# 10-605

William Cohen

# Summary to date

- Computational complexity: what and how to count
  - Memory *vs* disk access
  - Cost of scanning *vs* seeks for disk (and memory)
- Probability review
  - Classification with a "density estimator"
  - Naïve Bayes as a density estimator/classifier
- How to implement Naïve Bayes
  - Time is linear in size of data (one scan!)
  - Assuming the event counters fit in memory
  - We need to count

  C(*Y=label*), C( *X=word* ^ *Y=label*),…

# Naïve Bayes: Counts in Memory

- You have a *train* dataset and a *test* dataset
- Initialize an "event counter" (hashtable) C
- For each example *id, y,* $x_1,\ldots,x_d$ in *train:*
  - C("Y=ANY") ++; C("Y=*y*") ++
  - For *j* in *1..d*:
    - C("Y=*y* ^ X=$x_j$") ++
    - C("Y=*y* ^ X=*ANY*") ++
- For each example *id, y,* $x_1,\ldots,x_d$ in *test:*
  - For each *y'* in *dom(Y):*
    - Compute log Pr(*y'*,$x_1,\ldots,x_d$) =

where:
$q_x = 1/|V|$
$q_y = 1/|dom(Y)|$
$m=1$

$$= \left( \sum_j \log \frac{C(X = x_j \wedge Y = y') + mq_x}{C(X = ANY \wedge Y = y') + m} \right) + \log \frac{C(Y = y') + mq_y}{C(Y = ANY) + m}$$

  - Return the best *y'*

# SCALING TO LARGE VOCABULARIES: WHY?

Assume hashtable holding all counts fits in memory

# Complexity of Naïve Bayes

- You have a *train* dataset and a *test* dataset
- Initialize an "event counter" (hashtable) C
- For each example *id, y,* $x_1,\ldots,x_d$ in *train:*
  - C("Y=ANY") ++; C("Y=*y*") ++
  - For *j* in *1..d*:
    - C("Y=*y* ^ X=$x_j$") ++
    - …
- For each example *id, y,* $x_1,\ldots,x_d$ in *test:*
  - For each *y′* in *dom(Y):*
    - Compute log Pr(*y′*,$x_1,\ldots,x_d$) =

$$= \left( \sum_j \log \frac{C(X = x_j \wedge Y = y') + mq_x}{C(X = ANY \wedge Y = y') + m} \right) + \log \frac{C(Y = y') + mq_y}{C(Y = ANY) + m}$$

  - Return the best *y′*

Sequential reads

Complexity: O(*n*), *n*=size of *train*

where:
$q_x = 1/|V|$
$q_y = 1/|dom(Y)|$
$mq_x = 1$

Sequential reads

Complexity: O(|*dom(Y)*|**n′*), *n′*=size of *test*

# The Naïve Bayes classifier – v1

- Dataset: each example has
  - A unique id *id*
    - Why? For debugging the feature extractor
  - *d* attributes $X_1,\ldots,X_d$
    - Each $X_i$ takes a discrete value in $dom(X_i)$
  - One class label $Y$ in $dom(Y)$
- You have a *train* dataset and a *test* dataset
- Assume:
  - the dataset doesn't fit in memory
  - the model doesn't either

# What's next

- How to implement Naïve Bayes
  - Assuming the event counters do *not* fit in memory
- Why?

**Micro:**
0.5G memory
**Standard**:
S: 2Gb
XL: 8Gb
10xlarge:
160Gb
x1.32xlarge:
2Tb, 128 cores

$0.00652/hr

$0.03/hr

$0.104/hr

$2.34/hr

$13.33/hr

Region: US East (N. Virginia)

| | Linux/UNIX Usage | Windows Usage |
|---|---|---|
| **Standard On-Demand Instances** | | |
| Small (Default) | $0.065 per Hour | $0.115 per Hour |
| Medium | $0.130 per Hour | $0.230 per Hour |
| Large | $0.260 per Hour | $0.460 per Hour |
| Extra Large | $0.520 per Hour | $0.920 per Hour |
| **Second Generation Standard On-Demand Instances** | | |
| Extra Large | $0.580 per Hour | $0.980 per Hour |
| Double Extra Large | $1.160 per Hour | $1.960 per Hour |
| **Micro On-Demand Instances** | | |
| Micro | $0.020 per Hour | $0.020 per Hour |
| **High-Memory On-Demand Instances** | | |
| Extra Large | $0.450 per Hour | $0.570 per Hour |
| Double Extra Large | $0.900 per Hour | $1.140 per Hour |
| Quadruple Extra Large | $1.800 per Hour | $2.280 per Hour |
| **High-CPU On-Demand Instances** | | |
| Medium | $0.165 per Hour | $0.285 per Hour |
| Extra Large | $0.660 per Hour | $1.140 per Hour |
| **Cluster Compute Instances** | | |
| Quadruple Extra Large | $1.300 per Hour | $1.610 per Hour |
| Eight Extra Large | $2.400 per Hour | $2.970 per Hour |
| **High-Memory Cluster On-Demand Instances** | | |
| Eight Extra Large | $3.500 per Hour | $3.831 per Hour |
| **Cluster GPU Instances** | | |
| Quadruple Extra Large | $2.100 per Hour | $2.600 per Hour |
| **High-I/O On-Demand Instances** | | |
| Quadruple Extra Large | $3.100 per Hour | $3.580 per Hour |
| **High-Storage On-Demand Instances** | | |
| Eight Extra Large | $4.600 per Hour | $4.931 per Hour |

# What's next

- How to implement Naïve Bayes
  - Assuming the event counters do *not* fit in memory
- Why?
  - Zipf's law: many words that you see, you don't see often.

# [Via Bruce Croft]

| Number of Occurrences (n) | Predicted Proportion of Occurrences 1/n(n+1) | Actual Proportion occurring n times $I_n/D$ | Actual Number of Words occurring n times |
|---|---|---|---|
| 1 | .500 | .402 | 204,357 |
| 2 | .167 | .132 | 67,082 |
| 3 | .083 | .069 | 35,083 |
| 4 | .050 | .046 | 23,271 |
| 5 | .033 | .032 | 16,332 |
| 6 | .024 | .024 | 12,421 |
| 7 | .018 | .019 | 9,766 |
| 8 | .014 | .016 | 8,200 |
| 9 | .011 | .014 | 6,907 |
| 10 | .009 | .012 | 5,893 |

Frequencies from 336,310 documents in the 1GB TREC Volume 3 Corpus
125,720,891 total word occurrences; 508,209 unique words

# What's next

- How to implement Naïve Bayes
  - Assuming the event counters do *not* fit in memory
- Why?
- Heaps' Law: If $V$ is the size of the vocabulary and the $n$ is the length of the corpus in words:

$$V = Kn^{\beta} \quad \text{with constants } K,\ 0 < \beta < 1$$

- Typical constants:
  - $K \approx 1/10 - 1/100$
  - $\beta \approx 0.4–0.6$ (approx. square-root)
- Why?
  - Proper names, missspellings, neologisms, …
- Summary:
  - For text classification for a corpus with O(n) words, expect to use O(sqrt(n)) storage for vocabulary.
  - Scaling might be worse for other cases (e.g., hypertext, phrases, …)

# What's next

- How to implement Naïve Bayes
  - Assuming the event counters do *not* fit in memory
- Possible approaches:
  - Use a database? (or at least a key-value store)

# Numbers (Jeff Dean says) Everyone Should Know

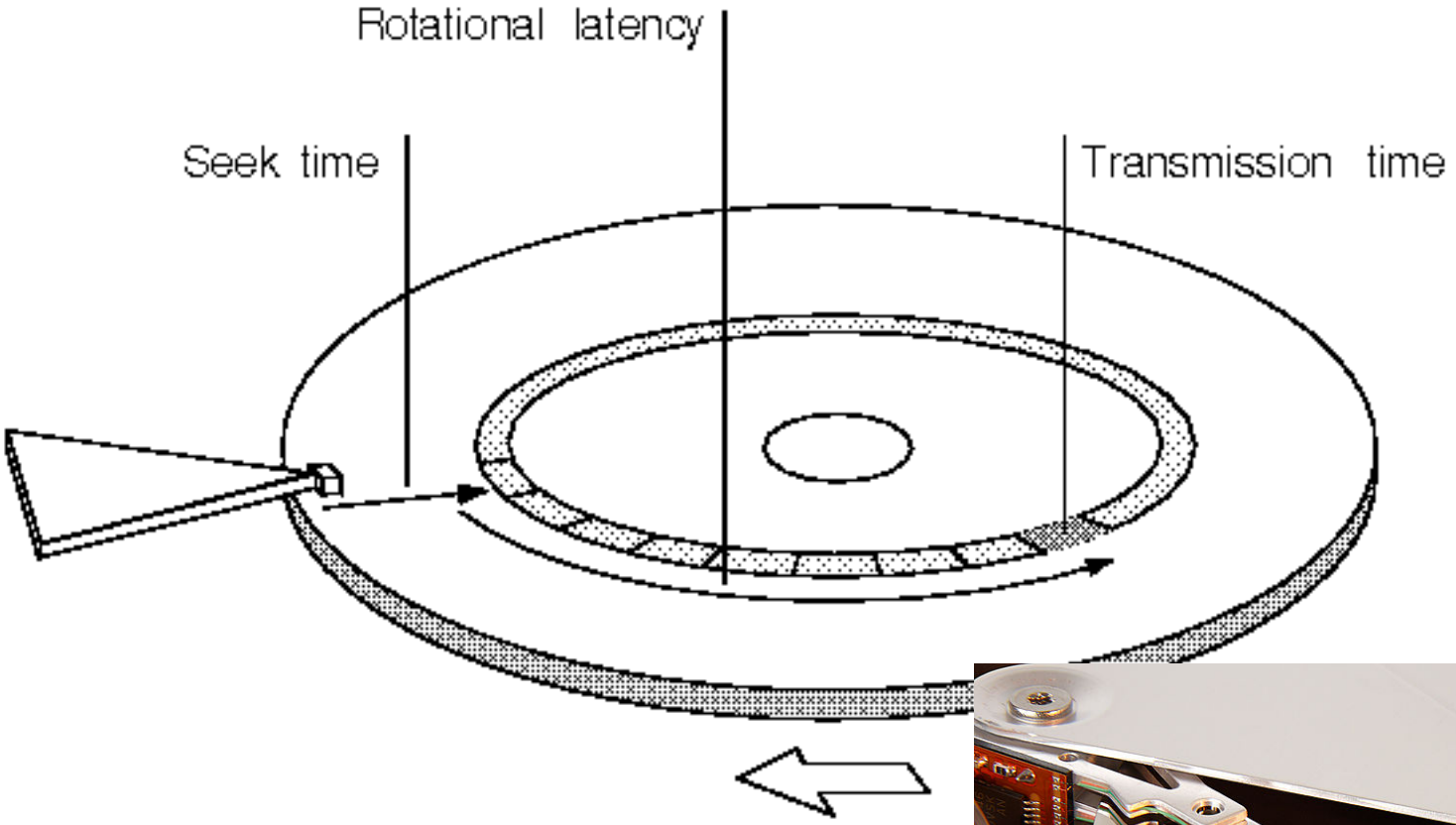
| | |
|---|---|
| L1 cache reference | 0.5 ns |
| Branch mispredict | 5 ns |
| L2 cache reference | 7 ns |
| Mutex lock/unlock | 100 ns |
| Main memory reference | 100 ns |
| Compress 1K bytes with Zippy | 10,000 ns |
| Send 2K bytes over 1 Gbps network | 20,000 ns |
| Read 1 MB sequentially from memory | 250,000 ns |
| Round trip within same datacenter | 500,000 ns |
| Disk seek | 10,000,000 ns |
| Read 1 MB sequentially from network | 10,000,000 ns |
| Read 1 MB sequentially from disk | 30,000,000 ns |
| Send packet CA->Netherlands->CA | 150,000,000 ns |

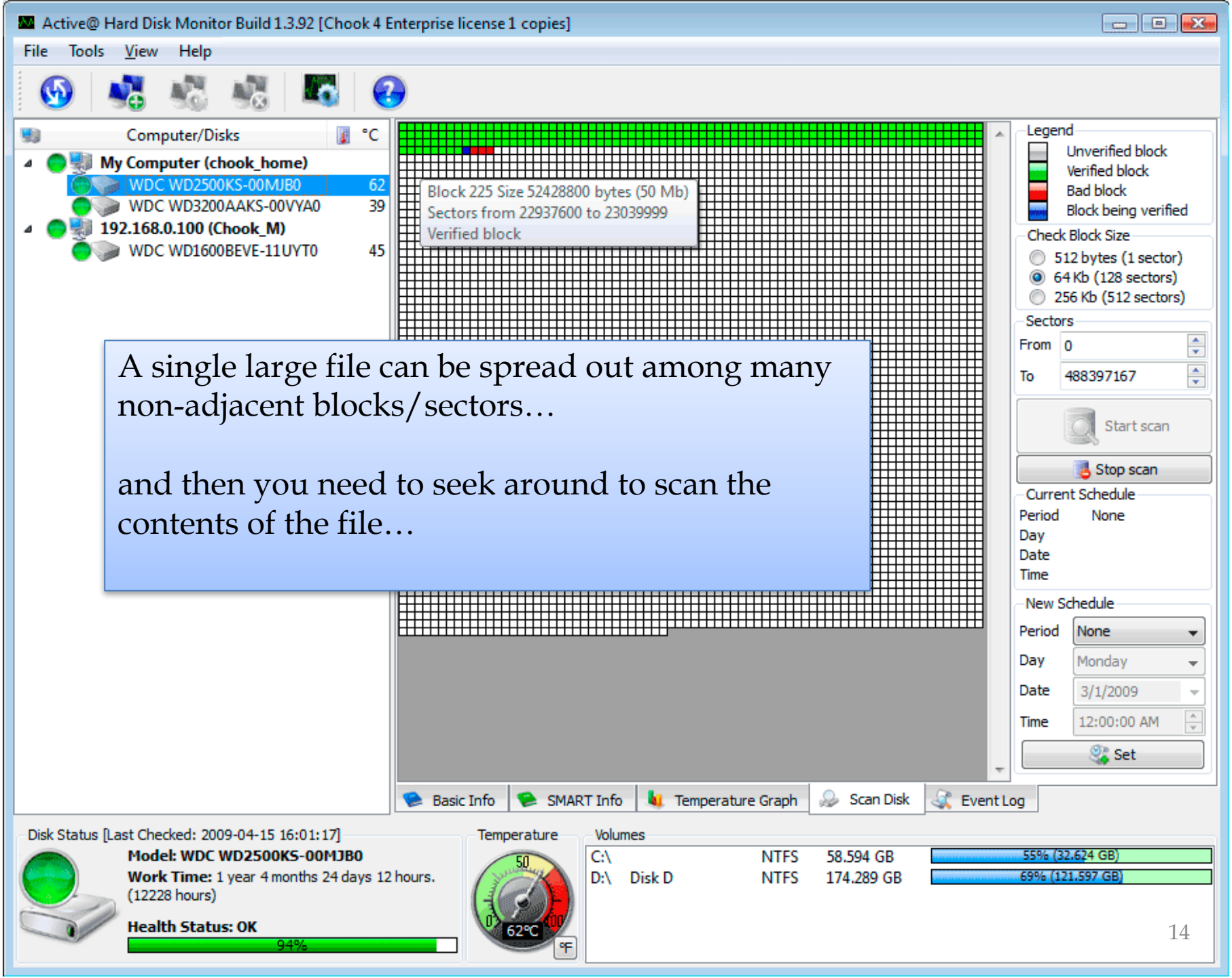
~= 10x

~= 15x

40x

~= 100,000x


Rotational latency
Seek time
Transmission time




A single large file can be spread out among many non-adjacent blocks/sectors…

and then you need to seek around to scan the contents of the file…

# What's next

- How to implement Naïve Bayes
  - Assuming the event counters do *not* fit in memory
- Possible approaches:
  - Use a database?
    - Counts are stored on disk, not in memory
    - …So, accessing a count might involve some seeks
      - Caveat: many DBs are good at caching frequently-used values, so seeks might be infrequent …..

O(n*scan) ➔ O(n*scan*seek)

# What's next

- How to implement Naïve Bayes
  - Assuming the event counters do *not* fit in memory
- Possible approaches:
  - Use a **memory-based distributed** database?
    - Counts are stored on disk, not in memory
    - …So, accessing a count might involve some seeks
      - Caveat: many DBs are good at caching frequently-used values, so seeks might be infrequent …..

O(n*scan) ➔ O(n*scan*???)

# Counting

- example 1
- example 2
- example 3
- ….

Counting logic

"increment C[*x*] by *D*"



Hash table, database, etc

# Counting

- example 1
- example 2
- example 3
- ….

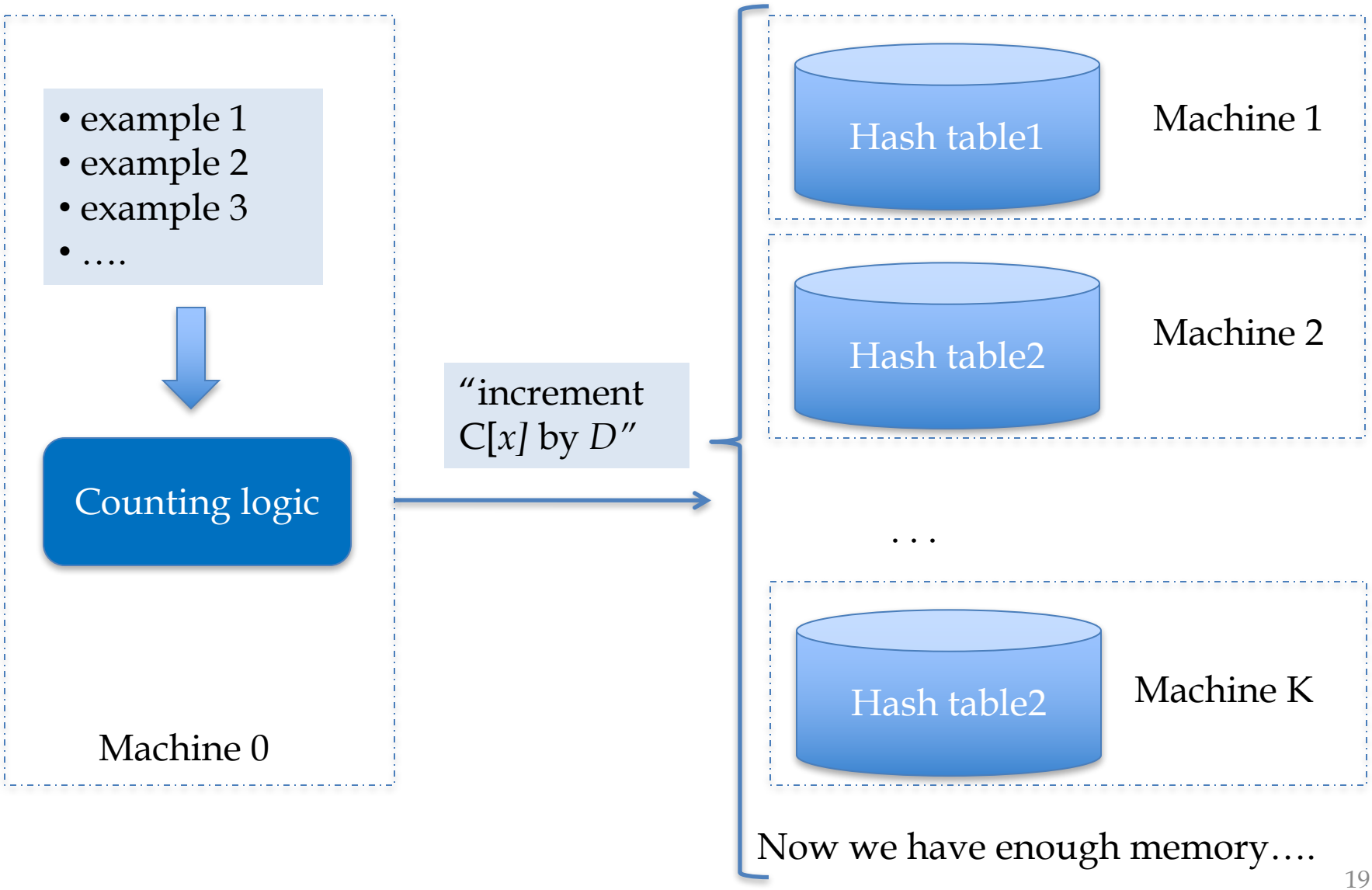
Counting logic

"increment C[*x*] by *D*"

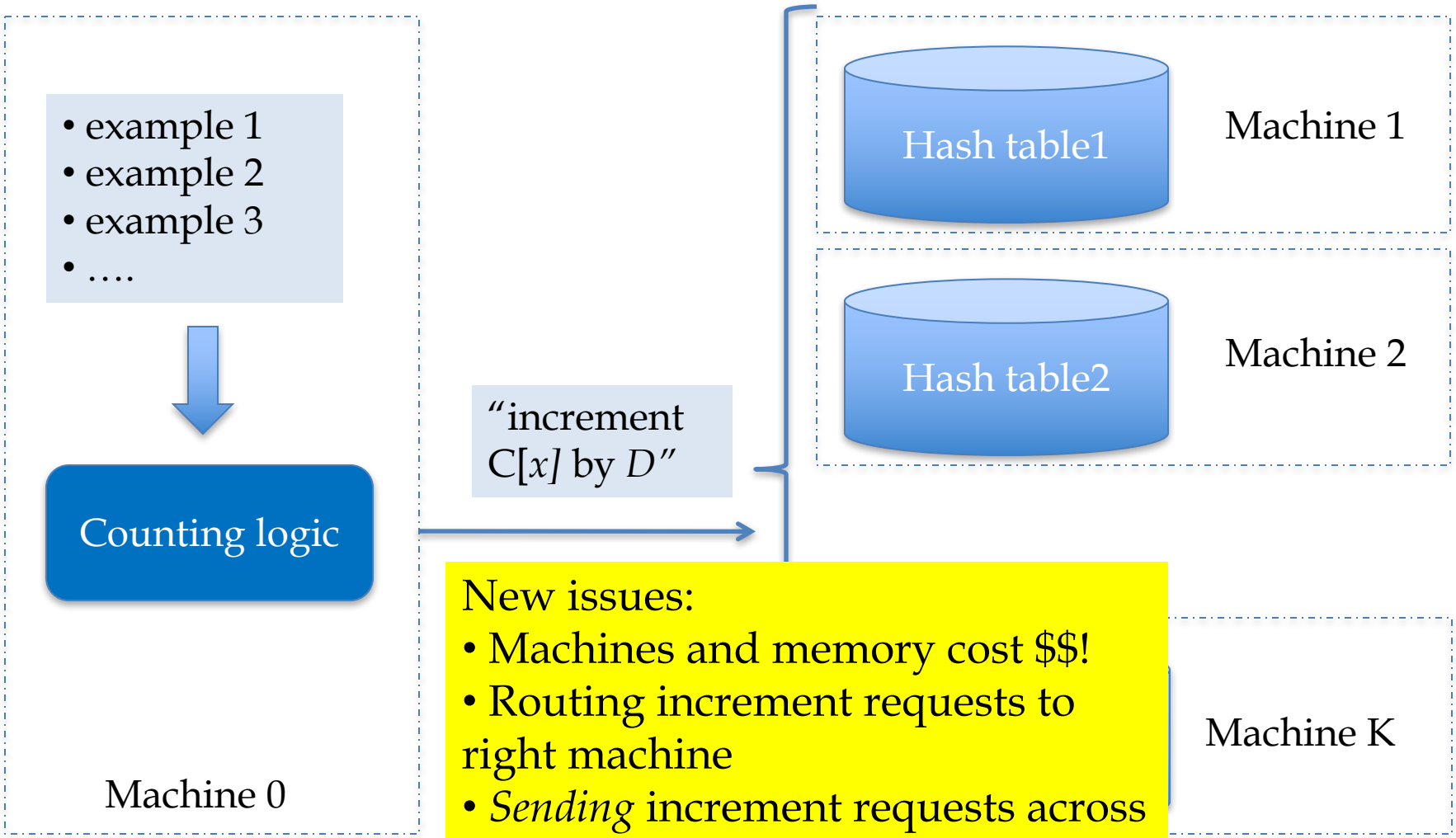



Hashtable issue: memory is too small
Database issue: seeks are slow

# Distributed Counting



# Distributed Counting



# Numbers (Jeff Dean says) Everyone Should Know

| | |
|---|---|
| L1 cache reference | 0.5 ns |
| Branch mispredict | 5 ns |
| L2 cache reference | 7 ns |
| Mutex lock/unlock | 100 ns |
| Main memory reference | 100 ns |
| Compress 1K bytes with Zippy | 10,000 ns |
| Send 2K bytes over 1 Gbps network | 20,000 ns |
| Read 1 MB sequentially from memory | 250,000 ns |
| Round trip within same datacenter | 500,000 ns |
| Disk seek | 10,000,000 ns |
| Read 1 MB sequentially from network | 10,000,000 ns |
| Read 1 MB sequentially from disk | 30,000,000 ns |
| Send packet CA->Netherlands->CA | 150,000,000 ns |

~= 10x

~= 15x

40x

~= 100,000x

# What's next

- How to implement Naïve Bayes
  - Assuming the event counters do *not* fit in memory
- Possible approaches:

O(n*scan) ➔ O(n*scan+n*send)

  - Use a **memory-based distributed** database?
    - Extra cost: Communication costs: O(n) … but that's "ok"
    - Extra complexity: routing requests correctly
      - Note: If the increment requests were ordered seeks would not be needed!

1) Distributing data in memory across machines is not *as cheap* as accessing memory locally because of **communication costs.**
2) The problem we're dealing with is not **size.** It's the interaction between **size** and **locality:** we have a large structure that's being accessed in a **non-local** way.

# What's next

- How to implement Naïve Bayes
  - Assuming the event counters do *not* fit in memory
- Possible approaches:

O(n*scan) ➔ O(n*scan+n*send)

  - Use a **memory-based distributed** database?
    - Extra cost: Communication costs: O(n) … but that's "ok"
    - Extra complexity: routing requests correctly
  - Compress the counter hash table?
    - Use integers as keys instead of strings?
    - Use approximate counts?
    - Discard infrequent/unhelpful words?
  - Trade off time for space somehow?
    - Observation: if the counter updates were better-ordered we could avoid using disk

Great ideas which we'll discuss more later

# Large-vocabulary ~~Naïve Bayes~~ Counting

- One way trade off time for space:
  - Assume you **need** *K* times as much memory as you actually **have**
  - Method:
    - Construct a hash function *h(event)*
    - For *i=0,…,K-1*:
      - Scan thru the *train* dataset
      - Increment counters for *event* only if *h(event) mod K* == *i*
      - Save this counter set to disk at the end of the scan
    - After K scans you have a complete counter set
- Comment:
  - this works for *any* counting task, not just naïve Bayes
  - What we're really doing here is organizing our "messages" to get more locality….

# HOW TO ORGANIZE DATA TO ENABLE LARGE-SCALE COUNTING

# Large vocabulary counting

- Another approach:
  - Start with
    - Q: "what can we do for large sets quickly"?
    - A: sorting
      - It's *O(n log n),* not much worse than linear
      - You can do it for very large datasets using a *merge sort*
        - » sort *k* subsets that fit in memory,
        - » merge results, which can be done in linear time


IBM


Alternative visualization

# ASIDE: MORE ON SORTING

# Bottom-Up Merge Sort

use: input array *A[n]*; buffer array *B[n]*

- *assert: A[ ] contains sorted runs of length r=1*
- for run-length *r=1*,2,4,8,…
  - merge adjacent length-*r* runs in *A[ ]*, copying the result into the buffer *B[ ]*
  - *assert: B[ ] contains sorted runs of length 2*r*
  - swap roles of A and B

```
BottomUpMerge(int A[], int iLeft, int iRight, int iEnd, int B[])
{
  int i0 = iLeft;
  int i1 = iRight;
  int j;

  /* While there are elements in the left or right lists */
  for (j = iLeft; j < iEnd; j++)
    {
      /* If left list head exists and is <= existing right list head */
      if (i0 < iRight && (i1 >= iEnd || A[i0] <= A[i1]))
        {
          B[j] = A[i0];
          i0 = i0 + 1;
        }
      else
        {
          B[j] = A[i1];
          i1 = i1 + 1;
        }
    }
}
```

# Wikipedia on Old-School Merge Sort

Use four tape drives A,B,C,D

1. merge runs from A,B and write them alternately into C,D

2. merge runs from C,D and write them alternately into A,B

3. And so on….

Requires only constant memory.

# Unix Sort

```
-s, --stable
       stabilize sort by disabli

-S, --buffer-size=SIZE
       use SIZE for main memory

-t, --field-separator=SEP
       use SEP instead of non-bl

-T, --temporary-directory=DIR
       use DIR for temporaries,
       ple directories

-u, --unique
       with  -c,  check  for str
       equal run
```

- Load as *much as you can* [actually --buffer-size=SIZE] into memory and do an *in-memory sort* [usually quicksort].
- If you have more to do, then spill this sorted buffer out on to disk, and get a another buffer's worth of data.
- Finally, merge your spill buffers.

# SORTING OUT OF MEMORY WITH PIPES

```
generate lines | sort | process lines
```

# How Unix Pipes Work

- Processes are all started at the same time
- Data streaming thru the pipeline is held in a queue: *writer* → *[…queue…]* → *reader*
- If the queue is *full*:
  - the *writing process* is blocked
- If the queue is *empty*:
  - the *reading process* is blocked

- (I think) queues are usually smallish: 64k

# How stream-and-sort works

- Pipeline is *stream* → *[…queue…]* → *sort*
- Algorithm you get:
  - *sort* reads *--buffer-size* lines in, sorts them, spills them to disk
  - *sort* merges spill files after *stream* closes
  - *stream* is blocked when sort falls behind
  - and *sort* is blocked if it gets ahead

# THE STREAM-AND-SORT DESIGN PATTERN FOR NAIVE BAYES

# Large-vocabulary Naïve Bayes

- Create a hashtable C
- For each example *id, y,* $x_1,\ldots,x_d$ in *train:*
  - C("*Y*=ANY") ++; C("*Y*=*y*") ++
  - For *j* in *1..d*:
    - C("*Y*=*y* ^ *X*=$x_j$") ++

# Large-vocabulary Naïve Bayes

- ~~Create a hashtable C~~
- For each example *id, y,* $x_1,\ldots,x_d$ in *train:*
  - ~~C("Y=ANY") ++; C("Y=y") ++~~
  - Print "Y=ANY += 1"
  - Print "Y=*y* += 1"
  - For *j* in *1..d*:
    - ~~C("Y=y ^ X=$x_j$") ++~~
    - Print "Y=*y* ^ X=$x_j$ += 1"

Think of these as "messages" to another component to increment the counters

- Sort the event-counter update "messages"
- Scan the sorted messages and compute and output the final counter values

```
python MyTrainer.py train | sort | python MyCountAdder.py > model
```

# Large-vocabulary Naïve Bayes

- ~~Create a hashtable C~~
- For each example *id, y,* $x_1,\ldots,x_d$ in *train:*
  - ~~C("Y=ANY") ++; C("Y=y") ++~~
  - Print "Y=ANY += 1"
  - Print "Y=*y* += 1"
  - For *j* in *1..d*:
    - ~~C("Y=y ^ X=$x_j$") ++~~
    - Print "Y=*y* ^ X=$x_j$ += 1"
- Sort the event-counter update "messages"
  - We're collecting together messages about the same counter
- Scan and add the sorted messages and output the final counter values

```
Y=business                 +=  1
Y=business                 +=  1
…
Y=business ^ X =aaa        +=  1
…
Y=business ^ X=zynga       +=  1
Y=sports ^ X=hat           +=  1
Y=sports ^ X=hockey        +=  1
Y=sports ^ X=hockey        +=  1
Y=sports ^ X=hockey        +=  1
…
Y=sports ^ X=hoe           +=  1
…
Y=sports                   +=  1
…
```

# Large-vocabulary Naïve Bayes

Scan-and-add:

```
Y=business                  +=   1
Y=business                  +=   1
…
Y=business ^ X =aaa         +=   1
…
Y=business ^ X=zynga        +=   1
Y=sports ^ X=hat            +=   1
Y=sports ^ X=hockey         +=   1
Y=sports ^ X=hockey         +=   1
Y=sports ^ X=hockey         +=   1
…
Y=sports ^ X=hoe            +=   1
…
Y=sports                    +=   1
…
```

streaming

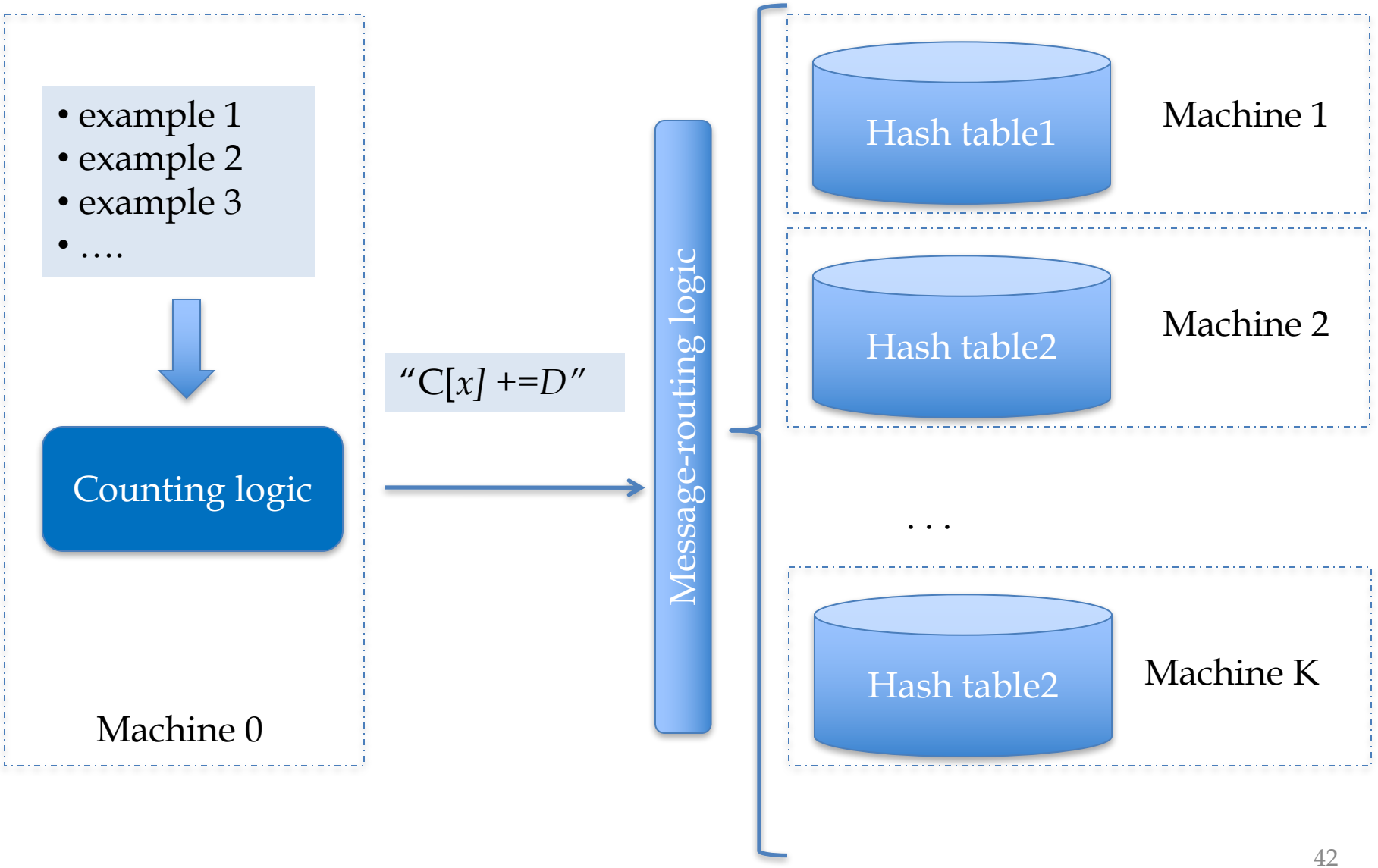
- previousKey = Null
- sumForPreviousKey = 0
- For each (*event*,*delta*) in input:
  - If *event*==previousKey
    - sumForPreviousKey += delta
  - Else
    - OutputPreviousKey()
    - previousKey = *event*
    - sumForPreviousKey = *delta*
- OutputPreviousKey()

define OutputPreviousKey():

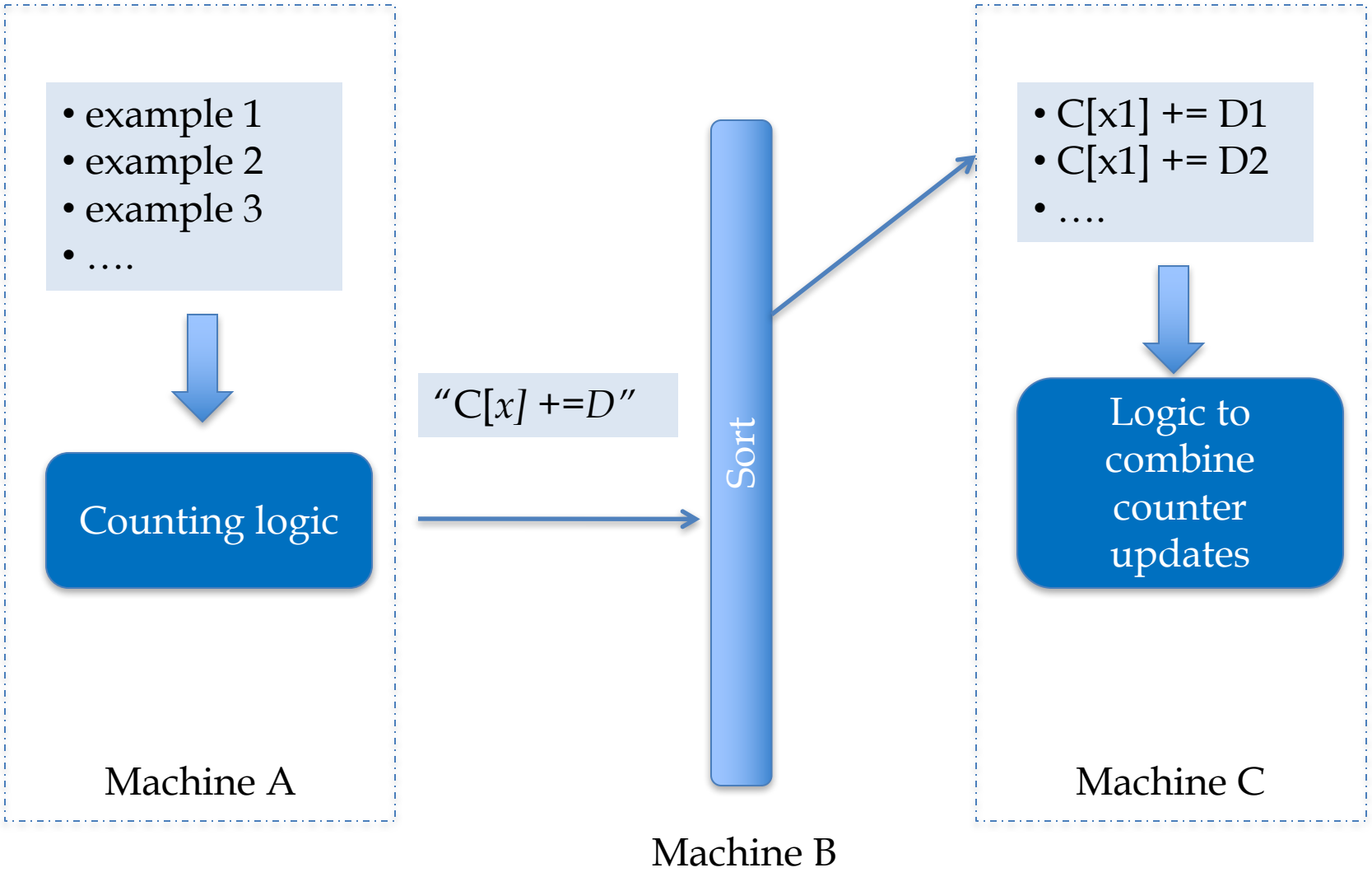
- If PreviousKey!=Null
  - print PreviousKey,sumForPreviousKey

Accumulating the event counts requires *constant* storage … as long as the input is sorted.

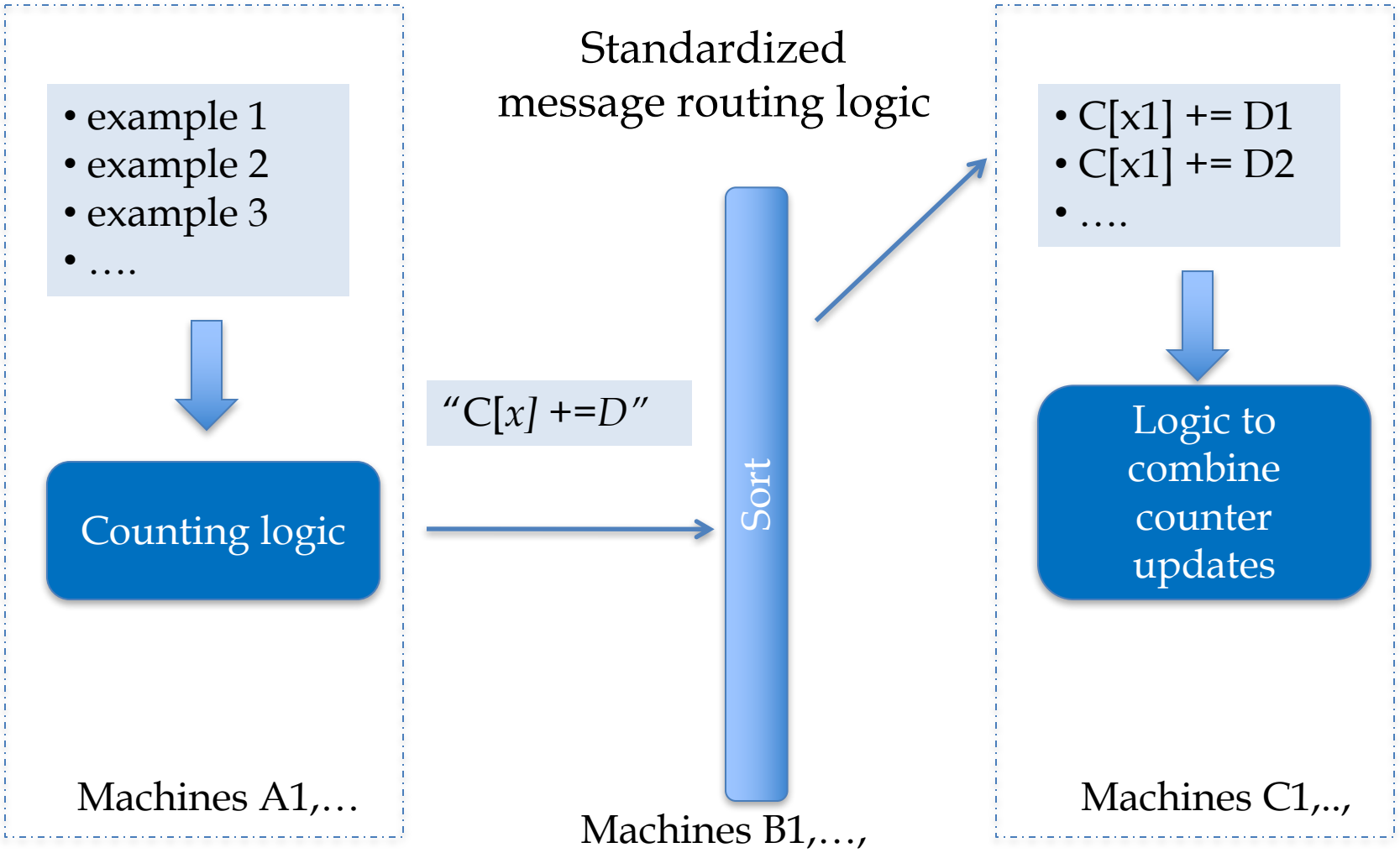
# Distributed Counting → Stream and Sort Counting



# Distributed Counting → Stream and Sort Counting



# Stream and Sort Counting → Distributed Counting




Trivial to parallelize!

Easy to parallelize!

# Locality is good

**Micro:**
0.6G memory
**Standard**:
S: 1.7Gb
L: 7.5Gb
XL: 15Mb
**Hi Memory**:
XXL: 34.2
XXXXL: 68.4

Region: US East (Virginia)

| | Linux/UNIX Usage | Windows Usage |
|---|---|---|
| **Standard On-Demand Instances** | | |
| Small (Default) | $0.085 per hour | $0.12 per hour |
| Large | $0.34 per hour | $0.48 per hour |
| Extra Large | $0.68 per hour | $0.96 per hour |
| **Micro On-Demand Instances** | | |
| Micro | $0.02 per hour | $0.03 per hour |
| **Hi-Memory On-Demand Instances** | | |
| Extra Large | $0.50 per hour | $0.62 per hour |
| Double Extra Large | $1.00 per hour | $1.24 per hour |
| Quadruple Extra Large | $2.00 per hour | $2.48 per hour |
| **Hi-CPU On-Demand Instances** | | |
| Medium | $0.17 per hour | $0.29 per hour |
| Extra Large | $0.68 per hour | $1.16 per hour |
| **Cluster Compute Instances** | | |
| Quadruple Extra Large | $1.30 per hour | $1.61 per hour |
| Eight Extra Large | $2.40 per hour | $2.97 per hour |
| **Cluster GPU Instances** | | |
| Quadruple Extra Large | $2.10 per hour | $2.60 per hour |

# Large-vocabulary Naïve Bayes

- For each example *id*, *y*, $x_1$,….,$x_d$ in *train:*
  - Print Y=ANY += 1
  - Print Y=*y* += 1
  - For *j* in *1..d*:
    - Print *Y=y* ^ *X*=$x_j$ += 1
- Sort the event-counter update "messages"
- Scan and add the sorted messages and output the final counter values

Complexity: O(*n*), *n*=size of *train*

(Assuming a constant number of labels apply to each document)

Complexity: O(*nlogn*)

Complexity: O(*n*)

```
python MyTrainer.py train | sort | python MyCountAdder.py > model
```

Model size: min( *O(n)*, *O(|V||dom(Y)|)*)

# STREAM-AND-SORT + LOCAL PARTIAL COUNTING

# Today

- Naïve Bayes with huge feature sets
  - i.e. ones that don't fit in memory
- Pros and cons of possible approaches
  - Traditional "DB" (actually, key-value store)
  - Memory-based distributed DB
  - Stream-and-sort counting
- **Optimizations**
- Other tasks for stream-and-sort

# Optimizations

`java MyTrainer`*`train`* `| sort | java MyCountAdder > `*`model`*

O(n)
Input size=n
Output size=n

O(nlogn)
Input size=n
Output size=n

O(n)
Input size=n
Output size=m

m<<n … say O(sqrt(n))

A useful optimization: decrease the *size* of the input to the sort

Reduces the size from O(n) to O(m)

1. Compress the output by using simpler messages
   ("C[*event*] += *1*") ➔ "*event 1*"
2. Compress the output more – e.g. string→integer code
   Tradeoff – ease of debugging vs efficiency – are messages meaningful or meaningful in context?

# Optimization: partial local counting

- For each example *id, y,* $x_1,\ldots,x_d$ in *train:*
  - Print "$Y$=$y$ += 1"
  - For $j$ in *1..d*:
    - Print "$Y$=$y$ ^ $X$=$x_j$ += 1"
- Sort the event-counter update "messages"
- Scan and add the sorted messages and output the final counter values

- Initialize hashtable C
- For each example *id, y,* $x_1,\ldots,x_d$ in *train:*
  - C[$Y$=$y$] += 1
  - For $j$ in *1..d*:
    - C[$Y$=$y$ ^ $X$=$x_j$] += 1
  - If memory is getting full: *output all values from C as messages and re-initialize C*
- Sort the event-counter update "messages"
- Scan and add the sorted messages

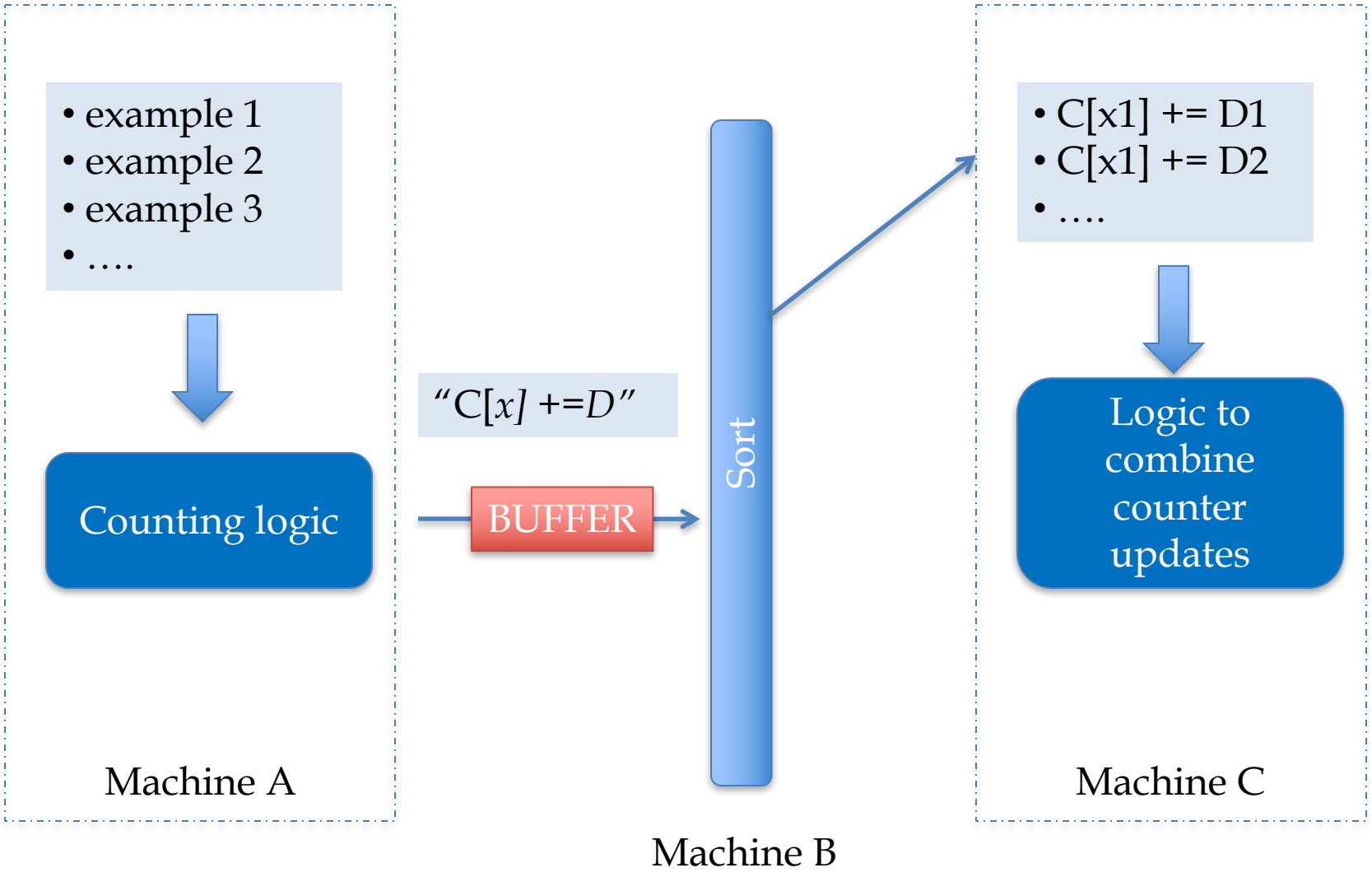
```
python MyTrainer.py train | sort | python MyCountAdder.py > model
```

# Review: Large-vocab Naïve Bayes

- Create a hashtable C
- For each example *id, y,* $x_1,\ldots,x_d$ in *train:*
  - C.inc("$Y$=$y$")
  - For $j$ in *1..d*:
    - C.inc("$Y$=$y$ ^ $X$=$x_j$")

```
class EventCounter(object):
    def __init__(self):
        self._ctr = {}
    def inc(self, event):
        // increment the counter for 'event'
        if (len(self._ctr) > BUFFER_SIZE):
            for (e,n) in self._ctr.items() : print '%s\t%d' % (e,n)
            // clear self._ctr
```

# Distributed Counting → Stream and Sort Counting



# How much does buffering help?

| BUFFER_SIZE | Time | Message Size |
|---|---|---|
| *none* | | 1.7M *words* |
| 100 | 47s | 1.2M |
| 1,000 | 42s | 1.0M |
| 10,000 | 30s | 0.7M |
| 100,000 | 16s | 0.24M |
| 1,000,000 | 13s | 0.16M |
| *limit* | | 0.05M |

# CONFESSION:
# THIS NAÏVE BAYES HAS A PROBLEM….

# Today

- Naïve Bayes with huge feature sets
  - i.e. ones that don't fit in memory
- Pros and cons of possible approaches
  - Traditional "DB" (actually, key-value store)
  - Memory-based distributed DB
  - Stream-and-sort counting
- Optimizations
- Other tasks for stream-and-sort
- **Finally**: A "detail" about large-vocabulary Naïve Bayes…..

Assume hashtable holding all counts fits in memory

# Complexity of Naïve Bayes

- You have a *train* dataset and a *test* dataset
- Initialize an "event counter" (hashtable) C
- For each example *id, y,* $x_1,\ldots,x_d$ in *train:*
  - C("$Y$=$y$") ++
  - For $j$ in *1..d*:
    - C("$Y$=$y$ ^ $X$=$x_j$") ++
    - ….
- For each example *id, y,* $x_1,\ldots,x_d$ in *test:*
  - For each $y'$ in *dom(Y):*
    - Compute log Pr($y'$,$x_1,\ldots,x_d$) =

$$= \left( \sum_j \log \frac{C(X = x_j \wedge Y = y') + mq_x}{C(X = ANY \wedge Y = y') + m} \right) + \log \frac{C(Y = y') + mq_y}{C(Y = ANY) + m}$$

  - Return the best $y'$

Sequential reads

Complexity: O(*n*), *n*=size of *train*

where:
$q_j = 1/|V|$
$q_y = 1/|dom(Y)|$
$mq_x = 1$

Sequential reads

Complexity: O(|*dom(Y)*|**n'*), *n'*=size of *test*

# Using Large-vocabulary Naïve Bayes

- For each example *id*, *y*, $x_1,\ldots,x_d$ in *train:*
- Sort the event-counter update "messages"
- Scan and add the sorted messages and output the final counter values

Model size: max $O(n)$, $O(|V| \cdot |dom(Y)|)$

- For each example *id*, *y*, $x_1,\ldots,x_d$ in *test:*
  - For each $y'$ in $dom(Y)$:
    - Compute log Pr$(y', x_1,\ldots,x_d)$ =

$$= \left( \sum_j \log \frac{C(X = x_j \wedge Y = y') + mq_x}{C(Y = y') + m} \right) + \log \frac{C(Y = y') + mq_y}{C(Y = ANY) + m}$$

# Using Large-vocabulary Naïve Bayes

[For assignment]

- For each example *id, y,* $x_1,\ldots,x_d$ in *train:*
- Sort the event-counter update "messages"
- Scan and add the sorted messages and output the final counter values

Model size: O(|*V*|)

- Initialize a HashSet NEEDED and a hashtable C
- For each example *id, y,* $x_1,\ldots,x_d$ in *test:*
  - Add $x_1,\ldots,x_d$ to NEEDED

Time: O($n_2$), *size of test*
Memory: same

- For each *event, C(event)* in the summed counters
  - If *event* involves a NEEDED term *x* read it into C

- For each example *id, y,* $x_1,\ldots,x_d$ in *test:*
  - For each *y'* in *dom(Y):*
    - Compute log Pr(*y'*,$x_1,\ldots,x_d$) = ….

Time: O($n_2$)
Memory: same

Time: O($n_2$)
Memory: same

# Large-Vocabulary Naïve Bayes

*Learning/Counting*

- Counts on disk with a key-value store
- Counts as messages to a set of distributed processes
- Repeated scans to build up partial counts
- Counts as messages in a stream-and-sort system
- **Assignment: Counts as messages but buffered in memory**

*Using Counts*

- **Assignment:**
  - **Scan through counts to find those needed for test set**
  - **Classify with counts in memory**
- Put counts in a database
- Use partial counts and repeated scans of the test data?
- Re-organize the counts and test set so that you can classify in a stream

# MORE STREAM-AND-SORT EXAMPLES

# Some other stream and sort tasks

- Coming up: classify Wikipedia pages
  - Features:
    - words on page: *src* $w_1$ $w_2$ ….
    - outlinks from page: *src* $dst_1$ $dst_2$ …
    - how about **inlinks** to the page?

# Some other stream and sort tasks

- outlinks from page: $src\ dst_1\ dst_2\ \ldots$

– Algorithm:

- For each input line $src\ dst_1\ dst_2\ \ldots\ dst_n$ print out
  - $dst_1$ inlinks.= $src$
  - $dst_2$ inlinks.= $src$
  - …
  - $dst_n$ inlinks.= $src$
- Sort this output
- Collect the messages and group to get
  - $dst\ src_1\ src_2\ \ldots\ src_n$

# Some other stream and sort tasks

- prevKey = Null
- sumForPrevKey = 0
- For each (*event* += *delta*) in input:
  - If *event*==prevKey
    - sumForPrevKey += *delta*
  - Else
    - OutputPrevKey()
    - prevKey = *event*
    - sumForPrevKey = *delta*
- OutputPrevKey()

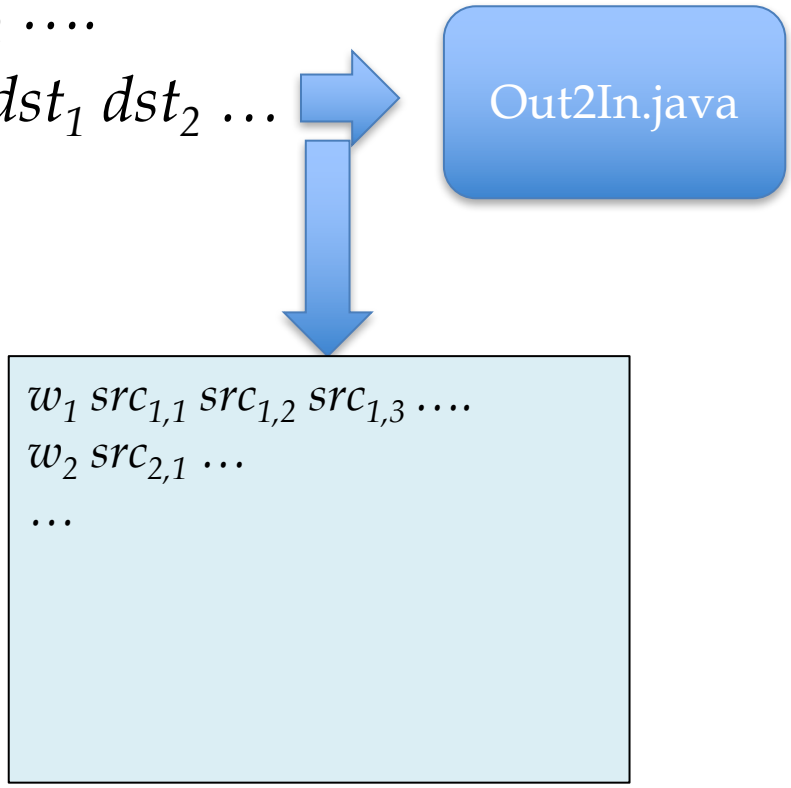
define OutputPrevKey():
- If PrevKey!=Null
  - print PrevKey,sumForPrevKey

- prevKey = Null
- linksToPrevKey = [ ]
- For each (*dst* inlinks.= *src*) in input:
  - If *dst*==prevKey
    - linksPrevKey.append(*src)*
  - Else
    - OutputPrevKey()
    - prevKey = *dst*
    - linksToPrevKey=[*src*]
- OutputPrevKey()

define OutputPrevKey():
- If PrevKey!=Null
  - print PrevKey, linksToPrevKey

# Some other stream and sort tasks

- What if we run this same program on the words on a page?
  - Features:
    - words on page: *src* $w_1$ $w_2$ ….
    - outlinks from page: *src* $dst_1$ $dst_2$ …




an *inverted index* for the documents

# Some other stream and sort tasks

- outlinks from page: $src\ dst_1\ dst_2\ \ldots$

– Algorithm:

- For each input line $src\ dst_1\ dst_2\ \ldots\ dst_n$ print out
  - $dst_1$ inlinks.= $src$
  - $dst_2$ inlinks.= $src$
  - …
  - $dst_n$ inlinks.= $src$
- Sort this output
- Collect the messages and group to get
  - $dst\ src_1\ src_2\ \ldots\ src_n$

# Some other stream and sort tasks

- Later on: distributional clustering of words

```
duty
|___responsibility 0.21 0.21
|    |___role 0.12 0.11
|    |    |___action 0.11 0.10
|    |    |    |___change 0.24 0.08
|    |    |    |    |___rule 0.16 0.08
|    |    |    |    |    |___restriction 0.27 0.08
|    |    |    |    |    |    |___ban 0.30 0.08
|    |    |    |    |    |    |___sanction 0.19 0.08
|    |    |    |    |    |___schedule 0.11 0.07
|    |    |    |    |    |___regulation 0.37 0.07
|    |    |    |___challenge 0.13 0.07
|    |    |    |    |___issue 0.13 0.07
|    |    |    |    |    |___reason 0.14 0.07
|    |    |    |    |    |___matter 0.28 0.07
|    |    |    |___measure 0.22 0.07
|    |___obligation 0.12 0.10
|    |___power 0.17 0.08
|    |    |___jurisdiction 0.13 0.08
|    |    |___right 0.12 0.07
|    |    |___control 0.20 0.07
|    |    |___ground 0.08 0.07
|    |___accountability 0.14 0.08
|    |___experience 0.12 0.07
|    post 0.14 0.14
|___post 0.14 0.14
|    |___job 0.17 0.10
|    |    |___work 0.17 0.10
|    |    |    |___training 0.11 0.07
|    |___position 0.25 0.10
|___task 0.10 0.10
|    |___chore 0.11 0.07
|___operation 0.10 0.10
|    |___function 0.10 0.08
|    |___mission 0.12 0.07
|    |    |___patrol 0.07 0.07
|    |___staff 0.10 0.07
|___penalty 0.09 0.09
|    |___fee 0.17 0.08
|    |    |___tariff 0.13 0.08
|    |    |___tax 0.19 0.07
|___reservist 0.07 0.07
```

# Some other stream and sort tasks

- Later on: distributional clustering of words

Algorithm:

- For each word $w$ in a corpus print $w$ and the words in a window around it
  - Print "$wi$ context .= $(w_{i-k},\ldots,w_{i-1},w_{i+1},\ldots,w_{i+k})$"
- Sort the messages and collect all contexts for each $w$ – thus creating an instance associated with $w$
- Cluster the dataset
  - Or train a classifier and classify it

# Some other stream and sort tasks

•prevKey = Null
• sumForPrevKey = 0
• For each (*event* += *delta*) in input:
    • If *event*==prevKey
        • sumForPrevKey += *delta*

    •Else
        • OutputPrevKey()
        • prevKey = *event*
        • sumForPrevKey = *delta*
• OutputPrevKey()

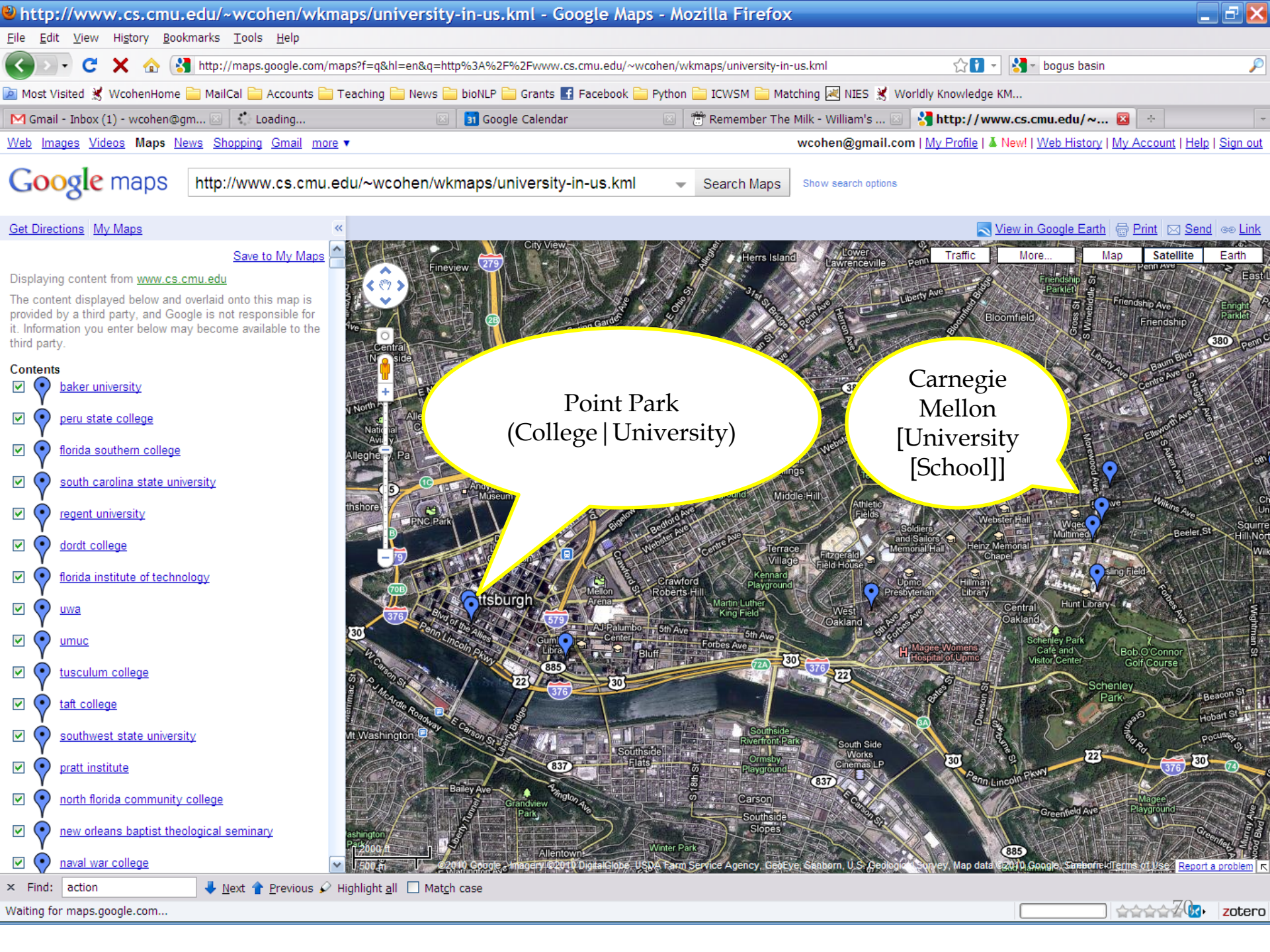
define OutputPrevKey():
• If PrevKey!=Null
    • print PrevKey,sumForPrevKey

•prevKey = Null
• ctxOfPrevKey = [ ]
• For each ($w$ c.= $w_1,\ldots,w_k$) in input:
    • If *dst*==prevKey
        • ctxOfPrevKey.append( $w_1,\ldots,w_k$ )
    • Else
        • OutputPrevKey()
        • prevKey = $w$
        • ctxOfPrevKey=[$w_1,..,w_k$]
• OutputPrevKey()

define OutputPrevKey():
• If PrevKey!=Null
    • print PrevKey, ctxOfPrevKey

# Some other stream and sort tasks

- Finding unambiguous geographical names
- GeoNames.org: for each place in its database, stores
  - Several alternative names
  - Latitude/Longitude
  - …
- Lets you put places on a map (e.g., Google Maps)
- Problem: many names are ambiguous, especially if you allow an approximate match
  - Paris, London, … even  Carnegie Mellon


http://www.cs.cmu.edu/~wcohen/wkmaps/university-in-us.kml - Google Maps - Mozilla Firefox
http://www.cs.cmu.edu/~wcohen/wkmaps/university-in-us.kml
Google maps
http://www.cs.cmu.edu/~wcohen/wkmaps/university-in-us.kml
Search Maps
Show search options
Get Directions
My Maps
Save to My Maps
Displaying content from www.cs.cmu.edu
The content displayed below and overlaid onto this map is provided by a third party, and Google is not responsible for it. Information you enter below may become available to the third party.
Contents
baker university
peru state college
florida southern college
south carolina state university
regent university
dordt college
florida institute of technology
uwa
umuc
tusculum college
taft college
southwest state university
pratt institute
north florida community college
new orleans baptist theological seminary
naval war college
View in Google Earth
Print
Send
Link
Traffic
More...
Map
Satellite
Earth
Point Park
(College | University)
Carnegie
Mellon
[University
[School]]
Find: action
Next
Previous
Highlight all
Match case
Waiting for maps.google.com...
zotero

# Some other stream and sort tasks

- Finding almost unambiguous geographical names
- GeoNames.org: for each place in the database
  - print all plausible soft-match substrings in each alternative name, paired with the lat/long, e.g.
    - Carnegie Mellon University at lat1,lon1
    - Carnegie Mellon at lat1,lon1
    - Mellon University at lat1,lon1
    - Carnegie Mellon School at lat2,lon2
    - Carnegie Mellon at lat2,lon2
    - Mellon School at lat2,lon2
    - …
  - Sort and collect… and filter

# Some other stream and sort tasks

•prevKey = Null
• sumForPrevKey = 0
• For each (*event* += *delta*) in input:
    • If *event*==prevKey
        • sumForPrevKey += *delta*
    •Else
        • OutputPrevKey()
        • prevKey = *event*
        • sumForPrevKey = *delta*

•OutputPrevKey()

define OutputPrevKey():
• If PrevKey!=Null
    • print PrevKey,sumForPrevKey

•prevKey = Null
• locOfPrevKey = Gaussian()
• For each (*place* at *lat,lon*) in input:
    • If *dst*==prevKey
        •locOfPrevKey.observe(*lat, lon)*
    • Else
        • OutputPrevKey()
        • prevKey = *place*
        • locOfPrevKey = Gaussian()
        • locOfPrevKey.observe(*lat, lon)*
• OutputPrevKey()

define OutputPrevKey():
• If PrevKey!=Null and
locOfPrevKey.stdDev() < 1 mile
    • print PrevKey, locOfPrevKey.avg()