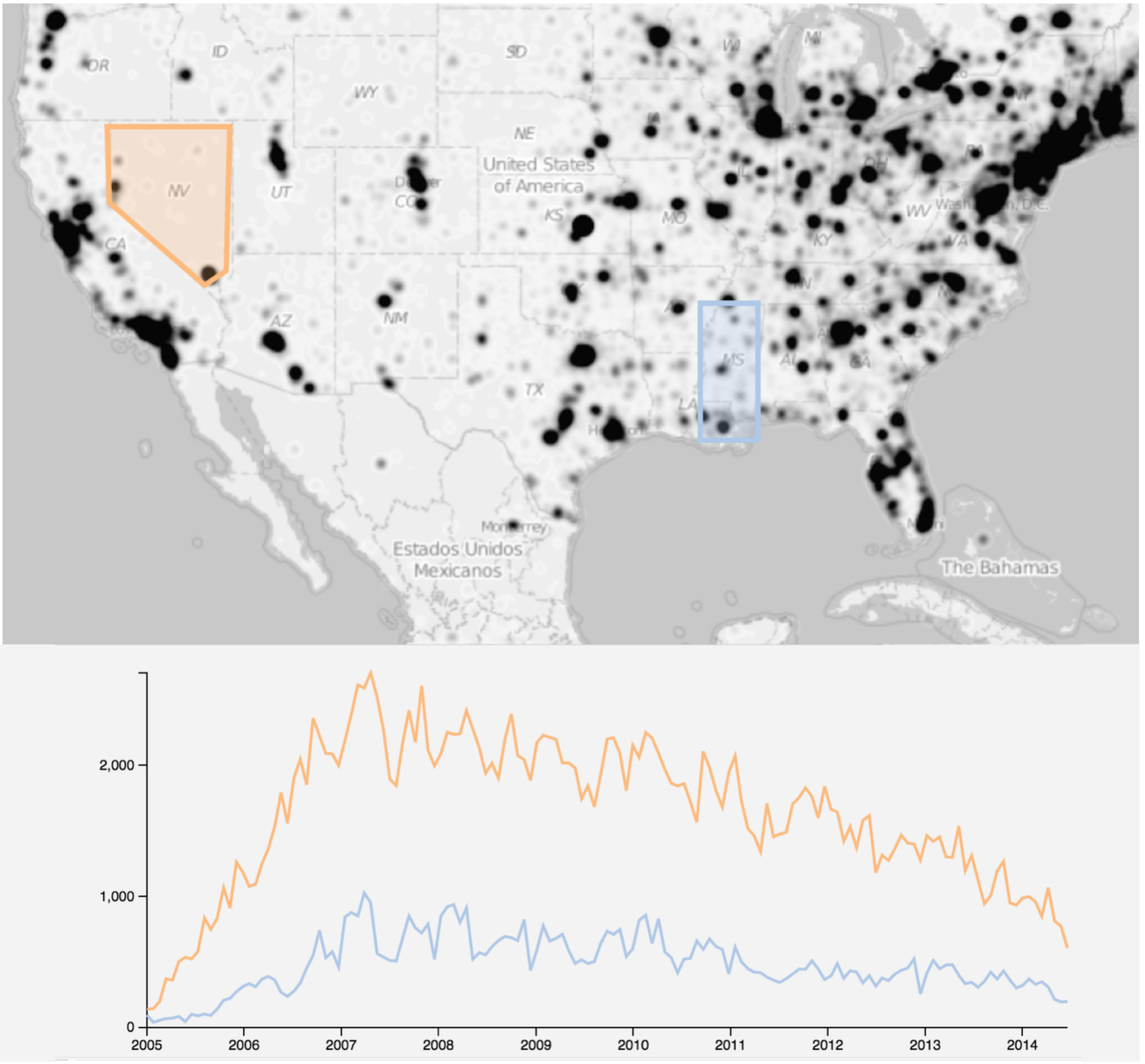


[F. Miranda et al., 2017]

Three Algorithms to Merge Bins

- Threshold: don't do a full scan, use extra information about ranking
- Sweep: Use a priority queue where the product bin with the current smallest key is on the top
- Hybrid:
  - Threshold has best theoretical guarantee but some sparse cases can be faster
  - Use Sweep on small input lists, Threshold on denser problem

Top-edited Wikipages in Nevada and Mississippi

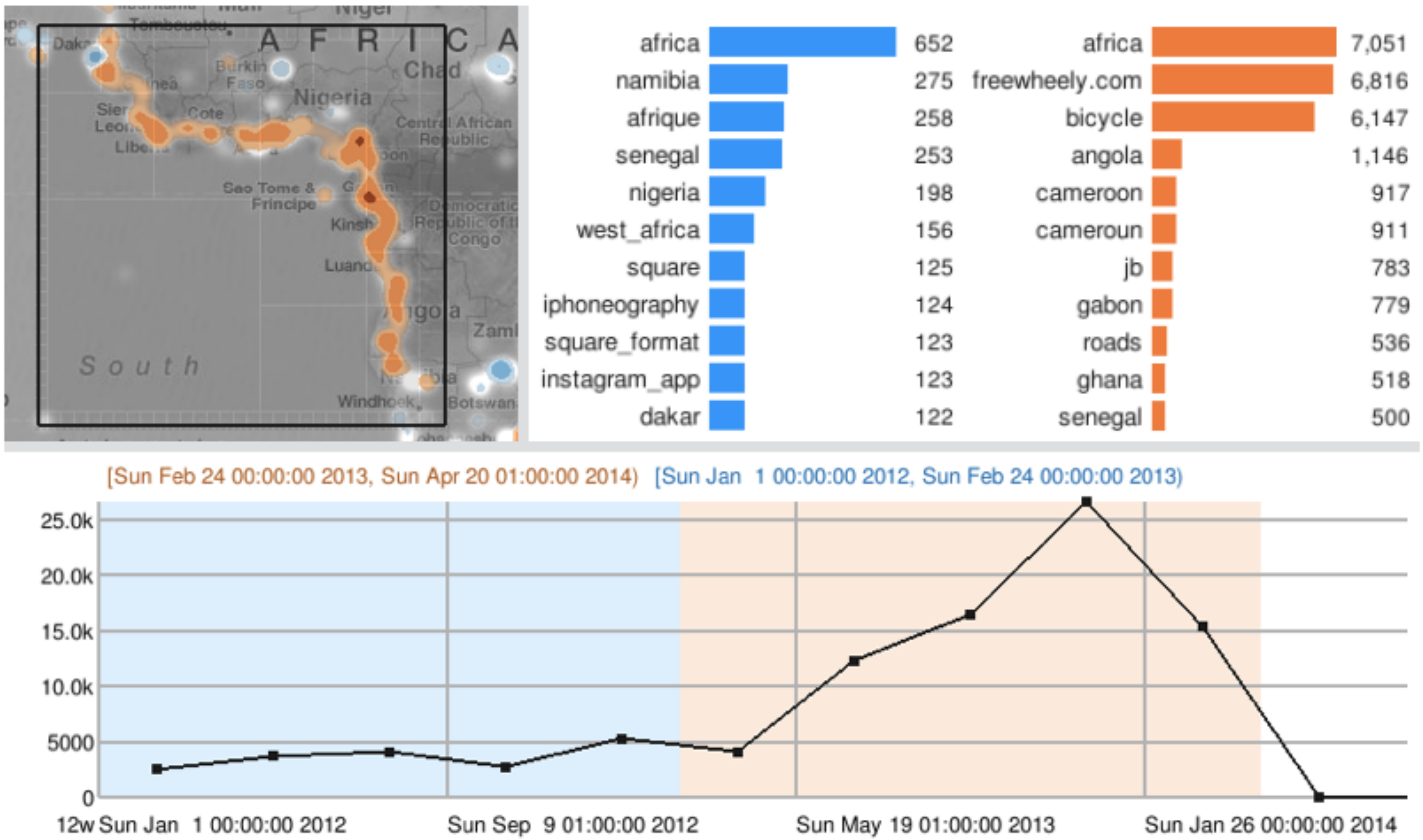


Baton_Rouge,_Louisiana	323
University_of_Mississippi...	230
Mississippi	216
Jackson,_Mississippi	208
Louisiana_State_University...	189
Mississippi_State_University...	169
WVLA-TV	158
Ole_Miss_Rebels_football...	155
List_of_Star_Wars_books...	131
Louisiana	122
New_Orleans_Saints	107

Reno,_Nevada	303
Early_Christianity	284
Comparison_of_the_AK-47_and_M1...	273
Las_Vegas_Academy	225
Timeline_of_Christianity...	216
Las_Vegas	204
Council_of_Jerusalem	192
Paul_the_Apostle	190
University_of_Nevada,_Las_Vega...	189
Nevada	188
Antinomianism	188

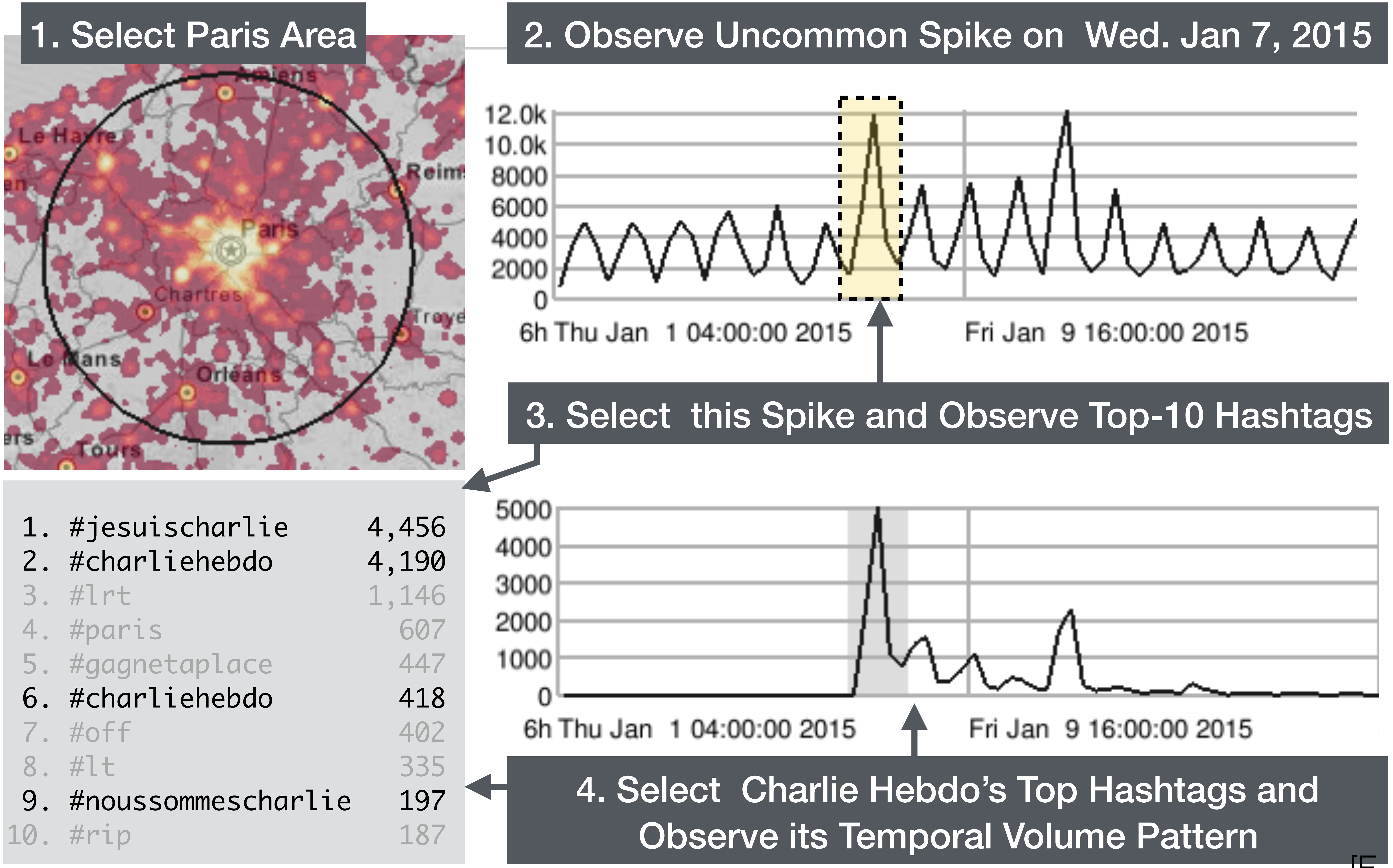
[F. Miranda et al., 2017]

Geolocated Flickr tags in Africa



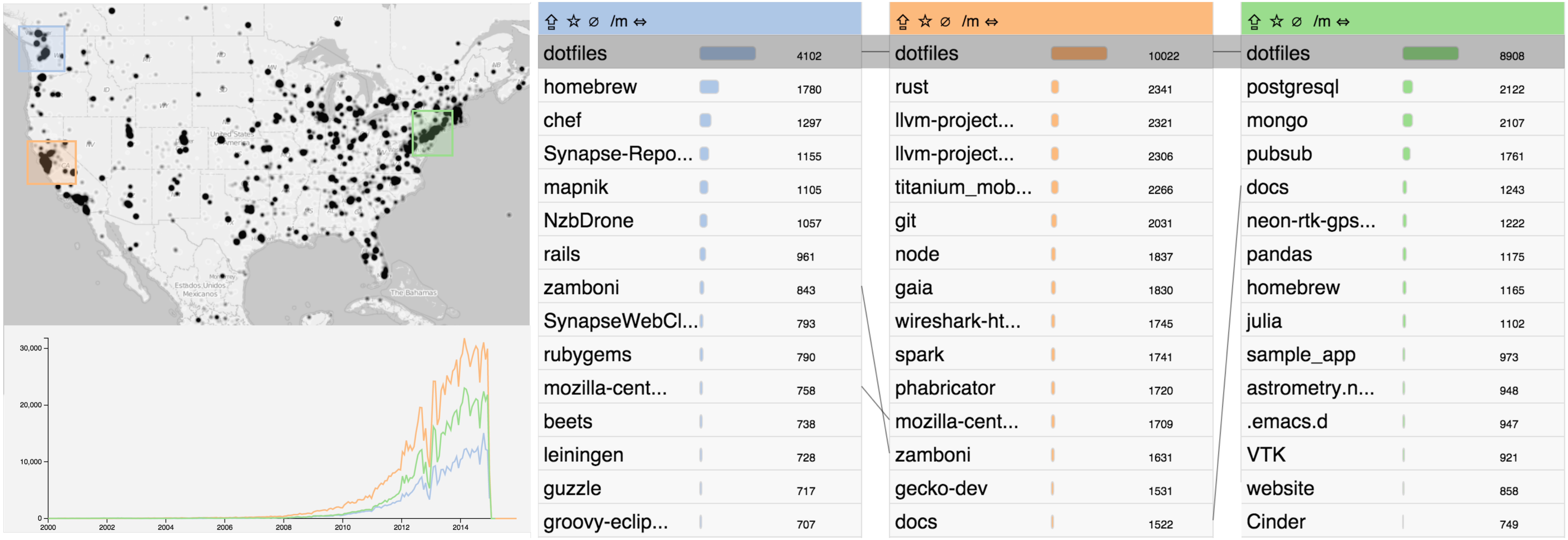
[F. Miranda et al., 2017]

Top Hashtags in Paris related to Charlie Hebdo



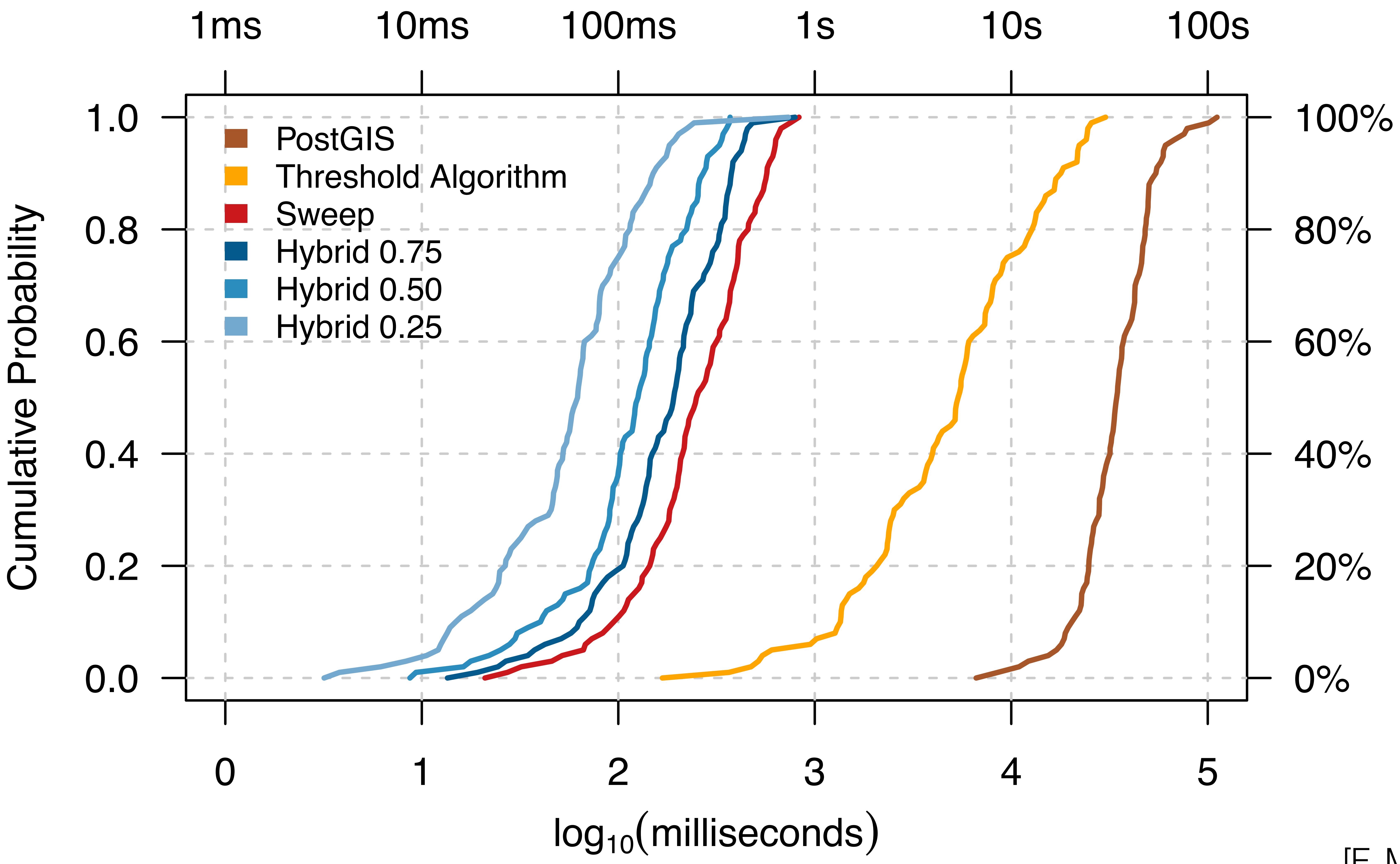
[F. Miranda et al., 2017]

GitHub Top commits near urban centers



[F. Miranda et al., 2017]

Evaluation



[F. Miranda et al., 2017]

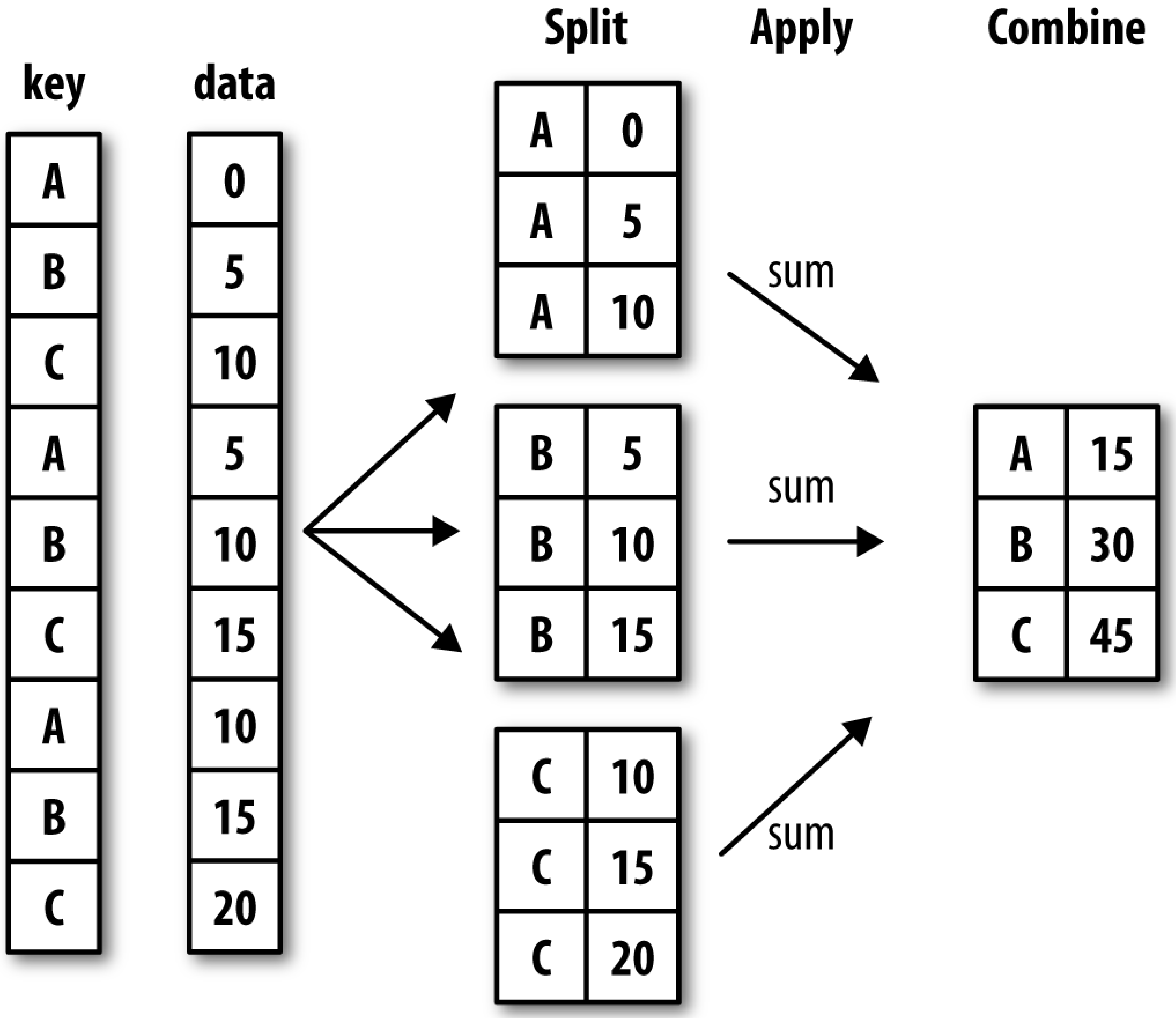
Aggregation

Split-Apply-Combine

- Coined by H. Wickham, 2011
- Similar to Map (split+apply) Reduce (combine) paradigm
- The Pattern:
  1. **Split** the data by some grouping variable
  2. **Apply** some function to each group independently
  3. **Combine** the data into some output dataset
- The apply step is usually one of :
  - Aggregate
  - Transform
  - Filter

[T. Brandt]

Split-Apply-Combine



[W. McKinney, Python for Data Analysis]

Splitting by Variables

name	age	sex
John	13	Male
Mary	15	Female
Alice	14	Female
Peter	13	Male
Roger	14	Male
Phyllis	13	Female

.(sex)

name	age	sex
John	13	*Male*
Peter	13	*Male*
Roger	14	*Male*

name	age	sex
Mary	15	*Female*
Alice	14	*Female*
Phyllis	13	*Female*

.(age)

name	age	sex
John	*13*	Male
Peter	*13*	Male
Phyllis	*13*	Female

name	age	sex
Alice	*14*	Female
Roger	*14*	Male

name	age	sex
Mary	*15*	Female

[H. Wickham, 2011]

Apply+Combine: Counting

.(sex)

sex	value
Male	3
Female	3

.(age)

age	value
13	3
14	2
15	1

.(sex, age)

sex	age	value
Male	13	2
Male	14	1
Female	13	1
Female	14	1
Female	15	1

[H. Wickham, 2011]

In Pandas

- `groupby` method creates a `GroupBy` object
- `groupby` doesn't actually compute anything until there is an apply/aggregate step or we wish to examine the groups
- Choose keys (columns) to group by
- `size()` is the count of each group

Aggregation

- Operations:
  - `count()`
  - `mean()`
  - `sum()`
- May also wish to aggregate only certain subsets
  - Use square brackets with column names
- Can also write your own functions for aggregation and pass then to `agg` function
  - ```
    def peak_to_peak(arr):
        return arr.max() - arr.min()
    grouped.agg(peak_to_peak)
    ```
    ```

Optimized groupby methods

Function name	Description
`count`	Number of non-NA values in the group
`sum`	Sum of non-NA values
`mean`	Mean of non-NA values
`median`	Arithmetic median of non-NA values
`std, var`	Unbiased (n – 1 denominator) standard deviation and variance
`min, max`	Minimum and maximum of non-NA values
`prod`	Product of non-NA values
`first, last`	First and last non-NA values

[W. McKinney, Python for Data Analysis]

Iterating over groups

- ```
  for name, group in df.groupby('key1'):
      print(name)
      print(group)
  ```
- Can also `.describe()` groups
  ```

Apply: Generalized methods

```
In [74]: def top(df, n=5, column='tip_pct'):
   ....:     return df.sort_values(by=column)[-n:]

In [75]: top(tips, n=6)
Out[75]:
     total_bill   tip smoker  day    time  size   tip_pct
109       14.31  4.00    Yes  Sat  Dinner     2  0.279525
183       23.17  6.50    Yes  Sun  Dinner     4  0.280535
232       11.61  3.39     No  Sat  Dinner     2  0.291990
67         3.07  1.00    Yes  Sat  Dinner     1  0.325733
178        9.60  4.00    Yes  Sun  Dinner     2  0.416667
172        7.25  5.15    Yes  Sun  Dinner     2  0.710345

In [76]: tips.groupby('smoker').apply(top)
Out[76]:
            total_bill   tip smoker   day    time  size   tip_pct
smoker
No     88        24.71  5.85     No  Thur   Lunch     2  0.236746
       185       20.69  5.00     No   Sun  Dinner     5  0.241663
       51        10.29  2.60     No   Sun  Dinner     2  0.252672
       149        7.51  2.00     No  Thur   Lunch     2  0.266312
       232       11.61  3.39     No   Sat  Dinner     2  0.291990
Yes    109       14.31  4.00    Yes   Sat  Dinner     2  0.279525
       183       23.17  6.50    Yes   Sun  Dinner     4  0.280535
       67         3.07  1.00    Yes   Sat  Dinner     1  0.325733
       178        9.60  4.00    Yes   Sun  Dinner     2  0.416667
       172        7.25  5.15    Yes   Sun  Dinner     2  0.710345
```

[W. McKinney]

Apply

- `tips.groupby('smoker').apply(top)`
- Function is an **argument**
- Function applied on each **row group**
- All row groups glued together using `concat`

Types of GroupBy

- Aggregation: `agg`
  - `n:1` n group values become one value
  - Examples: mean, min, median
- Apply: `apply`
  - `n:m` n group values become m values
  - Most general (could do aggregation or transform with apply)
  - Example: top 5 in each group, filter
- Transform: `transform`
  - `n:n` n group values become n values
  - Cannot mutate the input

Transform Example

```
In [76]: df
Out[76]:
   key  value
0    a    0.0
1    b    1.0
2    c    2.0
3    a    3.0
4    b    4.0
5    c    5.0
6    a    6.0
7    b    7.0
8    c    8.0
9    a    9.0
10   b   10.0
11   c   11.0
```

```
In [77]: g = df.groupby('key').value

In [78]: g.mean()
Out[78]:
key
a    4.5
b    5.5
c    6.5
Name: value, dtype: float64

In [79]: g.transform(lambda x: x.mean())
Out[79]:
0     4.5
1     5.5
2     6.5
3     4.5
4     5.5
5     6.5
6     4.5
7     5.5
8     6.5
9     4.5
10    5.5
11    6.5
Name: value, dtype: float64
```

[W. McKinney, Python for Data Analysis]

Transform Example

```
In [76]: df
Out[76]:
   key  value
0    a    0.0
1    b    1.0
2    c    2.0
3    a    3.0
4    b    4.0
5    c    5.0
6    a    6.0
7    b    7.0
8    c    8.0
9    a    9.0
10   b   10.0
11   c   11.0
```

```
In [77]: g = df.groupby('key').value

In [78]: g.mean()
Out[78]:
key
a    4.5
b    5.5
c    6.5
Name: value, dtype: float64

In [79]: g.transform(lambda x: x.mean())
Out[79]:
0     4.5
1     5.5
2     6.5
3     4.5
4     5.5
5     6.5
6     4.5
7     5.5
8     6.5
9     4.5
10    5.5
11    6.5
Name: value, dtype: float64
```

or `g.transform('mean')`

[W. McKinney, Python for Data Analysis]

Normalization

```
def normalize(x):
    return (x - x.mean()) / x.std()
```

```
In [84]: g.transform(normalize)
Out[84]:
0    -1.161895
1    -1.161895
2    -1.161895
3    -0.387298
4    -0.387298
5    -0.387298
6     0.387298
7     0.387298
8     0.387298
9     1.161895
10    1.161895
11    1.161895
Name: value, dtype: float64
```

==

```
In [85]: g.apply(normalize)
Out[85]:
0    -1.161895
1    -1.161895
2    -1.161895
3    -0.387298
4    -0.387298
5    -0.387298
6     0.387298
7     0.387298
8     0.387298
9     1.161895
10    1.161895
11    1.161895
Name: value, dtype: float64
```

[W. McKinney]

Normalization

```python
def normalize(x):
    return (x - x.mean()) / x.std()
```

```
In [84]: g.transform(normalize)
Out[84]:
0    -1.161895
1    -1.161895
2    -1.161895
3    -0.387298
4    -0.387298
5    -0.387298
6     0.387298
7     0.387298
8     0.387298
9     1.161895
10    1.161895
11    1.161895
Name: value, dtype: float64
```

==

```
In [85]: g.apply(normalize)
Out[85]:
0    -1.161895
1    -1.161895
2    -1.161895
3    -0.387298
4    -0.387298
5    -0.387298
6     0.387298
7     0.387298
8     0.387298
9     1.161895
10    1.161895
11    1.161895
Name: value, dtype: float64
```

```
In [87]: normalized = (df['value'] - g.transform('mean')) / g.transform('std')
```

Fastest: "Unwrapped" group operation

[W. McKinney]

Other Operations

- Quantiles: return values at particular splits
  - Median is a 0.5-quantile
  - `df.quantile(0.1)`
  - also works on groups
- Can return data from group-by without having the keys in the index (`as_index=False`) or use `reset_index` after computing
- Grouped weighted average via apply

Pivot Tables

- Data summarization tool in many spreadsheet programs
- Aggregates a table of data by one or more keys with some keys arranged on rows (`index`), others as columns (`columns`)
- Pandas supports via `pivot_table` method
- `margins=True` gives partial totals
- Can use different aggregation functions via `aggfunc` kwarg

Function name	Description
`values`	Column name or names to aggregate. By default aggregates all numeric columns
`rows`	Column names or other group keys to group on the rows of the resulting pivot table
`cols`	Column names or other group keys to group on the columns of the resulting pivot table
`aggfunc`	Aggregation function or list of functions; `'mean'` by default. Can be any function valid in a groupby context
`fill_value`	Replace missing values in result table
`margins`	Add row/column subtotals and grand total, False by default

[W. McKinney, Python for Data Analysis]

Pivot Tables in Pandas

- `tips`

	total_bill	tip	sex	smoker	day	time	size	tip_pct
0	16.99	1.01	Female	No	Sun	Dinner	2	0.059447
1	10.34	1.66	Male	No	Sun	Dinner	3	0.160542
2	21.01	3.50	Male	No	Sun	Dinner	3	0.166587
3	23.68	3.31	Male	No	Sun	Dinner	2	0.139780
4	24.59	3.61	Female	No	Sun	Dinner	4	0.146808
5	25.29	4.71	Male	No	Sun	Dinner	4	0.186240
6	8.77	2.00	Male	No	Sun	Dinner	2	0.228050

- `tips.pivot_table(index=['sex', 'smoker'])`

		size	tip	tip_pct	total_bill
sex	**smoker**				
Female	**No**	2.592593	2.773519	0.156921	18.105185
	Yes	2.242424	2.931515	0.182150	17.977879
Male	**No**	2.711340	3.113402	0.160669	19.791237
	Yes	2.500000	3.051167	0.152771	22.284500

Pivot Tables with Margins and Aggfunc

- `tips.pivot_table(['size'], index=['sex', 'day'], columns='smoker', aggfunc='sum', margins=True)`

		size		
	smoker	**No**	**Yes**	**All**
sex	**day**			
Female	**Fri**	2.0	7.0	9.0
	Sat	13.0	15.0	28.0
	Sun	14.0	4.0	18.0
	Thur	25.0	7.0	32.0
Male	**Fri**	2.0	8.0	10.0
	Sat	32.0	27.0	59.0
	Sun	43.0	15.0	58.0
	Thur	20.0	10.0	30.0
All		151.0	93.0	244.0

Crosstabs

- `crosstab` is a special case for group frequencies (`aggfunc='count'`)

```
In [293]: pd.crosstab(data.Gender, data.Handedness, margins=True)
Out[293]:
Handedness  Left-handed  Right-handed  All
Gender
Female                1             4    5
Male                  2             3    5
All                   3             7   10
```

- Tipping example
- Also see the Federal Election Database example in the book

Crosstabs

- `pd.crosstab([tips.time, tips.day], tips.smoker, margins=True)`

	smoker	No	Yes	All
time	day			
Dinner	Fri	3	9	12
	Sat	45	42	87
	Sun	57	19	76
	Thur	1	0	1
Lunch	Fri	1	6	7
	Thur	44	17	61
All		151	93	244

- or… `tips.pivot_table('total_bill',index=['time', 'day'], columns=['smoker'], aggfunc='count', margins=True, fill value=0)`

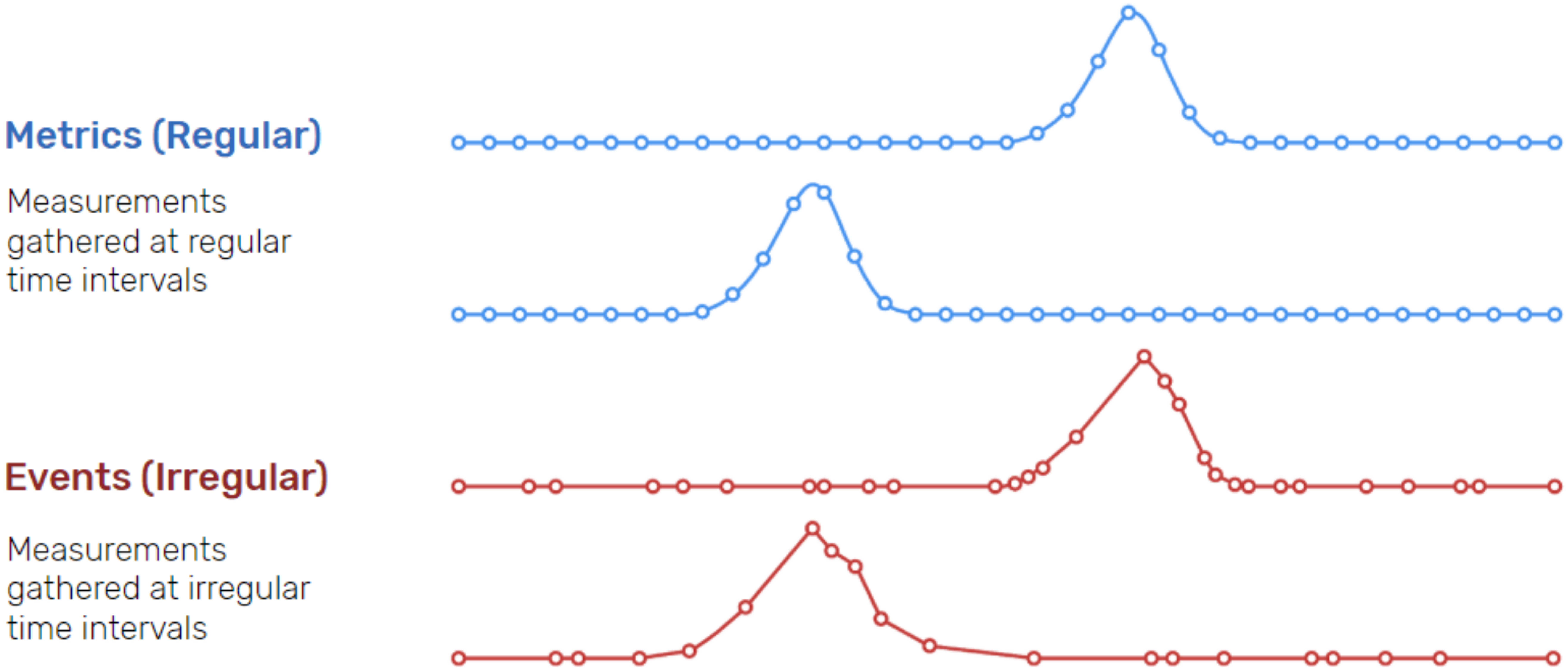
Time Series Data

What is time series data?

- Technically, it's normal tabular data with a timestamp attached
- But… we have observations of the same values over time, usually **in order**
- This allows more analysis
- Example: Web site database that tracks the last time a user logged in
  - 1: Keep an attribute `lastLogin` that is **overwritten** every time user logs in
  - 2: **Add a new row** with login information every time the user logs in
  - Option 2 takes more storage, but we can also do a lot more analysis!

Time Series Data

- Metrics: measurements at regular intervals
- Events: measurements that are not gathered at regular intervals



[InfluxDB]

Types of Time Series Data

- time series: observations for a **single** entity at **different** time intervals
  - one patient's heart rate every minute
- cross-section: observations for **multiple** entities at the **same** point in time
  - heart rates of 100 patients at 8:01pm
- panel data: observations for **multiple** entities at **different** time intervals
  - heart rates of 100 patients every minute over the past hour

[InfluxDB]

Time Series Databases

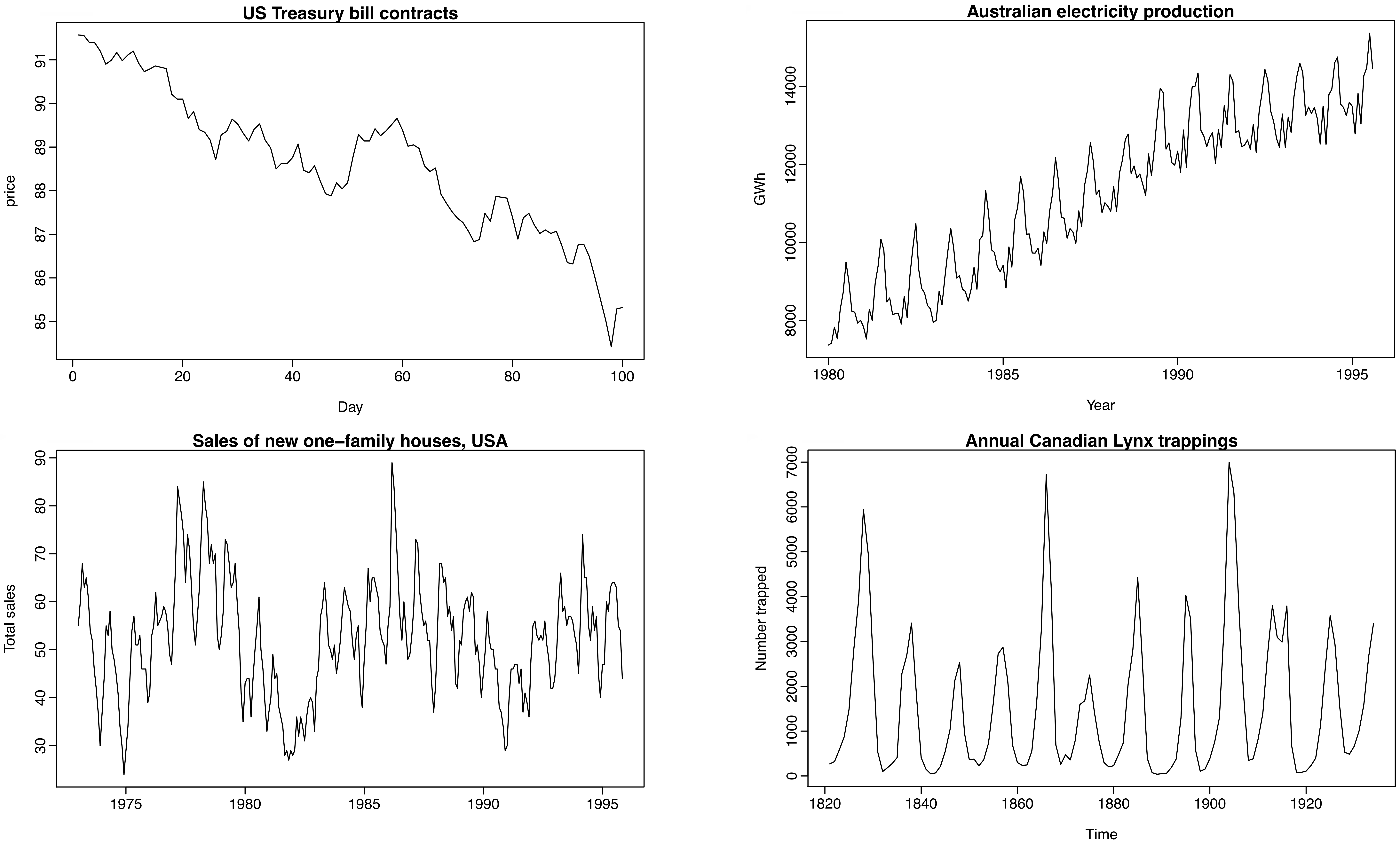
- Most time series data is heavy **inserts**, few updates
- Also analysis tends to be on ordered data with trends, prediction, etc.
- Can also consider **stream** processing
- Focus on time series allows databases to specialize
- Examples:
  - InfluxDB (noSQL)
  - TimescaleDB (SQL-based)

Features of Time Series Data

- Trend: long-term increase or decrease in the data
- Seasonal Pattern: time series is affected by seasonal factors such as the time of the year or the day of the week (fixed and of known frequency)
- Cyclic Pattern: rises and falls that are not of a fixed frequency
- Stationary: no predictable patterns (roughly horizontal with constant variance)
  - White noise series is stationary
  - Will look the basically the same whenever you observe it

[Hyndman and Athanosopoulos]

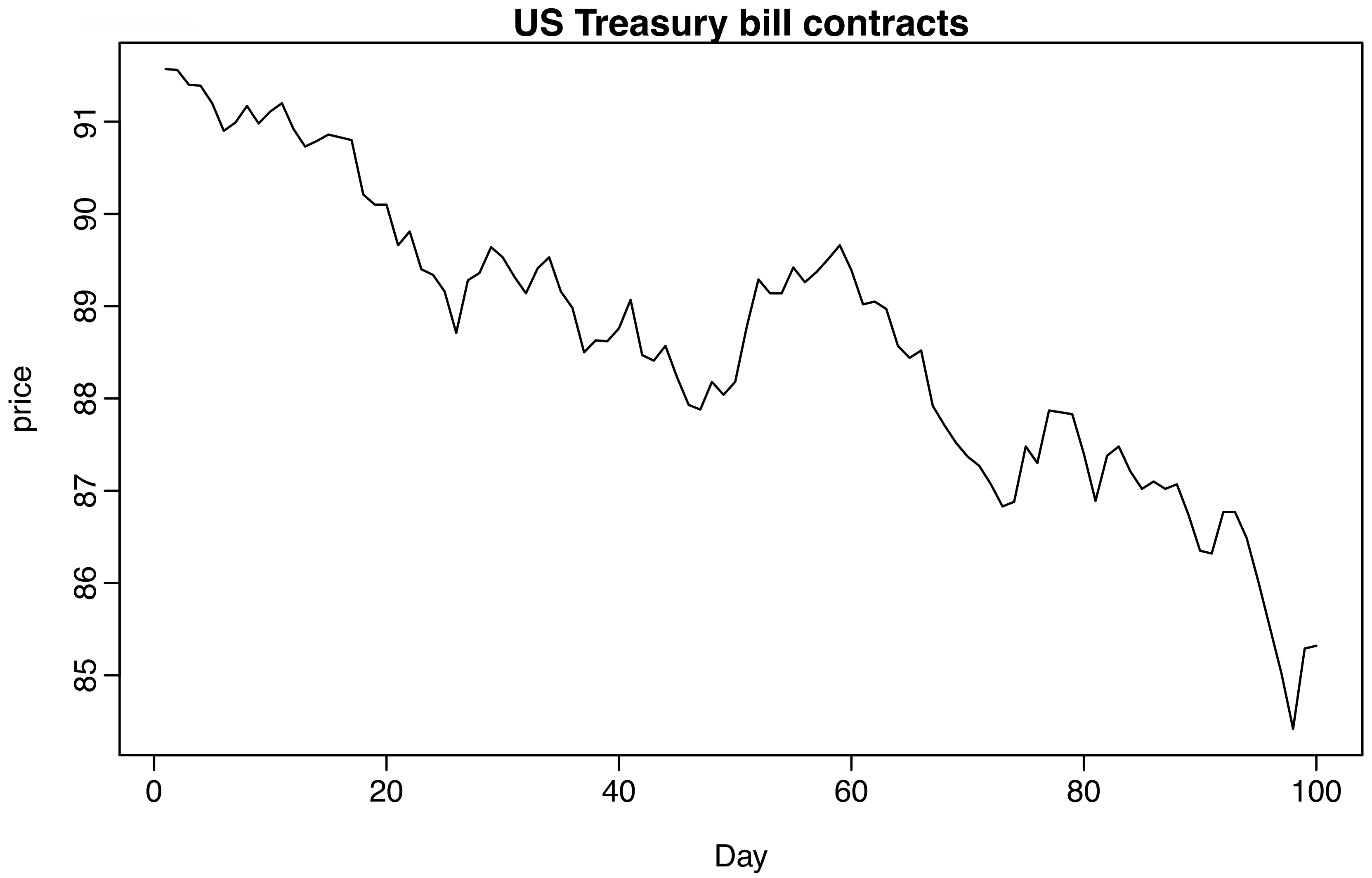
Examples

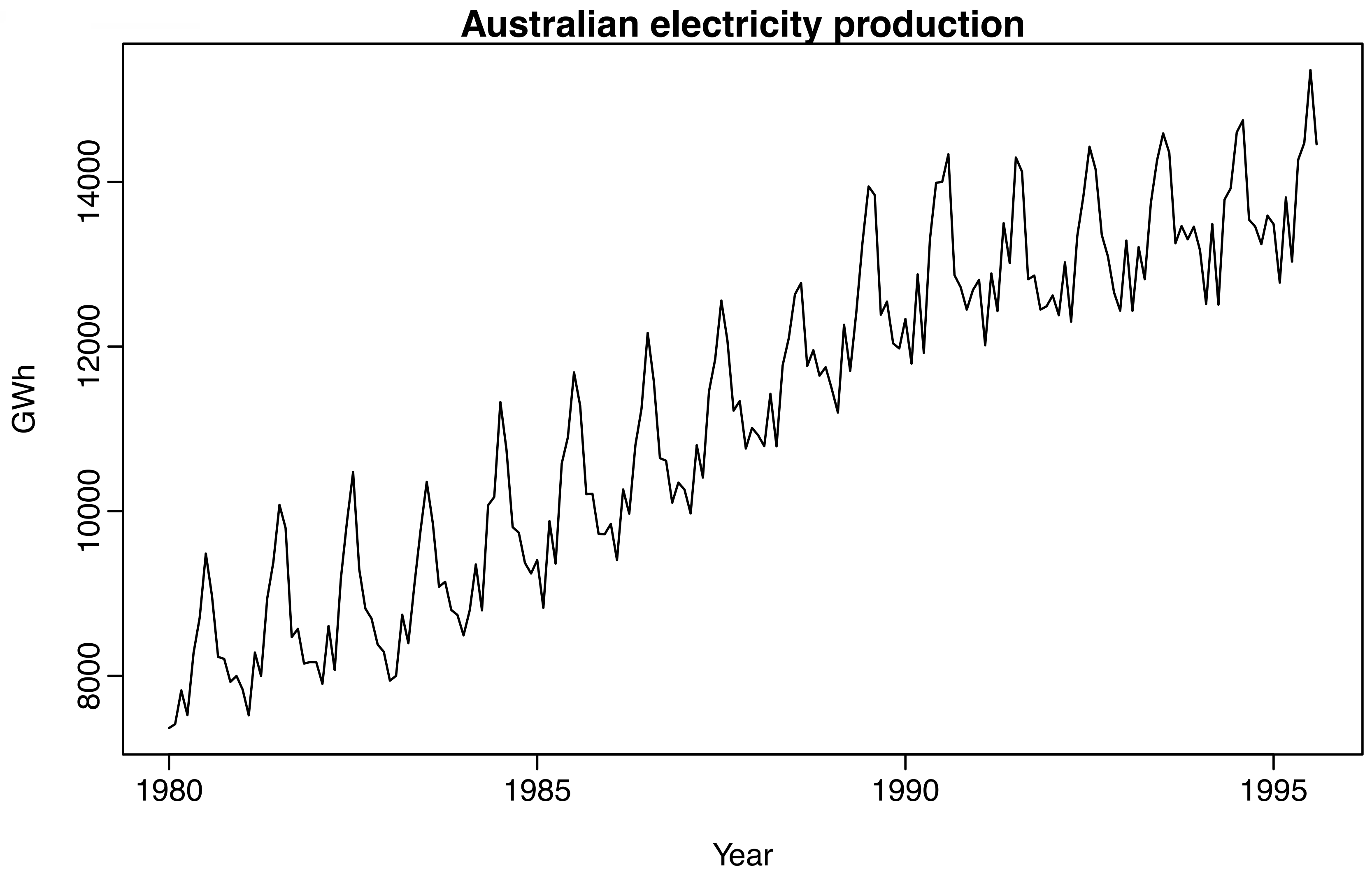


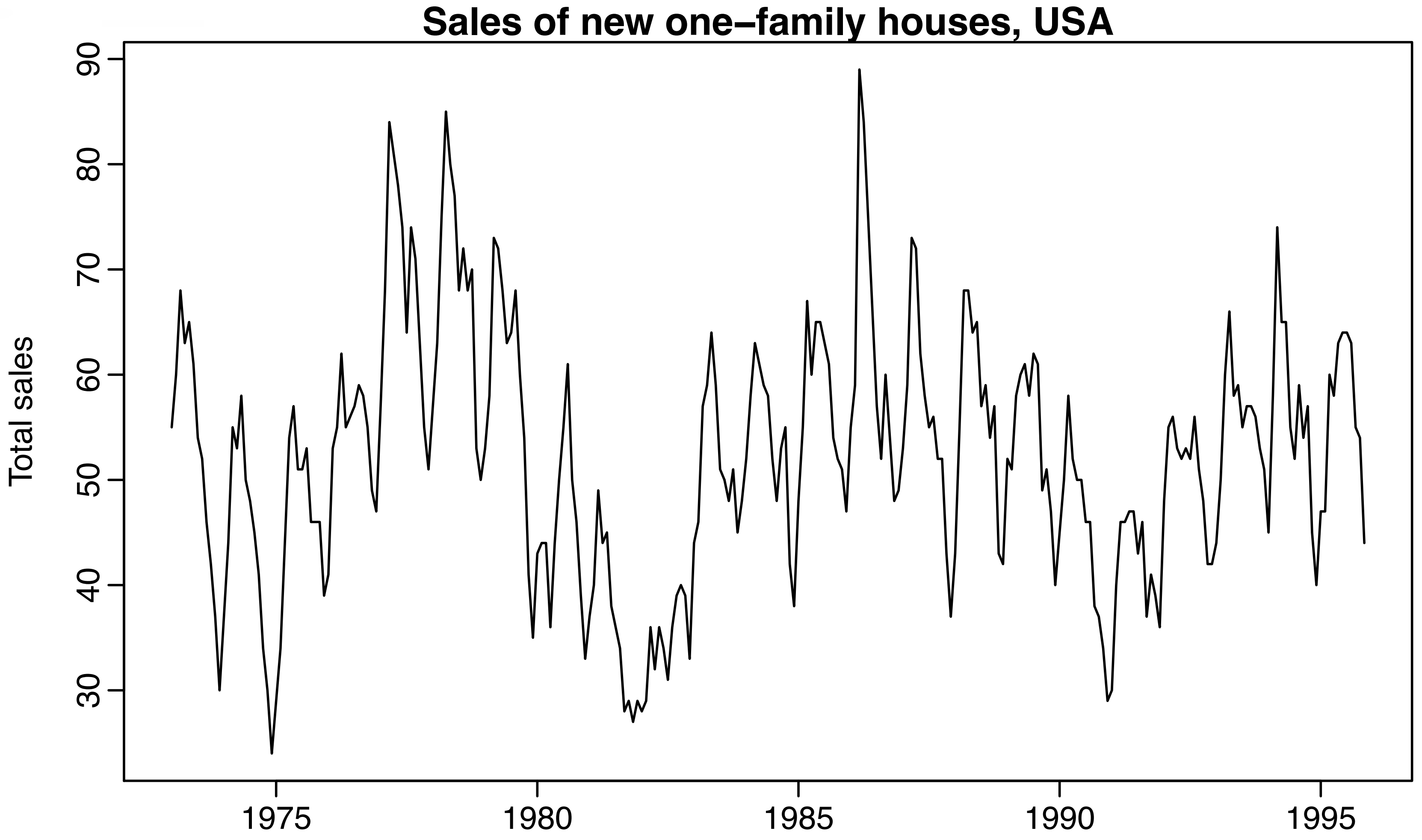
[R. J. Hyndman]

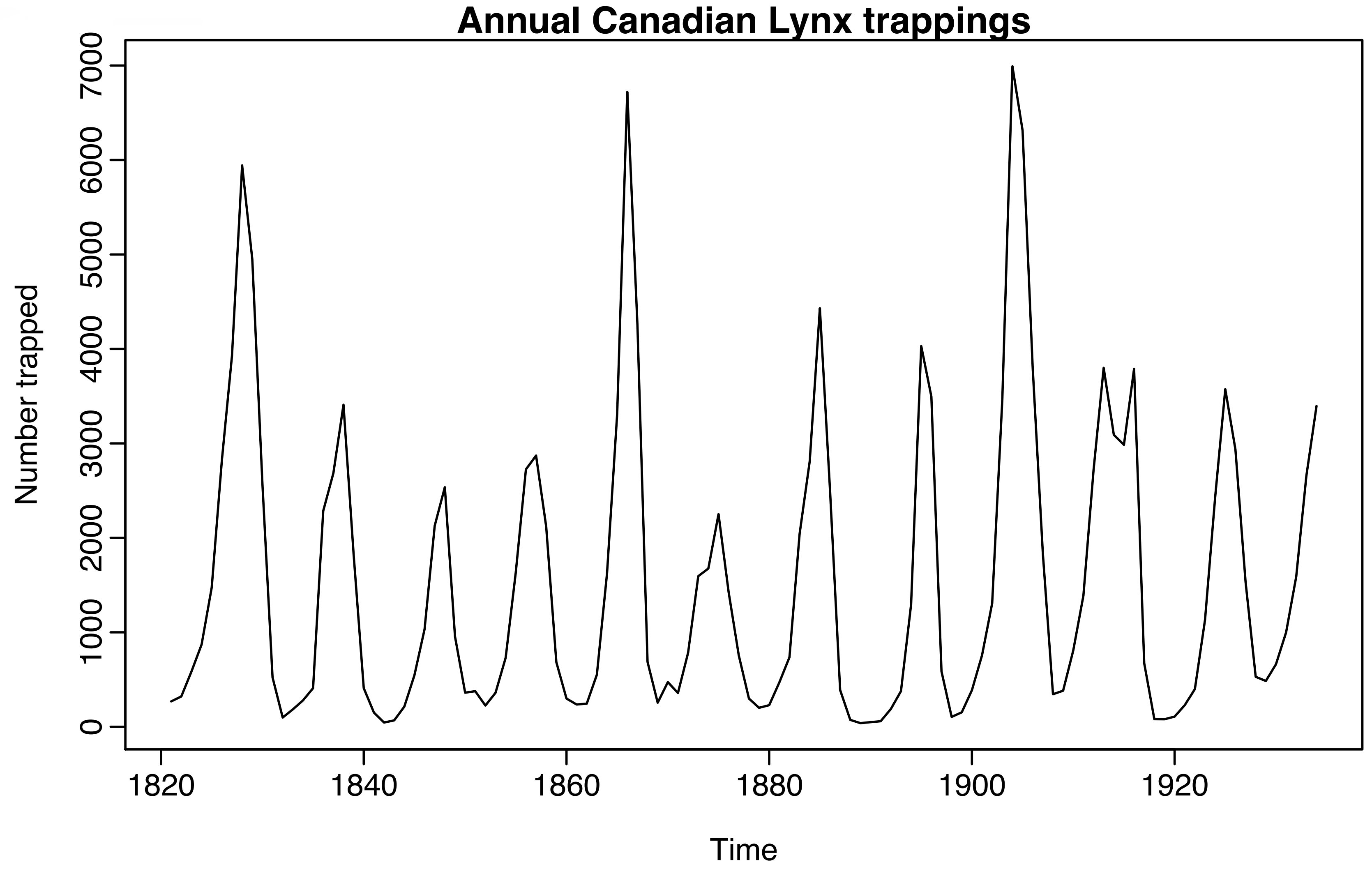
Examples

Trend





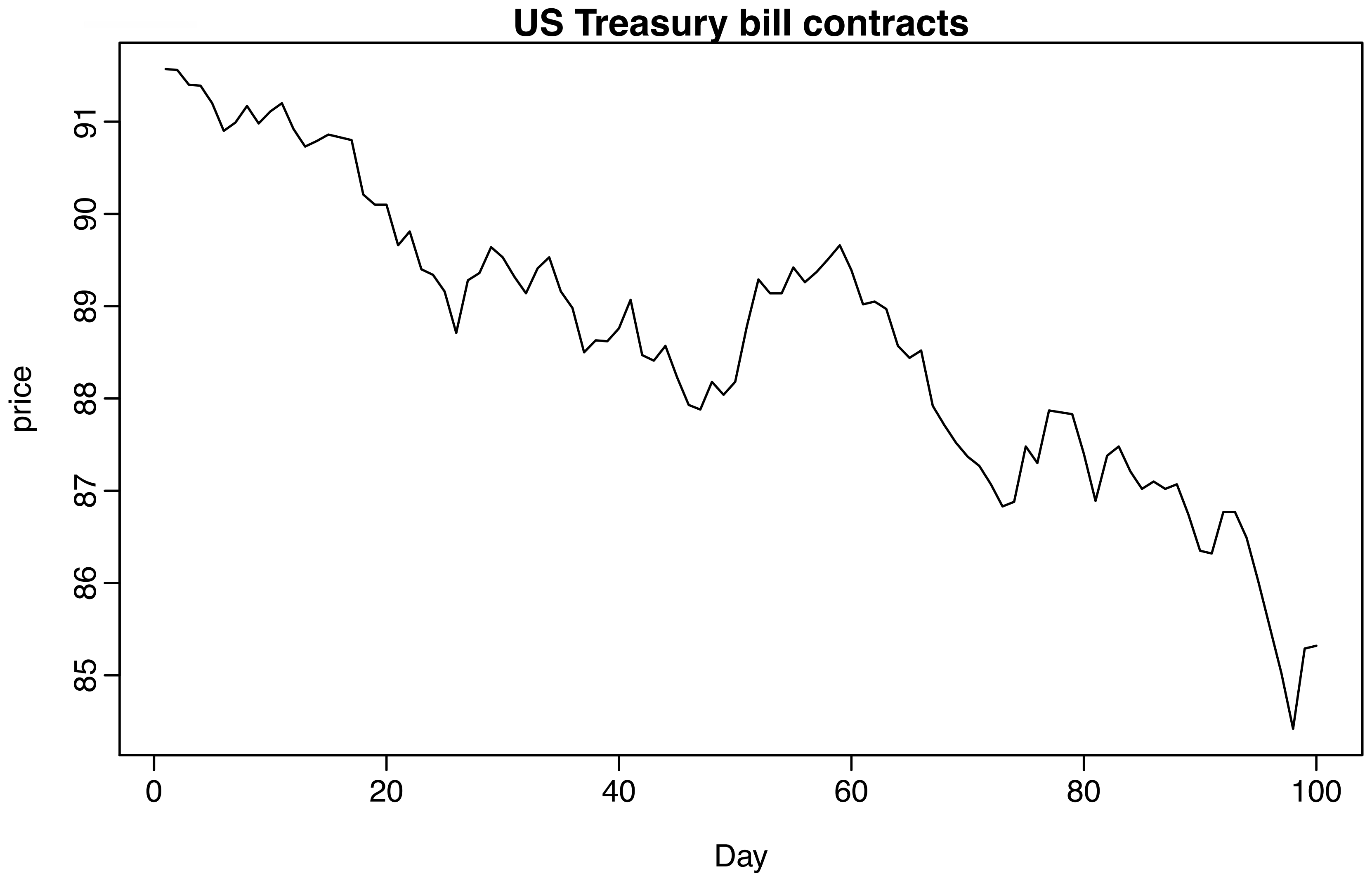


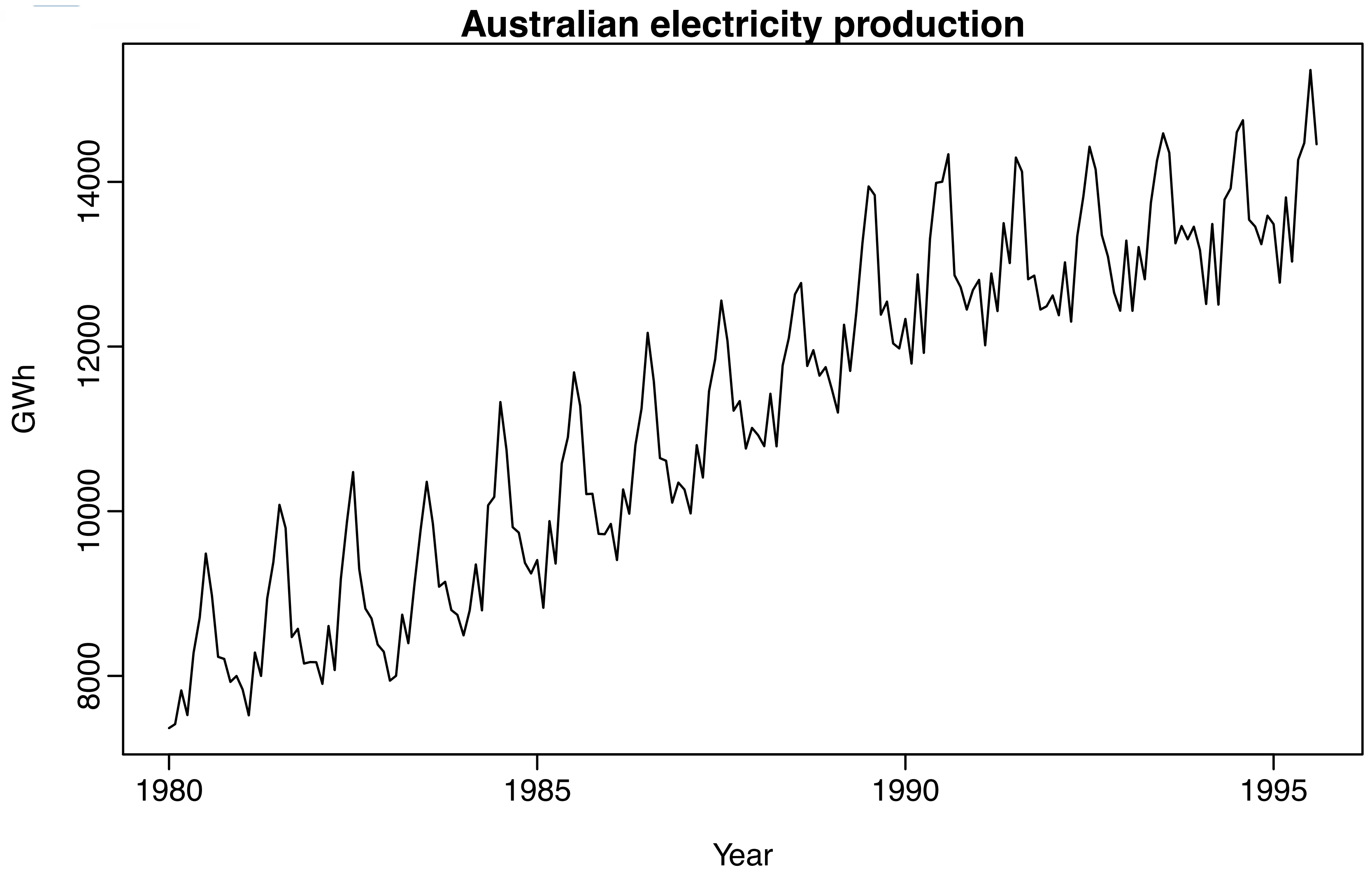


[R. J. Hyndman]

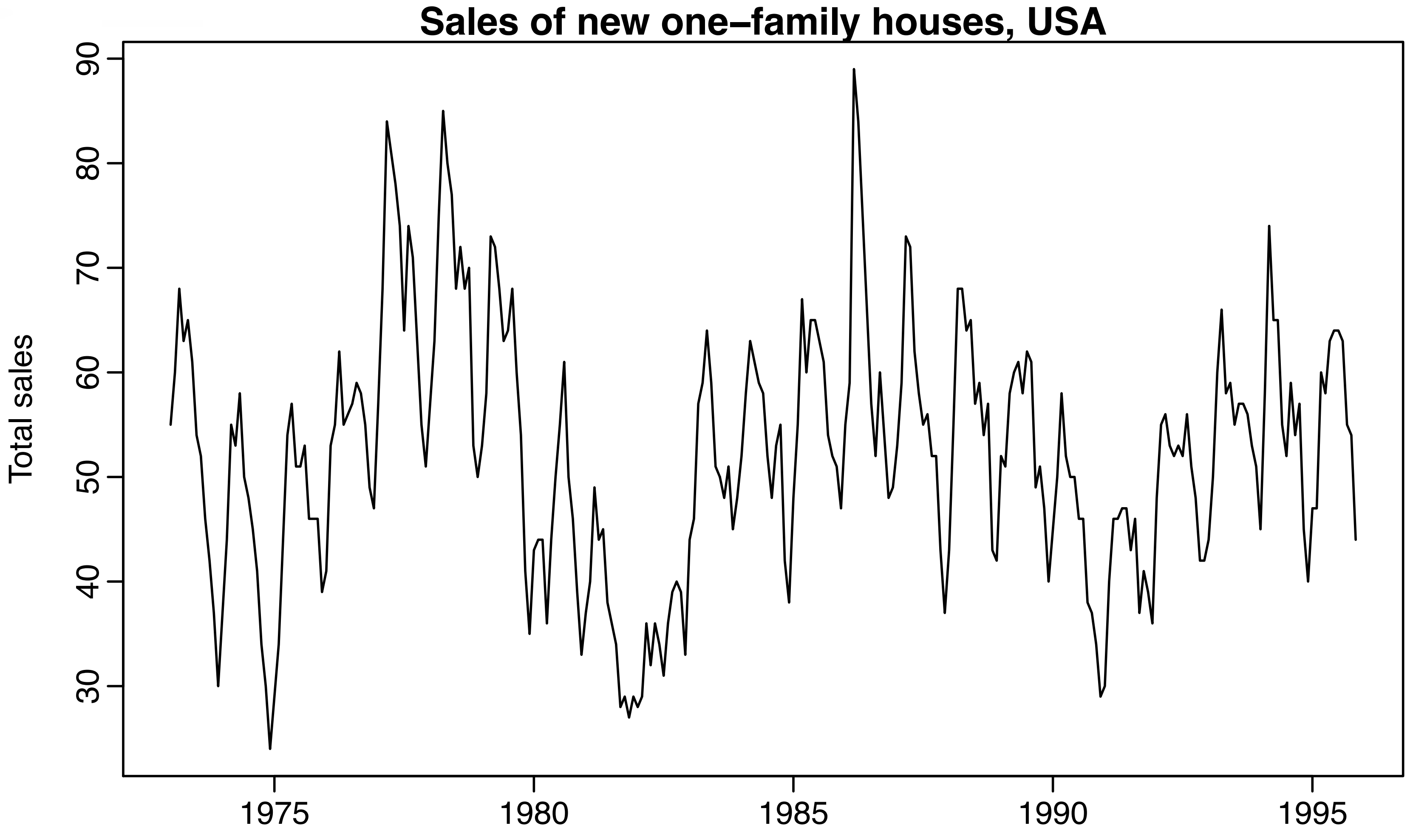
Examples

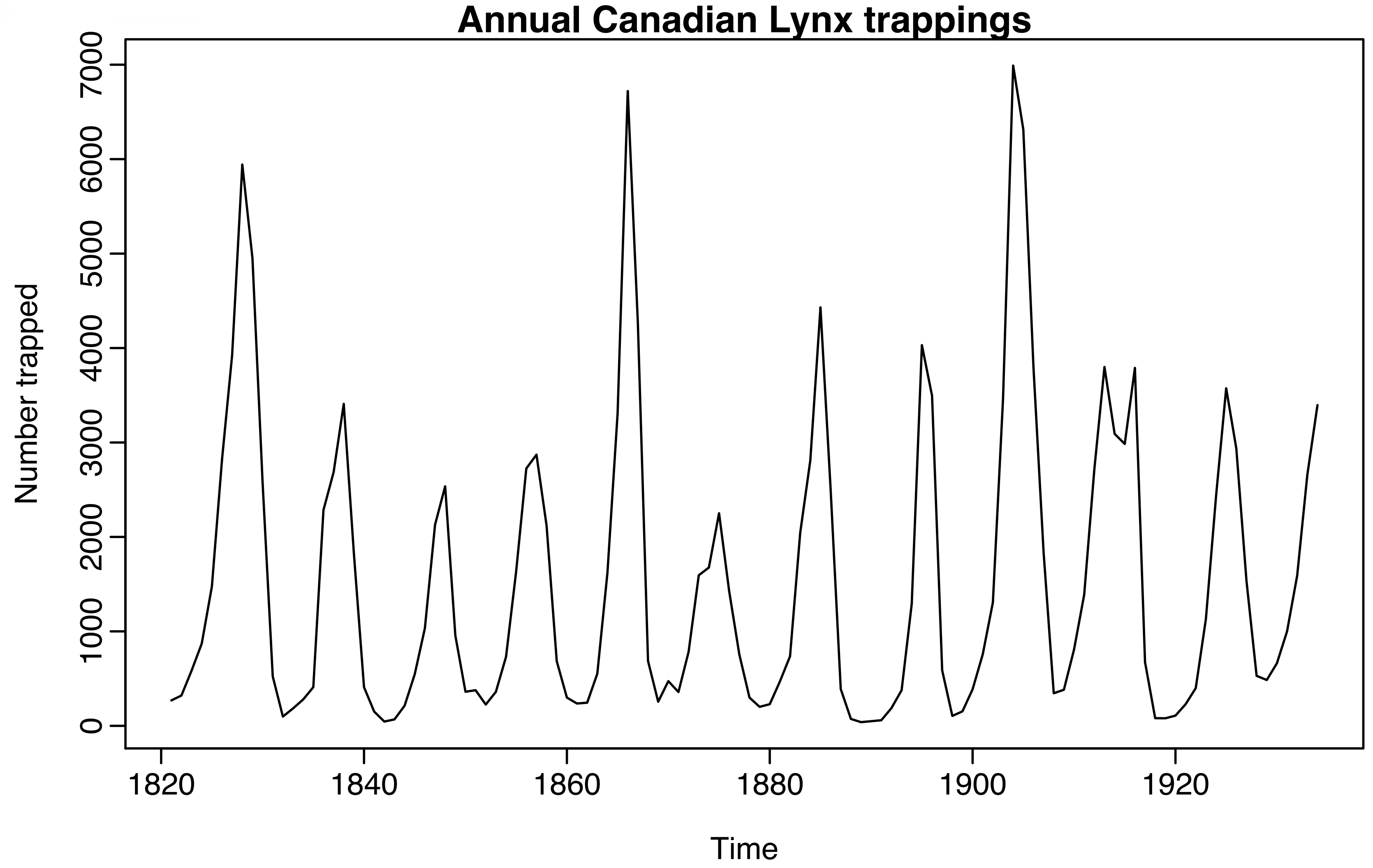
Trend





Trend + Seasonality

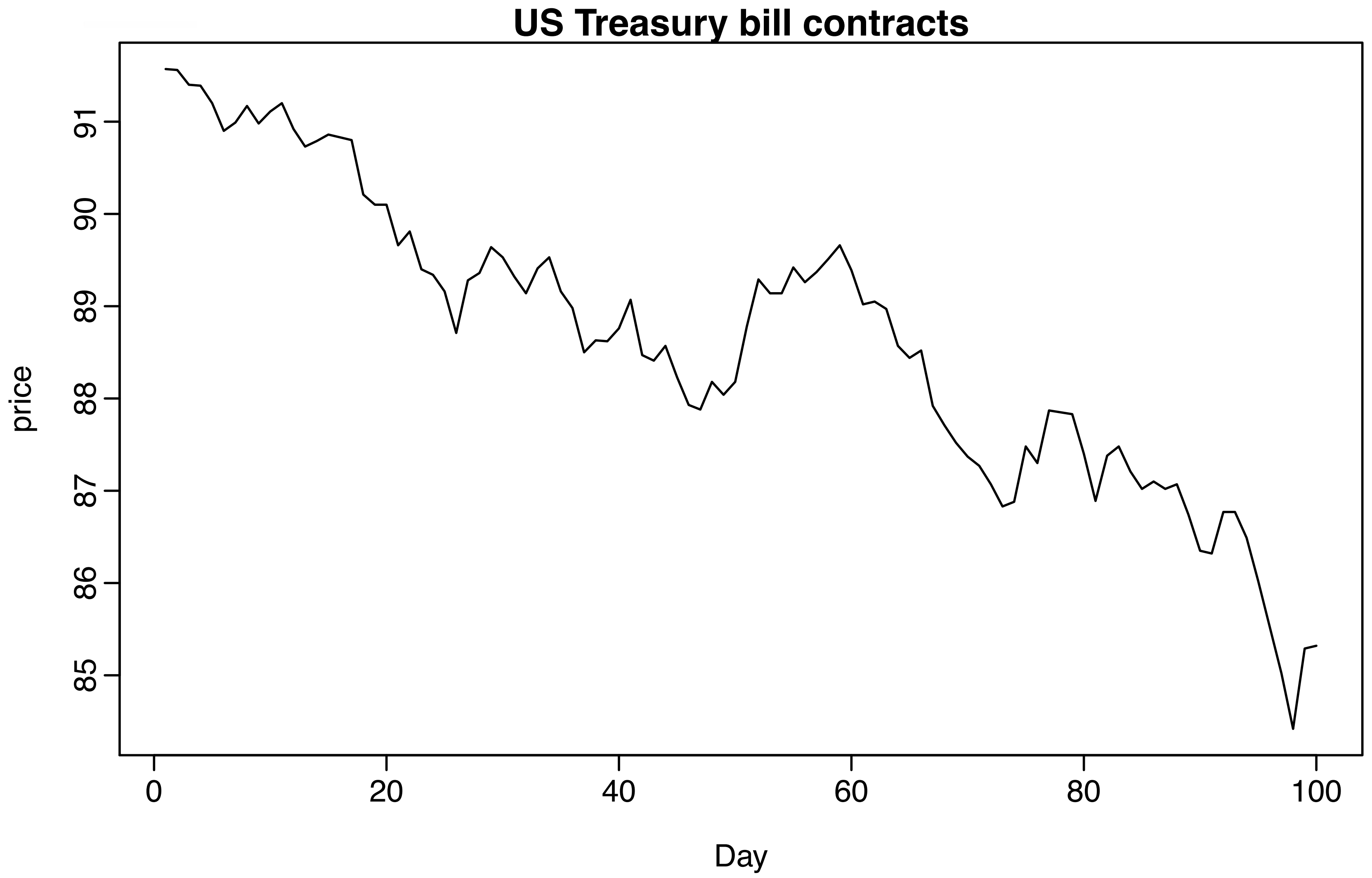


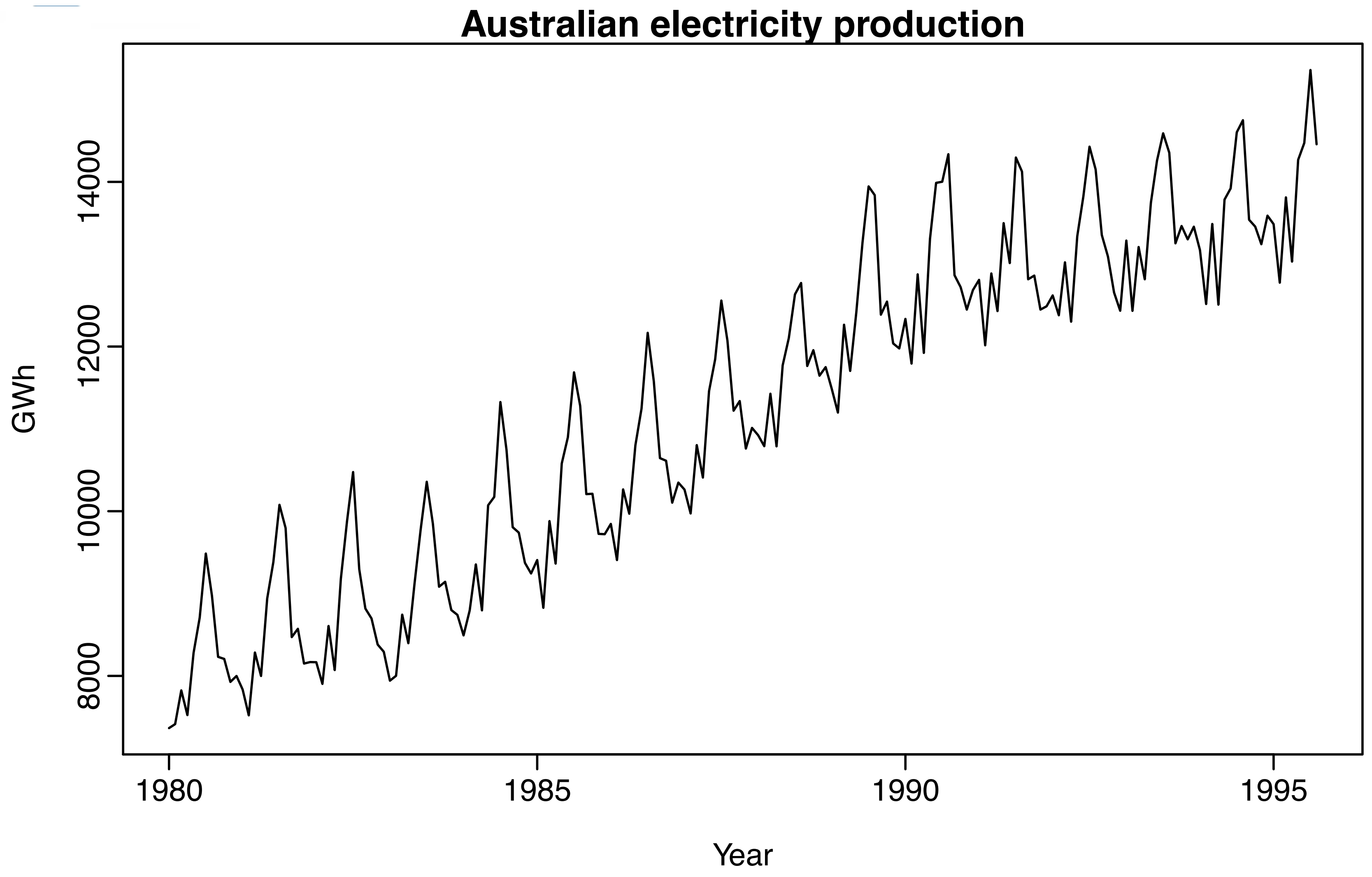


[R. J. Hyndman]

Examples

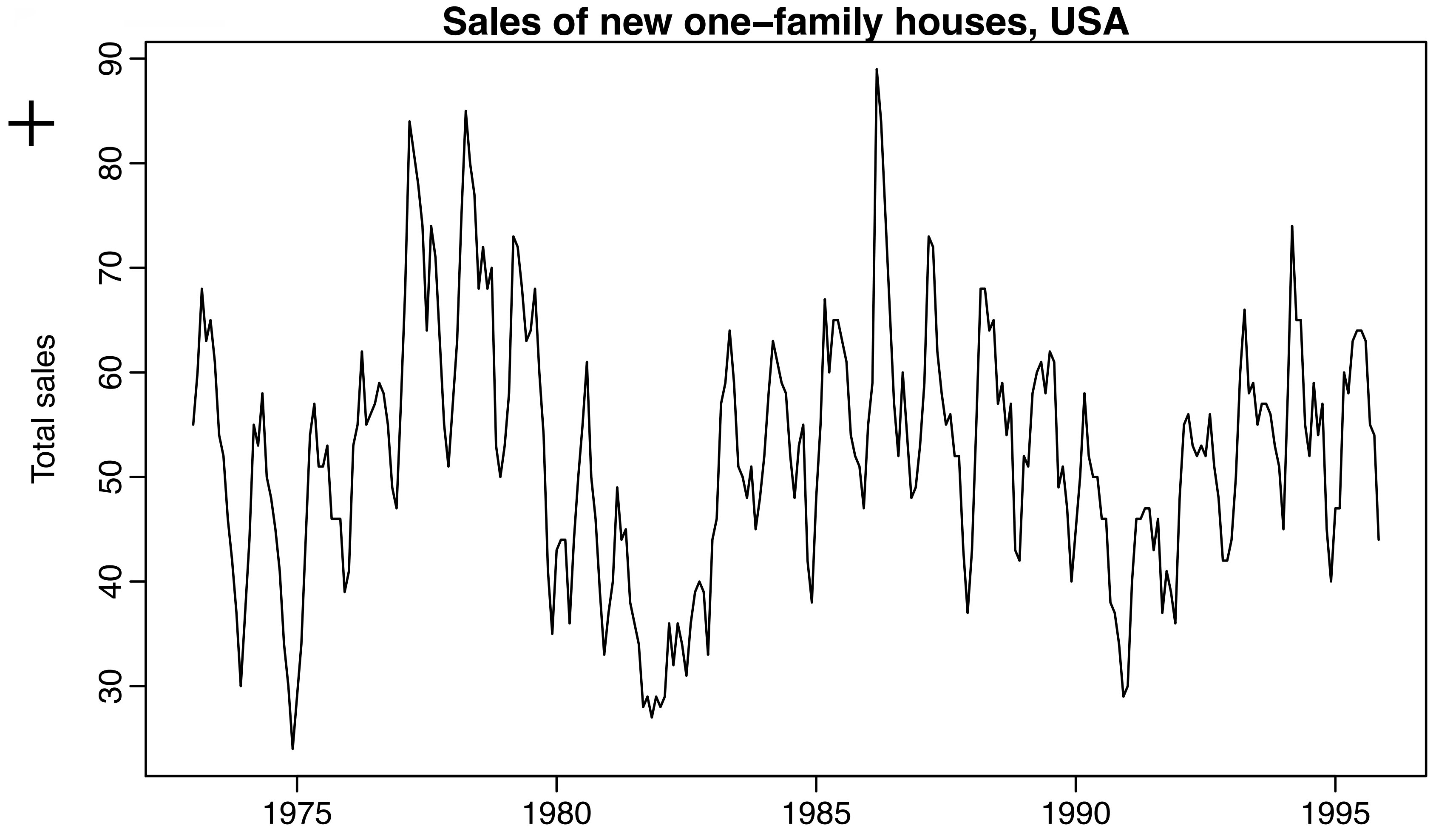
Trend

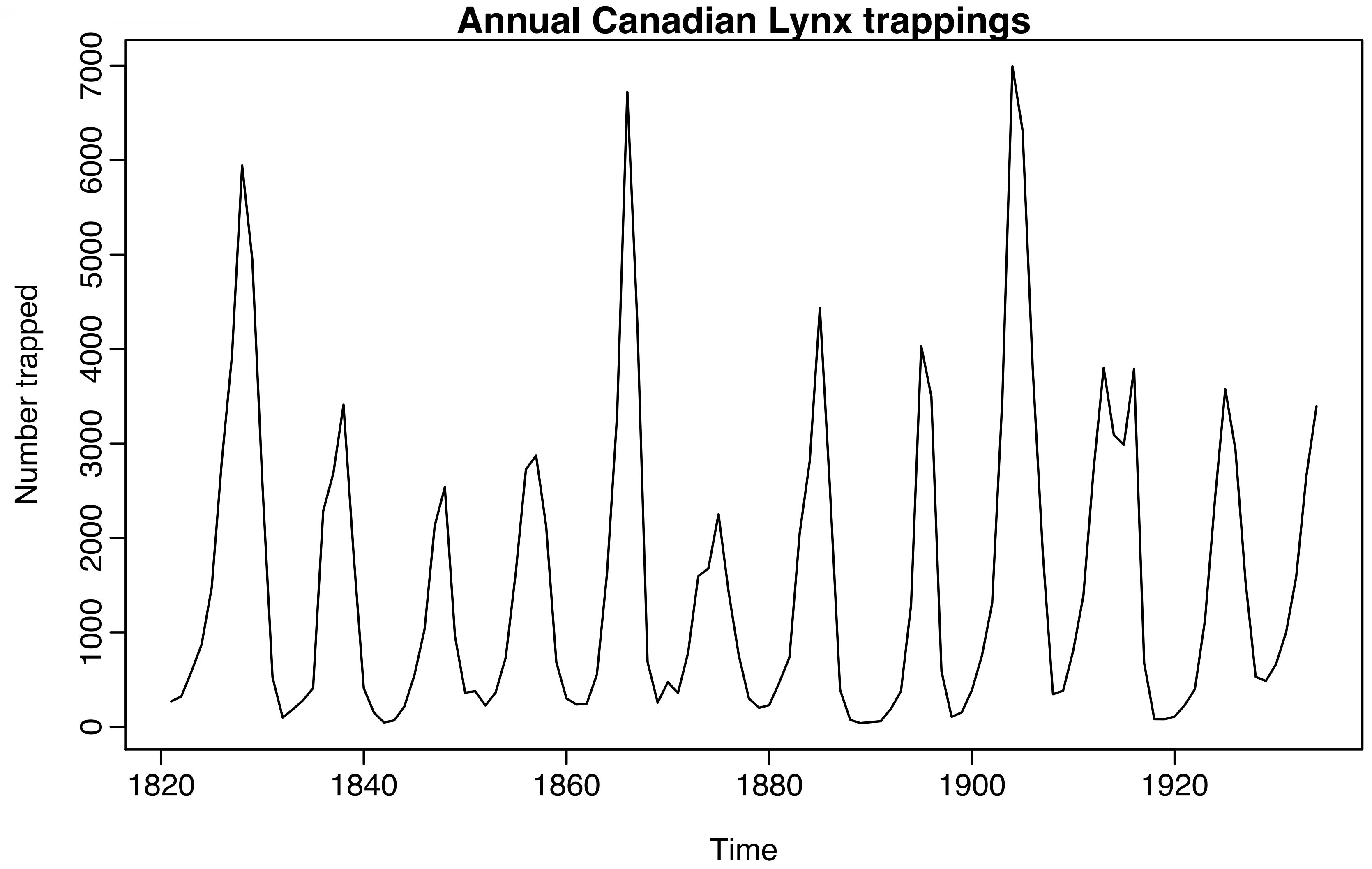




Trend + Seasonality

Seasonality + Cyclic

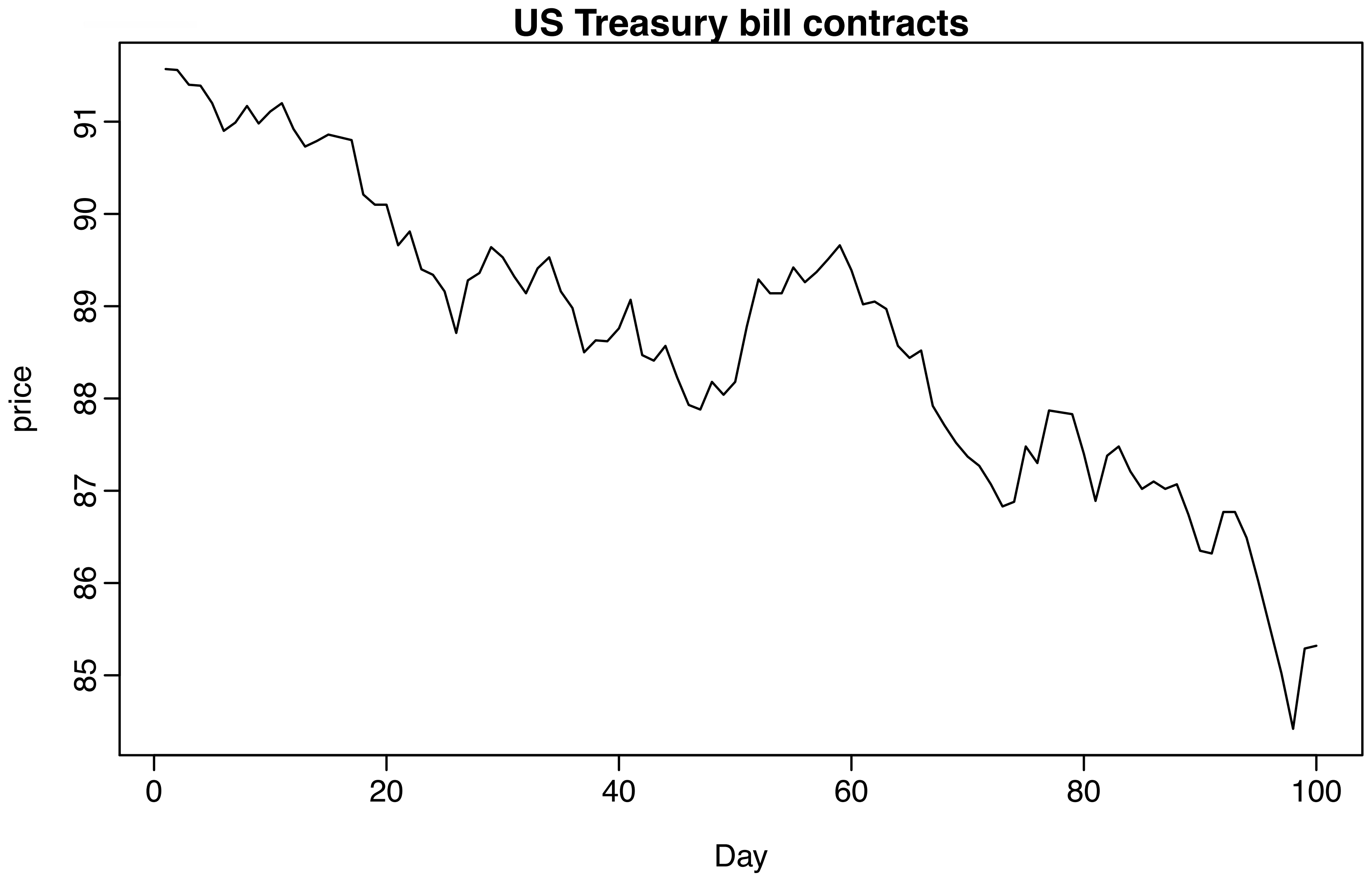


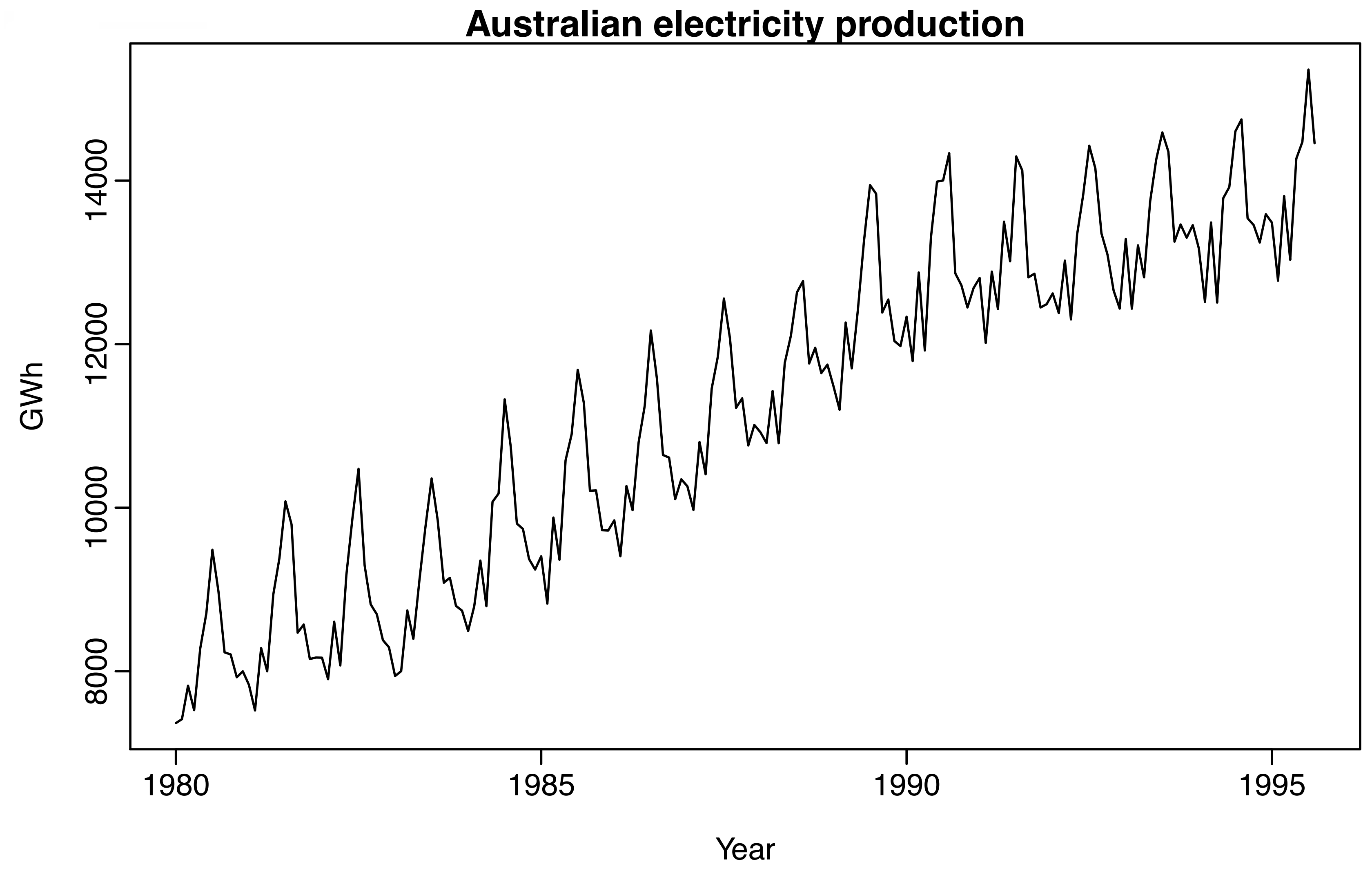


[R. J. Hyndman]

Examples

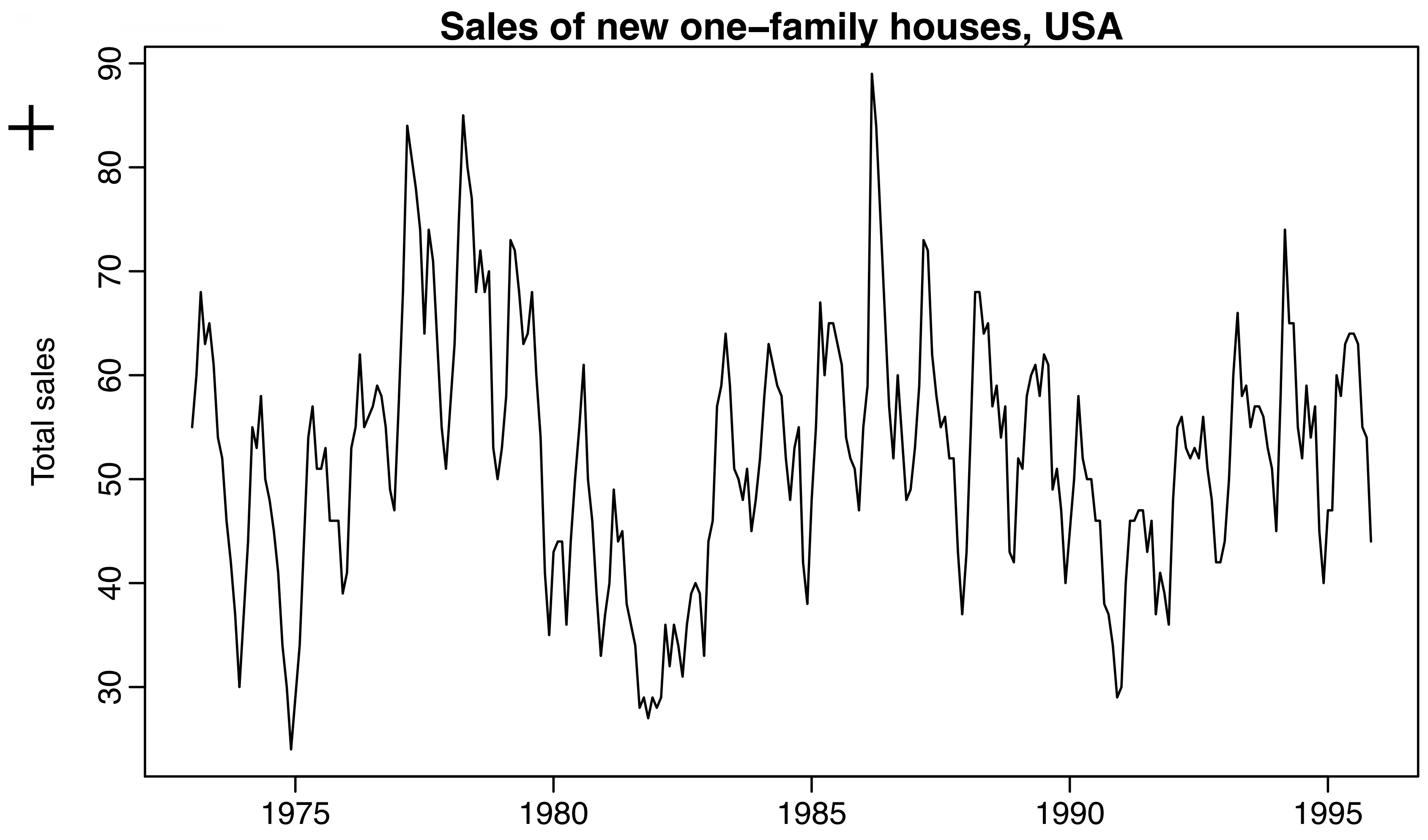
Trend

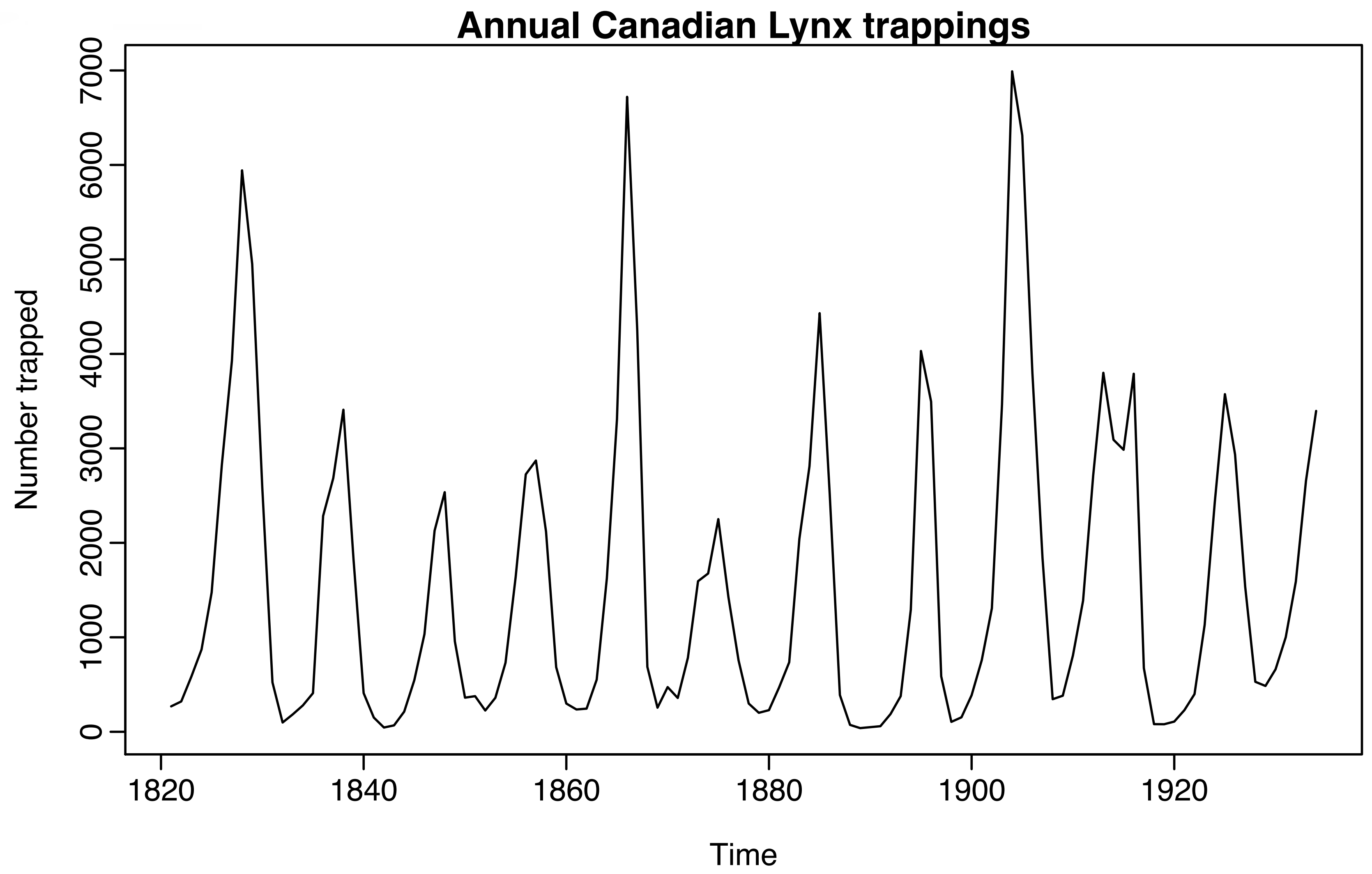




Trend + Seasonality

Seasonality + Cyclic





Stationary

[R. J. Hyndman]

Types of Time Data

- Timestamps: specific instants in time (e.g. `2018-11-27 14:15:00`)
- Periods: have a standard start and length (e.g. the month `November 2018`)
- Intervals: have a start and end timestamp
  - Periods are special case
  - Example: `2018-11-21 14:15:00` — `2018-12-01 05:15:00`
- Elapsed time: measure of time relative to a start time (`15 minutes`)

Dates and Times

- What is time to a computer?
  - Can be stored as seconds since Unix Epoch (January 1st, 1970)
- Often useful to break down into minutes, hours, days, months, years…
- Lots of different ways to write time:
  - How could you write `"November 29, 2016"`?
  - European vs. American ordering…
- What about time zones?

Python Support for Time

- The `datetime` package
  - Has date, time, and datetime classes
  - `.now()` method: the current datetime
  - Can access properties of the time (year, month, seconds, etc.)
- Converting from strings to datetimes:
  - `datetime.strptime`: good for known formats
  - `dateutil.parser.parse`: good for unknown formats
- Converting to strings
  - `str(dt)` or `dt.strftime(<format>)`

Datetime format specification

- Look it up:
  - http://strftime.org
- Generally, can create whatever format you need using these format strings

Code	Meaning	Example
%a	Weekday as locale's abbreviated name.	Mon
%A	Weekday as locale's full name.	Monday
%w	Weekday as a decimal number, where 0 is Sunday and 6 is Saturday.	1
%d	Day of the month as a zero-padded decimal number.	30
%-d	Day of the month as a decimal number. (Platform specific)	30
%b	Month as locale's abbreviated name.	Sep
%B	Month as locale's full name.	September
%m	Month as a zero-padded decimal number.	09
%-m	Month as a decimal number. (Platform specific)	9
%y	Year without century as a zero-padded decimal number.	13
%Y	Year with century as a decimal number.	2013
%H	Hour (24-hour clock) as a zero-padded decimal number.	07
%-H	Hour (24-hour clock) as a decimal number. (Platform specific)	7
%I	Hour (12-hour clock) as a zero-padded decimal number.	07
%-I	Hour (12-hour clock) as a decimal number. (Platform specific)	7
%p	Locale's equivalent of either AM or PM.	AM
%M	Minute as a zero-padded decimal number.	06
%-M	Minute as a decimal number. (Platform specific)	6
%S	Second as a zero-padded decimal number.	05
%-S	Second as a decimal number. (Platform specific)	5

Pandas Support for Datetime

- `pd.to_datetime`:
  - convenience method
  - can convert an entire column to datetime
- Has a `NaT` to indicate a missing time value
- Stores in a `numpy.datetime64` format
- `pd.Timestamp`: a wrapper for the `datetime64` objects

More Pandas Support

- Accessing a particular time or checking equivalence allows any string that can be interpreted as a date:
  - `ts['1/10/2011']` or `ts['20110110']`
- Date ranges: `pd.date_range('4/1/2012','6/1/2012',freq='4h')`
- Slicing works as expected
- Can do operations (add, subtract) on data indexed by datetime and the indexes will match up
- As with strings, to treat a column as datetime, you can use the `.dt` accessor

Generating Date Ranges

- `index = pd.date_range('4/1/2012', '6/1/2012')`
- Can generate based on a number of periods as well
  - `index = pd.date_range('4/1/2012', periods=20)`
- Frequency (`freq`) controls how the range is divided
  - Codes for specifying this (e.g. `4h`, `D`, `M`)
  - ```
    In [90]: pd.date_range('1/1/2000', '1/3/2000 23:59', freq='4h')
    Out[90]:
    <class 'pandas.tseries.index.DatetimeIndex'>
    [2000-01-01 00:00:00, ..., 2000-01-03 20:00:00]
    Length: 18, Freq: 4H, Timezone: None
    ```
  - Can also mix them: `'2h30m'`
    ```

Time Series Frequencies

Alias	Offset Type	Description
D	Day	Calendar daily
B	BusinessDay	Business daily
H	Hour	Hourly
T or min	Minute	Minutely
S	Second	Secondly
L or ms	Milli	Millisecond (1/1000th of 1 second)
U	Micro	Microsecond (1/1000000th of 1 second)
M	MonthEnd	Last calendar day of month
BM	BusinessMonthEnd	Last business day (weekday) of month
MS	MonthBegin	First calendar day of month
BMS	BusinessMonthBegin	First weekday of month
`W-MON, W-TUE, ...`	Week	Weekly on given day of week: MON, TUE, WED, THU, FRI, SAT, or SUN.
`WOM-1MON, WOM-2MON, ...`	WeekOfMonth	Generate weekly dates in the first, second, third, or fourth week of the month. For example, `WOM-3FRI` for the 3rd Friday of each month.

[W. McKinney, Python for Data Analysis]

DatetimeIndex

- Can use time as an **index**
- ```
  data = [('2017-11-30', 48),
          ('2017-12-02', 45),
          ('2017-12-03', 44),
          ('2017-12-04', 48)]
  dates, temps = zip(*data)
  s = pd.Series(temps, pd.to_datetime(dates))
  ```
- Accessing a particular time or checking equivalence allows any string that can be interpreted as a date:
  - `s['12/04/2017']` or `s['20171204']`
- Using a less specific string will get all matching data:
  - `s['2017-12']` returns the three December entries
  ```

DatetimeIndex

- Time slices do not need to exist:
  - `s['2017-12-01':'2017-12-31']`

Shifting Data

- Leading or Lagging Data

```
In [95]: ts = Series(np.random.randn(4),
   ....:             index=pd.date_range('1/1/2000', periods=4, freq='M'))
```

```
In [96]: ts
Out[96]:
2000-01-31   -0.066748
2000-02-29    0.838639
2000-03-31   -0.117388
2000-04-30   -0.517795
Freq: M, dtype: float64
```

```
In [97]: ts.shift(2)
Out[97]:
2000-01-31         NaN
2000-02-29         NaN
2000-03-31   -0.066748
2000-04-30    0.838639
Freq: M, dtype: float64
```

```
In [98]: ts.shift(-2)
Out[98]:
2000-01-31   -0.117388
2000-02-29   -0.517795
2000-03-31         NaN
2000-04-30         NaN
Freq: M, dtype: float64
```

- Shifting by time:

```
In [99]: ts.shift(2, freq='M')
Out[99]:
2000-03-31   -0.066748
2000-04-30    0.838639
2000-05-31   -0.117388
2000-06-30   -0.517795
Freq: M, dtype: float64
```

Shifting Time Series

- Data:

```
[('2017-11-30', 48), ('2017-12-02', 45),
 ('2017-12-03', 44), ('2017-12-04', 48)]
```

- Compute day-to-day difference in high temperature:
  - `s - s.shift(1)` (same as `s.diff()`)
  - ```
    2017-11-30    NaN
    2017-12-02   -3.0
    2017-12-03   -1.0
    2017-12-04    4.0
    ```
  - `s - s.shift(1, 'd')`
  - ```
    2017-11-30    NaN
    2017-12-01    NaN
    2017-12-02    NaN
    2017-12-03   -1.0
    2017-12-04    4.0
    2017-12-05    NaN
    ```

Timedelta

- Compute differences between dates
- Lives in `datetime` module
- `diff = parse_date("1 Jan 2017") - datetime.now().date()`
  `diff.days`
- Also a `pd.Timedelta` object that take strings:
  - `datetime.now().date() + pd.Timedelta("4 days")`
- Also, Roll dates using anchored offsets

```
from pandas.tseries.offsets import Day, MonthEnd

now = datetime(2011, 11, 17)
In [107]: now + MonthEnd(2)
Out[107]: Timestamp('2011-12-31 00:00:00')
```