ASYNC Lecture Slides

Machine Learning at Scale

Welcome to this lecture on Spark. Spark is an open source cluster computing framew ork. It has emerged as a next generation big data processing engine, overtaking Hado op, which helped ignite this big data revolution. At least this is true in certain capacities, especially around machine learning.

Spark maintains MapReduce's linear scalability and fault tolerance, but extends us in a number of ways. It enables us to use memory-backed computer processing, which helps accelerate processing of large-scale data by machine learning algorithms.

In this lecture, we're going to introduce the basics of Spark, an almost true functional programming language. We'll introduce the core data structure in Spark, the RDD, the Resilient Distributed Dataset. Basically this structure stores key value pairs in a very effective way, both on disk but also in memory.

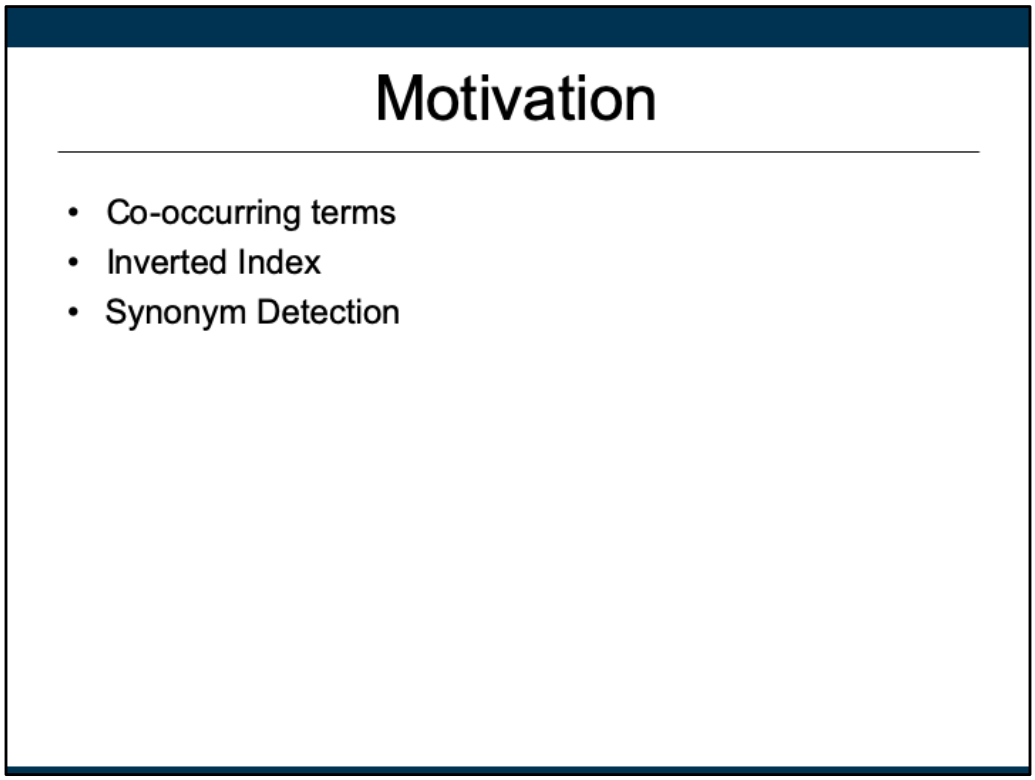
We'll learn how to program in Spark. We'll do the Hello World of MapReduce framew orks, which is word counting. We'll do it three or four different ways.
We'll learn how to get data in and out of Spark. We'll learn how to write iterative algo rithms, such as k-means and page-rank in Spark, how one should use broadcast variables, and get data to all our mappers.

We'll even revisit some of the design patterns that we covered earlier in the Hadoop MapReduce framework and see how similar principles apply here.

ASYNC Lecture Slides

The End

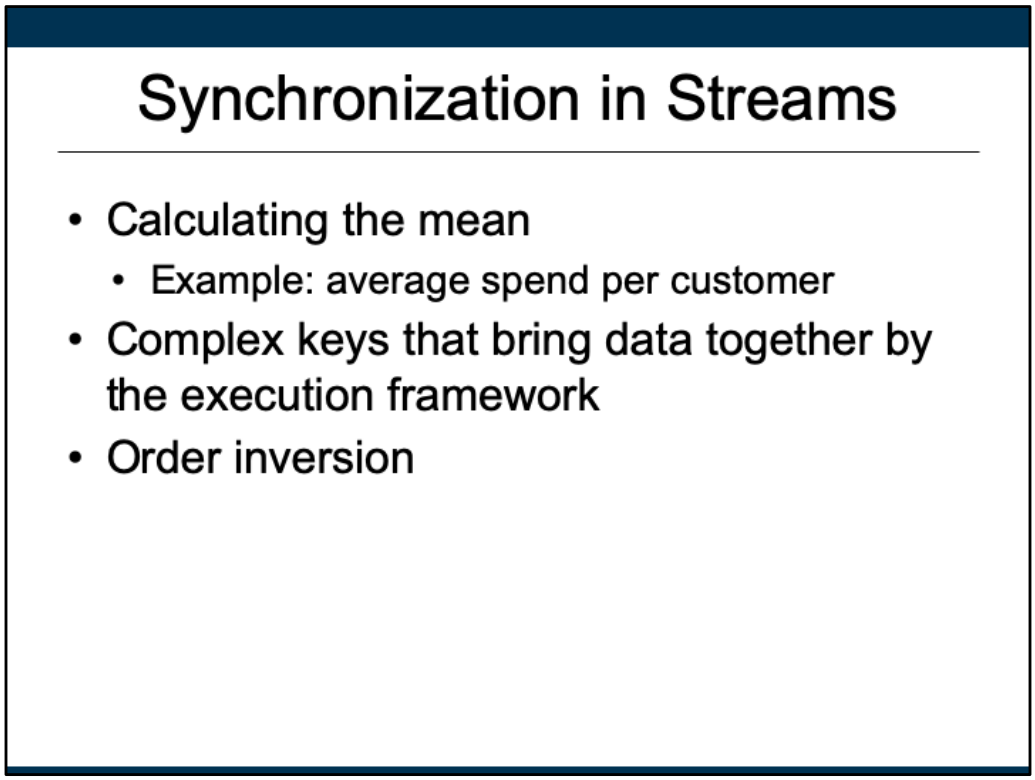
Pairs and Stripes



With the fundamentals covered, let's revisit some design patterns we learned in Hadoop and go one step further, looking at how we can utilize smart data structures to design even more efficient algorithms.

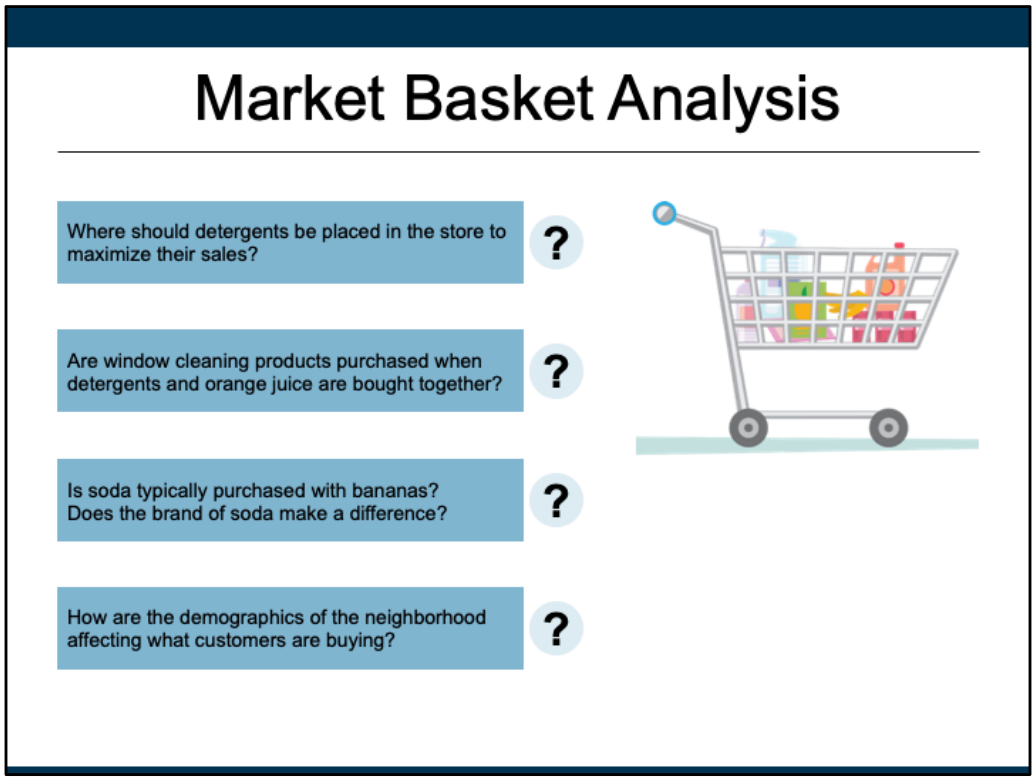
We'll take our word count to next level by calculating not just single words, but pairs of words that occur together (i.e., "co-occurring terms"); later, we'll see how we can leverage this new knowledge to build an "inverted index"—a fundamental data structure in the domain of information retrieval.

You'll use these patterns in the homework to do "synonym detection."



OK, we're going to continue our discussion here around synchronization in streams. Previously we have looked at pretty straightforward examples of this, where we synchronized around a particular customer ID, and we got the average amount spent by the customer. Now let's up the ante and explore more challenging examples which they are more combinatorically explosive before synchronizing down to a solution space.

So one common approach for synchronization in a MapReduce style framework, is to construct complex keys and values in such a way that the data necessary for computation are naturally brought together by the execution framework. We first touched on this technique last week in terms of packaging together partial sums and counts in a complex value or pair that is passed from the mapper to the combiner to the reducer.



Let’s look at an example of market basket analysis. Here, if I'm a shop owner, I'm very interested in the buying habits or patterns of customers. And this is very much part of the data mining field. We're always looking for patterns, we're always looking for correlations, and sometimes we leverage these patterns to make our business more optimal.

So for example, we might co-locate the diapers and beer on the same aisle because on Friday evening we've noticed that people generally buy beer and diapers on their way home from work.

That's an urban myth, by the way.

Co-Occurence Matrix

	OJ	Window Cleaner	Milk	Soda	Detergent
OJ	4	1	1	2	1
Window Cleaner	1	2	1	1	0
Milk	1	1	1	0	0
Soda	2	1	0	3	1
Detergent	1	0	0	1	2

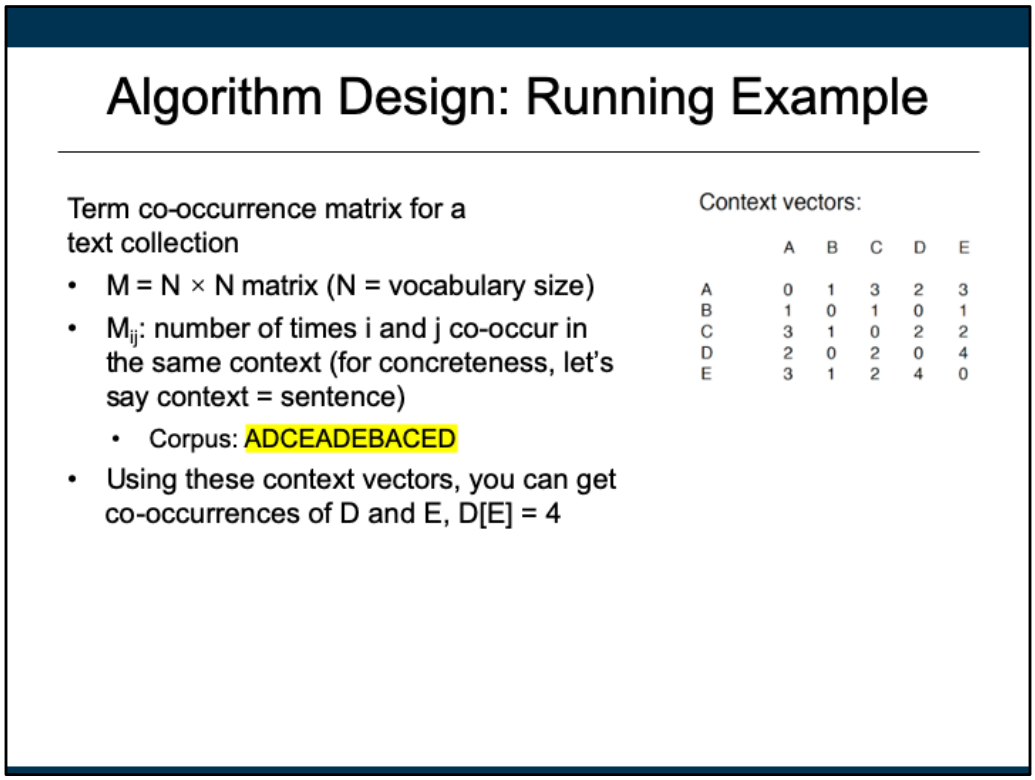
So what we're looking at here, is a co-occurrence matrix. We're looking at where events or items co-occur.

In this case, we're looking at where two items are in the same basket of a customer.

Effectively what we're doing here is we're building a square matrix which looks at, say, orange juice. We're saying, how many baskets were there where a customer bought orange juice and window cleaner together? As it turns out here, we've got one such customer.

Similarly, if we look at window cleaner and soda, we see that there's one customer.

So this is our co-occurrence matrix for market basket analysis, and it has a symmetric relationship.



Now, this also happens to be a very important concept in natural language processing, where we might be looking at term co-occurrence or word co-occurrence. We want to see where do words co-occur? And this can be very useful practice in determining the meaning or semantics of a particular word or phrase.

So here, how we construct this co-occurrence matrix is a little bit different to the basket analysis of the previous example. The idea here is that we might have a corpus of, let's say, letters, here in this case in this highly contrived example. And we might look at co-occurrence being defined as words that occur in a window of size 2 on either side of a particular word of interest.

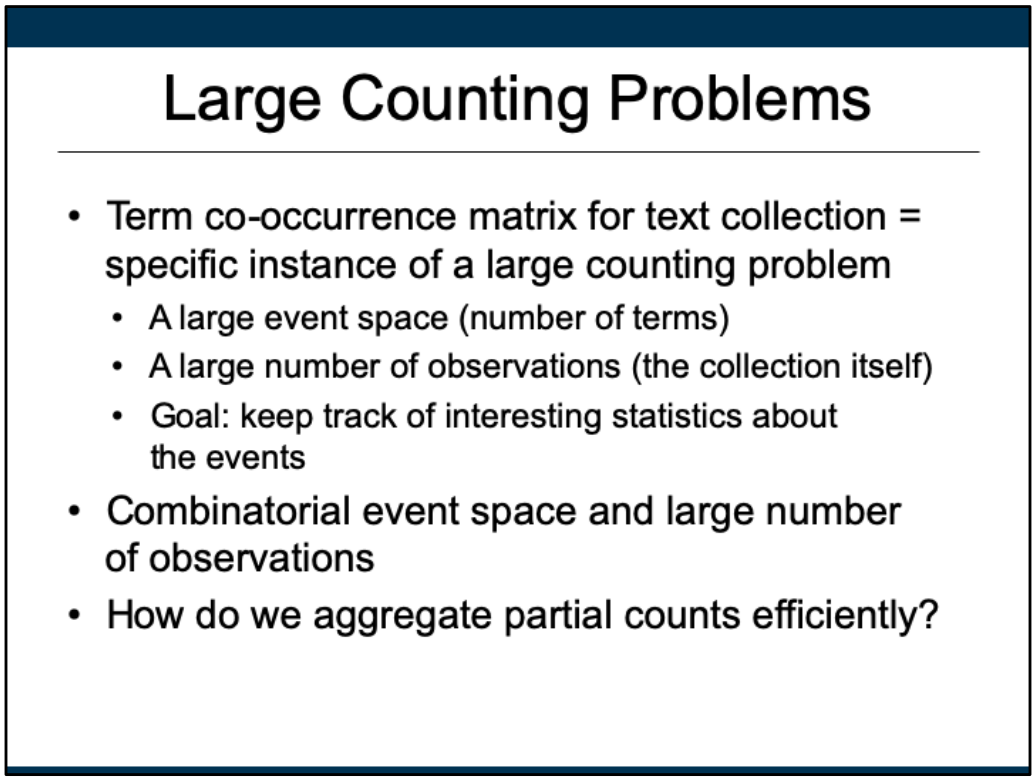
So for example here, if you look at the first word, A, then it occurs with D and with C. And in our process to generate this co-occurrence matrix, we're going to basically window across the corpus, constructing these co-occurrence instances, and then we're going to update our counters in our co-occurrence matrix.

So for example, we will update the count for A, D initially. Then we'll take the next co-occurring pair, A, C and update the corresponding cell in the co-occurrence matrix. And we will keep windowing across our corpus here until we come to the end. So the last update from this corpus will be for the co-occurrence of E and D. And we can see that for a D and E, they co-occur at a frequency of 4. In this case here, there's no particular order associated with the terms in the corpus, so it's a symmetric relationship. But sometimes we might have the requirement that the first word is always the first word, and so having a word pair like "the dog" is not the same as "dog the."

Algorithm Design: Running Example (cont.)

- M = N × N matrix (N = vocabulary size)
- M_{ij}: number of times i and j co-occur in the same context (for concreteness, let's say context = sentence)
- Why?
  - Distributional profiles as a way of measuring semantic distance
  - Semantic distance useful for many language processing tasks

Next, we want to explore how we could build such a matrix at scale over a large text collection, for example. And so the reasons for building this type of co-occurrence matrix are many, but as I alluded to earlier, by having these distribution profiles, they can be used to measure the semantic similarity or semantic distance between concepts. And this is a very useful notion in natural language processing.



So how do we do this at scale? Clearly we're going to have a combinatorial event space and a large number of observations, but that's the kind of thing that framework like Spark is really good at.

So the big question here is what do our transformations and actions look like?

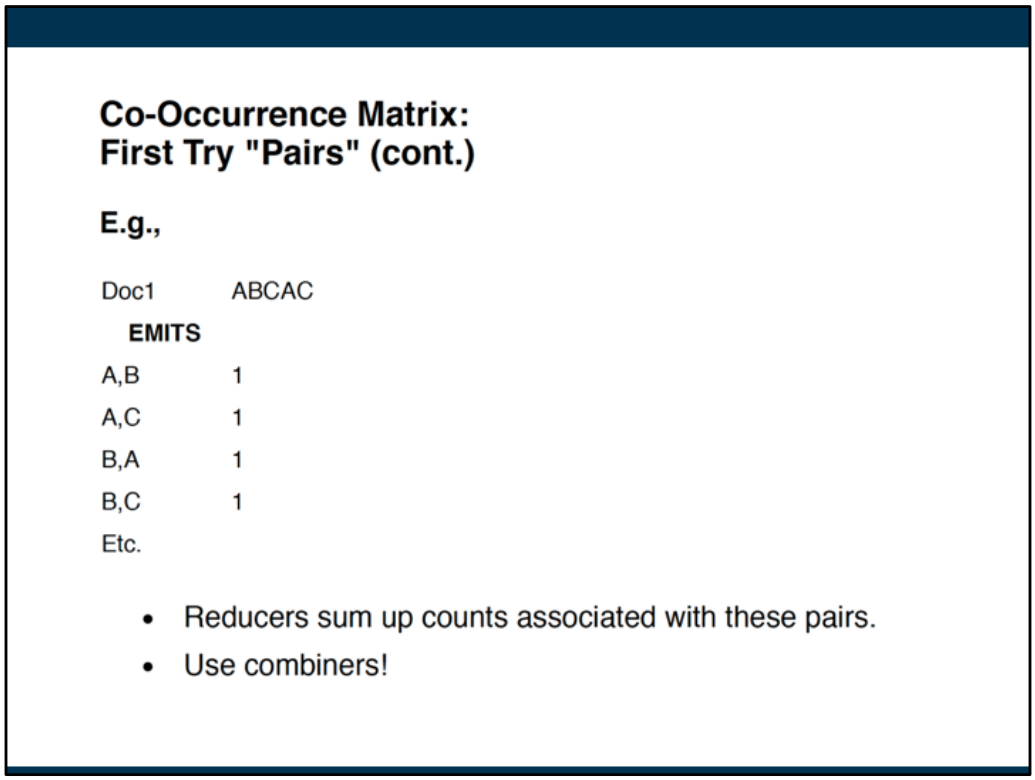
Co-Occurrence Matrix: First Try "Pairs"

Each pair corresponds to a cell in the word co-occurrence matrix. This algorithm illustrates the use of complex keys in order to coordinate distributed computations.

	A	B	C	D	E
A	0	1	3	2	3
B	1	0	1	0	1
C	3	1	0	2	2
D	2	0	2	0	4
E	3	1	2	4	0

- Each mapper takes a sentence:
  - Generate all co-occurring term pairs
  - For all pairs, emit (a, b) → count

So we've already seen that the term co-occurrence matrix for a text collection is a large event space. It's basically a square matrix, and it’s very sparse. And basically what we would do is build up this matrix using mapping and aggregating transformations. So as you can imagine, we can do this as follows. We can take a particular stream of key value pairs. In this case, the key could be a document or sentence id, and the value could be the document or sentence string.



And in this case, let's assume that we're doing co-occurrence defined by the terms occurring in the same sentence. So you can imagine that a natural mapper task here is to window over the input document string emitting pairs of tokens that co-occur, as we have defined previously. So here, we've got our first document, which is A, B, C, A, C, and we're going to emit the following pairs of tokens, and a count of one. So A, B is the first token to be admitted here with a count of 1, A, C, B, A, and B, C, et cetera. So then a simple reduceByKey() could basically say, let's synchronize around the pairs of words, and let's sum up the counts associated with those pairs. And lucky for us, Spark will do combining for us automatically whenever possible!

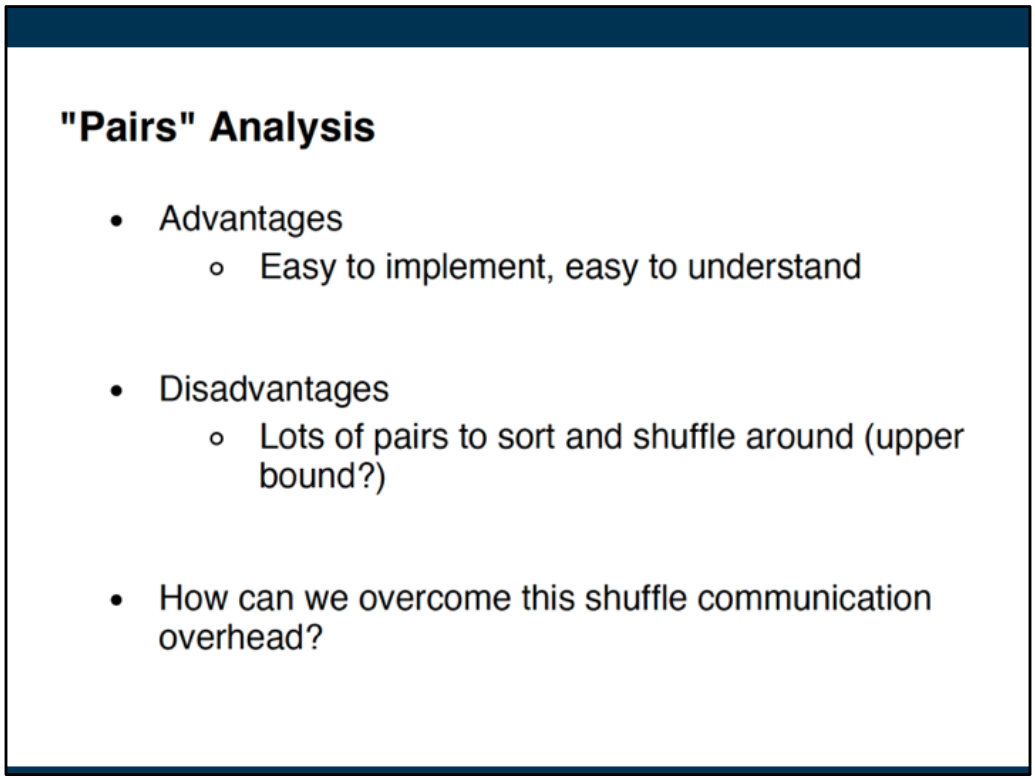
Pairs: Pseudo-Code

```
def emitPairs(row):
        words = row.split(“ ”)
        for all word in words:
                for all neighbor of word:
                        emit ((word,neighbor), 1) # emit count for co-
occurrences

rdd.flatMap(emitPairs)\
   .reduceByKey(lambda a,b: a+b) # sum co-occurrence counts
```

Each pair corresponds to a cell in the word co-occurrence matrix. This algorithm illustrates the use of complex keys in order to coordinate distributed computations.

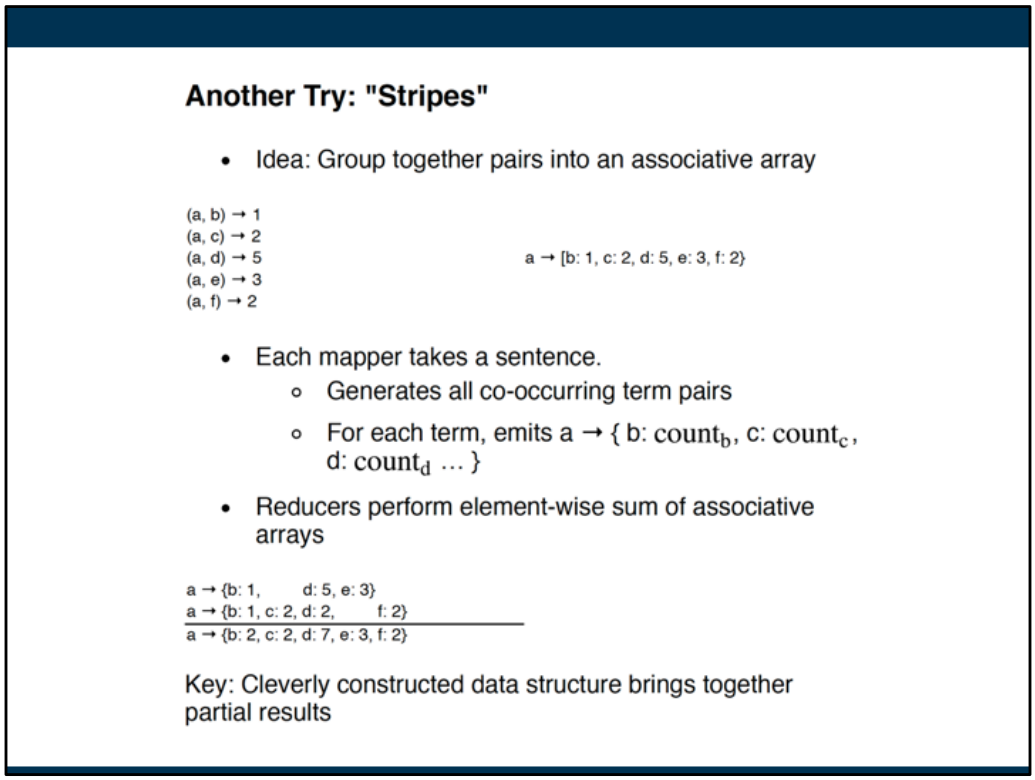
So here, the RDD pseudocode looks like this. Essentially, what we're doing here is that the mapper processes each document and emits intermediate key value pairs with each co-occurring word pair as the key, and the integer 1 as the value. And this is accomplished by two nested loops -- the outer loop iterates over all words, and the inner loop iterates over all neighbors of each word.



This the Pairs approach. It has some advantages and disadvantages.

The biggest advantage here is that it's very easy to implement!

Of course the downside is that we're creating a ton of (key, value) pairs that need to be shuffled around the network!



So instead of emitting all possible pairs, let's see what happens when we take a "stripes" approach.

The input to the mapper is a sentence as before.
The mapper emits an associative array where the key is a term, and the array holds (key, value) pairs where each key is a term which co-occurs with the primary key, and the values are the counts.
The reducers then aggregate the associative arrays by doing an element-wise sum of the arrays for each primary key

Stripes: Pseudo-Code

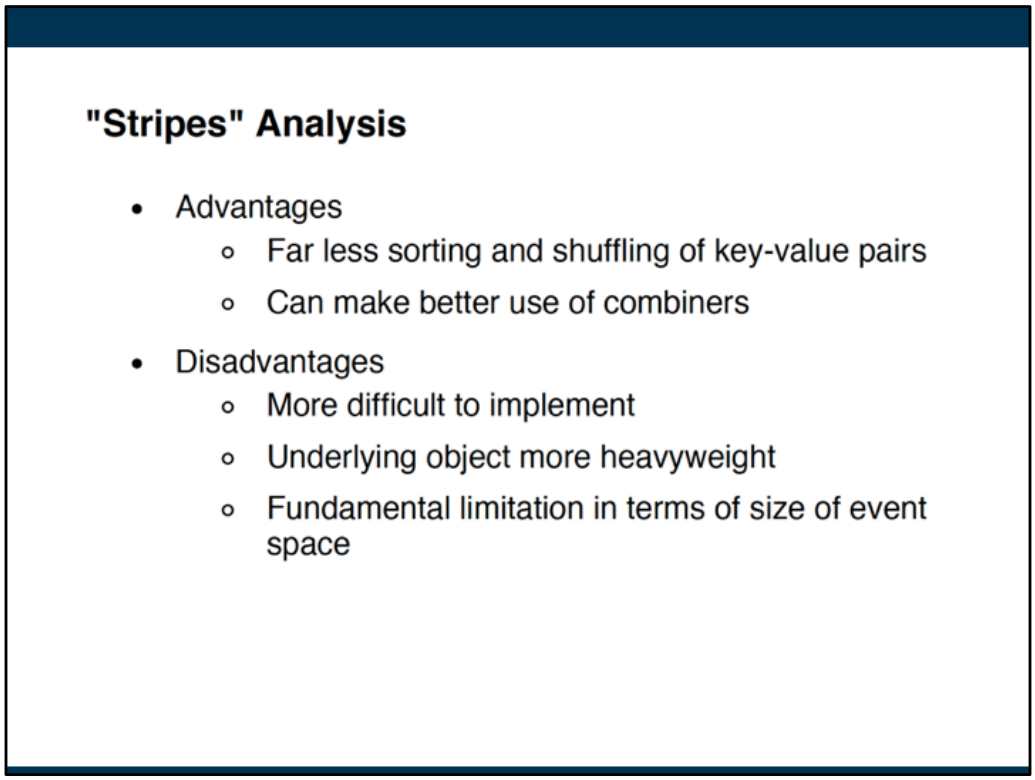
```
a → {b:1, c:2, d:5, e:3, f: 2}

def emitStripes(row):
        words = row.split(“ ”)
        for all word in words:
             H ← new ASSOCIATIVE_ARRAY
             for all neighbor of word:
                H{word} ← H{word} + 1
             emit (word, H) # emit stripe

def sumStripes(a,b):
        emit sum(a,b) # elementwise sum of {}

rdd.flatMap(emitStripes)\
   .reduceByKey(sumStripes)
```

Again, here’s the pseudo code for the stripes approach. We still have an outer loop and an inner loop in the map phase, but this time we’re accumulating a dictionary in order to emit entire stripes at the end of the outer loop. The reducer then aggregates the dictionaries by doing element-wise sums.

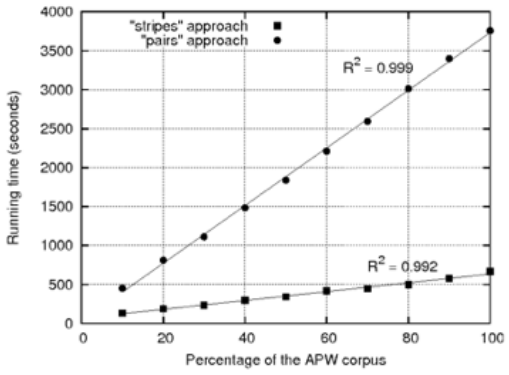


The main advantage here is that we are shuffling a lot less data. And by using reduceByKey, we're giving Spark an opportunity to do some combining for us behind the scenes.

On the flip side, we are limited in terms of the event space. In other words, we cannot afford to make our associative arrays larger than will fit in memory on a single machine.

Performance

- Which is faster, stripes or pairs?
- Stripes has a bigger value per key—watch out for memory usage!
- Pairs has more partition and sort overhead.



Lin, J.J., & Dyer, C. *Data-intensive text processing with MapReduce*

The chart presented here is from experiments on a Hadoop cluster, with 19 slaves, each with two single-core processors and two disks. More information can be found in the Lin and Dyer book, chapter 3. We can see that the stripes approach (bottom line) is much faster, and doesn't grow as fast as the pairs one.

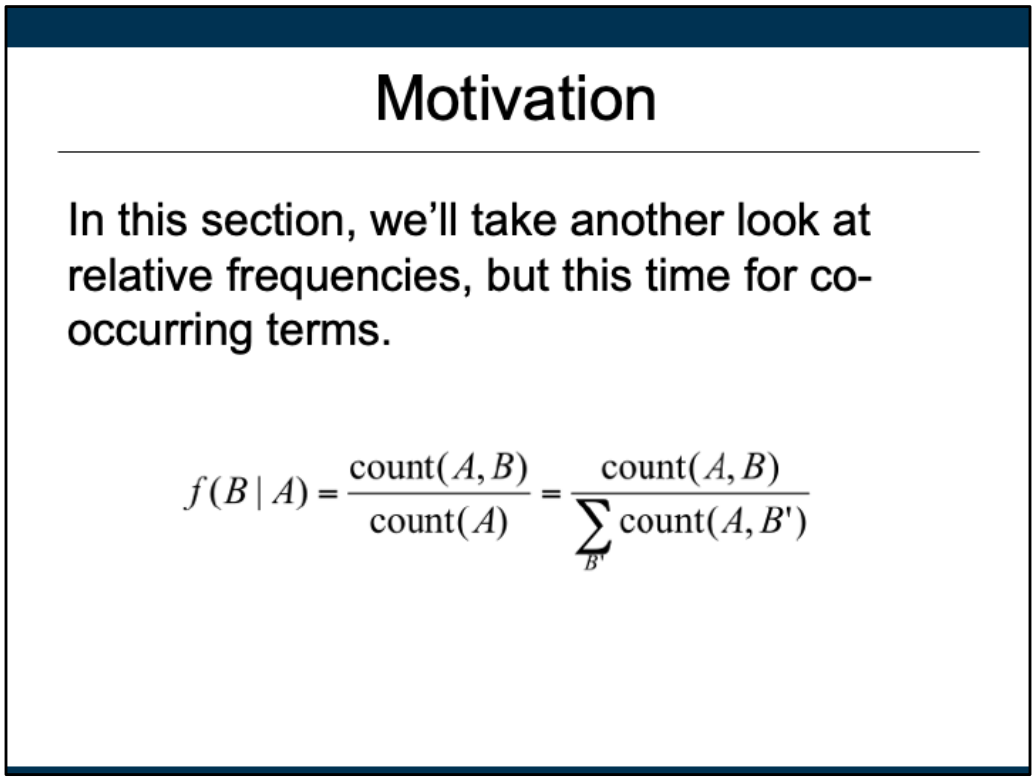
It would be interesting to see a similar experiment conducted for the Spark implementation, though we should be able to expect similar results. More on that in our live session!

In summary, the stripes approach is faster, but we have to be mindful of managing memory!
Pairs has more partition and sort overhead, but can handle data with potentially very large key skew.

Pairs and Stripes

The End

Relative Frequencies Revisited

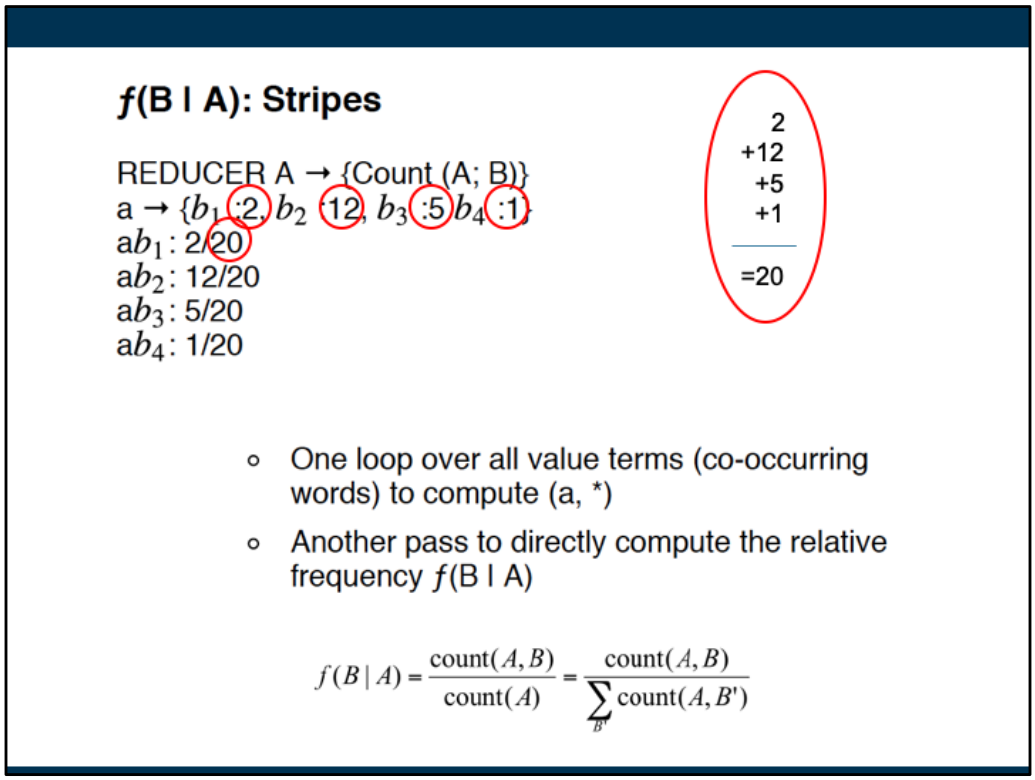


We talked about co-occurring terms, and we talked about relative frequencies, now let's talk about relative frequencies of co-occurring terms.

A very prominent task in text mining applications is computing the *word co-occurrence* of pairs of words in a large collection of documents. These co-occurrence counts can be converted into relative frequencies using "order inversion". Relative frequencies qualify better for tasks like **document classification**, **information retrieval**, **search**, or **plagiarism detection** as they are normalized co-occurrence counts.

Relative frequencies of word co-occurrences can be computed by **partitioning** and **grouping** a composite key similar to how we do secondary

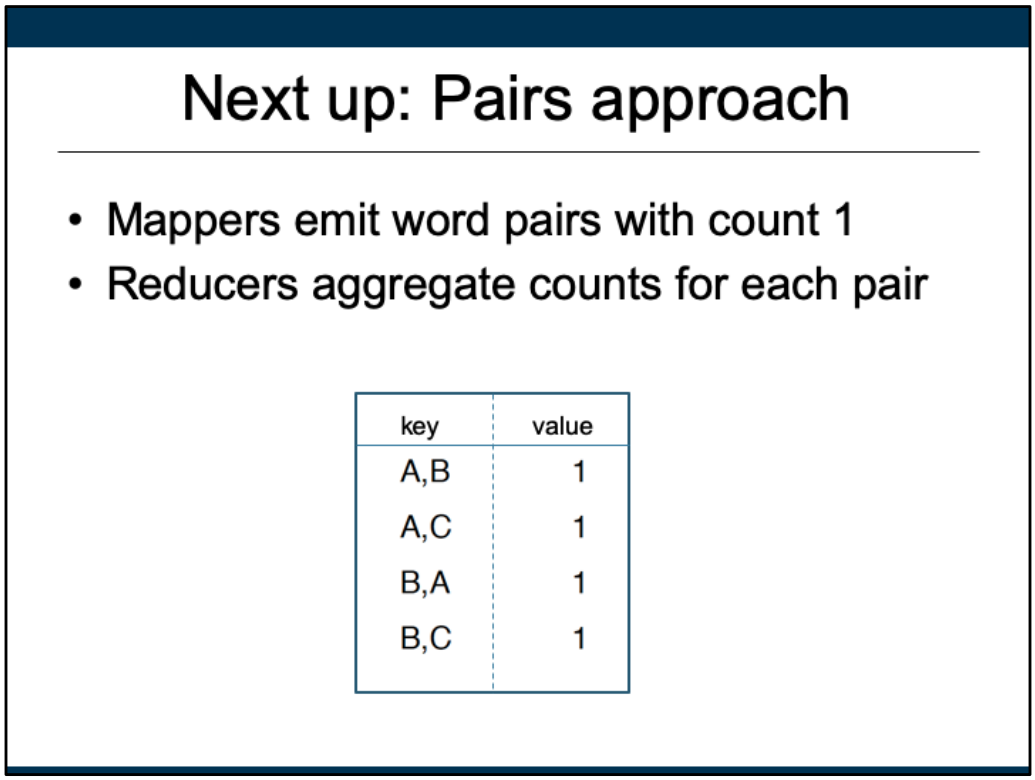
sort.



Previously we looked at calculating the number of times that pairs of words co-occur - now the goal is to calculate the relative frequencies of these pairs of words.

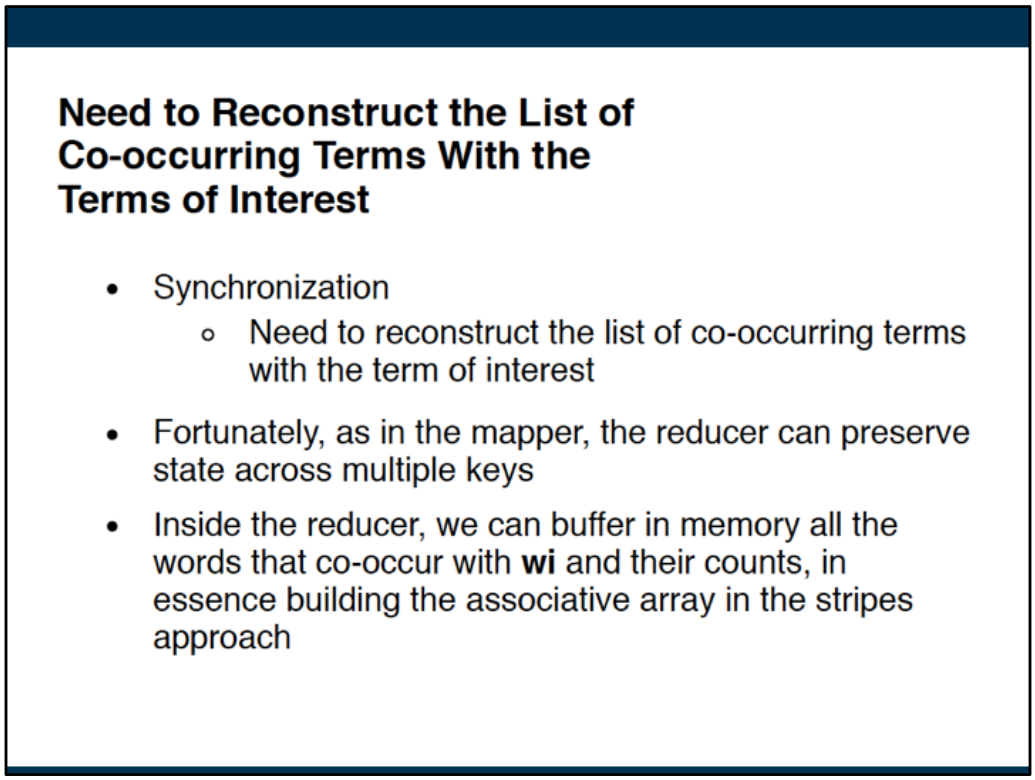
In the stripes approach, using the output from the previous section, it's relatively trivial to calculate all relative frequencies for a given word in a single pass over the data.

For each term, we simply sum the number of all co-occurrences inside the associative array, and place this in the denominator. Notice that no shuffling is required!



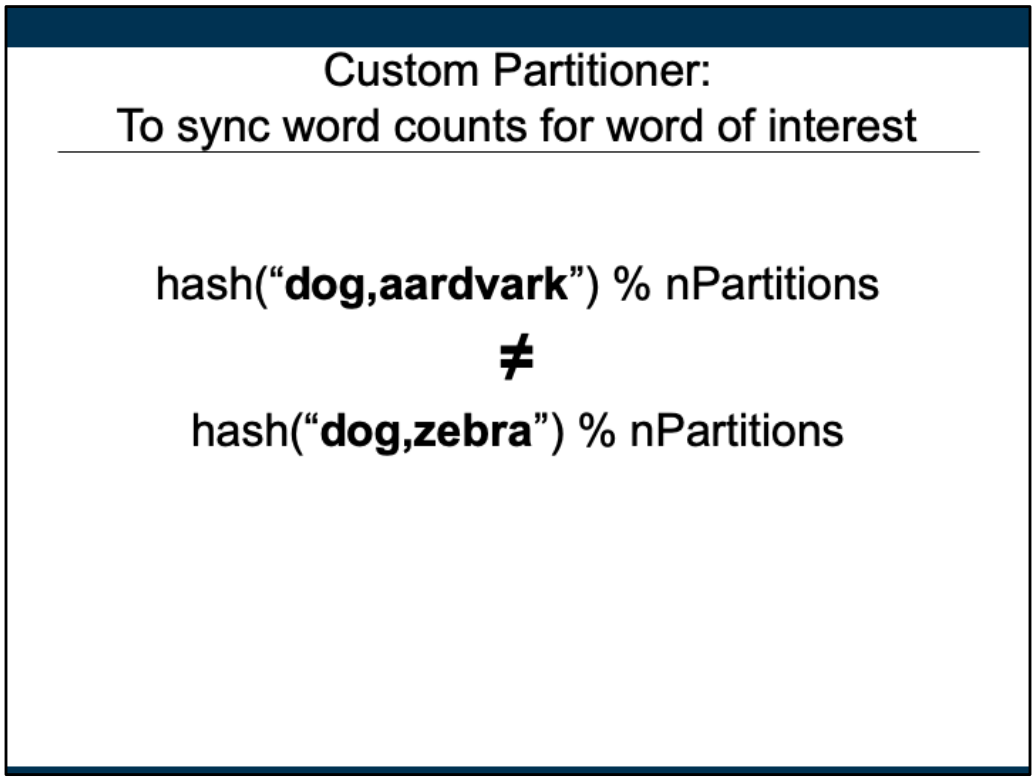
Next up: Pairs approach

So how might one compute relative frequencies with the pairs approach? Remember that in the pairs approach, the reducer receives the term pair as the key, and the count as the value. From this alone, it is not possible to compute relative frequencies, since we do not have the marginal – the part that goes in the denominator.



To make this work, we must define the sort order of the pair so that keys are first sorted by the left word, and then by the right word. Given this ordering, we can easily detect if all pairs associated with the word we are conditioning on (the first word) have been encountered – think back to our Hadoop wordcount logic.

Then in the reduce phase, we can go back through the in-memory buffer, compute the relative frequencies, and then emit those results in the final key-value pairs.



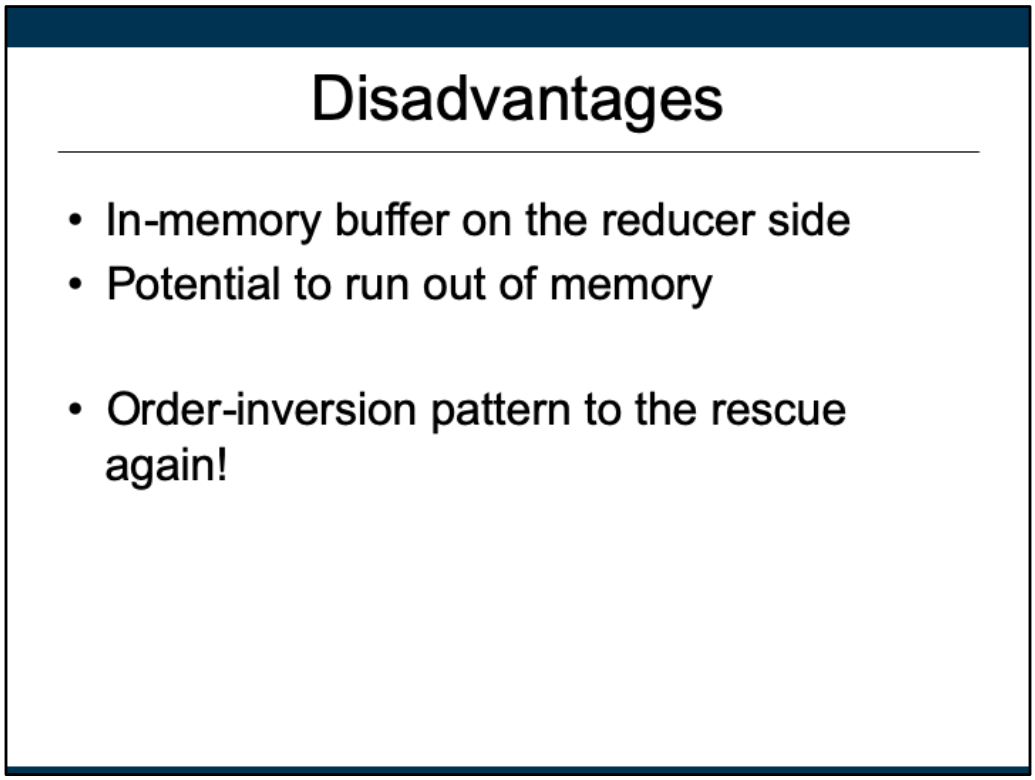
There is one more modification necessary to make this algorithm work.
We must ensure that all pairs with the same left words are sent to the same partition.

This, unfortunately, does not happen automatically. Recall that the default partitioner is based on the hash value of the key, modulo the number of partitions.

For a complex key, the raw byte representation is used to compute the hash value. As a result, there is not guarantee that, for example, (dog, aardvark) and (dog, zebra) are assigned to the same partition.

To produce the desired behavior, we must define a custom partitioner that pays attention only to the left word.

For example, we must make sure that all pairs with the first word dog, get sent to the same reducer.

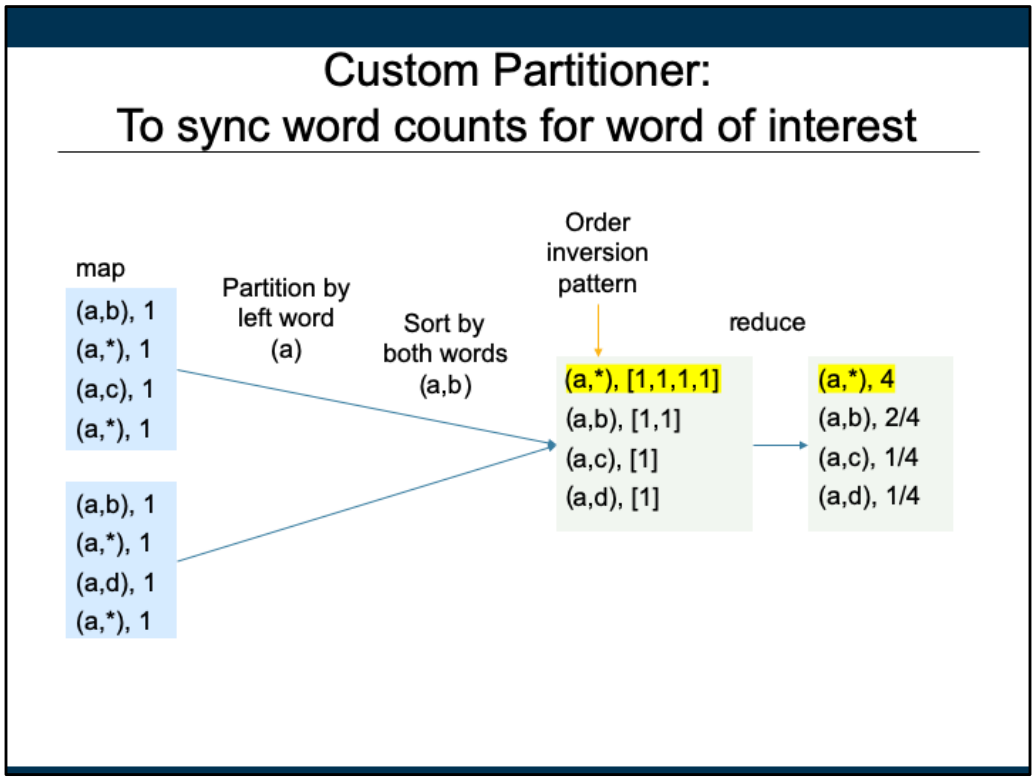


This algorithm will indeed work, but it suffers from the same drawback as the stripes approach: as the size of the corpus grows, so does that vocabulary size, and at some point there will not be sufficient memory to store all co-occurring words and their counts for the word we are conditioning on. For computing the co-occurrence matrix, the advantage of the pairs approach is that it doesn't suffer from any memory bottlenecks.

Is there a way to modify the basic pairs approach so that this advantage is retained?

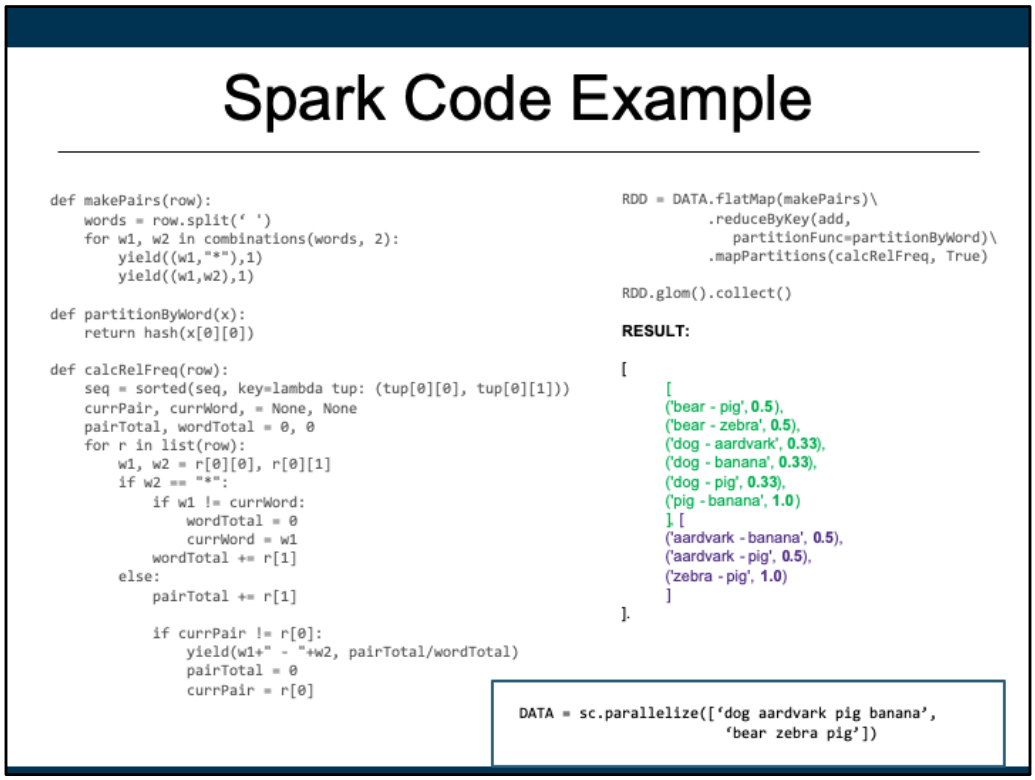
As it turns out, such an algorithm is indeed possible, although it requires the coordination of several mechanisms. The insight lies in properly sequencing data presented to the reduce phase. If it were possible to somehow compute (or otherwise obtain access to) the marginal in the reducer **before** processing the joint counts, the reducer could simply divide the joint counts by the marginal to compute the relative frequencies.

This notion of **before**" and **after**" can be captured in the ordering of key-value pairs, which can be explicitly controlled by the programmer.



To compute relative frequencies, we modify the mapper so that it additionally emits a \special" key of the form (wi; *),
with a value of one, that represents the contribution of the word pair to the marginal.

Through use of combiners, these partial marginal counts will be aggregated before being sent to the reducers. Alternatively, the in-mapper combining pattern can be used to even more efficiently aggregate marginal counts.



Let's walk through this code example.

On the right we have the Spark API calls, and on the left we have our function definitions.
Our dataset consists of 2 sentences:

Summary

Computing Relative frequencies with the order-inversion pattern requires:

- Emitting a special key-value pair for each co-occurring word pair in the mapper to capture its contribution to the marginal.
- Controlling the sort order of the intermediate key so that the key-value pairs representing the marginal contributions are processed by the reducer before any of the pairs representing the joint word co-occurrence counts.
- Defining a custom partitioner to ensure that all pairs with the same left word are shuffled to the same reducer.
- Preserving state across multiple keys in the reducer to first compute the marginal based on the special key-value pairs and then dividing the joint counts by the marginals to arrive at the relative frequencies.

Relative Frequencies Revisited

The End

Inverted Index

Definitions

- **Inverted index:** Given a term provides access to the list of documents that contain the term, an inverted index consists of postings lists, one associated with each term that appears in the collection.
- **Postings:** A postings list is comprised of individual postings, each of which consists of a document ID and a payload|information about occurrences of the term in the document. The simplest payload is …nothing! The most common payload, however, is term frequency (tf), or the number of times the term occurs in the document.

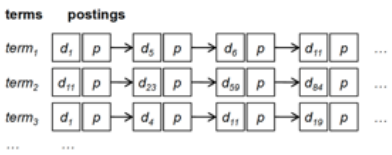


Figure 4.1: Simple illustration of an inverted index. Each term is associated with a list of postings. Each posting is comprised of a document id and a payload, denoted by p in this case. An inverted index provides quick access to documents ids that contain a term.

Lin and Dyer, Chapter 4, pg74.

Previously we talked about calculating relative frequencies for co-occurring terms. In this section we'll see how to calculate similarity score between co-occurring terms, and introduce the inverted index.

In its basic form, an inverted index consists of postings lists, one associated with each term that appears in the collection

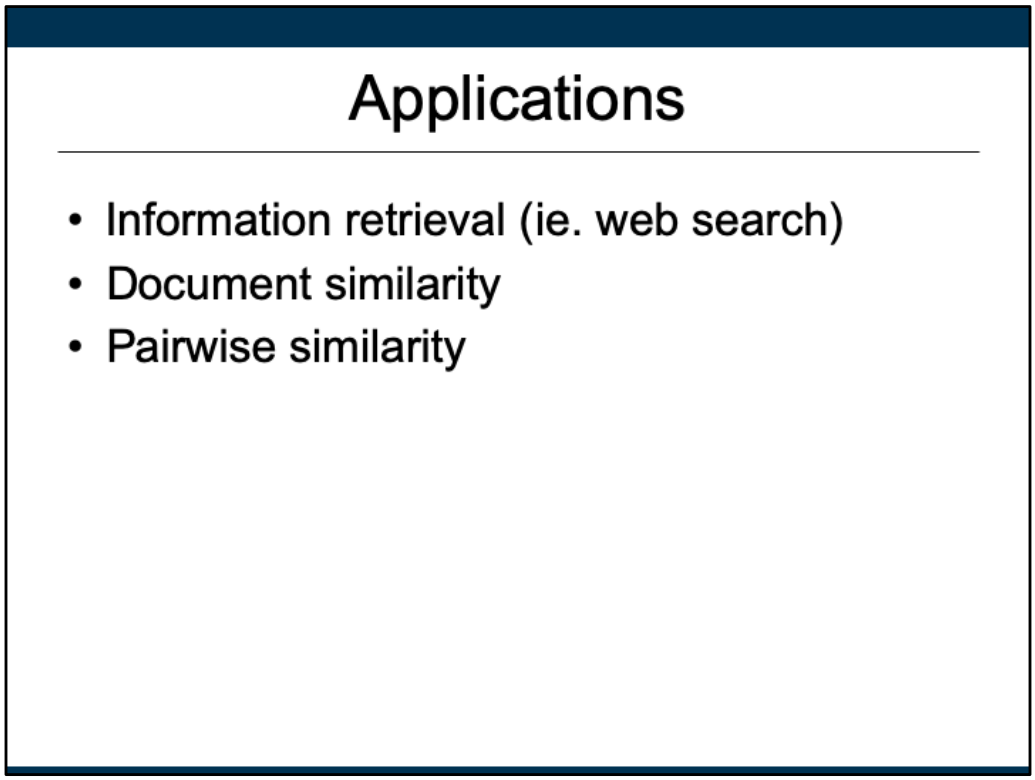
A postings list is comprised of individual postings, each of which consists of a document id and a payload - information about occurrences of the term in the document. The simplest payload is. . . nothing!

For simple boolean retrieval, no additional information is needed in the posting other than the document id; the existence of the posting itself indicates that presence of the term in the document. The most common payload, however, is term frequency (tf), or the number of times the term occurs in the document. ~~More complex payloads include positions of every occurrence of the term in the document (to support phrase queries and document scoring based on term proximity), properties of the term (such as if it occurred in the page title or not, to support document ranking based on notions of importance), or even the results of additional linguistic~~

~~processing (for example, indicating that the term is part of a place name, to support address searches).~~

The size of an inverted index varies, depending on the payload stored in each posting. If only term frequency is stored, a well-optimized inverted index can be a tenth of the size of the original document collection. An inverted index that stores positional information would easily be several times larger than one that does not.

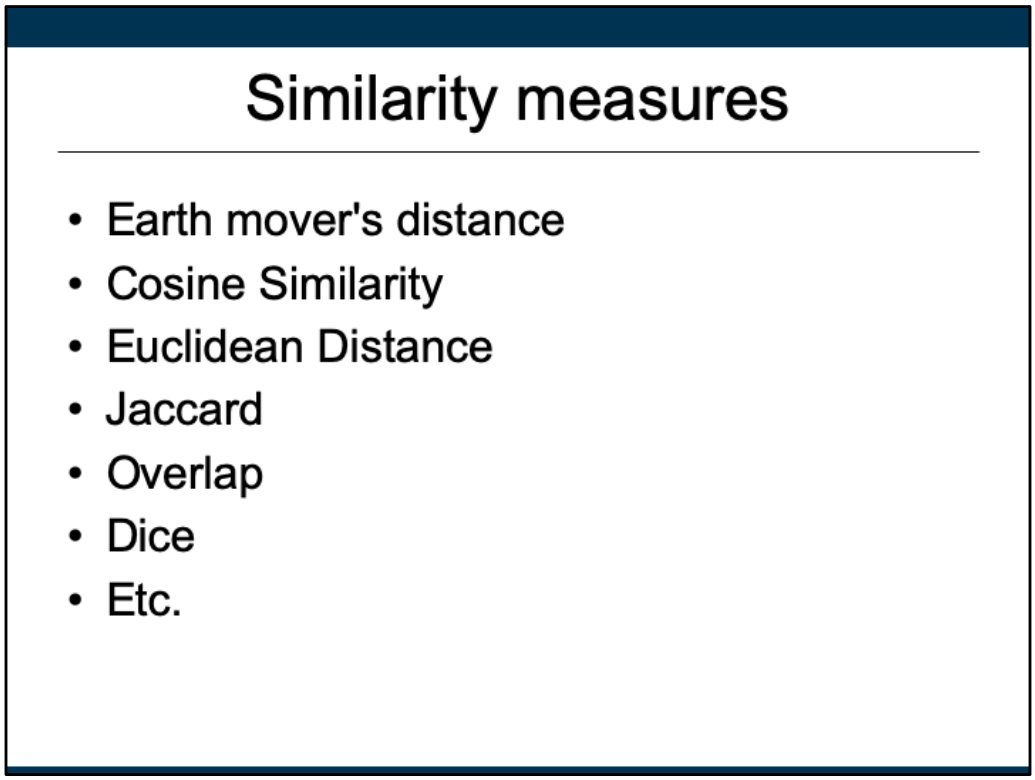
Generally, it is possible to hold the entire vocabulary (i.e., dictionary of all the terms) in memory,



The original and probably most common application of the Inverted Index is in the context of information retrieval, as just discussed.

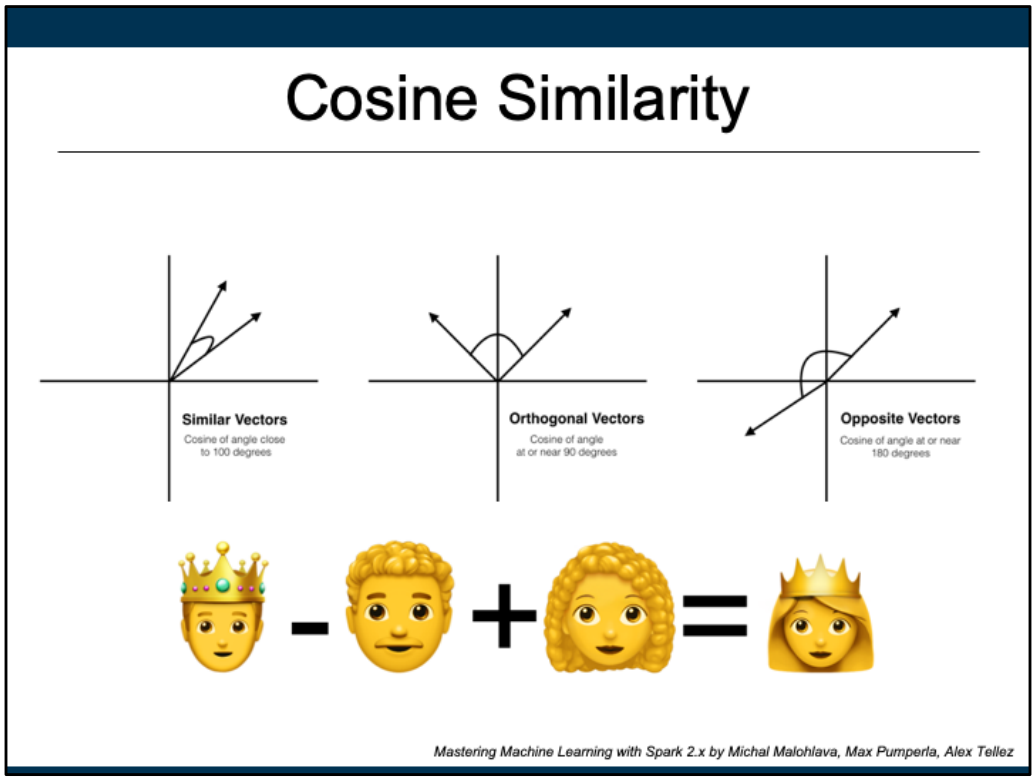
However, the inverted index data structure can be very useful for algorithm design at scale in a map/reduce type framework.

We'll be focusing on similarity calculations for the rest of this section.



Some examples of similarity measures used in Machine Learning are Earth mover's distance, Cosine Similarity, Euclidean Distance, Jaccard, Overlap, Dice.. Each one is sensitive to different aspects of the data.

In Machine Learning, Cosine Similarity is often used in applications such as data mining and information retrieval. For example, a database of documents can be processed such that each term is assigned a dimension and associated vector corresponding to the frequency of that term in the document. This allows for a Cosine Similarity measurement to distinguish and compare documents to each other based upon their similarities and overlap of subject matter.

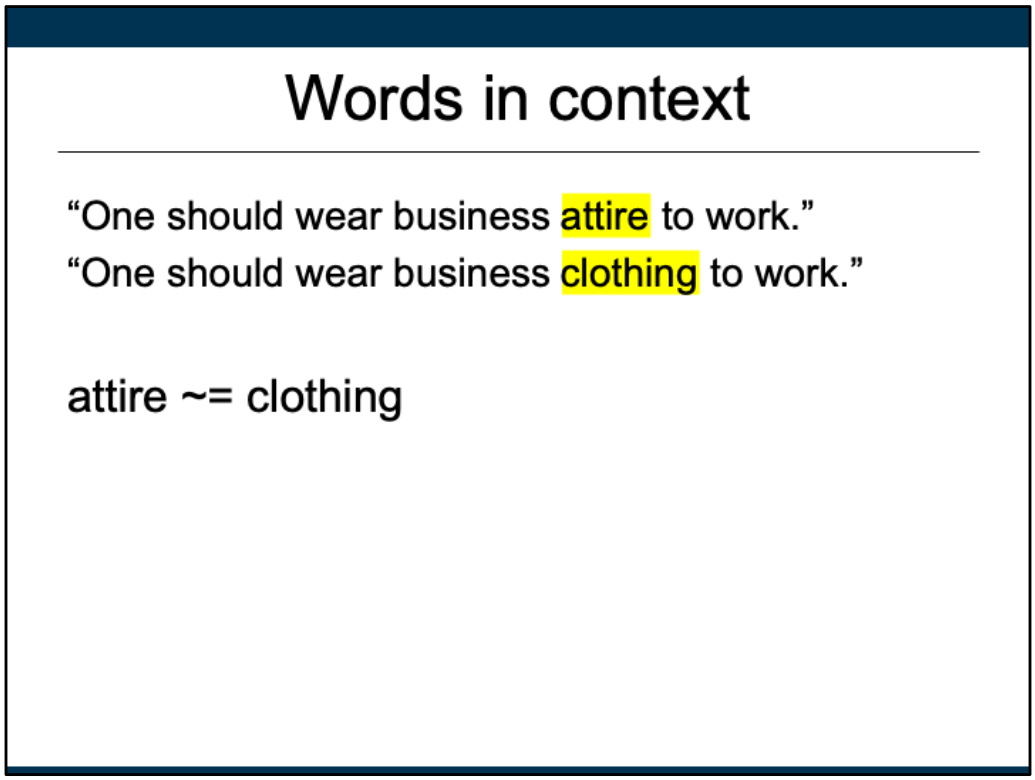


Cosine Similarity measures the cosine of the angle between two non-zero vectors of an inner product space. This similarity measurement is particularly concerned with orientation, rather than magnitude. In short, two cosine vectors that are aligned in the same orientation will have a similarity measurement of 1, whereas two vectors aligned perpendicularly will have a similarity of 0. If two vectors are diametrically opposed, meaning they are oriented in exactly opposite directions (i.e. back-to-back), then the similarity measurement is -1. Cosine Similarity is not concerned, and does not measure, differences is magnitude (length), and is only a representation of similarities in orientation.

Word similarity/dissimilarity is measured via cosine similarity, which has a very nice property of being bound between -1 and 1. Perfect similarity between two words will yield a score of 1, no relation will yield 0, and -1 means that they are opposites.

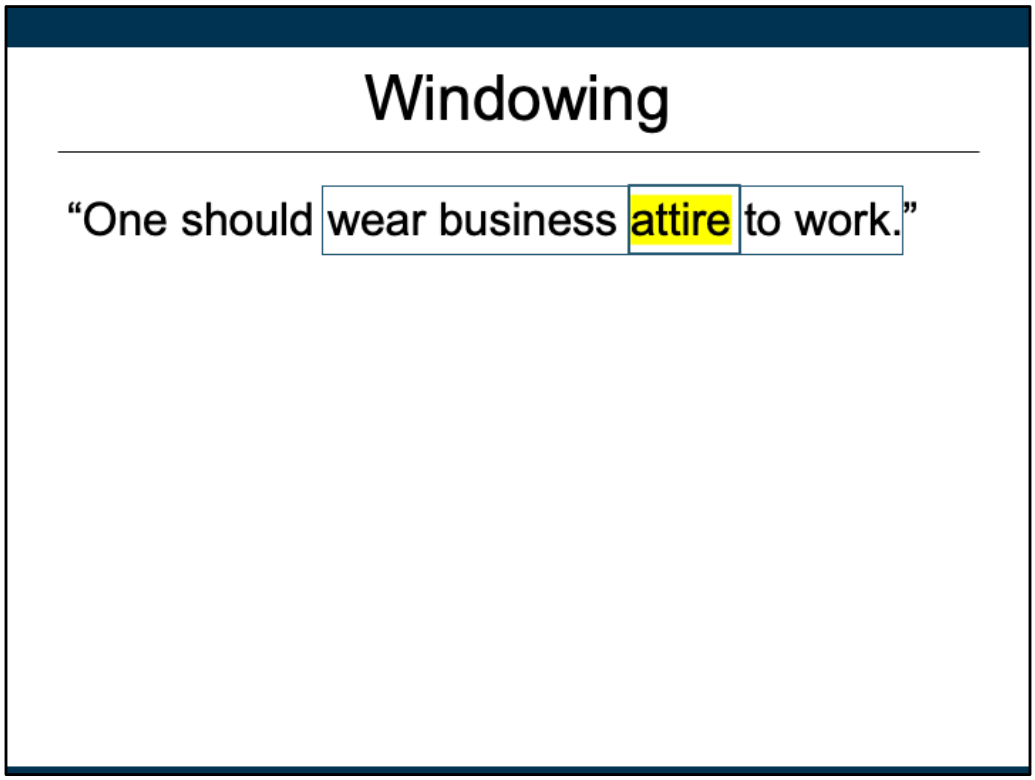
You've probably heard of word2vec for word embeddings as vectors. Cosine similarity is most often used with word2vec. Regardless though of how we generate them, once the words are represented by vectors, the task of finding similar or dissimilar words becomes easier. Any combination vectors result in a new vector and the cosine distances or other similarity measures can be used to calculate similarities between these words. These operations underlie the famous equation 'king - man + woman = queen'.

Note that the cosine similarity function for the word2vec algorithm (again, just the CBOW implementation in Spark, for now) is already baked into MLlib



We're not going to implement word2vec here. However, we'll use cosine similarity to measure the similarity between word pairs, based on the intuition, that if two words occur in similar contexts enough times, then these words are similar. Here's an example.

A single example isn't going to be particularly informative, however, if attire and clothing appear in similar contexts many times, one can infer that they are similar in meaning.



So how do we go about computing these at scale?

Let’s first consider how we’re going to define the context. Let’s say that the context of a word is a five word window with 2 words on the left, and 2 words on the right. In other words, we can say that the word ‘attire’ co-occurs with the words, wear, business, to, and work.

Next, we’ll want to represent these co-occurrences in a data structure that will be amenable to computation.

Matrix vs Stripes

	f1	f2	f3
wear	1	1	1
business	0	1	0
attire	1	0	1

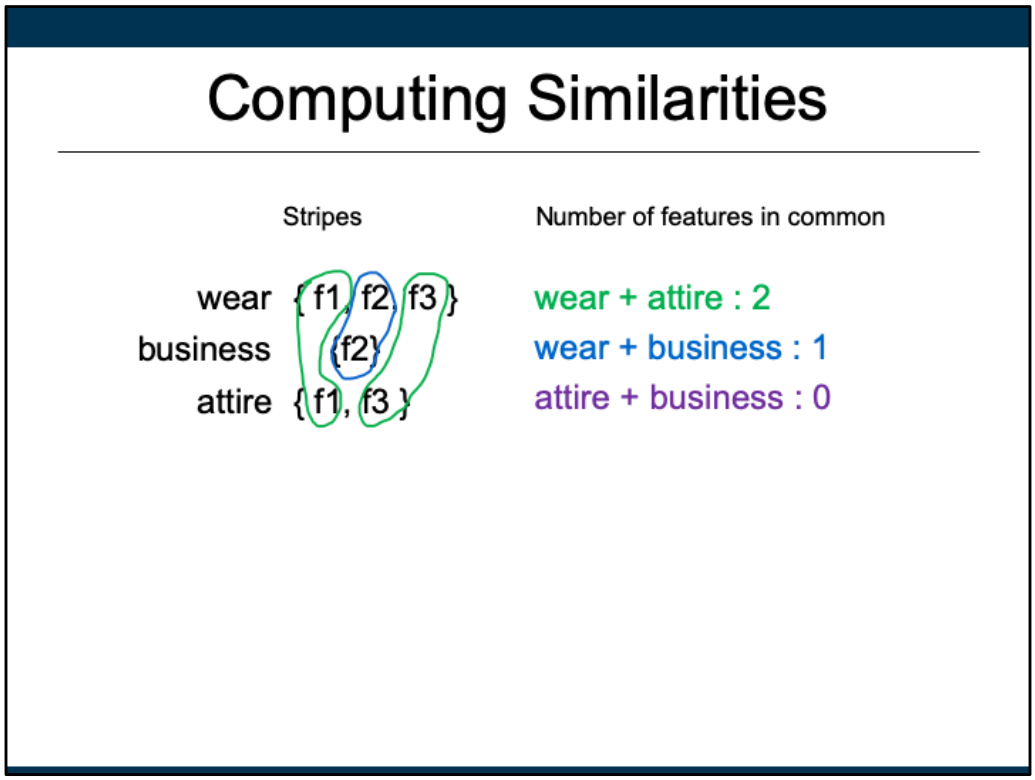
wear { f1, f2, f3 }

business { f2 }

attire { f1, f3 }

Where f1 – fn are the "features", or in other words, the vocabulary

We can construct a binary matrix, where the columns are words from our chosen vocabulary – you can think of these as the features. And rows are words from the corpus. An entry of 1 indicates that the words co-occur in a 5 word window, and a 0 that they do not. This will inevitably result in a very sparse matrix with many 0s which will waste a lot of disk space. A more compact representation like the stripes on the right, is a better format, especially when working at scale.

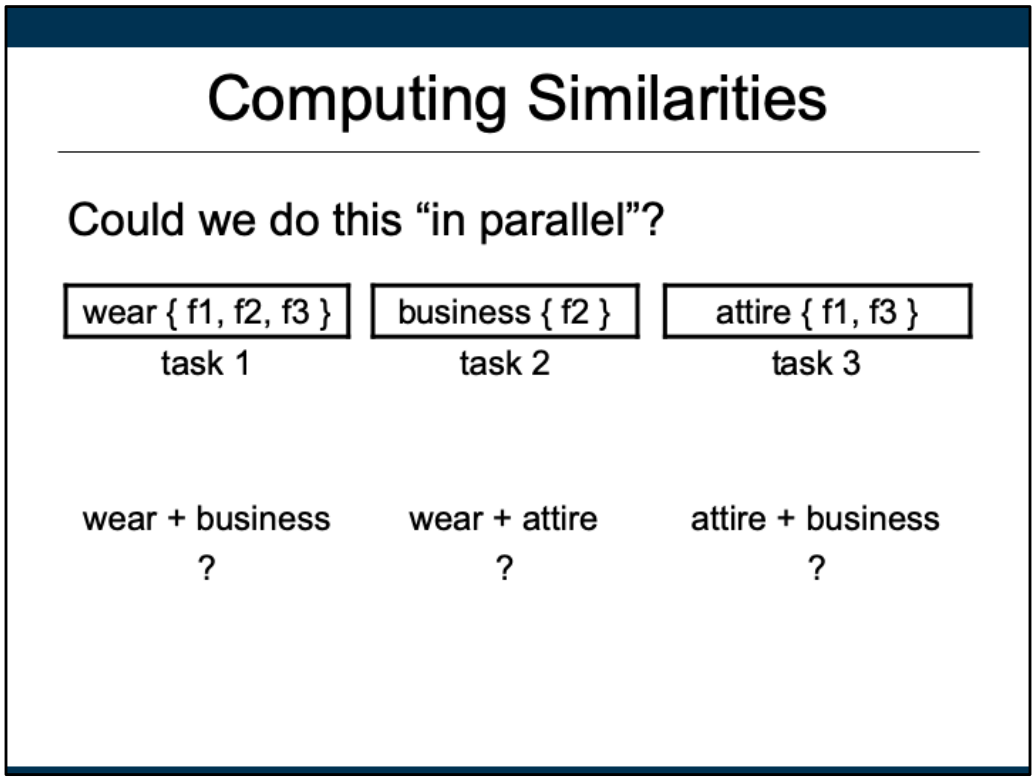


For the sake of simplicity of example, we'll say that words which share the most features are most similar.

Just a note, that this is not cosine similarity. In order to calculate cosine similarity we would need to also include other pieces of information which would just obscure our example here. Bear with me for the time being.

In our small example, the words wear and attire have two features in common, thus their similarity score is equal to 2. wear and business share a single feature, hence the score is 1, and attire and business have no features in common at all.

Notice that in order to count the number of features in common, we need access to both all of the rows as well as all of the columns, and as you can imagine, doing this in memory for a very large dataset is not tractable.



So can we do this in parallel?

Imagine that each stripe resides on its own block, and thus, by default, we’ll have three tasks which should be executed in parallel.

A brute-force approach is possible, but…
Given that each task has no knowledge of any other task, in order to bring the information together, we would need to do some pretty fancy footwork in how we design our keys, but more importantly, we’d have to emit a lot of redundant key-value pairs to the framework, potentially causing expensive spills to memory and/or disk.

Computing Similarities

A third data structure…

	f1	f2	f3
wear	1	1	1
business	0	1	0
attire	1	0	1

matrix

wear	{ f1, f2, f3 }
business	{ f2 }
attire	{ f1, f3 }

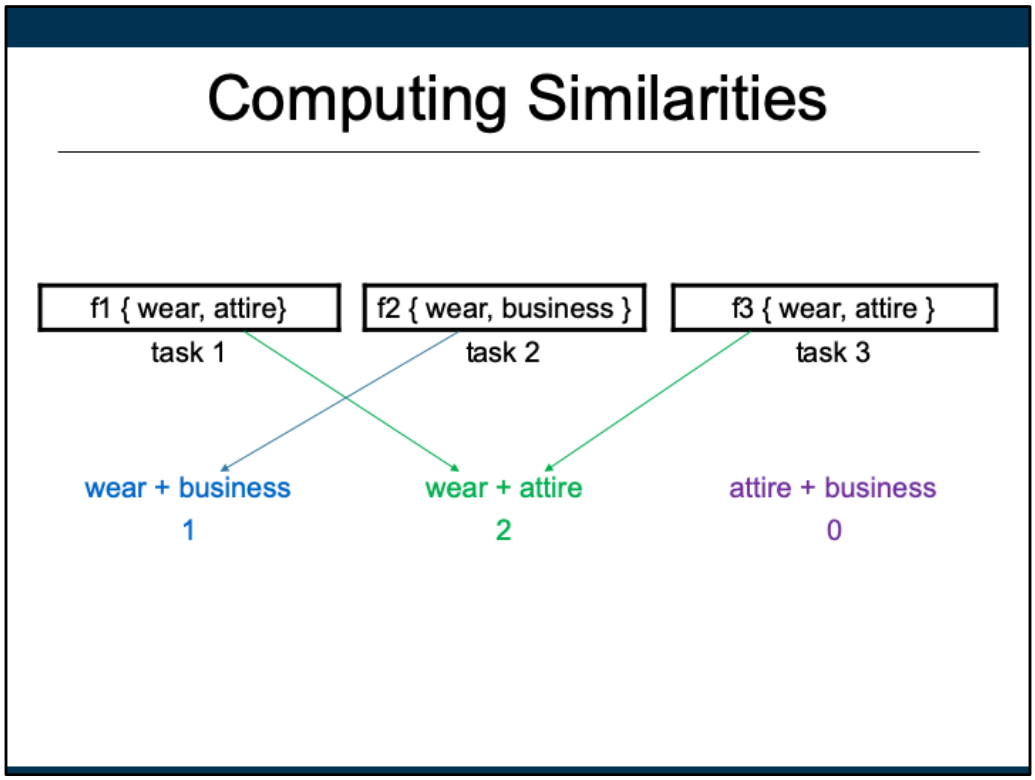
stripes

f1	{ wear:3, attire }
f2	{ wear, business }
f3	{ wear, attire }

inverted index

It turns out there is another way! But you knew this was coming ;)
The inverted index. This is nothing more than the stripes representation of the transpose of our original matrix.

This time the words which share a feature, reside on the same partition.



If each task outputs its own word pair combinations, the reduce phase can simply aggregate these pairs by doing a simple sum. No unnecessary data is emitted into the stream, and the algorithm has a simplicity that is a welcome departure from the potentially complex and error prone composite keys and values from the previous approach.

To calculate actual cosine similarities we'll need to pass along some additional information in the payload from our mappers, but the principle of transposing our original matrix to make this computationally feasible, remains the same.

And with that, you should now be well equipped to complete the homework!

Inverted Index

The End