LEARNED INDEXES: A NEW IDEA FOR EFFICIENT DATA ACCESS

ROBERT RODGER - GODATADRIVEN - 12 JUNE 2018

Thanks for the introduction, [session chair].

[intro]

Hello everyone, and thanks for coming to my talk. My name is

Robert Rodger. I am

Edward Hopper - Hotel by a Railroad

American but I am based in

Amsterdam, where I work for a small data science and -engineering consultancy called

GoDataDriven

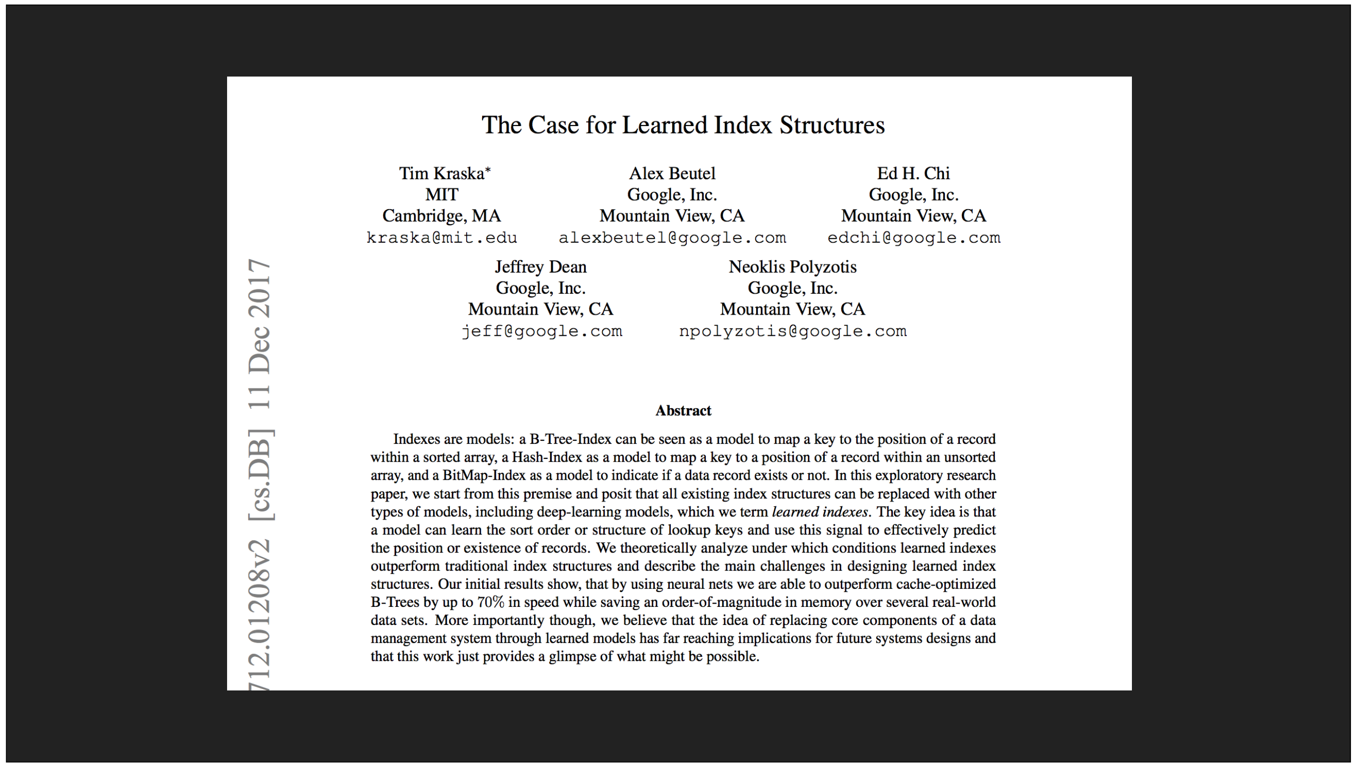
https://godatadriven.com

@GoDataDriven

GoDataDriven. Now, of the two aforementioned job tracks at my place of employment, I am a data scientist - that is to say, I am **not** an engineer. So…

what is a data scientist like me doing on the Store track stage?

Well, the answer is this paper:



712.01208v2 [cs.DB] 11 Dec 2017

The Case for Learned Index Structures

Tim Kraska*
MIT
Cambridge, MA
kraska@mit.edu

Alex Beutel
Google, Inc.
Mountain View, CA
alexbeutel@google.com

Ed H. Chi
Google, Inc.
Mountain View, CA
edchi@google.com

Jeffrey Dean
Google, Inc.
Mountain View, CA
jeff@google.com

Neoklis Polyzotis
Google, Inc.
Mountain View, CA
npolyzotis@google.com

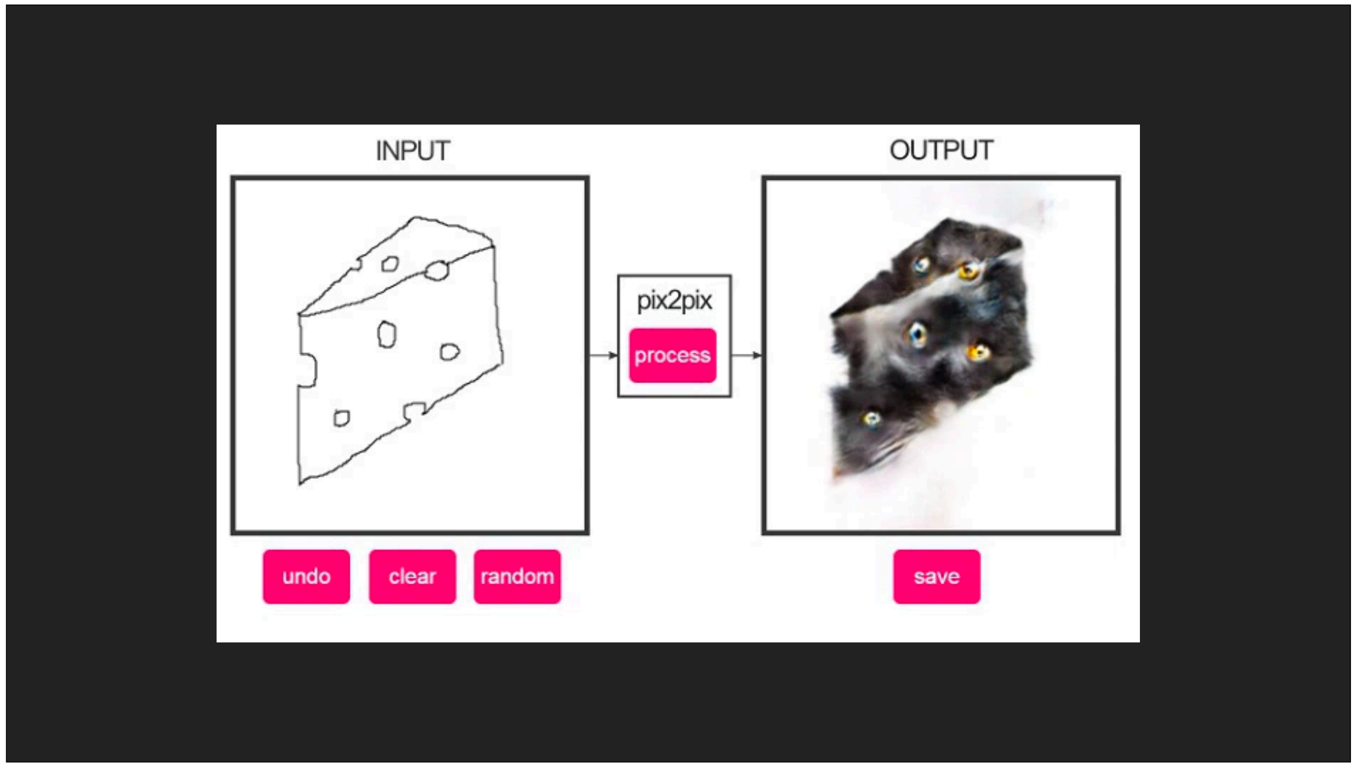
Abstract

Indexes are models: a B-Tree-Index can be seen as a model to map a key to the position of a record within a sorted array, a Hash-Index as a model to map a key to a position of a record within an unsorted array, and a BitMap-Index as a model to indicate if a data record exists or not. In this exploratory research paper, we start from this premise and posit that all existing index structures can be replaced with other types of models, including deep-learning models, which we term *learned indexes*. The key idea is that a model can learn the sort order or structure of lookup keys and use this signal to effectively predict the position or existence of records. We theoretically analyze under which conditions learned indexes outperform traditional index structures and describe the main challenges in designing learned index structures. Our initial results show, that by using neural nets we are able to outperform cache-optimized B-Trees by up to 70% in speed while saving an order-of-magnitude in memory over several real-world data sets. More importantly though, we believe that the idea of replacing core components of a data management system through learned models has far reaching implications for future systems designs and that this work just provides a glimpse of what might be possible.

“The Case for Learned Index Structures". Written by Tim Kraska of MIT together with a team at Google, it made a splash at the data scientist watercooler back in December because it proposed a really novel idea: that machine learning - something we data scientists know and care a lot about - had the potential to replace indexes in certain types of database systems - something **you** hardcore storage systems engineers in the audience all know and care a lot about.

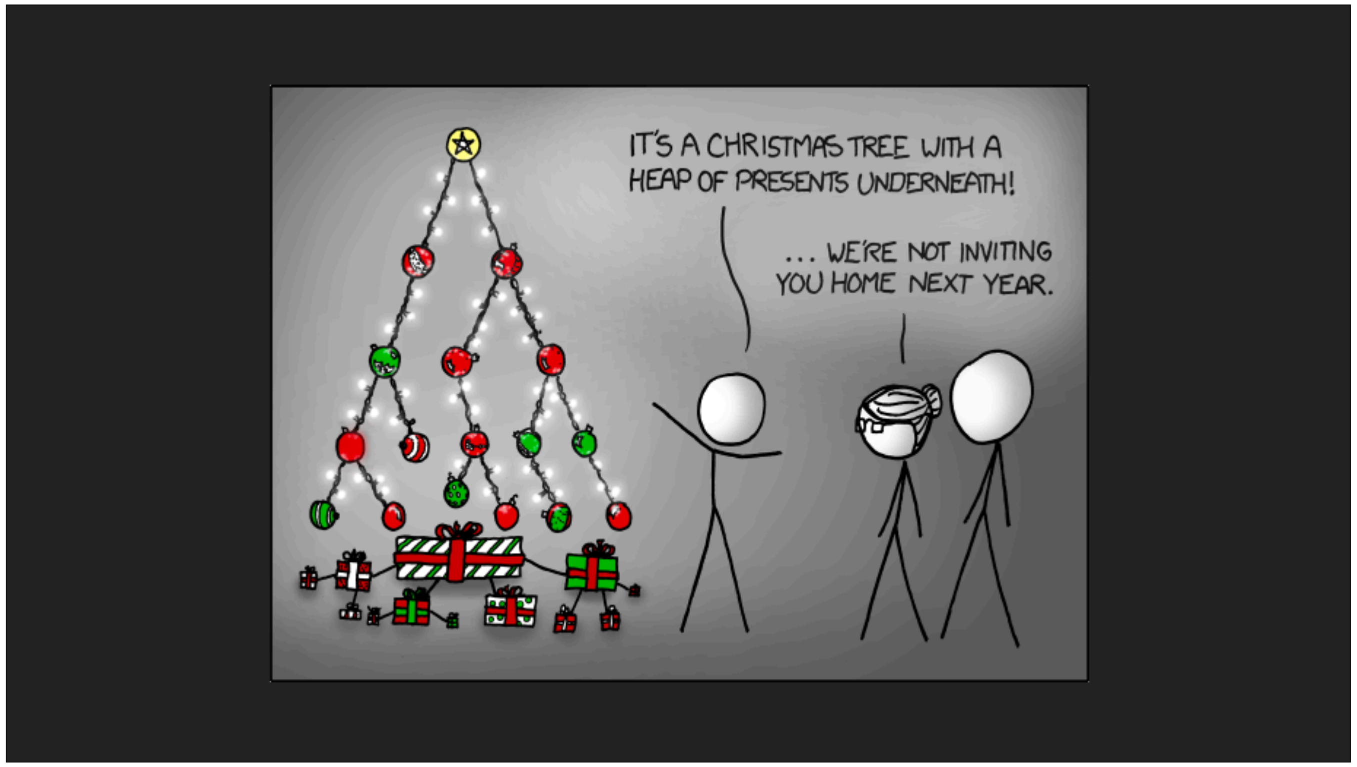
Now, not only was this paper interesting in a general sense because at first glance the use cases don't seem to match up:

on the one hand, you have



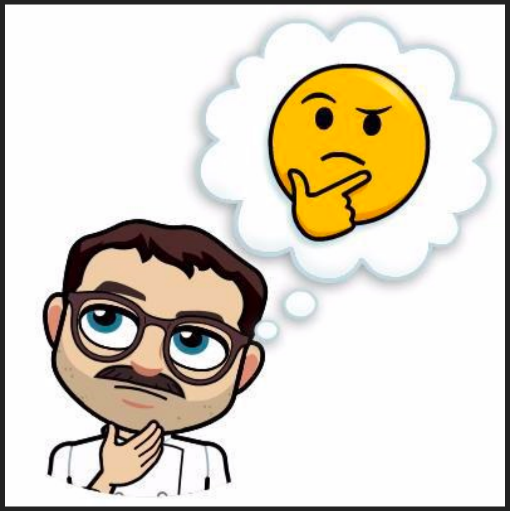
statistical inference, which is for learning patterns for purposes of make predictions based on future input,

and on the other you have


IT'S A CHRISTMAS TREE WITH A HEAP OF PRESENTS UNDERNEATH!
... WE'RE NOT INVITING YOU HOME NEXT YEAR.

data structures, which are useful for optimized lookup of already-seen information,

but it was also interesting for



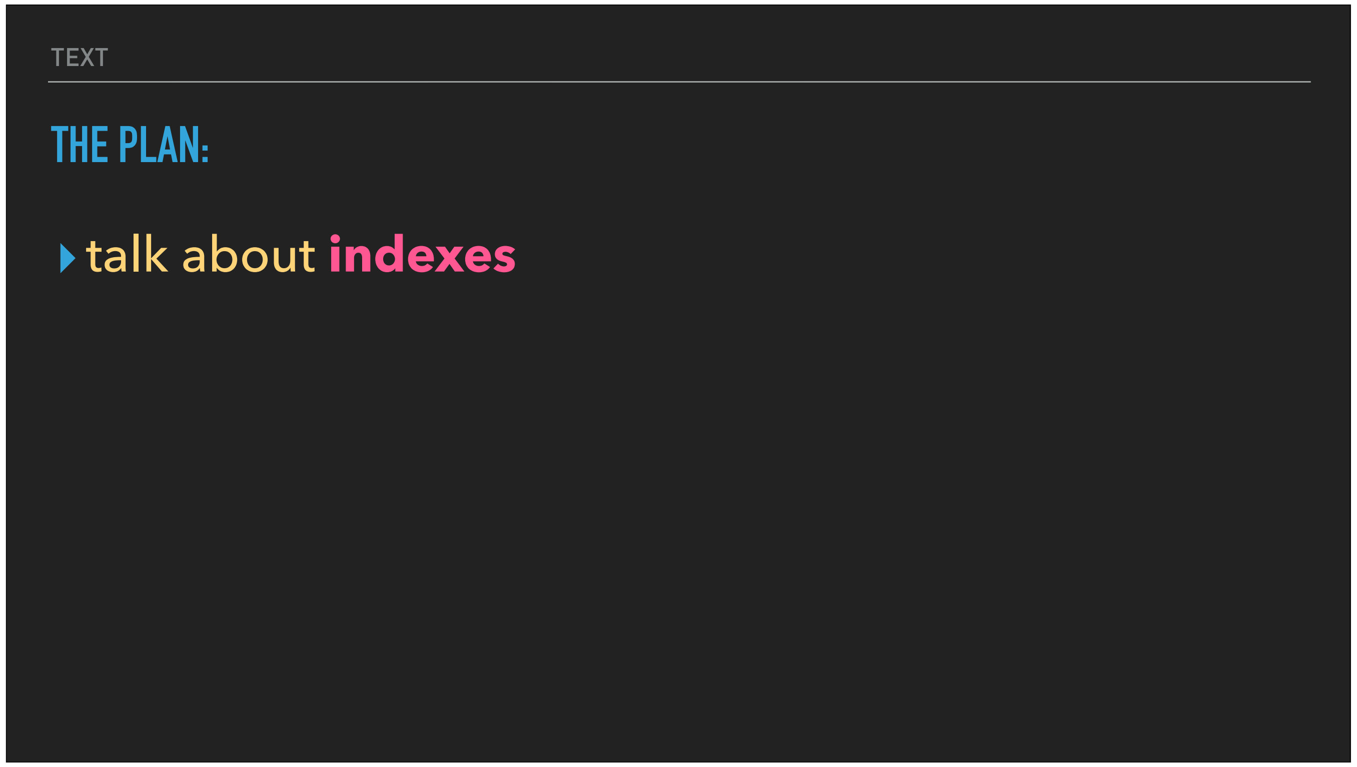
me, personally, because, while I feel like I know a fair amount about machine learning, despite their cruciality in my daily professional life, I knew next to nothing about database internals, and wanting to understand this paper gave me a good excuse to dive in and learn all I could about the subject.

So I want to share with you this idea, and let's hope Doug Turnbull had it right yesterday when he said these sorts of autodidactic experiences tend to lead to great talks.

In any case,

THE PLAN:

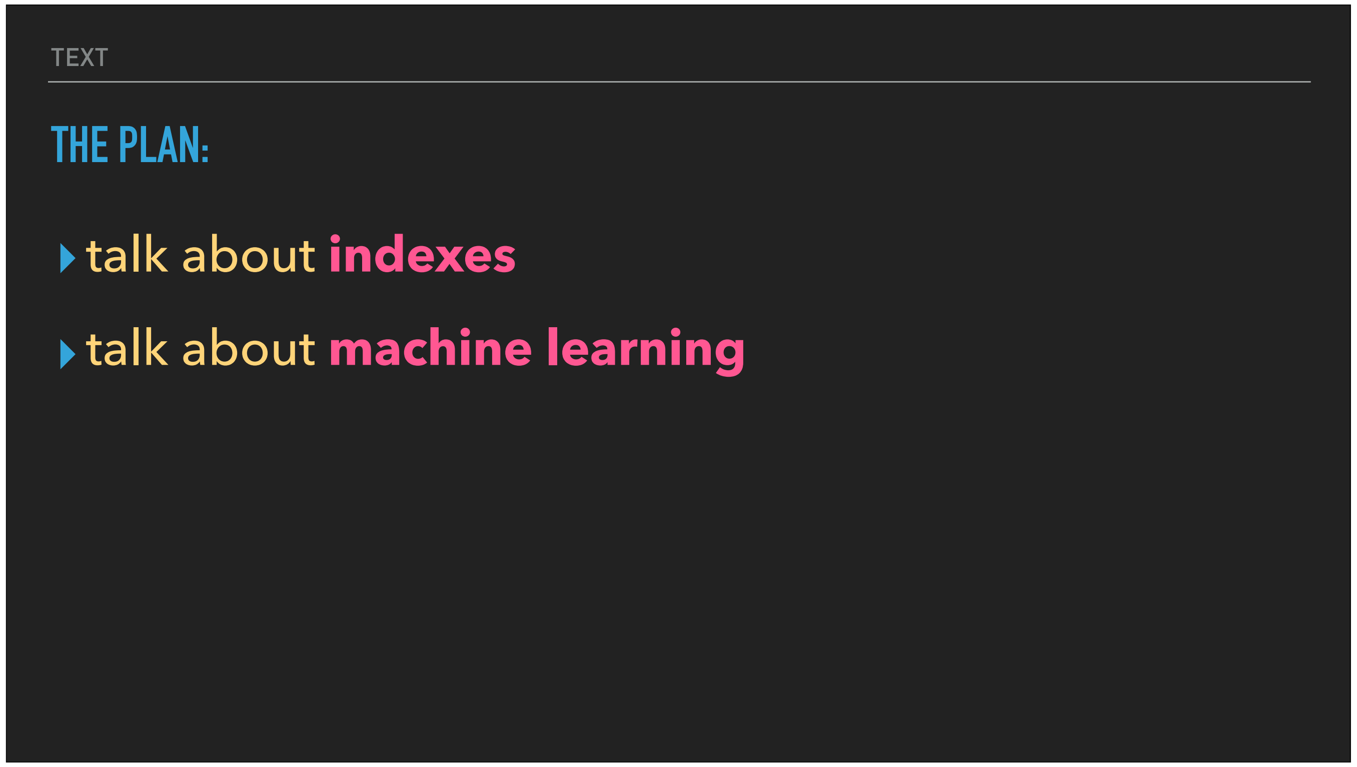
here’s the plan:


TEXT

THE PLAN:

- talk about **indexes**

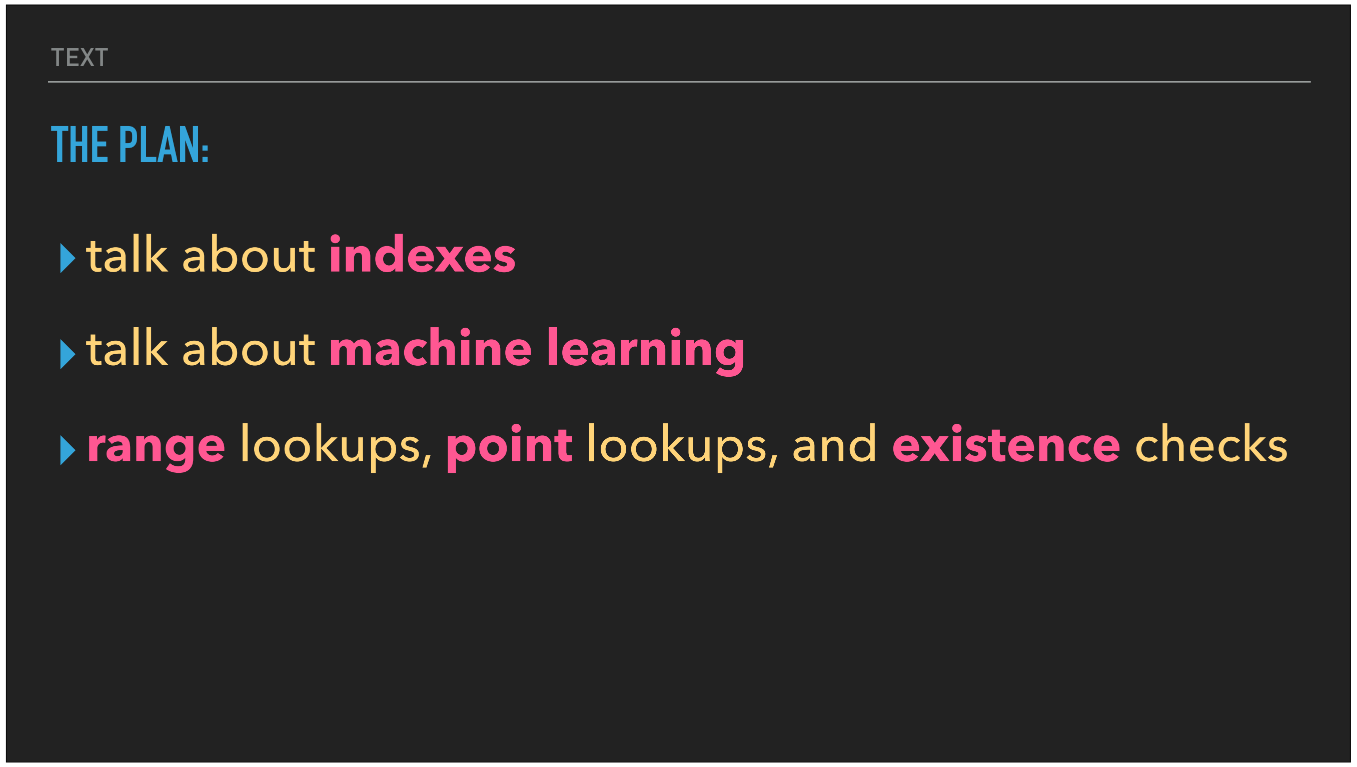
first, i'm going to talk about database indexes, and while i’m sure this will be old hat to most of you in the room, i want to do it anyway, both to ensure everyone is on the same page and also to reframe how we think about just what it is that database indexes do,



THE PLAN:

- talk about **indexes**
- talk about **machine learning**

second, i'm going to talk on a relatively high level about what machine learning tries to accomplish and how this can be adapted to the database index domain,



THE PLAN:

- talk about **indexes**
- talk about **machine learning**
- **range** lookups, **point** lookups, and **existence** checks

and lastly, i'm going to show how machine learning can be utilized to replace database indexes in three different types of tasks: range requests, point lookups, and existence checks.

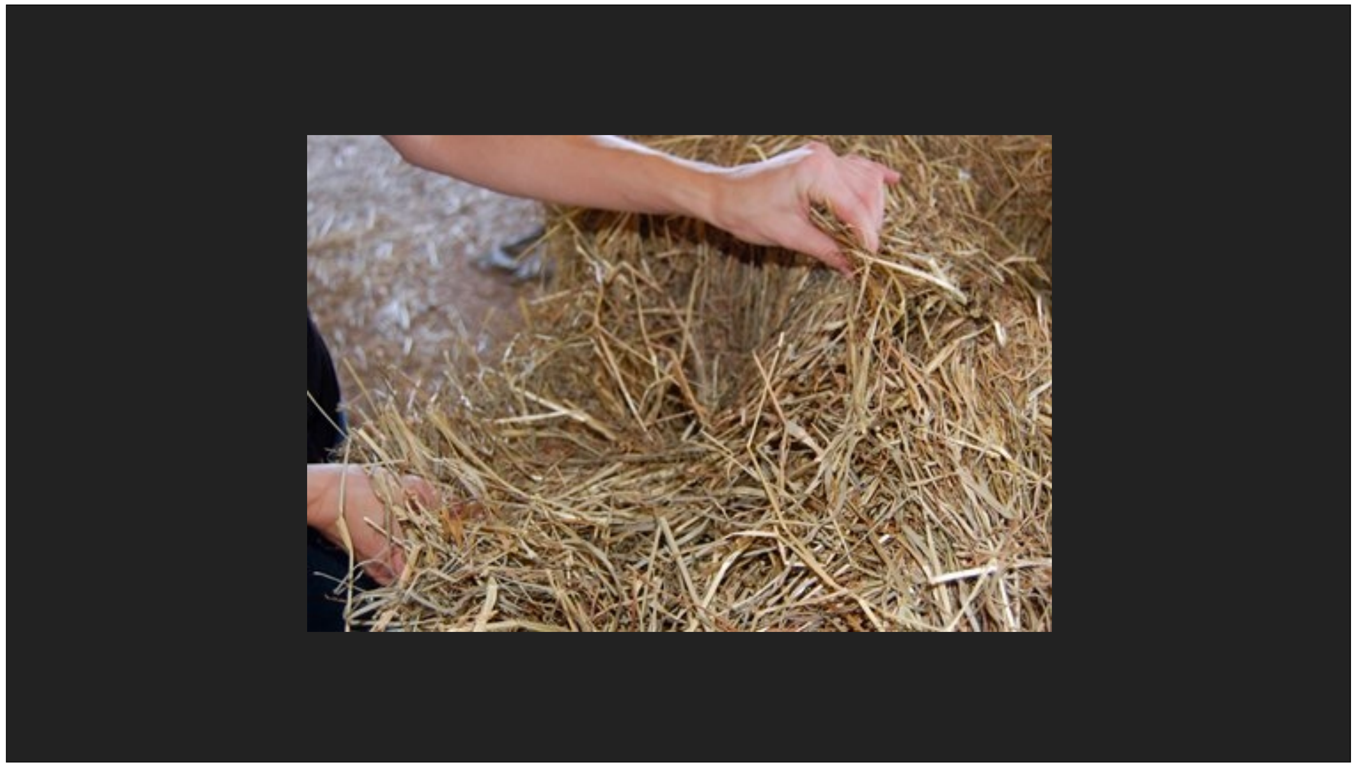
everybody ready? so let's go.

[1]

Let's start with an analogy. A database is like

a haystack (and I hope the analogy is not too English-centric for this crowd). As these databases can consist of thousands, or millions, or even billions of records, I think that it would not be unfair to say that the challenge of finding specific records in our database is very much like finding needles in a haystack.

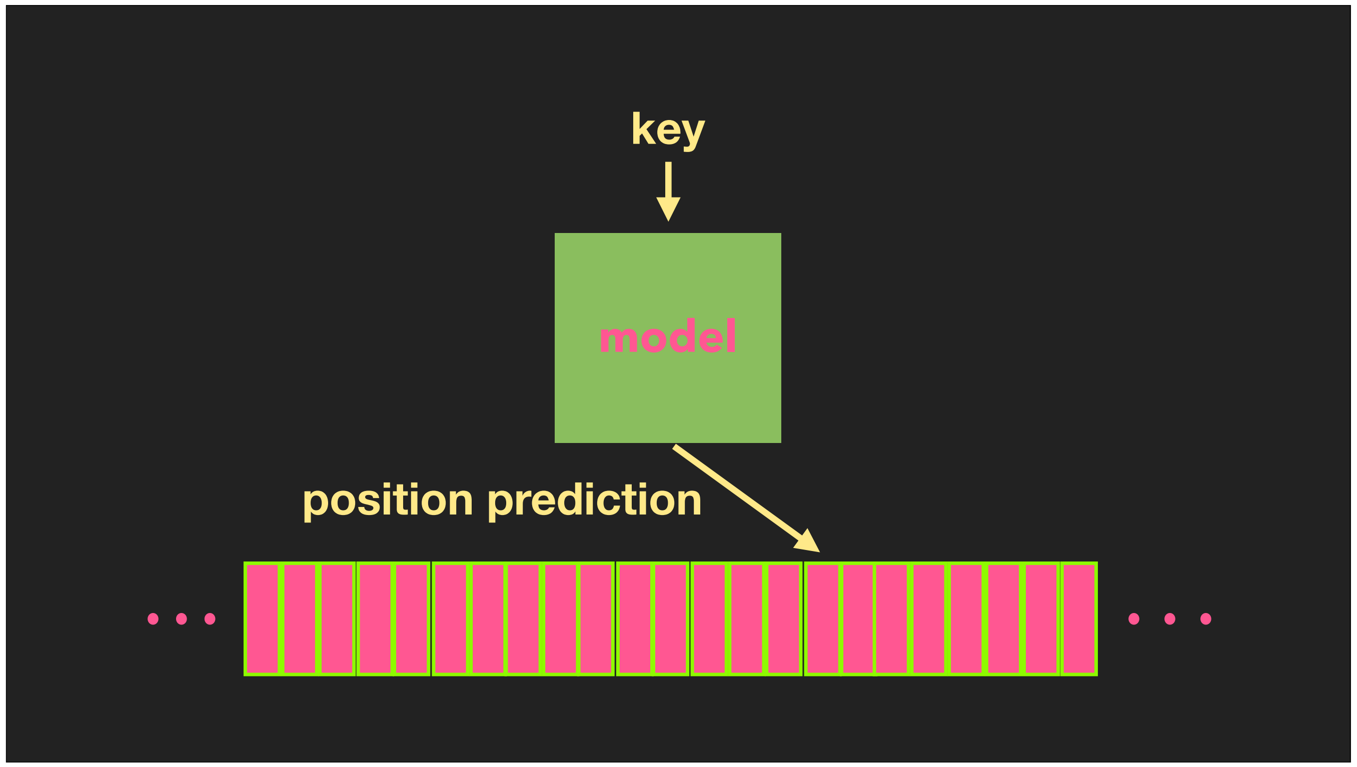
Now following the naive approach, every time we need to look up a record in our database we would have to



comb through our haystack one entry at a time until we either find what we were looking for, or, convince ourselves that what we're looking for is not to be found. This might be acceptable for databases with a few hundred records, but even at moderate scales, waiting for answers from our database, whether it be me doing some sort of analysis or you trying to populate the catalog of your online retail shop, would take too much time. What we would rather have is some means of knowing precisely where our desired record is stored in that database and then

skipping over all of the other records to touch just that one.

Because of this, database systems often come paired with an ancillary system, called an **index**, whose function it is to tell us more or less where in the database our desired record is, and to do so in an efficient manner. How it works is roughly as follows:



say each record in our database is identified via a unique key; an index is a black box or - and this is the insight we'll need for part two - a **model**, where the unique key goes in and a prediction for the position of the associated record comes out.

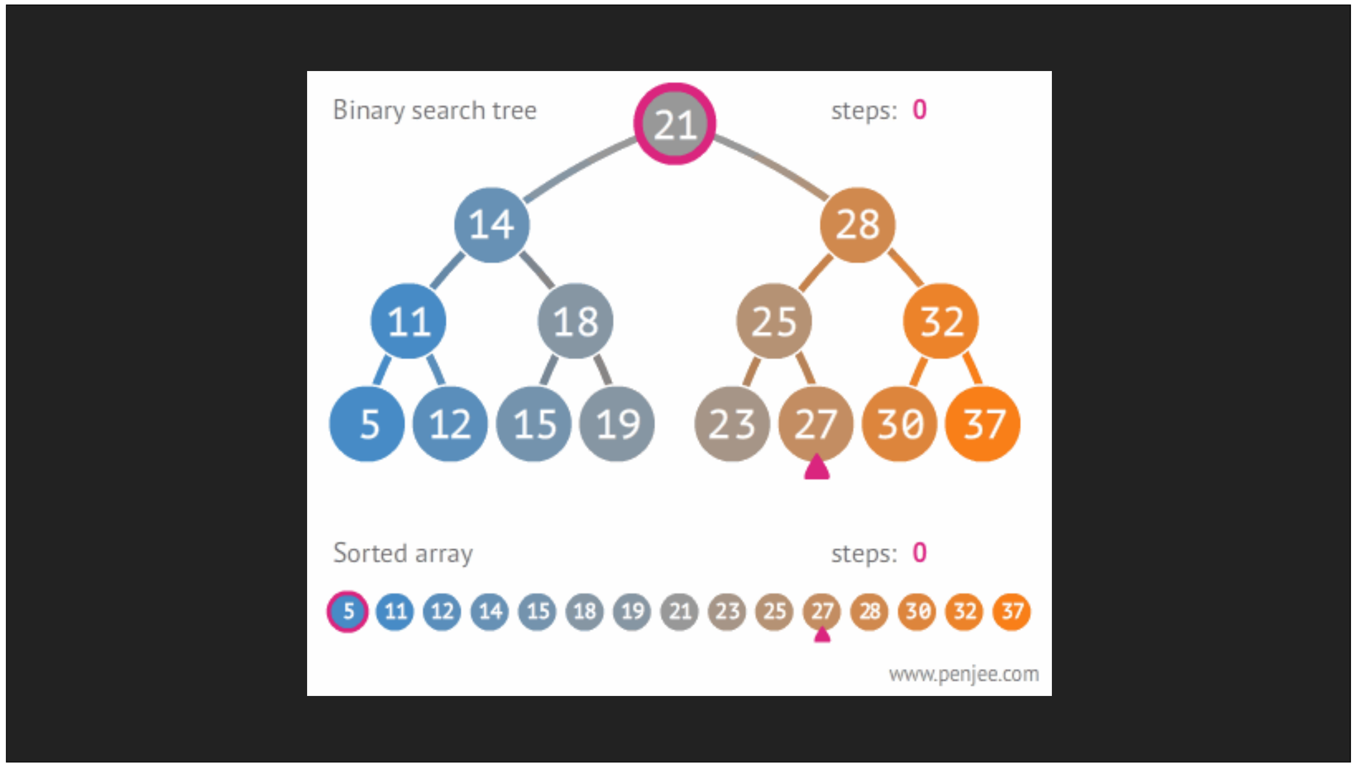
(This is very much like how

By Dr. Marcus Gossler - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=483741

card catalogs in the library used to work, for those of you like me old enough to remember having to do high school research projects using one.)

Now, that's a rather abstract view, so we should ask ourselves: if we were to actually implement such a system, what types of black boxes or models could be used? And, to begin this discussion with a concrete task, what could we use to handle range requests - that is, locating all the records whose keys fall between two specific values?

Anyone who's ever taken an algorithms course or recently had to sit through a tech interview would probably think: well, if I can order the records, then I could probably use some sort of binary search to find each of the endpoint records on either side of my range and then grab everything in between. At least, as a first guess. So let's follow up on that hunch: we want to facilitate binary search, so for our model let's use a binary search tree.

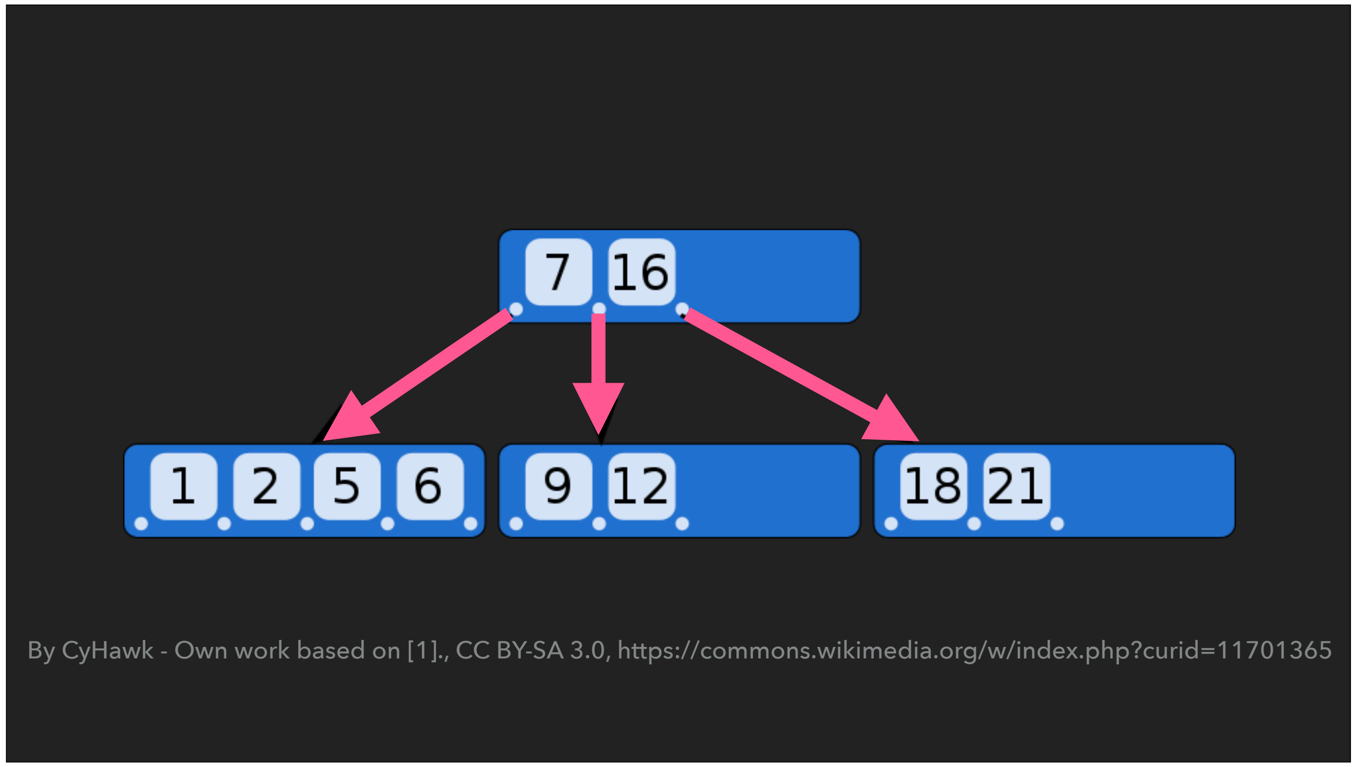


For those of you who haven't seen one before, the way a binary search tree works is as follows:

for each record key, you make a **node**, which is uniquely identified by the same id as the record and also carries the position of the record in the database. These nodes are stored in a tree structure, where every node can have at most two children, which themselves are trees, and you have requirement that all the keys in the left child tree will be smaller than the current node's key, and those in the right child tree will all be larger than the current node's key. And if you further use one of the variations on a binary search tree that do their best to keep the bottommost nodes all at the same depth, you can get a guarantee (as this animation is attempting to demonstrate) that the maximum number of nodes you have to examine during a lookup is O(log_2 N), where N is the number of keys or records in your database. For example, for one million records this is 20 and for a billion it's only 30.

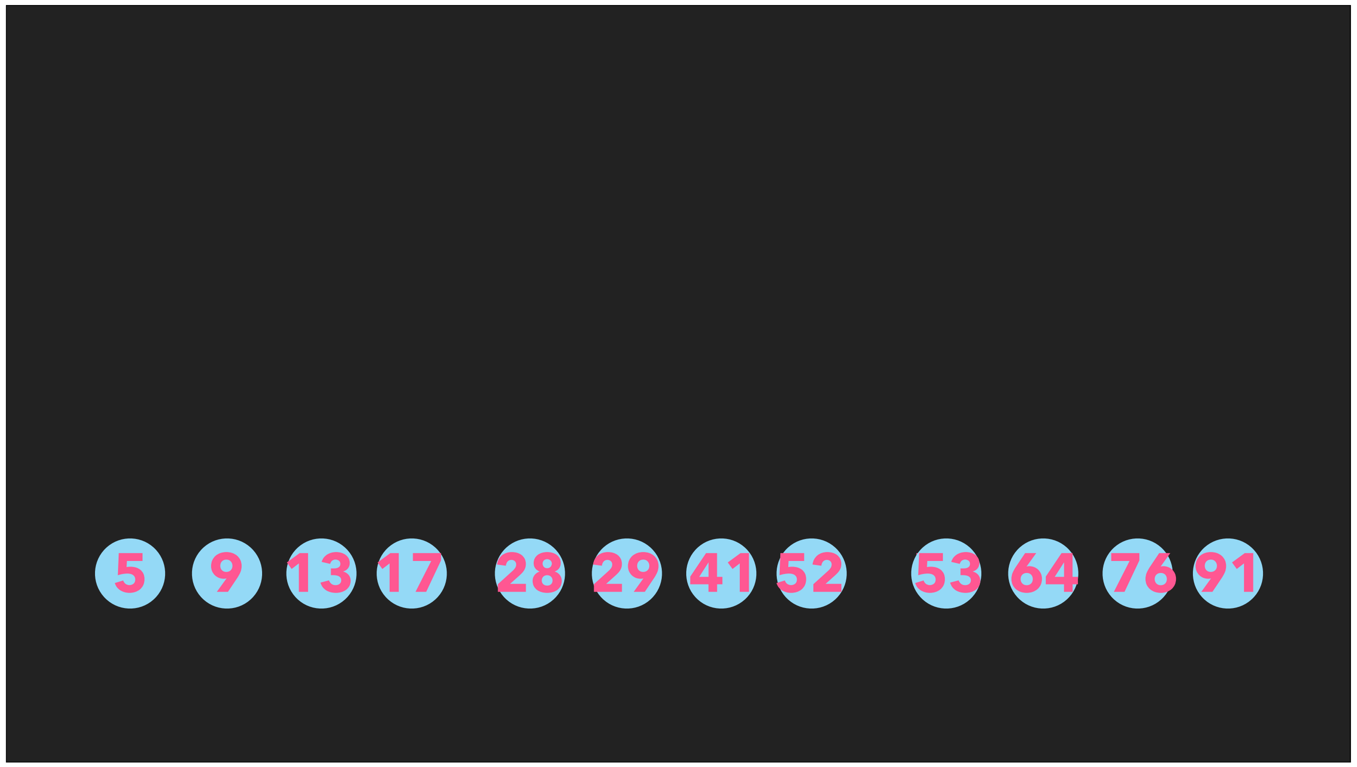
!

This seems like a nice improvement over brute search! And you might rightfully ask yourself: well, why stop at two children per node, or one key per node? And if you continue with that line of thinking (and again think up some clever rebalancing rules to maintain an even tree depth), you'll eventually discover improvement two, the B-tree, seen here.

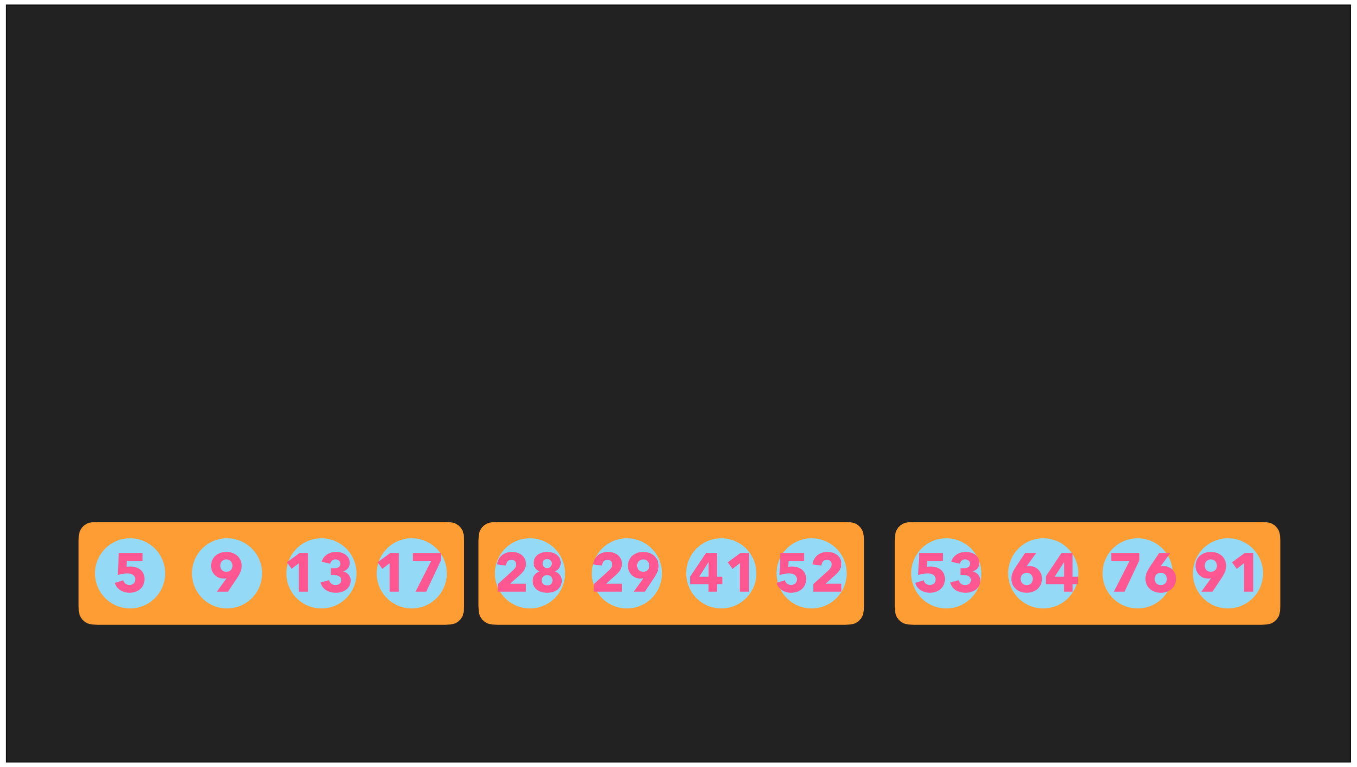

7
16
1
2
5
6
9
12
18
21
By CyHawk - Own work based on [1]., CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=11701365

Ignoring the costs of scanning the keys inside of an individual node (which, because the number of keys in a node is much, much smaller than the total number of keys in our database, should be justifiable), we've now guaranteed ourselves lookup times on the order of O(log_k N), where k is the number of node children and again N is the number of records in our database, which means that if we go back to our previous figures, with a 100 children to a node, for a million records we're down to a tree depth of 3 and for a billion we have a solid **5**.

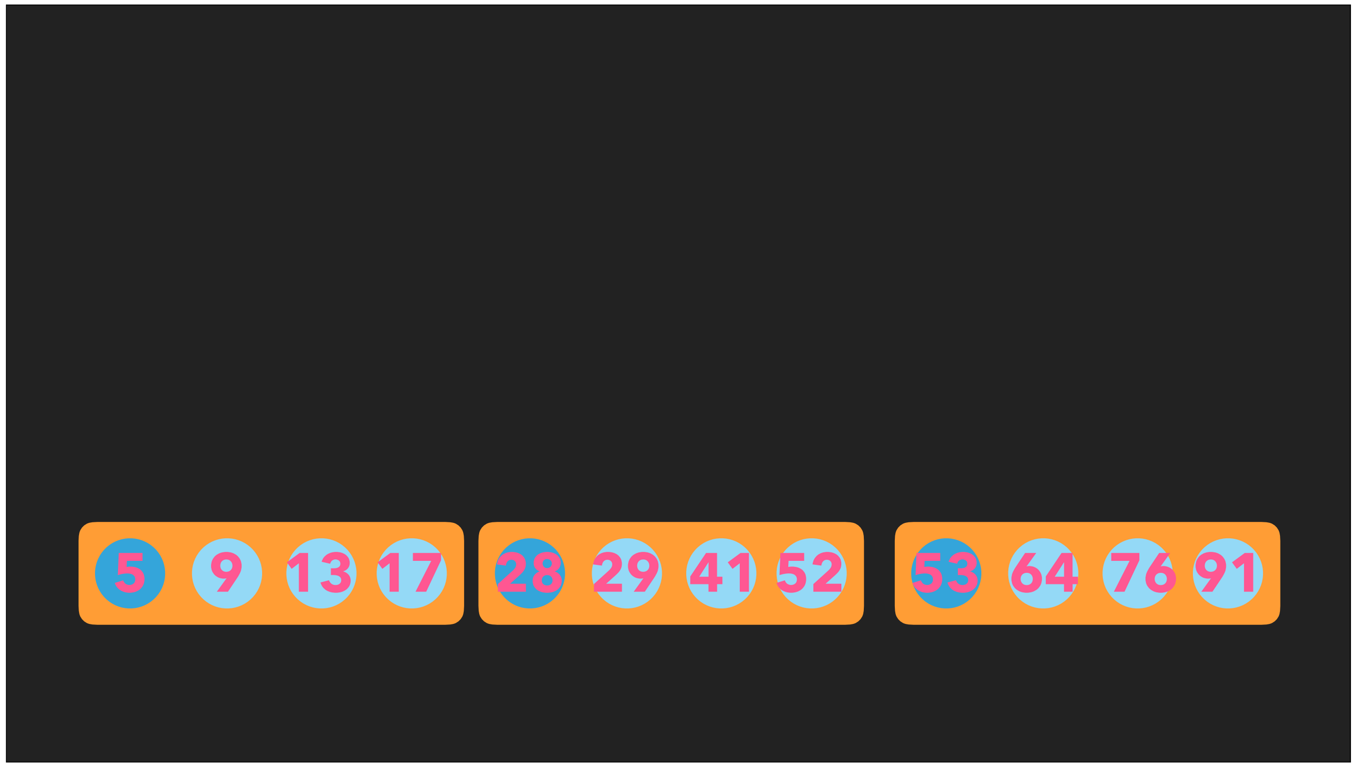
With this, we seem to have optimized the lookup complexity aspect of the problem. But having a node for every record seems like overkill and likely there is room for optimization in terms of space requirements. This leads us to the third and final improvement we can make, which works as follows:



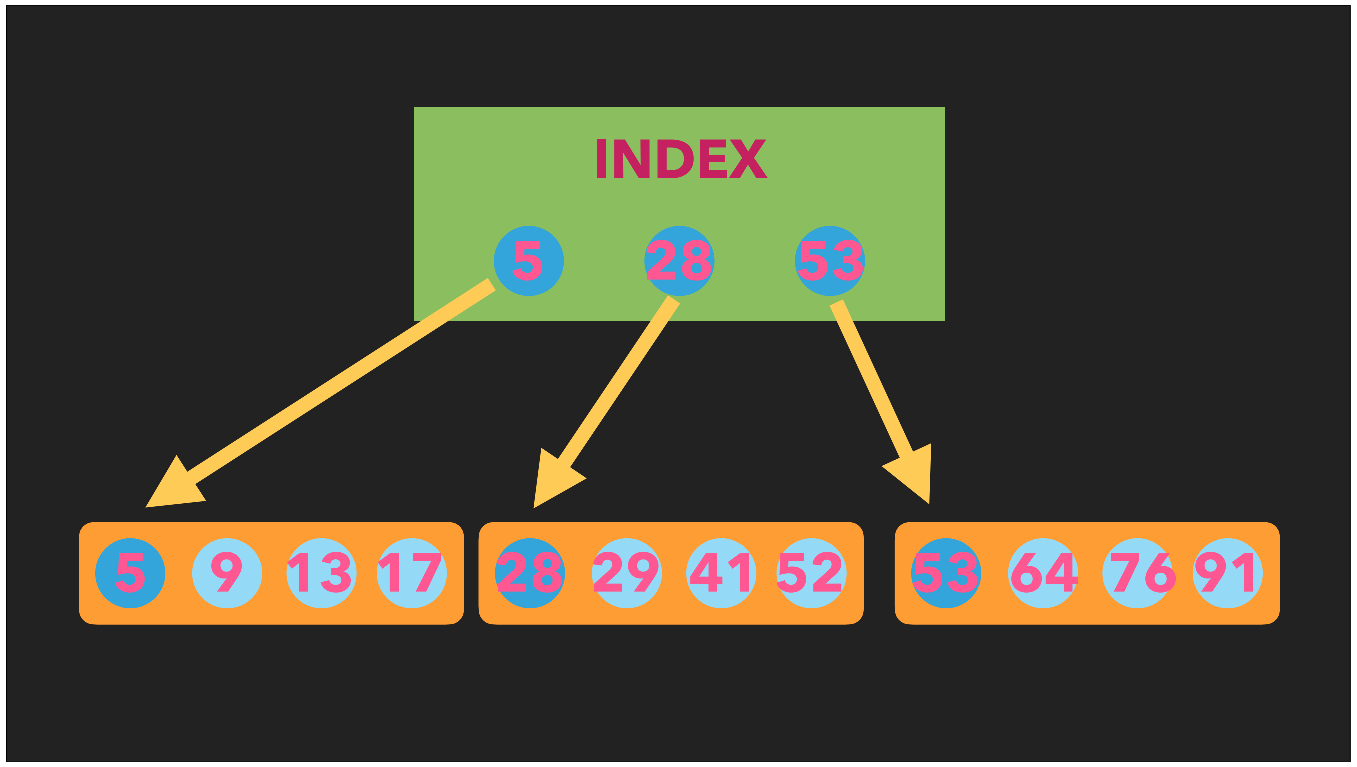
we take our sorted records and divide them up into continuous groups, called



pages, of a fixed size, called the **pagesize**. We then store in our index, not every key of every record, but only

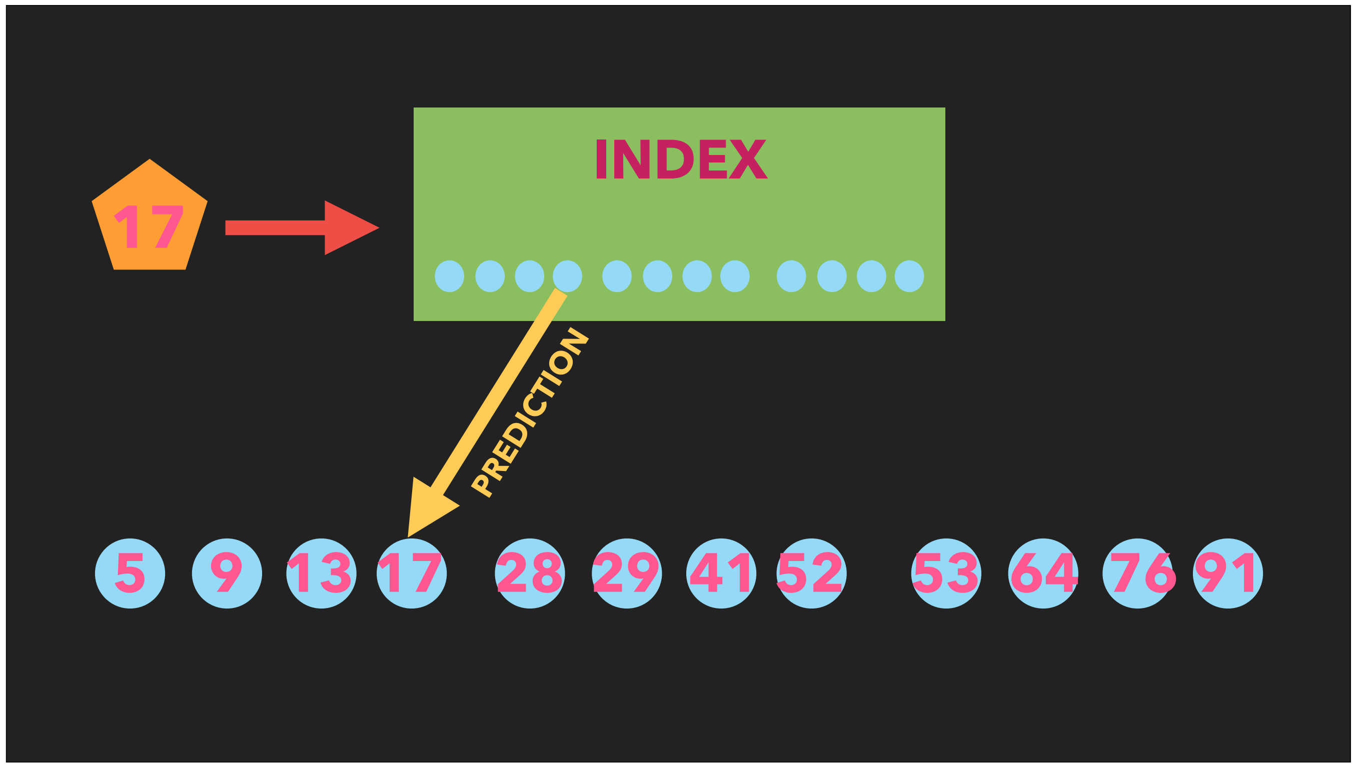


the first key of every page. This allows a great deal of space efficiency, and although our index now

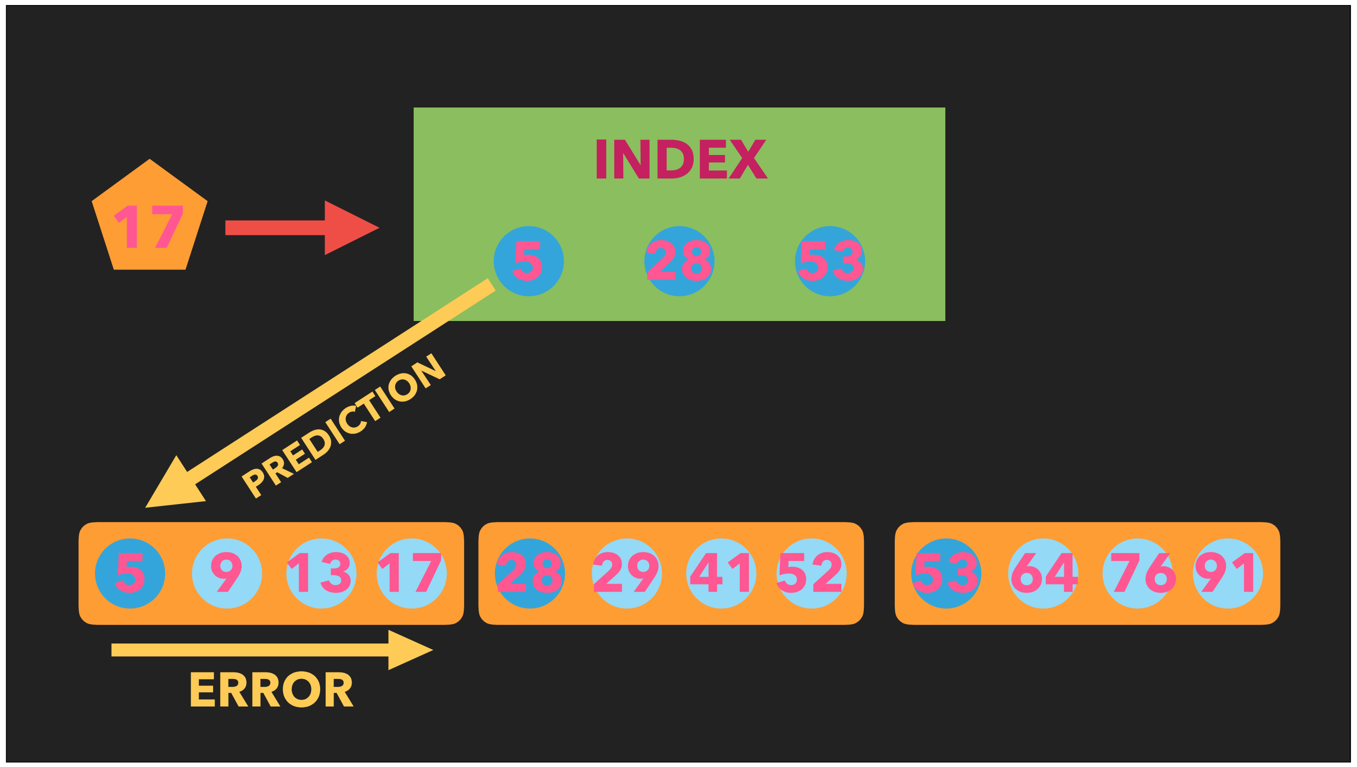


only points to a page and not a record - meaning after descending the index we still have to perform a search on the page - since we choose the pagesize to be tiny in comparison to the size of the number of records, on the balance of increased computation versus decreased storage, we still come out in the black. And with this third improvement, we now have, modulo some optimizations involving a cache, arguably the most common type of range index out in the wild.

And it's this last improvement, making our index sparse, that I think really makes the analogy of index-as-model work. And the reason is, when we think of the word model, we usually think of something that makes predictions, and where these predictions have associated errors. With binary search trees and B-trees we had the notion of



prediction - that is, where we could find the record - but now with sparsity our model predictions gain the notion of



error, as the prediction is not of the exact location of the record but of its page. Further, these errors come with hard guarantees - after all, the record, if it's in the database, is definitely not to the left of the first record in the page and is definitely no more than pagesize records to the right.

This is, all in all, a very nice system, in that we’ve got hard guarantees on both prediction compute complexity and error magnitude. And seeing as how B-trees have been around since

1971

1971, surely, one would think, nothing could work better, as otherwise that newer technology would have long ago replaced the B-tree as the model of choice for range indexes. Except that they are **not** necessarily the best option out there. Here's a

counterexample

simple counterexample (and this is the one given in the paper): say your records were of a fixed size and the keys were the continuous integers between 1 and 100 million - then we could have constant-time lookup simply by using the key as an offset.

And of course this not the most realistic example, but it serves to illustrate the following point:

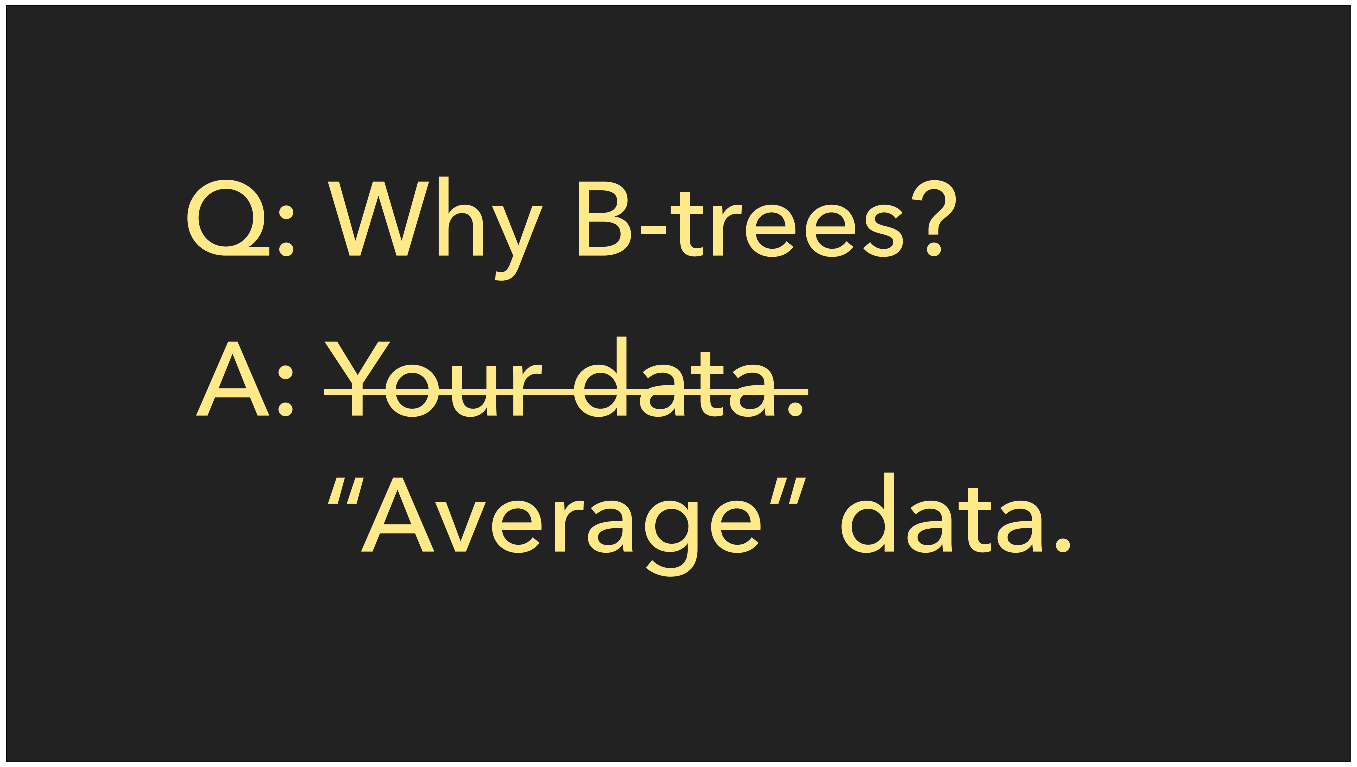
Q: Why B-trees?

the reason B-trees are so widespread in generally-available database systems is

Q: Why B-trees?

A: Your data.

not because they're the best model for fitting **your** data distribution,



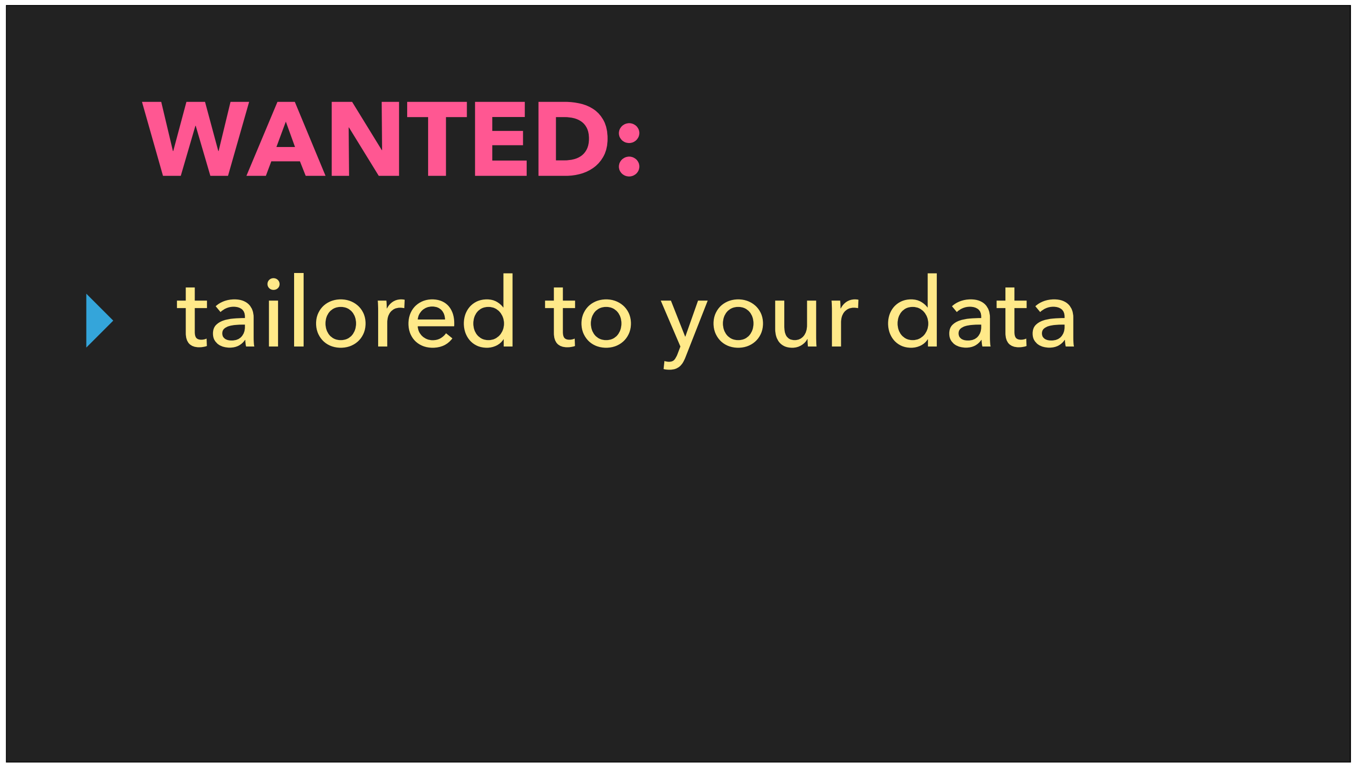
Q: Why B-trees?

A: ~~Your data.~~
"Average" data.

but because they're the best model for fitting unknown data distributions. **Of course**, if your database engineers were to know your exact data distribution was, they could engineer a tailored index, but this engineering cost would likely be too high for your project and would be unrealistic to expect from a database available for general use (think: your Redises, your Postgreses). Which leads us to the following wanted ad:

WANTED:

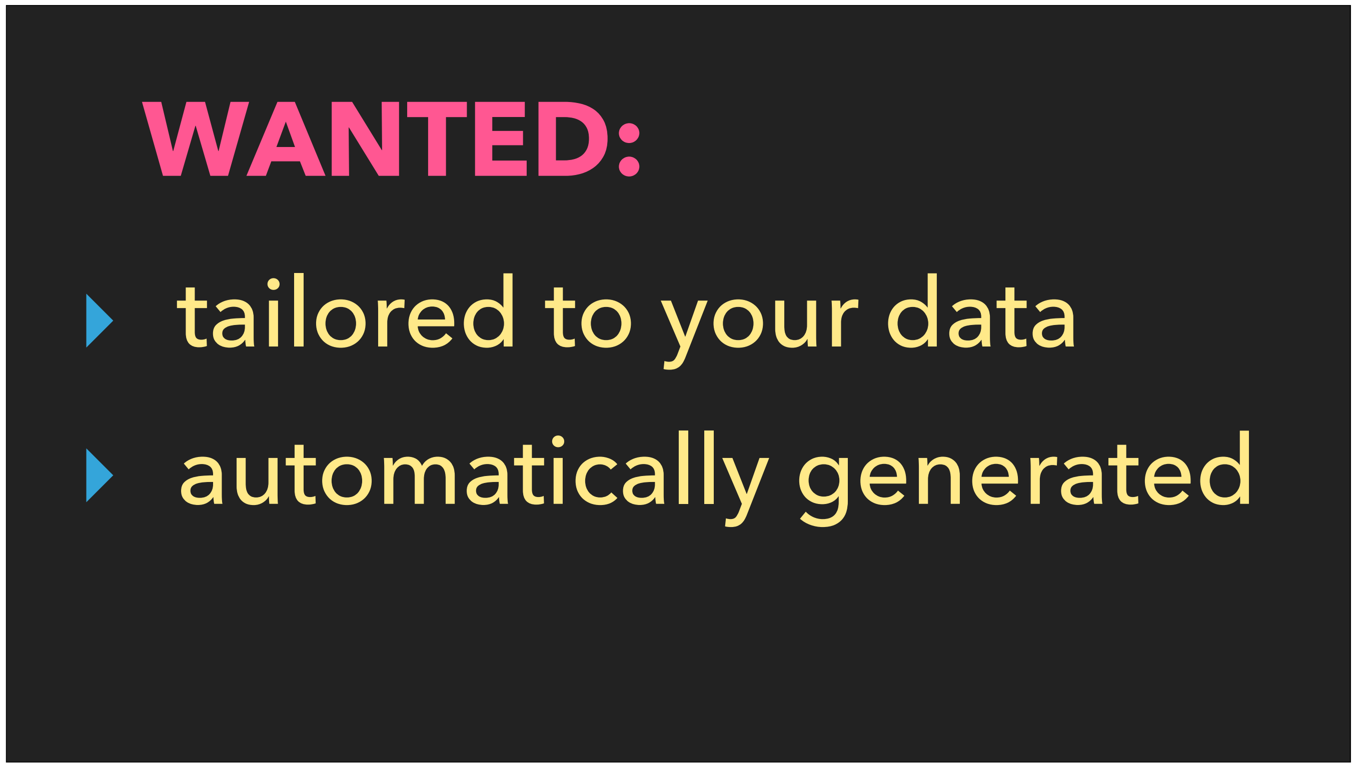
Wanted:



WANTED:

- tailored to your data

an index structure tailored to **your** data distribution, which can be



WANTED:

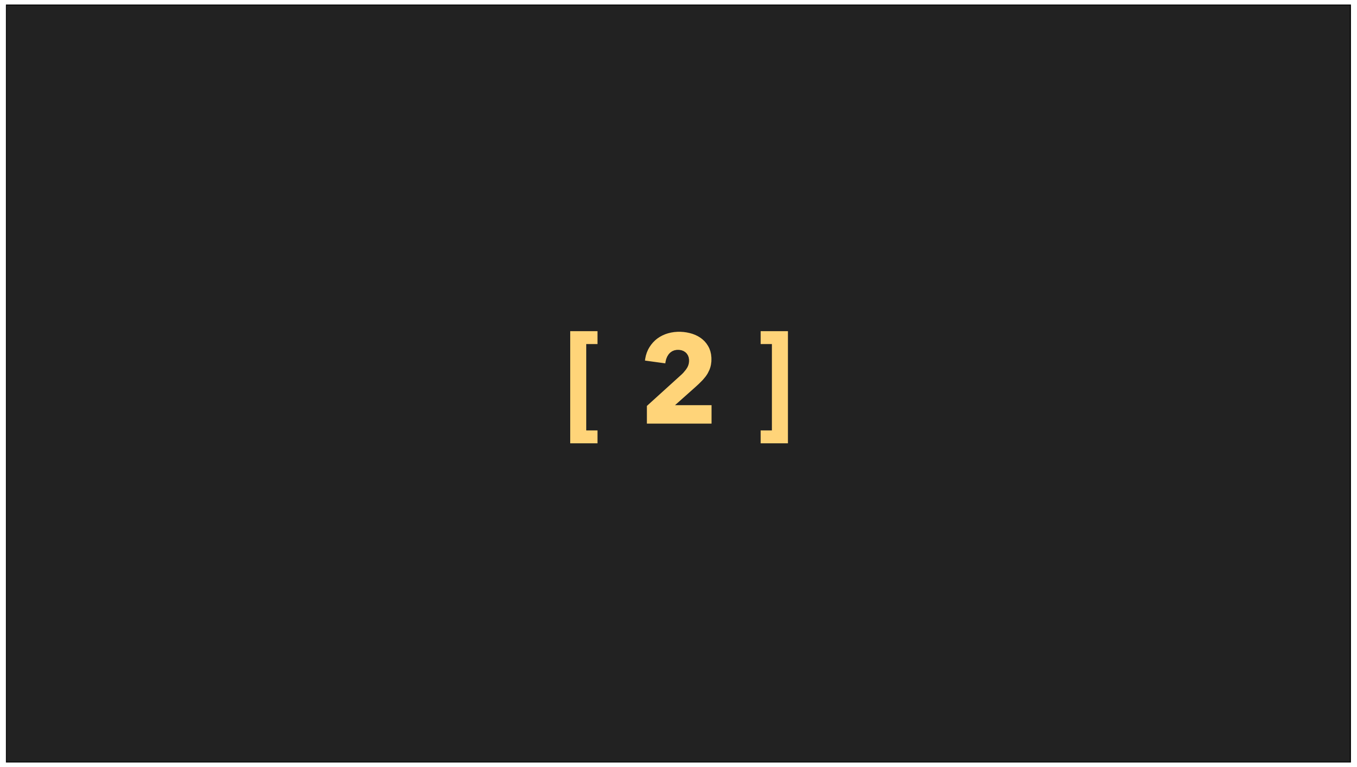
- tailored to your data
- automatically generated

automatically synthesized to avoid engineering costs, and which comes with

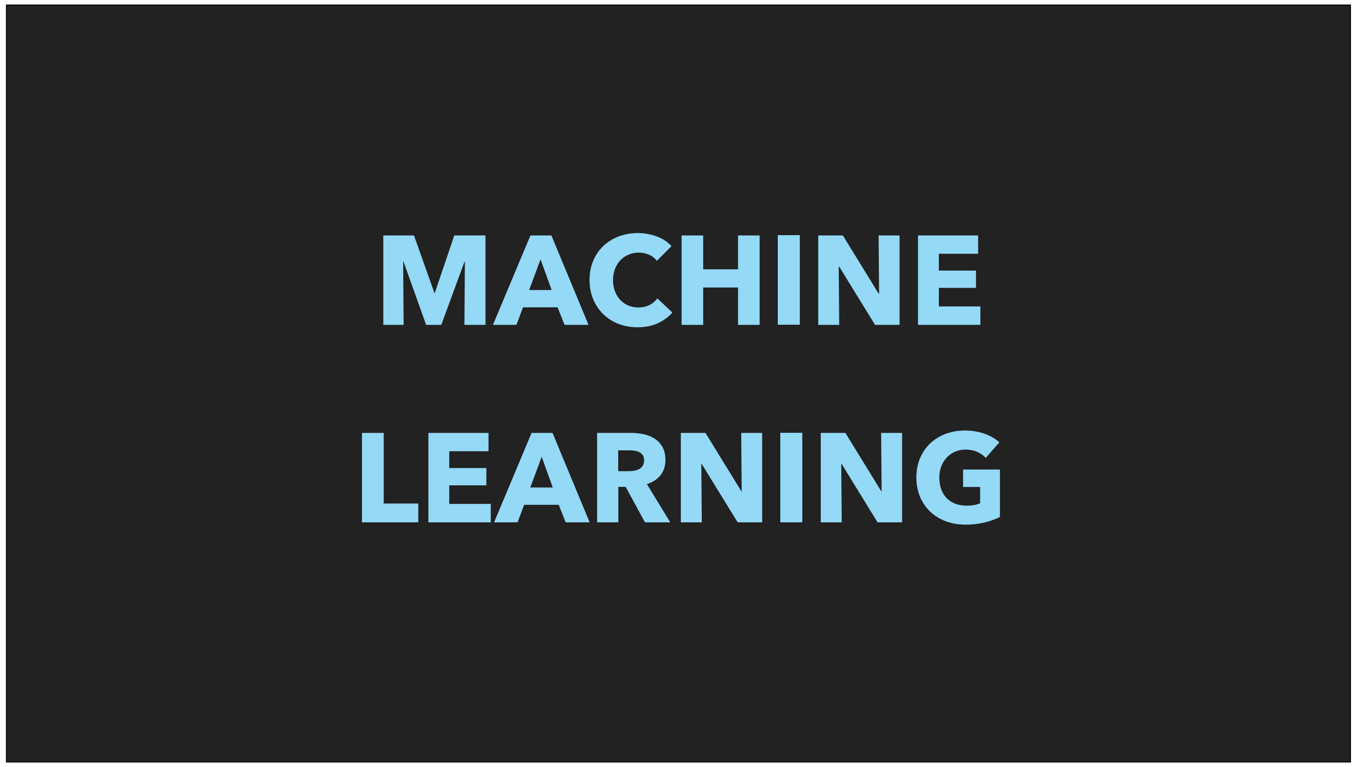
WANTED:

- tailored to your data
- automatically generated
- hard error guarantees

hard error guarantees. (As otherwise, the performance gains we get at prediction time might be lost at seek time.) So, what could fit the bill?


[2]

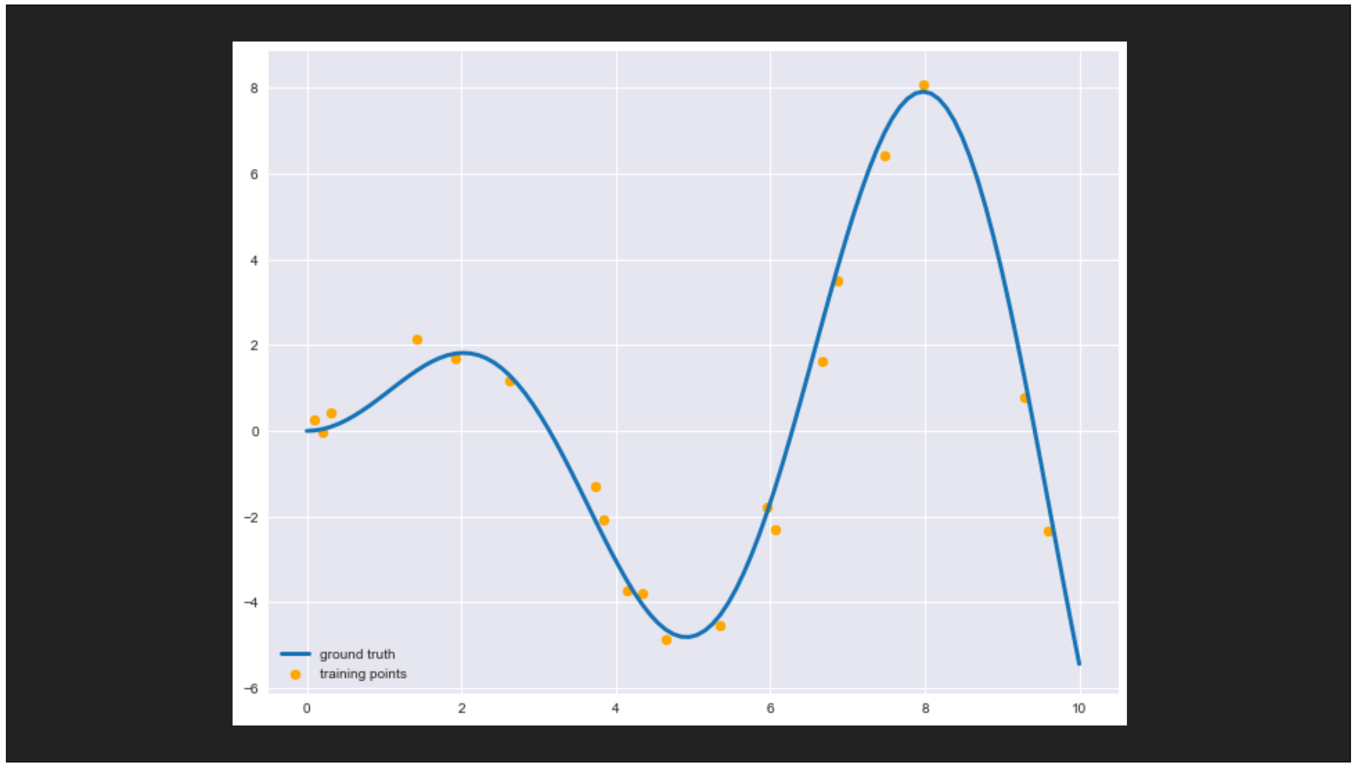
Well, as you might have guessed, the answer according to Kraska et al is:



MACHINE LEARNING

machine learning. And as for the general reason why, recall that what we want the index to learn is the distribution of the keys. This distribution is a function, and it turns out that

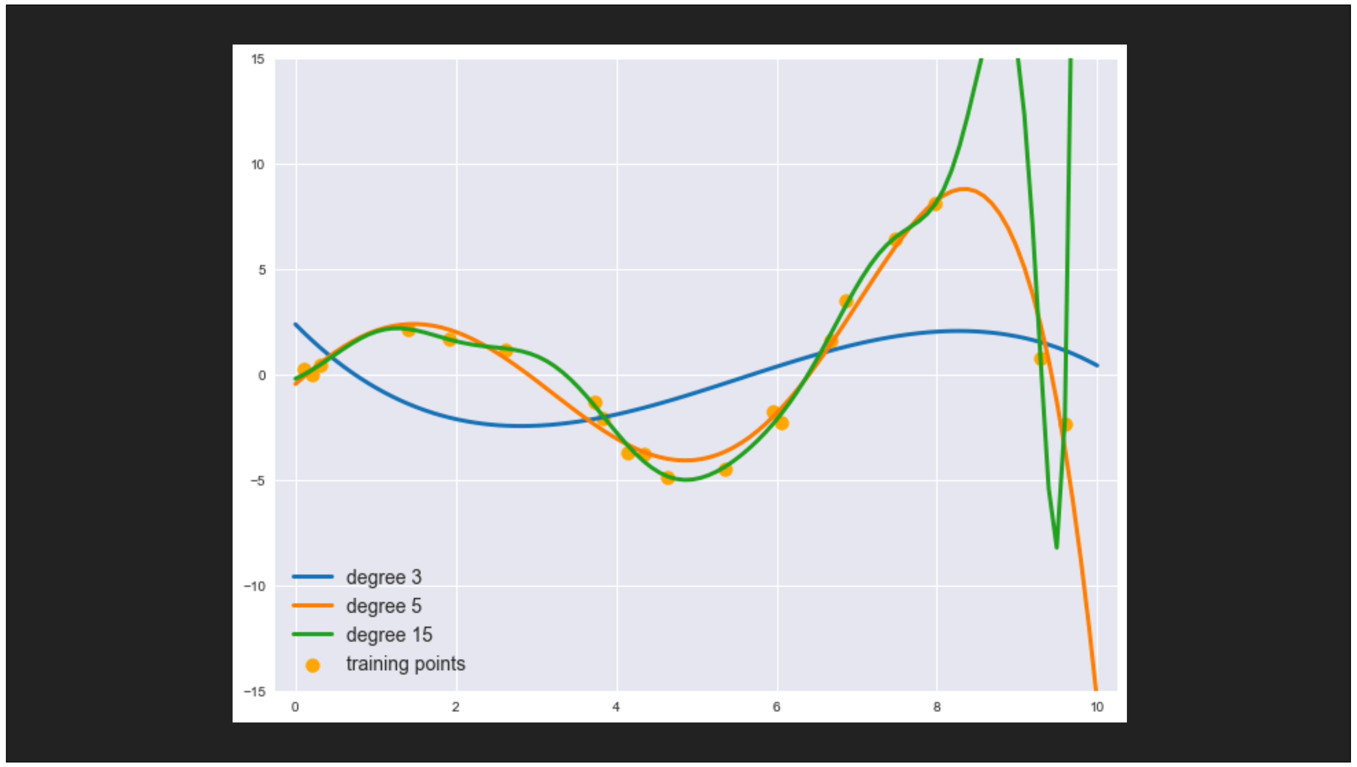
machine learning is a **really good** tool for learning functions, in particular the class of machine learning algorithms called neural nets, which form the basis of all the **deep learning** you've been hearing about for the past few years. in fact, machine learning algorithms are so good at learning functions that data scientists and other users of machine learning algorithms typically introduce restrictions on the allowed complexity of trained machine learning models simply to keep them from learning the data too well. Here's an example of what I mean:



Say the function you want to learn is represented by this blue curve here. In machine learning problems, you don't know what the function is, but you **can** make observations of that function, here represented by the yellow points. By the way, the reason that those yellow points don't actually coincide with the blue curve is because these observations are, in machine learning problems, corrupted by noise.

(As a real-world example, we could be trying to fit, say, the function of apartment prices in Berlin, and our observations would include information for each apartment about the number of rooms, the area in square meters, the distance to the metro, etc., together with actual sale prices, whose deviations from the latent price function could be explained by prejudices of the seller, time pressures experienced by potential buyers, etc.).

Our machine learning algorithm would then make a guess about what that function could be, use the observations together with some measure of loss to calculate an error on that guess, and consequently use the error to make a better guess, and so on, until the change in error between guesses falls below some tolerance threshold.



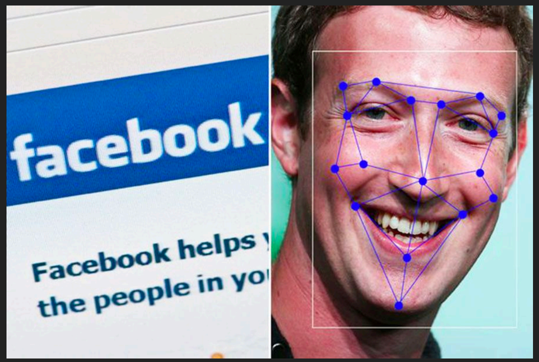
So we try to fit curves of varying complexity to these observations. The most simple, here in blue, and we see that, even with the best selection of function parameters, the resulting curve is unable to approach the vast majority of the observations. A machine learning practitioner would say that this model is **underfitting**, and this arises when the allowed complexity of the model is not sufficient to describe the function underlying the observations.

The most complex curve, here in green, this is also doing a terrible job but for a different reason: this one is trying to pass through all of the points, no matter how illogical the resulting shape. This phenomenon is called **overfitting**, and what's happening is that the machine learning algorithm is fitting, not to the latent function, but to the noisy observations of that function. (Remember this; it'll be important later.)

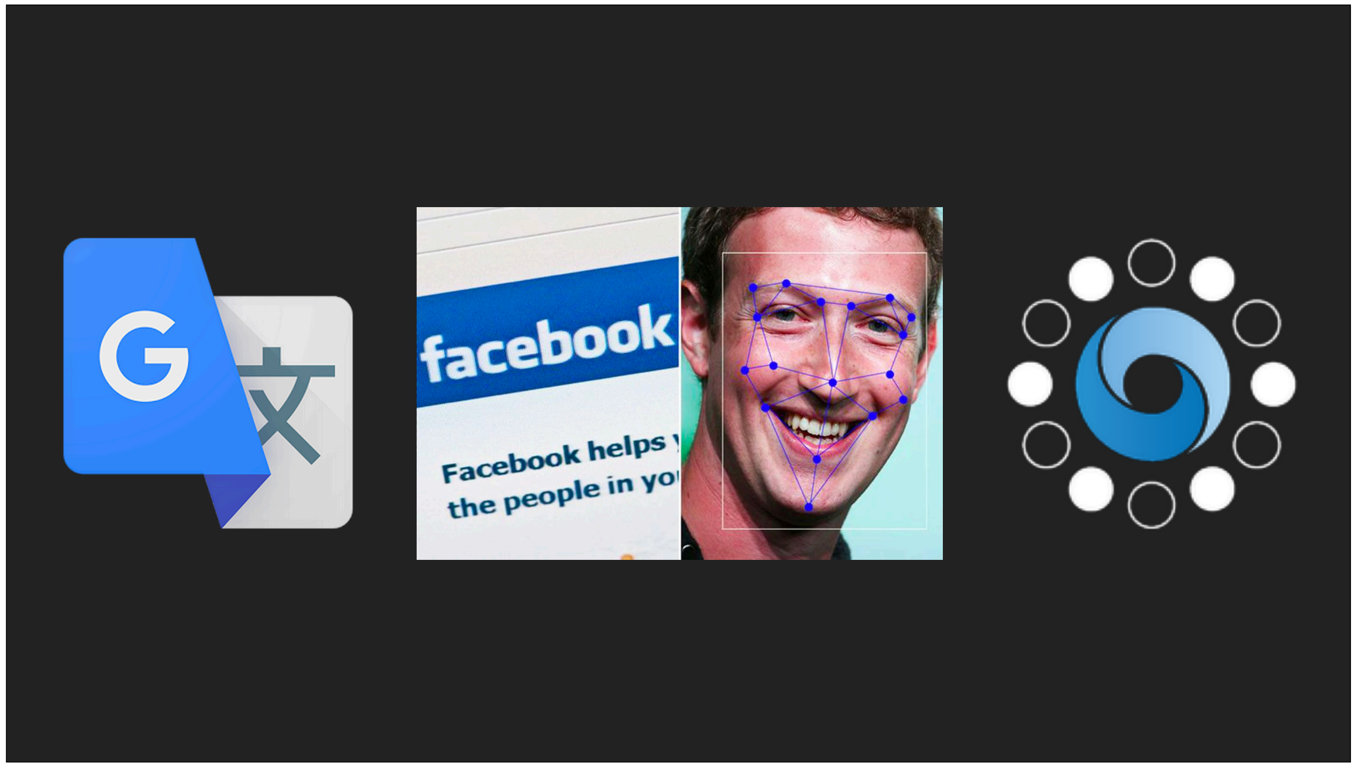
So we need some curve whose complexity is somewhere in between the blue curve and green curve, which is here shown in orange, and actually finding that perfect balance between underfitting and overfitting is one of the hardest parts about doing machine learning.

Now that was an example of using a machine learning algorithm to fit a simple function, but actually machine learning algorithms are capable of fitting immensely complicated functions of hundreds of millions of parameters.

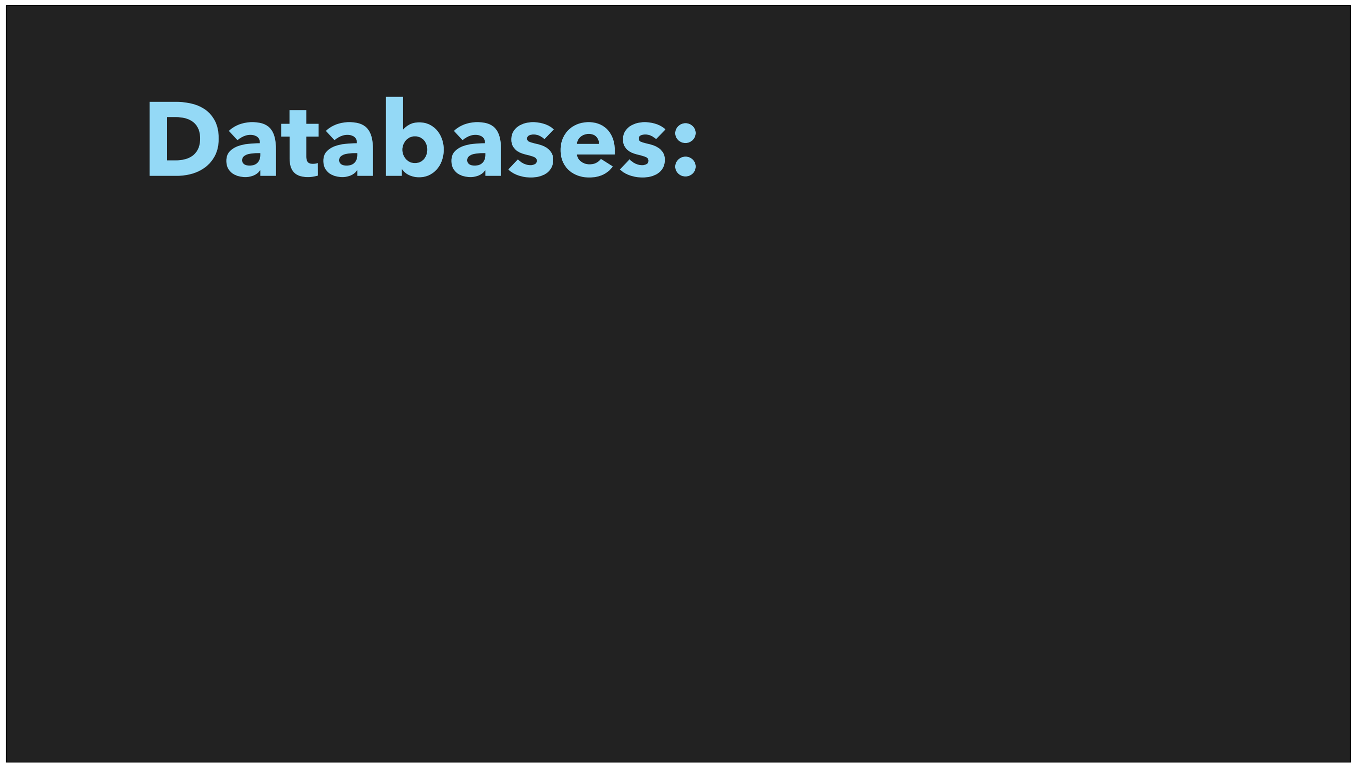
Google Translate,



Facebook's facial recognition software,



and DeepMind's AlphaGo all boil down to machine learning systems that have learned incredibly complicated functions.



Databases:

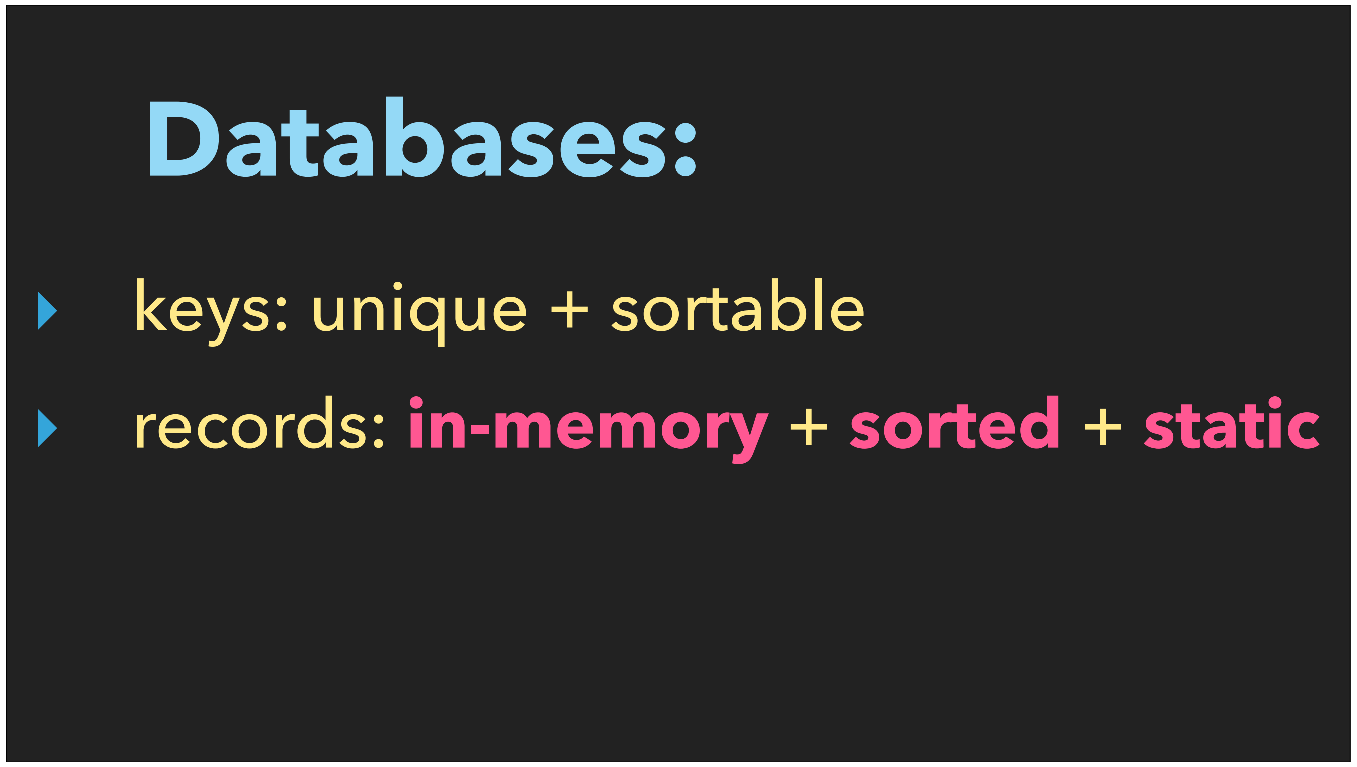
So machine learning is useful for learning functions; how do we apply this to the database domain?

Say the situation is the following:

Databases:

- keys: **unique + sortable**

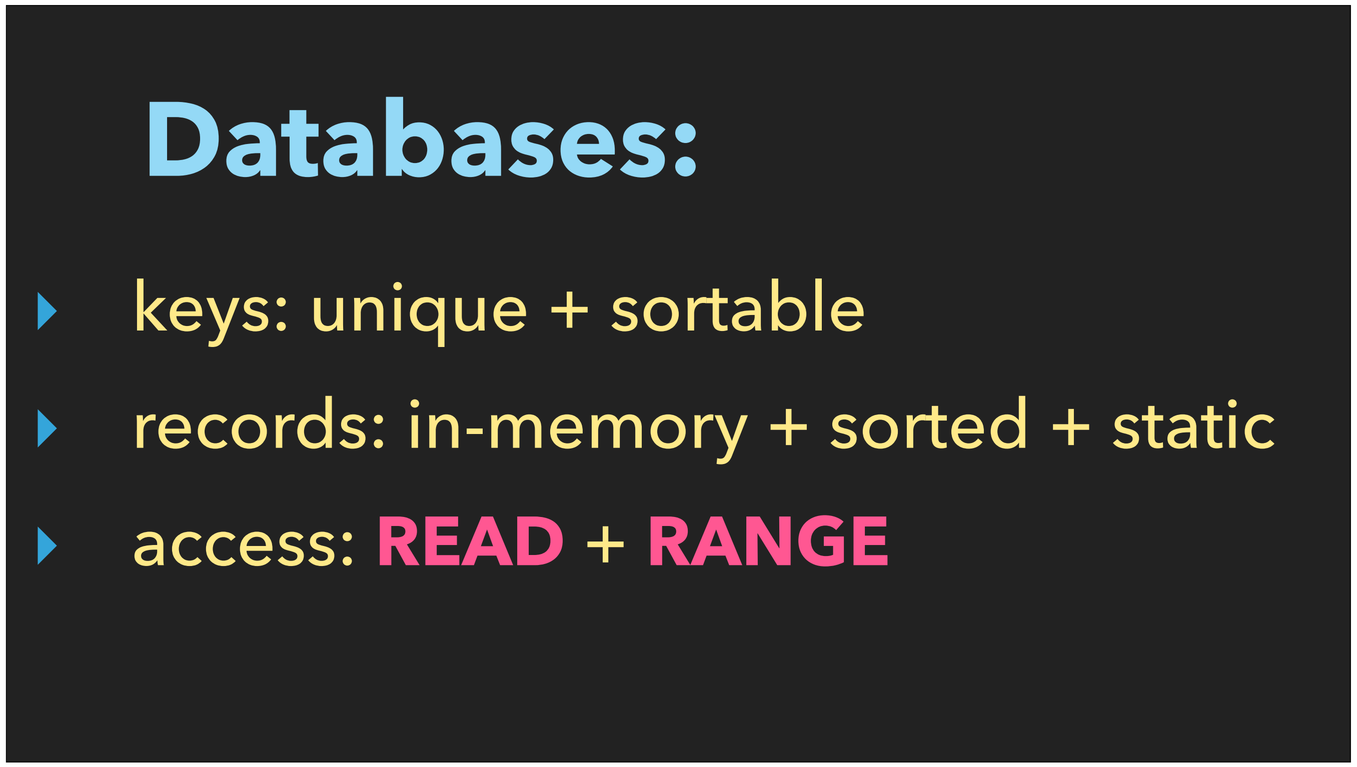
+ our records each have a unique key and the collection of these keys is orderable



Databases:

- keys: unique + sortable
- records: **in-memory** + **sorted** + **static**

+ our set of records is held in memory, sorted by key, and static (static or are only updated infrequently, as in a data warehouse)

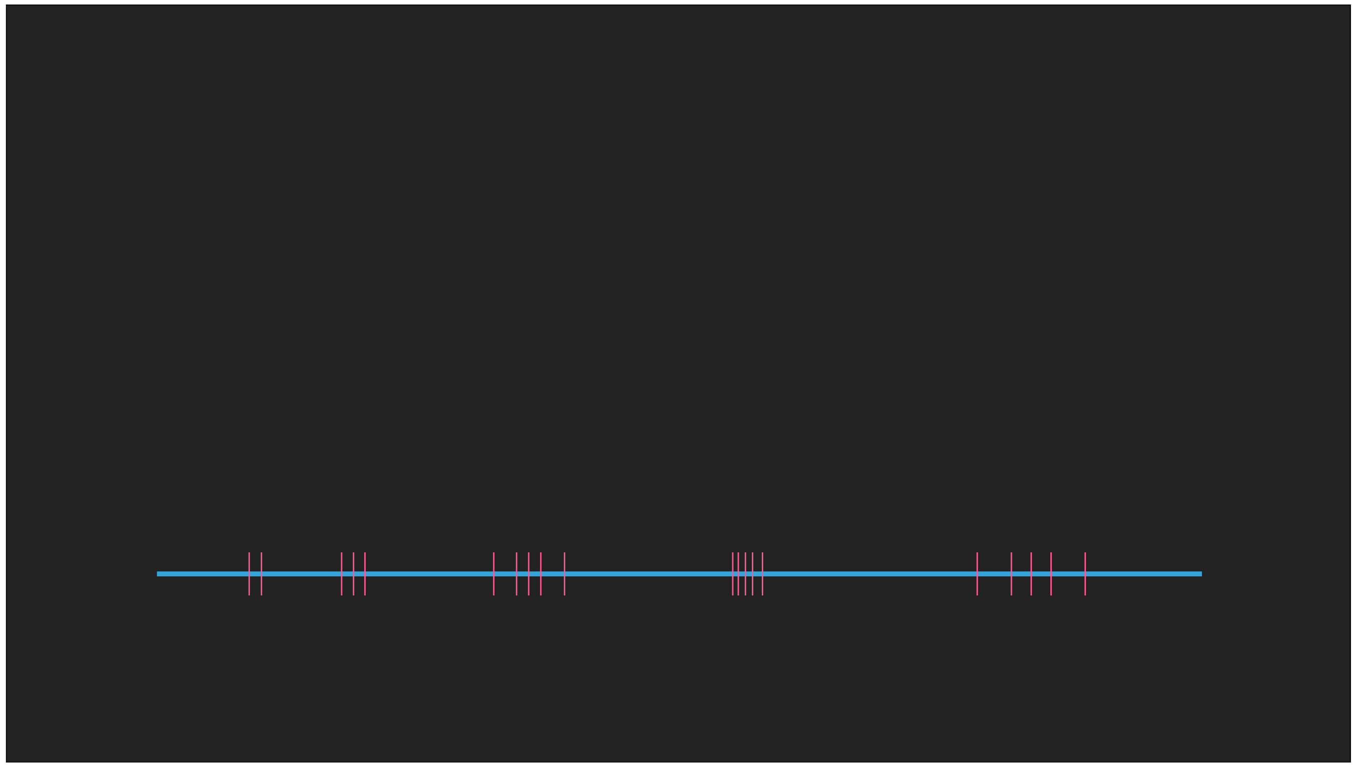


Databases:

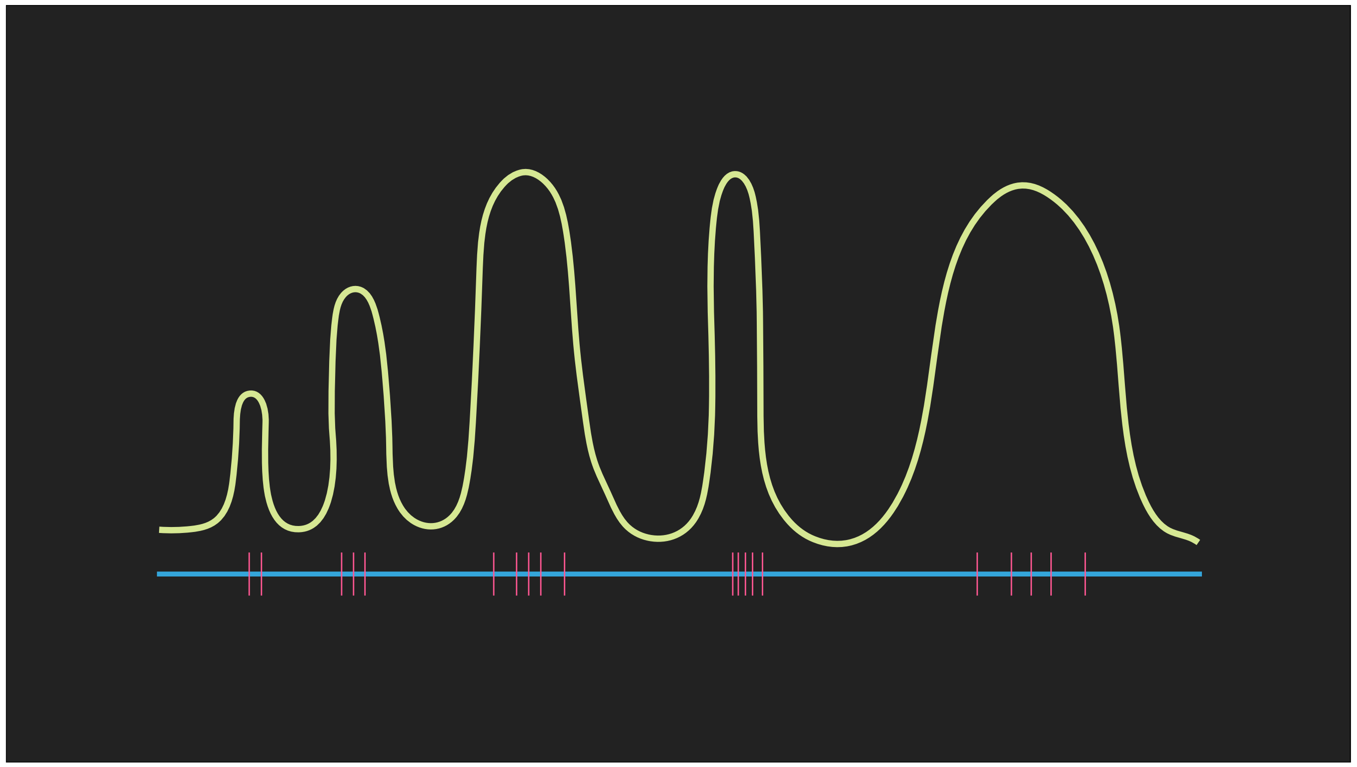
- keys: unique + sortable
- records: in-memory + sorted + static
- access: **READ + RANGE**

+ we are interested in READ access and RANGE queries

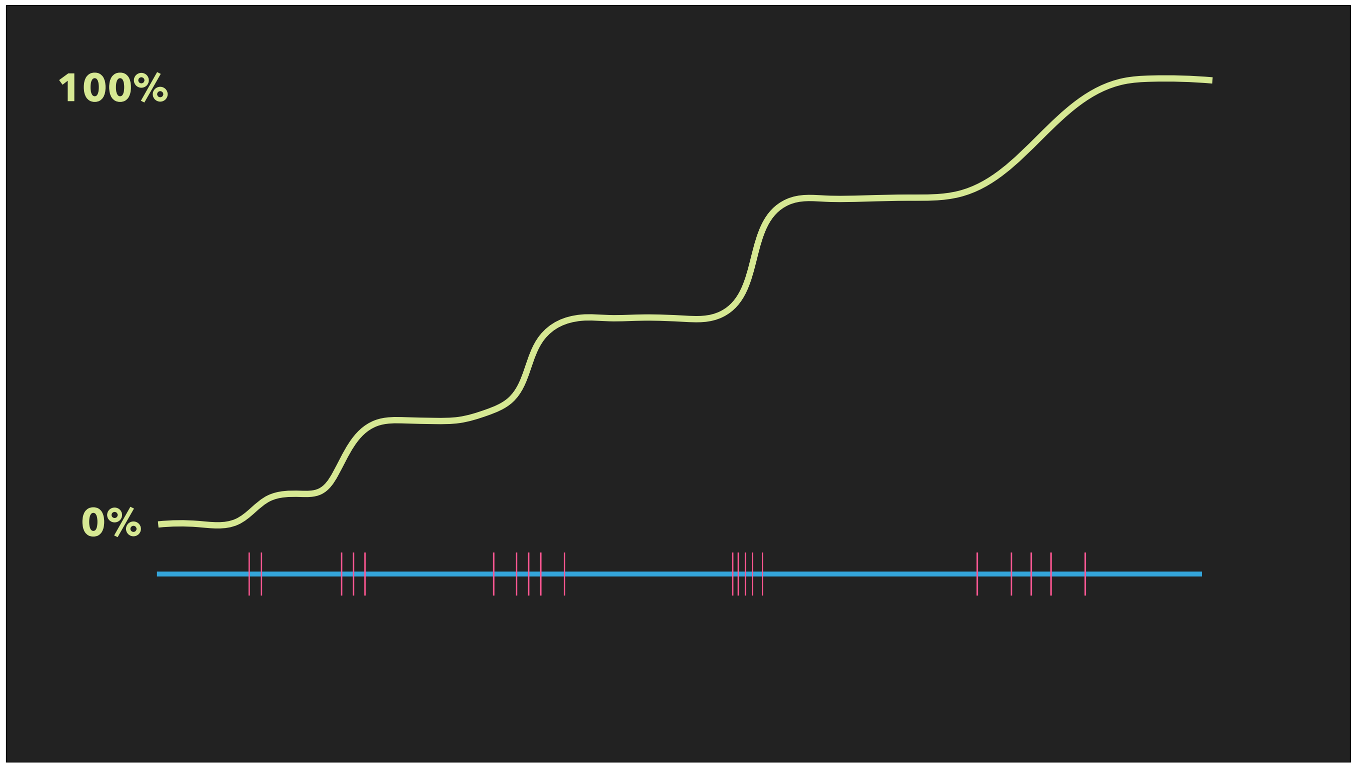
Given these conditions, here's another function-learning situation more along the lines of what we're interested in doing.



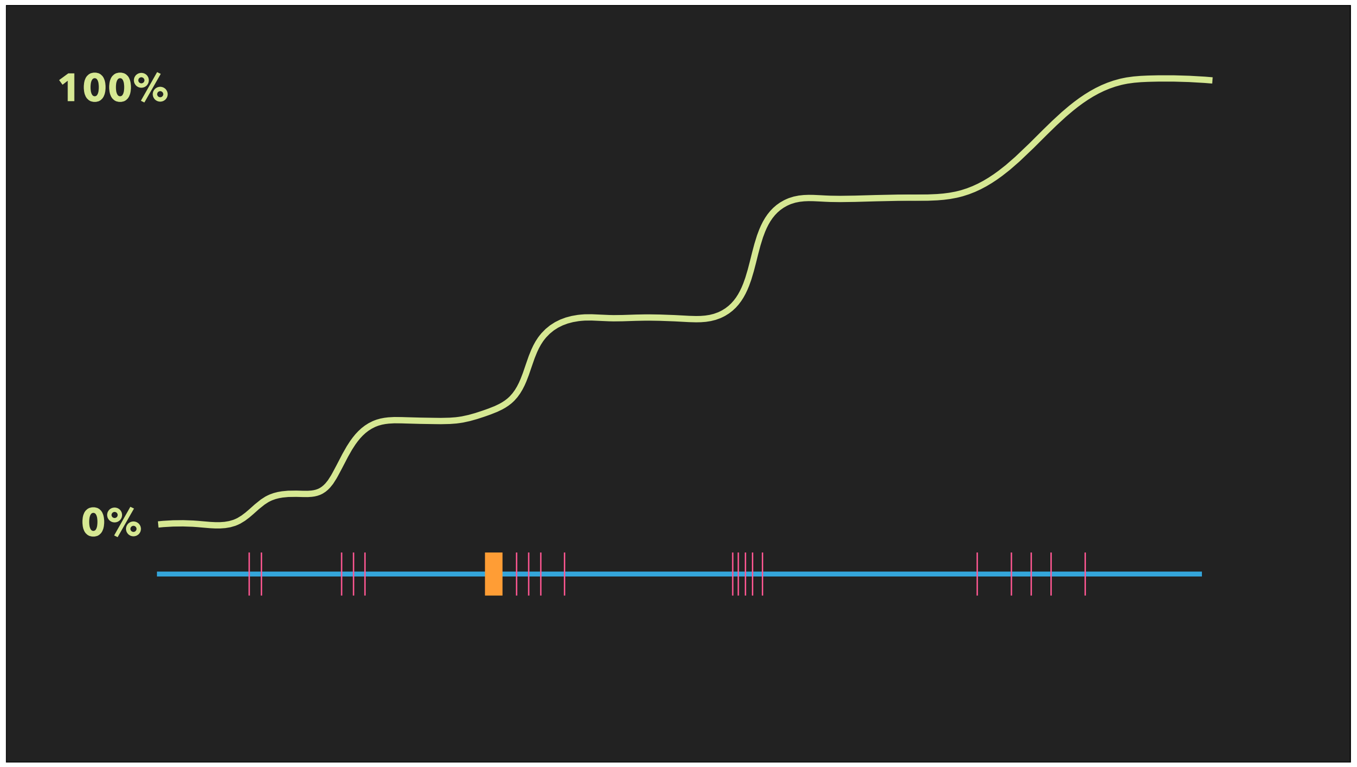
Say we have our keys and they are spread out in some way amongst the allowed values: we're interested in learning this



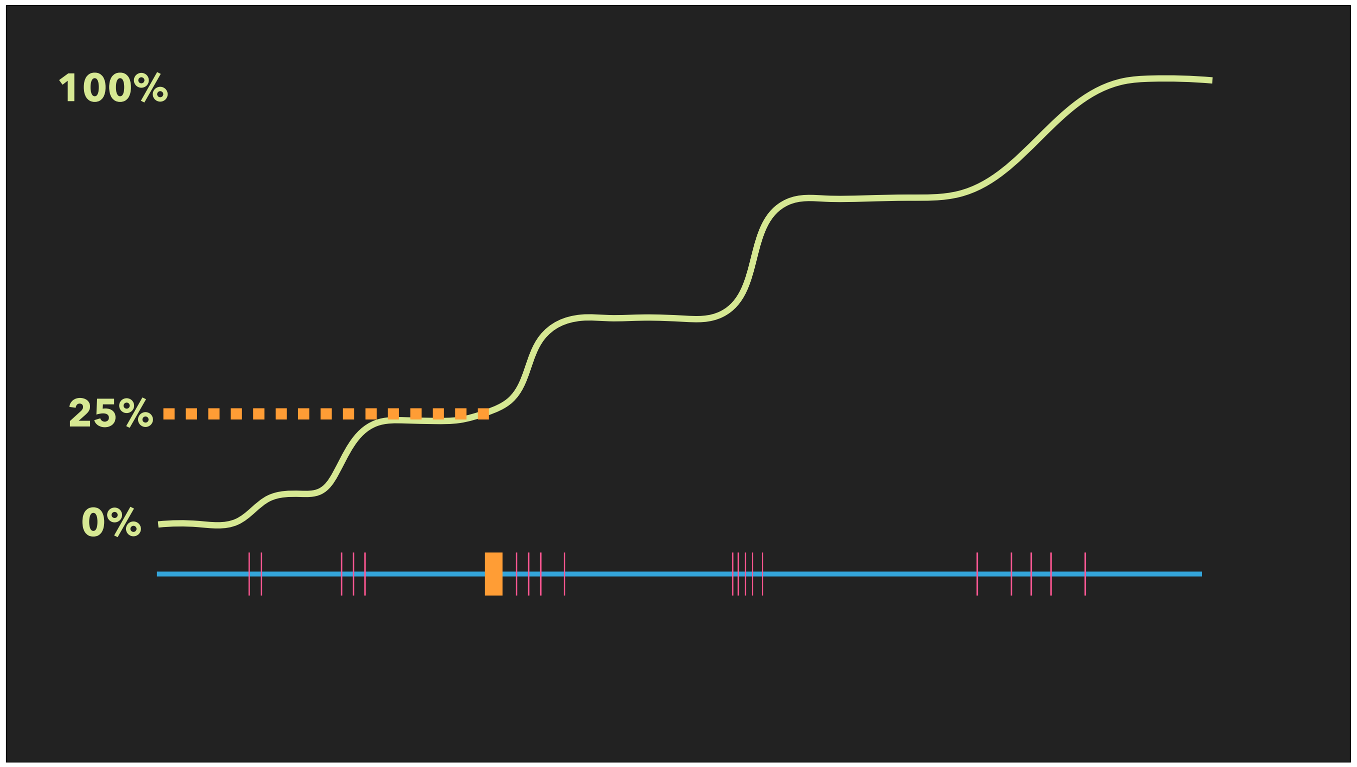
key distribution (and please forgive the lack of rigor in my illustration). Now, machine learning algorithms could learn this naked distribution perfectly well (it's a task called "density estimation"), but actually from an engineering point of view, the function we would rather learn is the



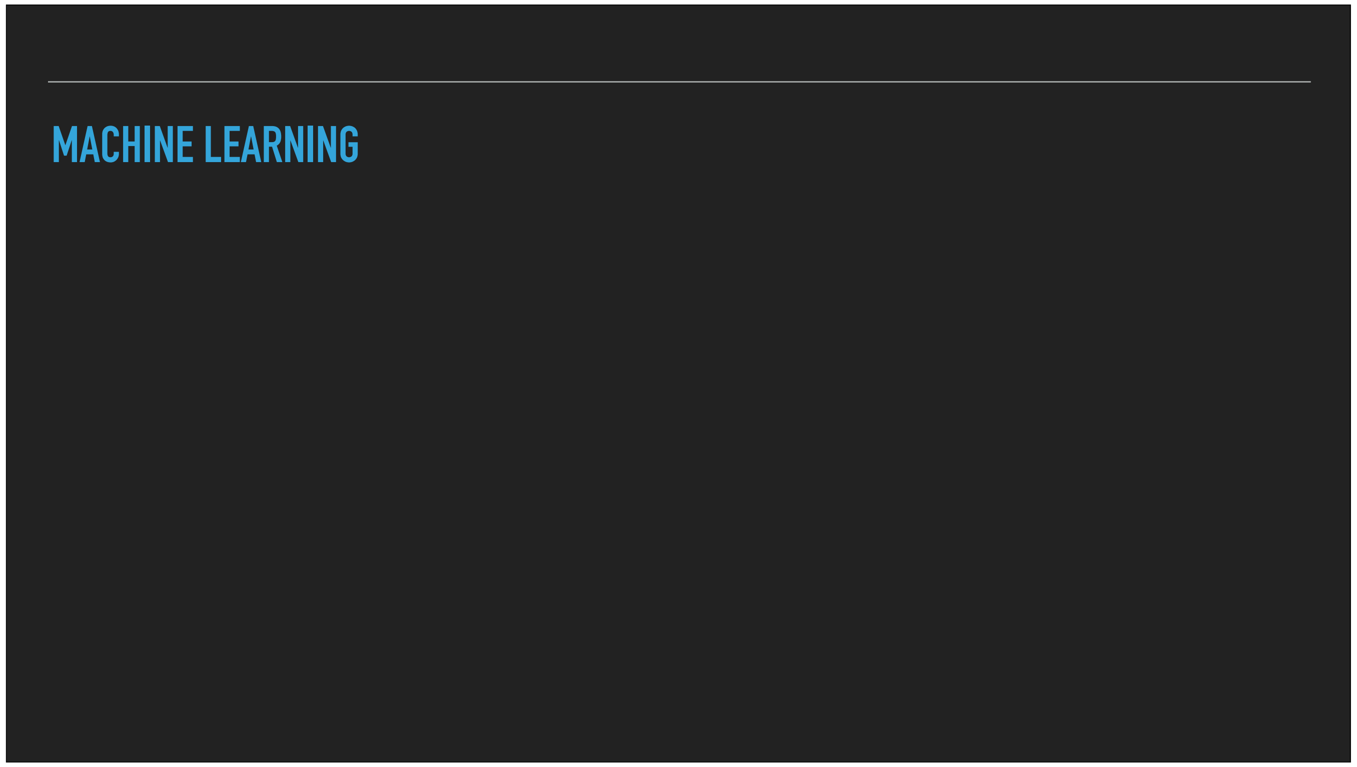
cumulative key distribution. that is to say, we want to give our model



a key and have it predict, say, that this particular key is

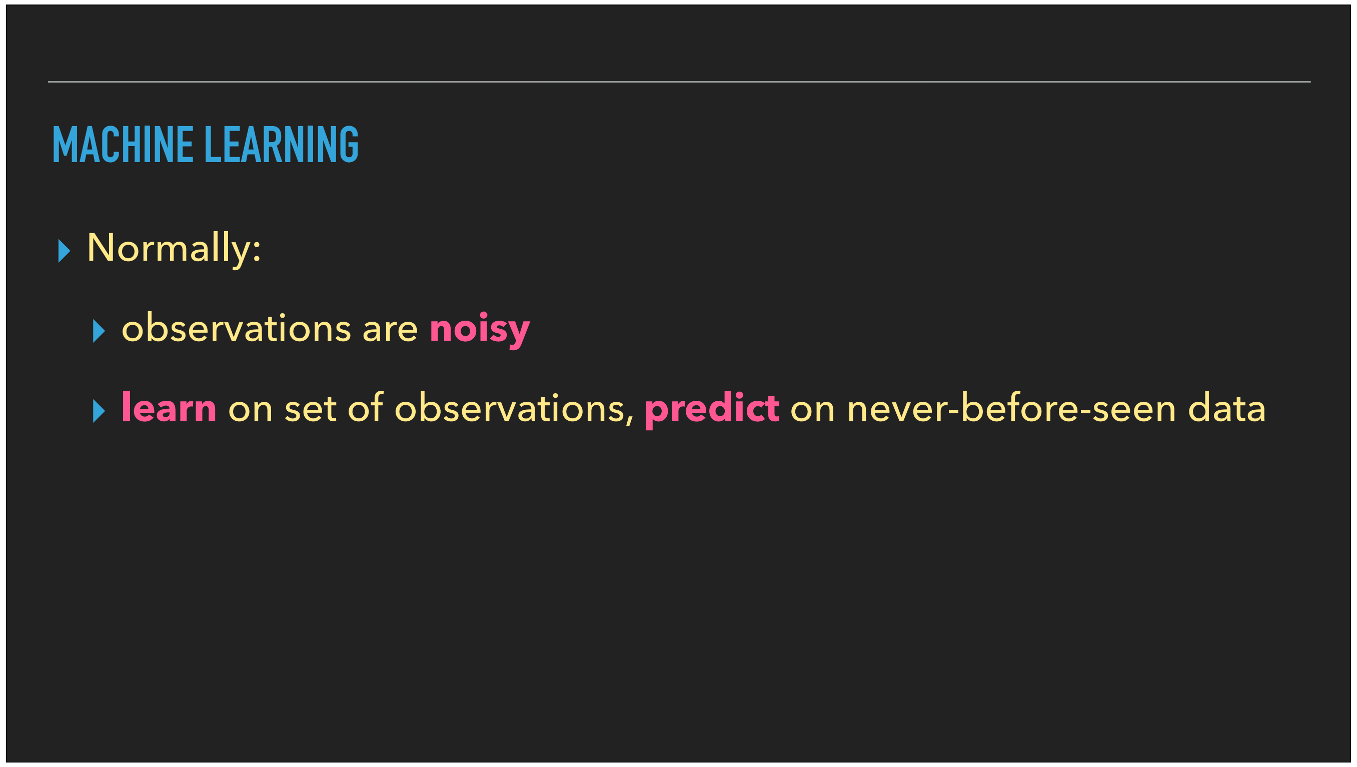


greater than 25% of the keys according to their ordering. This way, we immediately know that we should skip the first 25% of the records to retrieve the record of interest.



MACHINE LEARNING

Now, what I just described about learning distributions could be termed "normal machine learning". However, there is a very important difference between our database index scenario and normal machine learning: in normal machine learning,

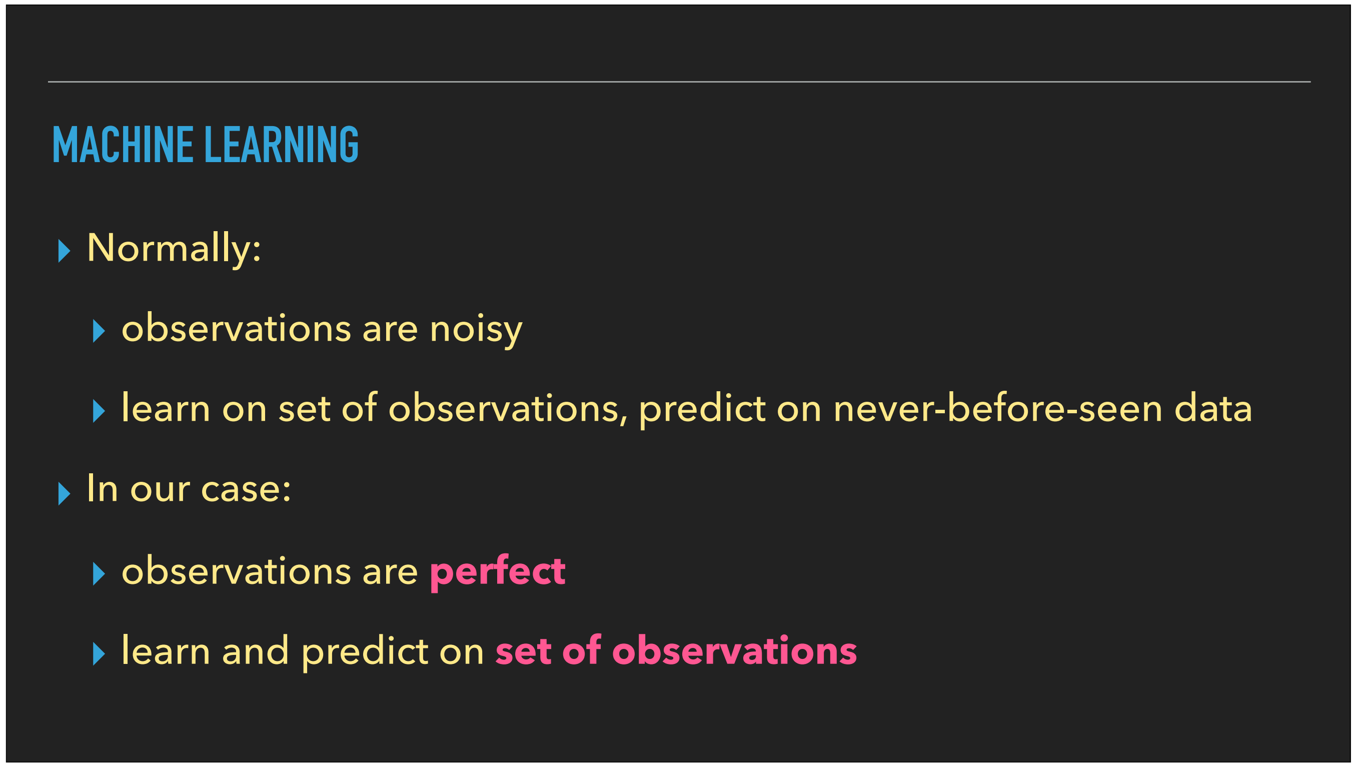


MACHINE LEARNING

- Normally:
  - observations are **noisy**
  - **learn** on set of observations, **predict** on never-before-seen data

you learn a function based on your noisy observations of that function and then make predictions for input values that you haven't seen before. For instance, going back to our Berlin apartment pricing model, we were fitting this model based on historical prices of sold apartments, but actually the reason we want to use it for is not to explain apartment prices in the past, but rather to make a predictions of the price of an apartment whose exact combination of features we'd never seen before.

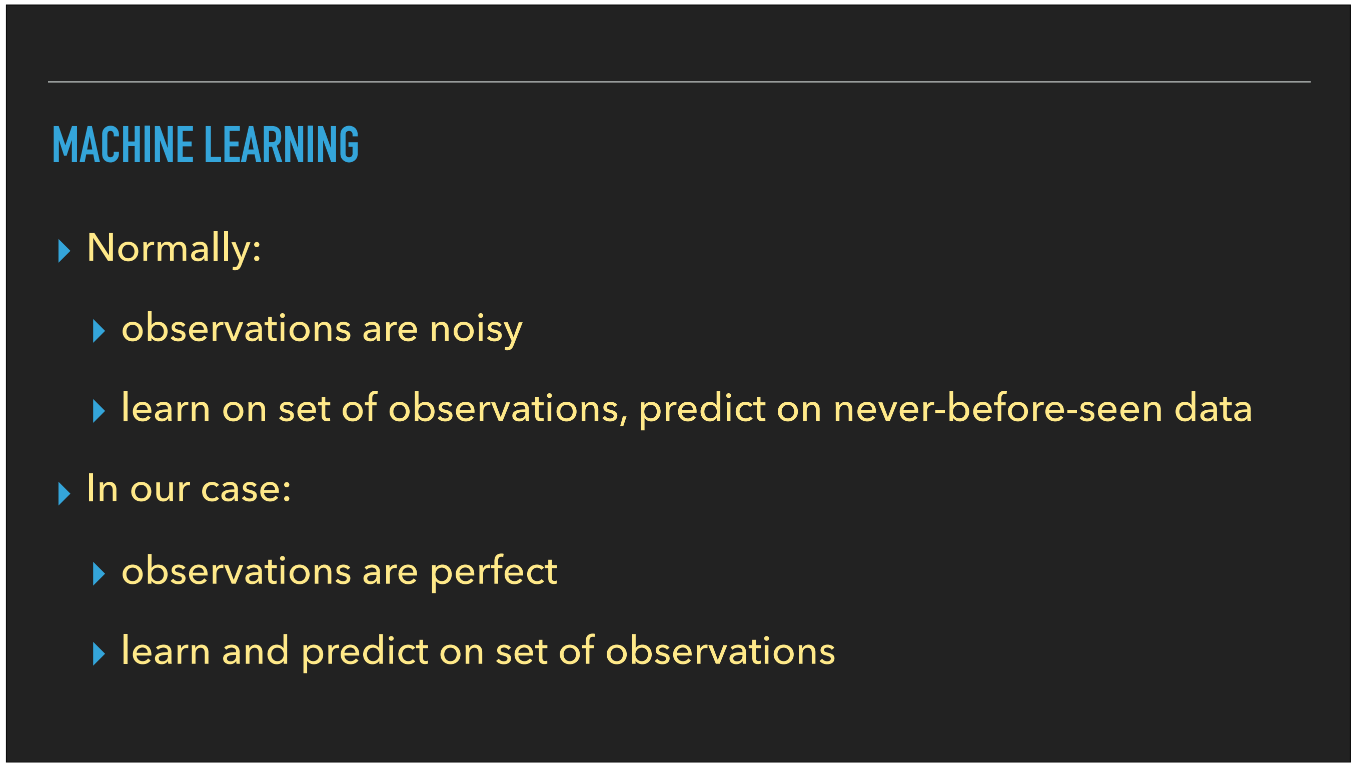
But with an index model,



MACHINE LEARNING

- Normally:
  - observations are noisy
  - learn on set of observations, predict on never-before-seen data
- In our case:
  - observations are **perfect**
  - learn and predict on **set of observations**

not only are your observations of the keys noise-free, but when it comes time to make predictions, you're actually going to make predictions **on inputs the model has already seen before**, namely the keys themselves. This break with normal machine learning methodology means, in fact, that in this situation **our observations and the underlying function we are trying to learn are one and the same**. That is, there is nothing distinguishing the blue curve and the yellow dots, which in turn means that in our previous example, we actually would have preferred the highly-overfitting model that wildly jumps about, because it always predicts what it had seen before, and because there are **no values** of the function that the model hasn't seen before.



MACHINE LEARNING

- Normally:
  - observations are noisy
  - learn on set of observations, predict on never-before-seen data
- In our case:
  - observations are perfect
  - learn and predict on set of observations

Additionally, this break with traditional methodology gives us hard error guarantees on our predictions: because after training our model we'll only be making predictions on what the model has already seen, and because the training data doesn't change, to calculate our error guarantees, all we have to do is to remember the worst errors that the model makes on the training data, and that's it.

Now, I mentioned earlier that a machine learning algorithm particularly adept at overfitting is the neural network. So to

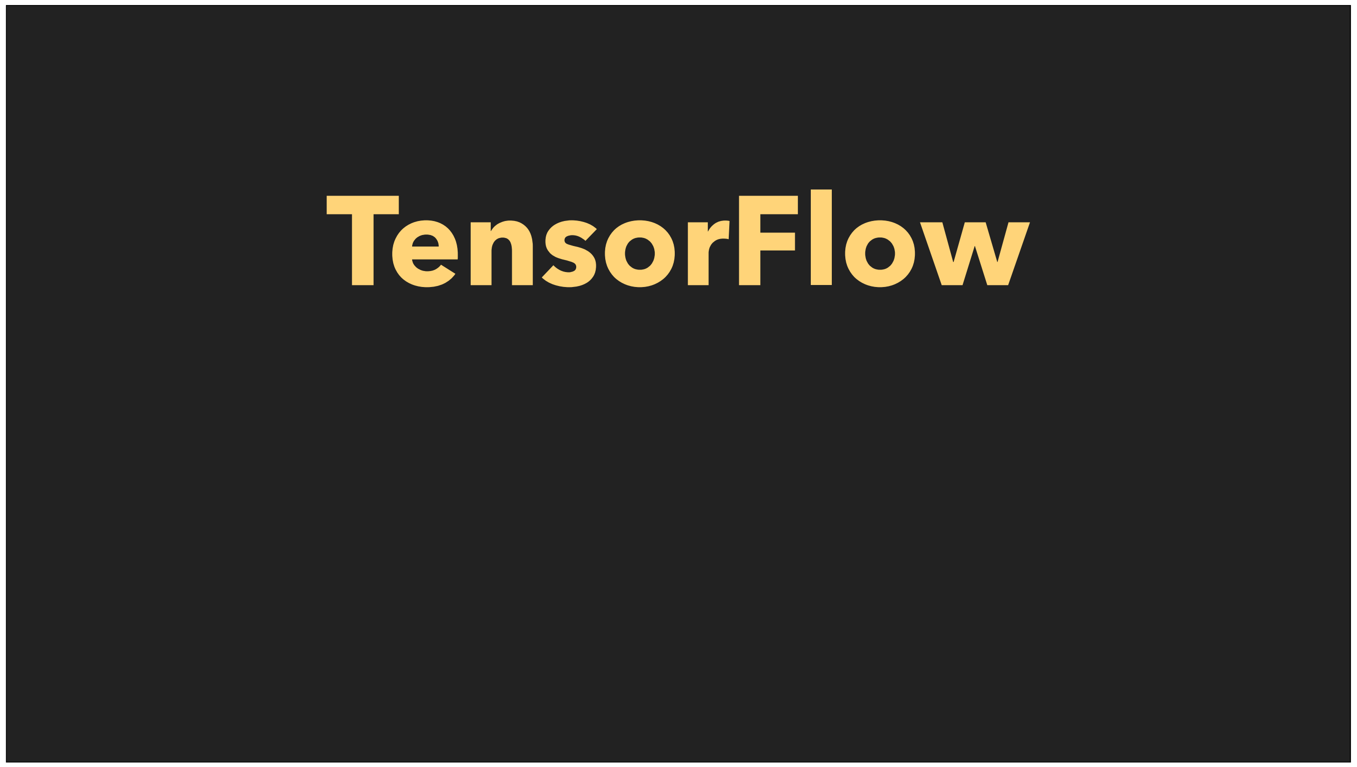
test

test their idea, the researchers took a dataset of 200 million webserver log records, trained a neural network index over their timestamps, and examined the results. And what did they find?

Well, they found that the model did

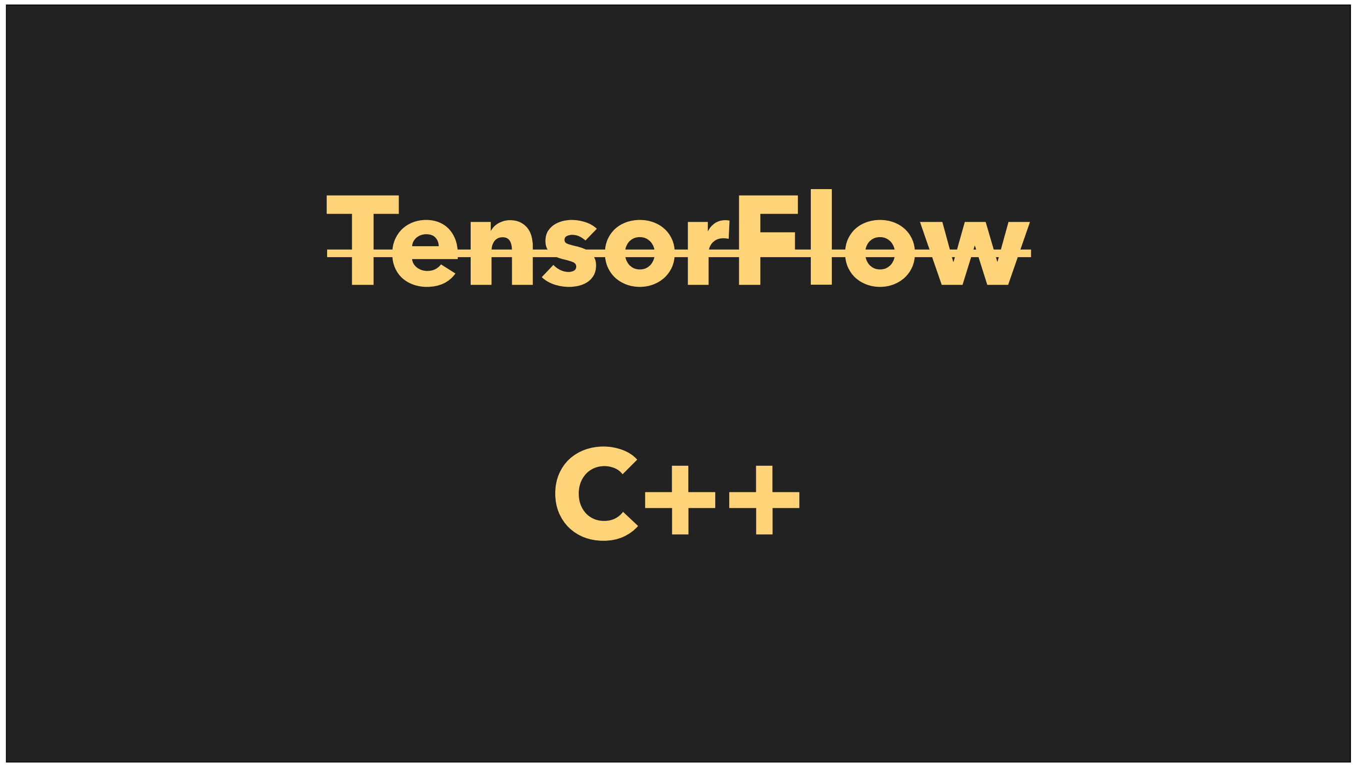
FAIL

terribly in comparison with a standard B-tree index: two orders of magnitude slower for making predictions and two to three times slower for searching the error margins. The authors offer a number of reasons for the poor performance. Much of it could be attributed to



TensorFlow

their choice of machine learning framework, namely Python's Tensorflow, used for both training the model and making for predictions. Now, Tensorflow was optimized for **big** models, and as a result has a significant invocation overhead that just killed the performance of this relatively small neural network. This problem, however, could be straightforwardly dealt with: simply train the model with Tensorflow and then export the optimized parameter values and recreate the model architecture using

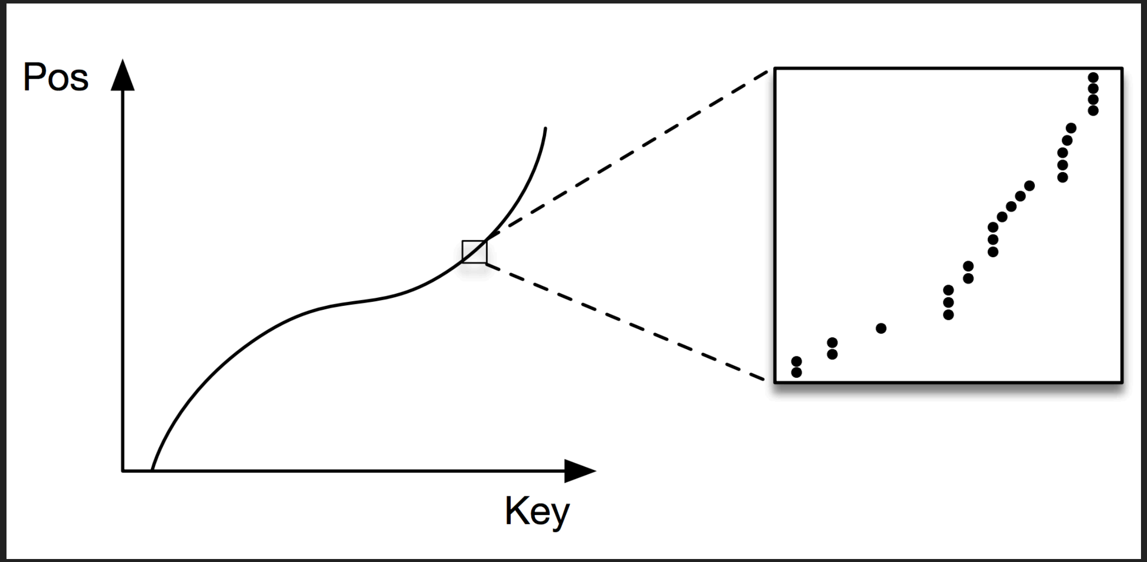


~~TensorFlow~~

C++

C++ for making predictions.

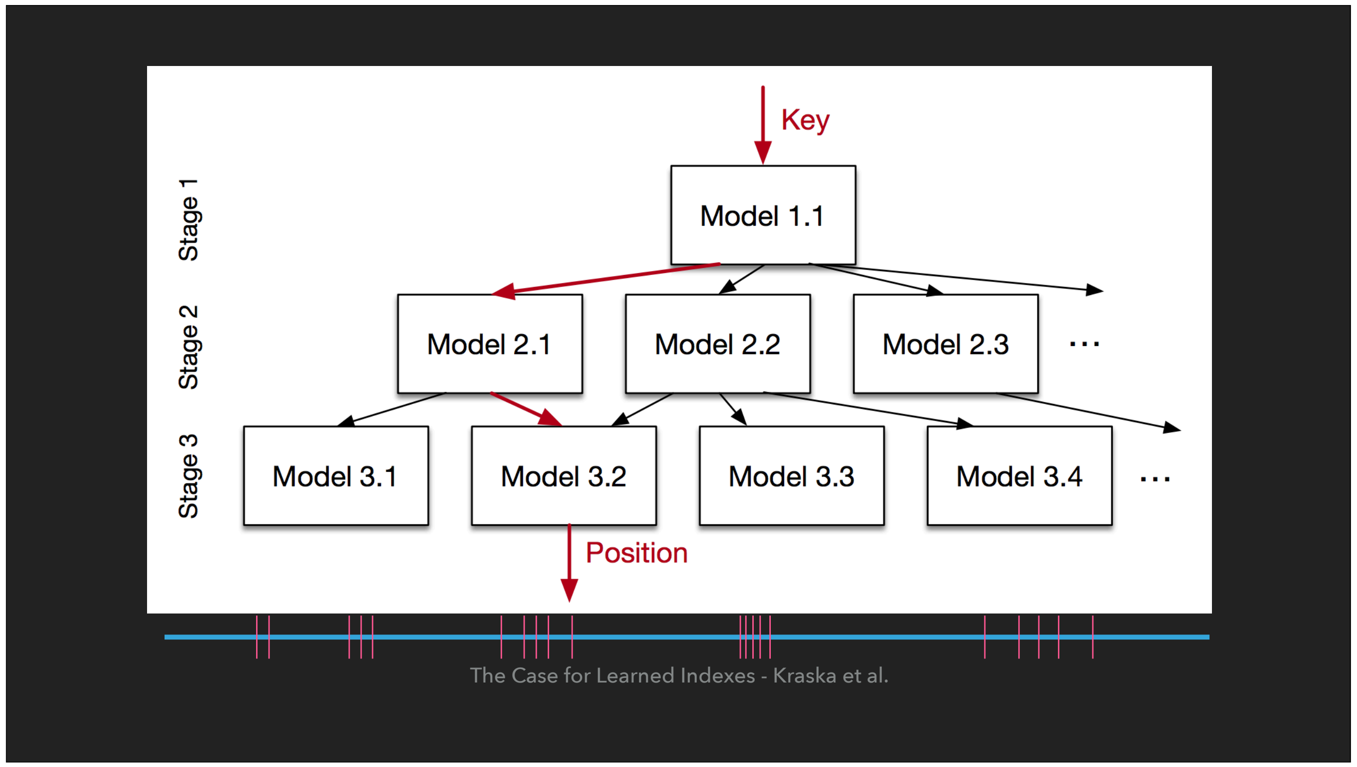
But there was another problem, less straightforward, which was that,



The Case for Learned Indexes - Kraska et al.

though neural networks are comparably good in terms of CPU and space efficiency at overfitting to the general shape of the cumulative data distribution, they lose their competitive advantage over B-trees when "going the last mile", so to say, and fine-tuning their predictions. Put another way, with a sufficient number of keys, from 10 thousand feet up the cumulative distribution looks smooth, as we can see in this image, which is **good** for machine learning models, but under the microscope the distribution appears grainy, which is **bad** for machine learning models.

So the solution of the authors was this:



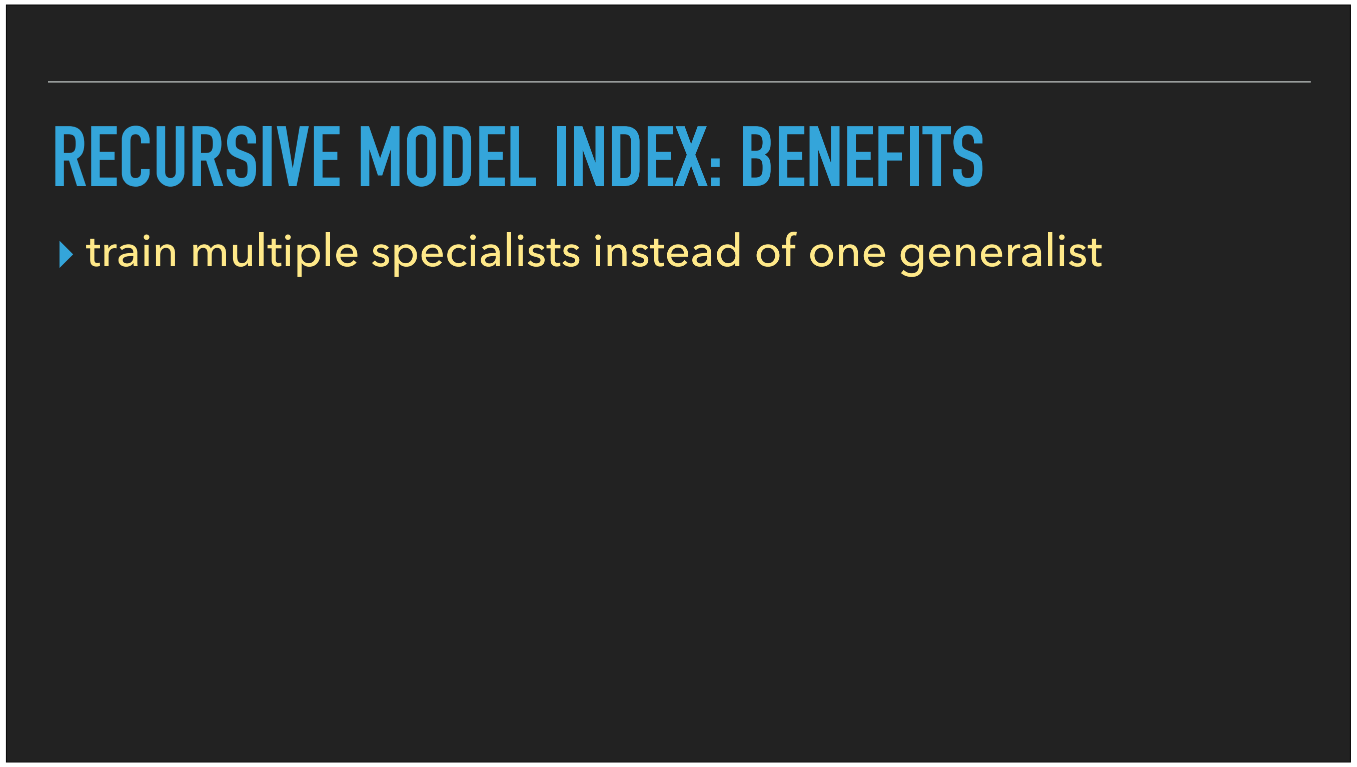
what they termed a **recursive model index**. The idea is to build a "model of experts", such that the models at the bottom are extremely knowledgable about a small, localized part of keyspace and the models above them are good at steering queries to the appropriate expert below.

Note, by the way, that this is **not** a tree structure: multiple models at one level can point to the same model in the level below.

This architecture has three principal benefits:

RECURSIVE MODEL INDEX: BENEFITS

one,

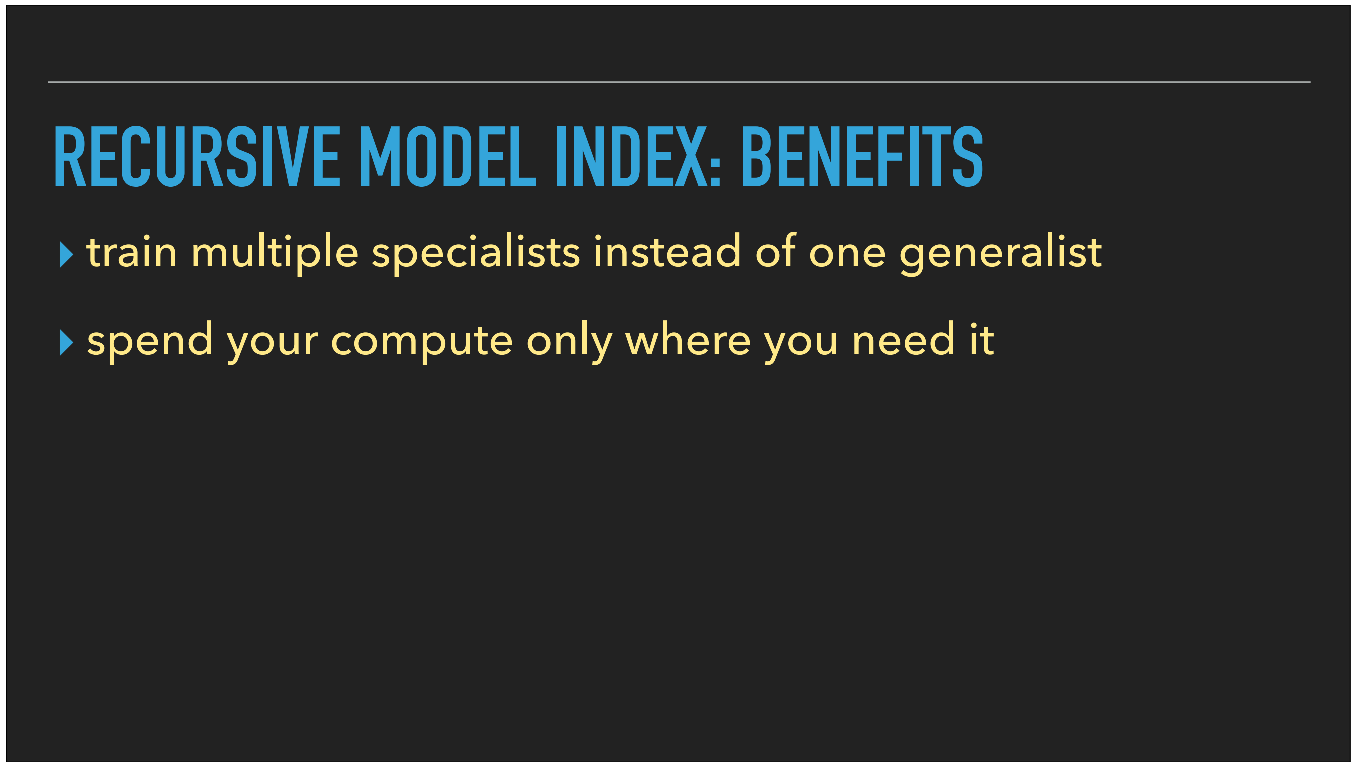


RECURSIVE MODEL INDEX: BENEFITS

- train multiple specialists instead of one generalist

instead of training one model based on its accuracy across the entire keyspace, you train multiple models each accountable only for a small region of the keyspace, decreasing overall loss;

two,

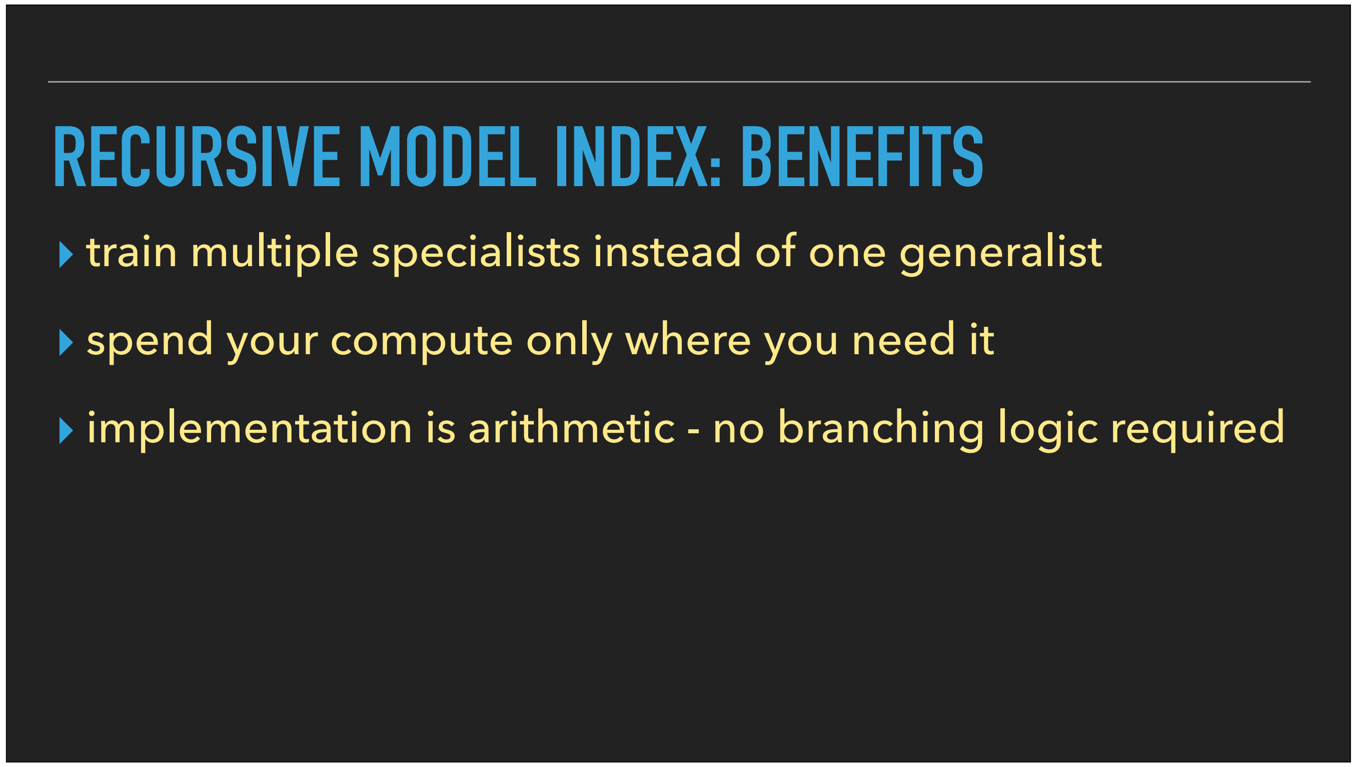


RECURSIVE MODEL INDEX: BENEFITS

- train multiple specialists instead of one generalist
- spend your compute only where you need it

complex and expensive models which are better at capturing general shape can be used as the first line of experts while simple and cheap models can be used on the smaller mini-domains;

and three,



RECURSIVE MODEL INDEX: BENEFITS

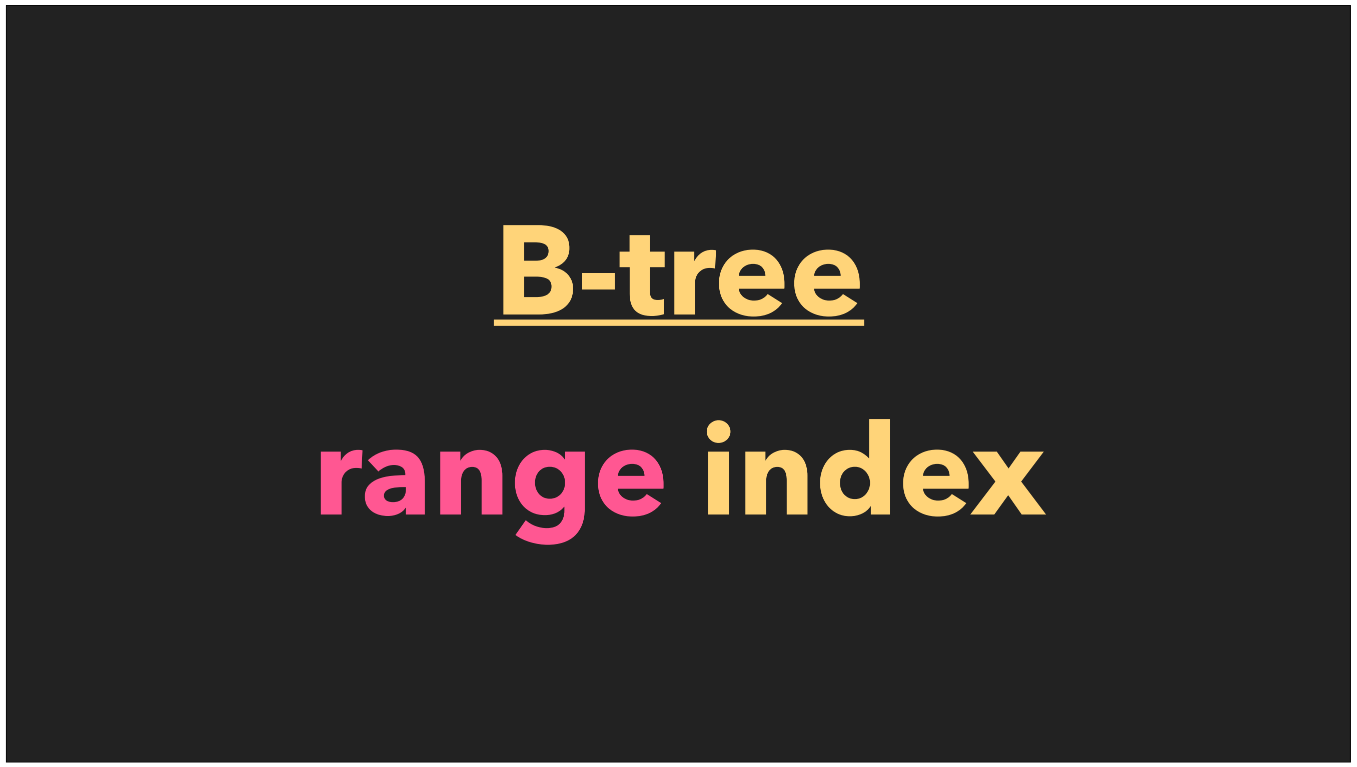
- train multiple specialists instead of one generalist
- spend your compute only where you need it
- implementation is arithmetic - no branching logic required

there is no search process required in-between the stages like in a B-tree (recall that when searching a B-tree, one must search through the keys in each node before deciding which child node to go to next): model outputs are simply offsets, and as a result the entire index can be represented as a sparse matrix multiplication, which means that predicts occur with constant complexity instead of O(log_k N).

I should mention that, up until now, we've been discussing

B-tree

B-tree database indexes as though they were strictly for looking up individual records. And while they **are** adept at that, their true utility lies in accessing



B-tree

range index

ranges of records; remember that our records are sorted, so that if we predict the positions of the two endpoints of our range of interest, we very quickly know the locations of all the records we'd like to retrieve. A logical follow-up question could then be: are there other index structures where machine learning could also play a role? That's the subject of the next section of this talk.

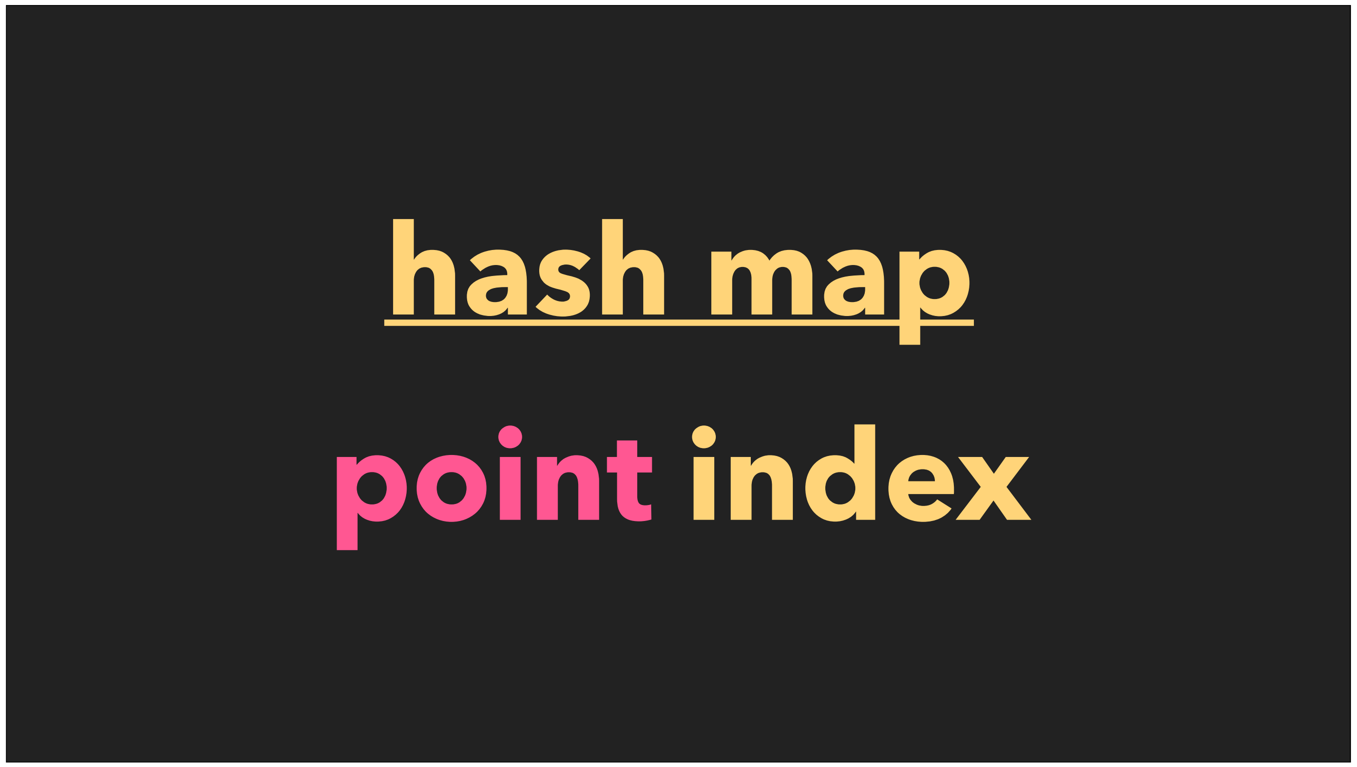
[3]

I'd like now to talk about two additional types of index structures: hash maps and bloom filters. Let's start with

hash map

hash maps.

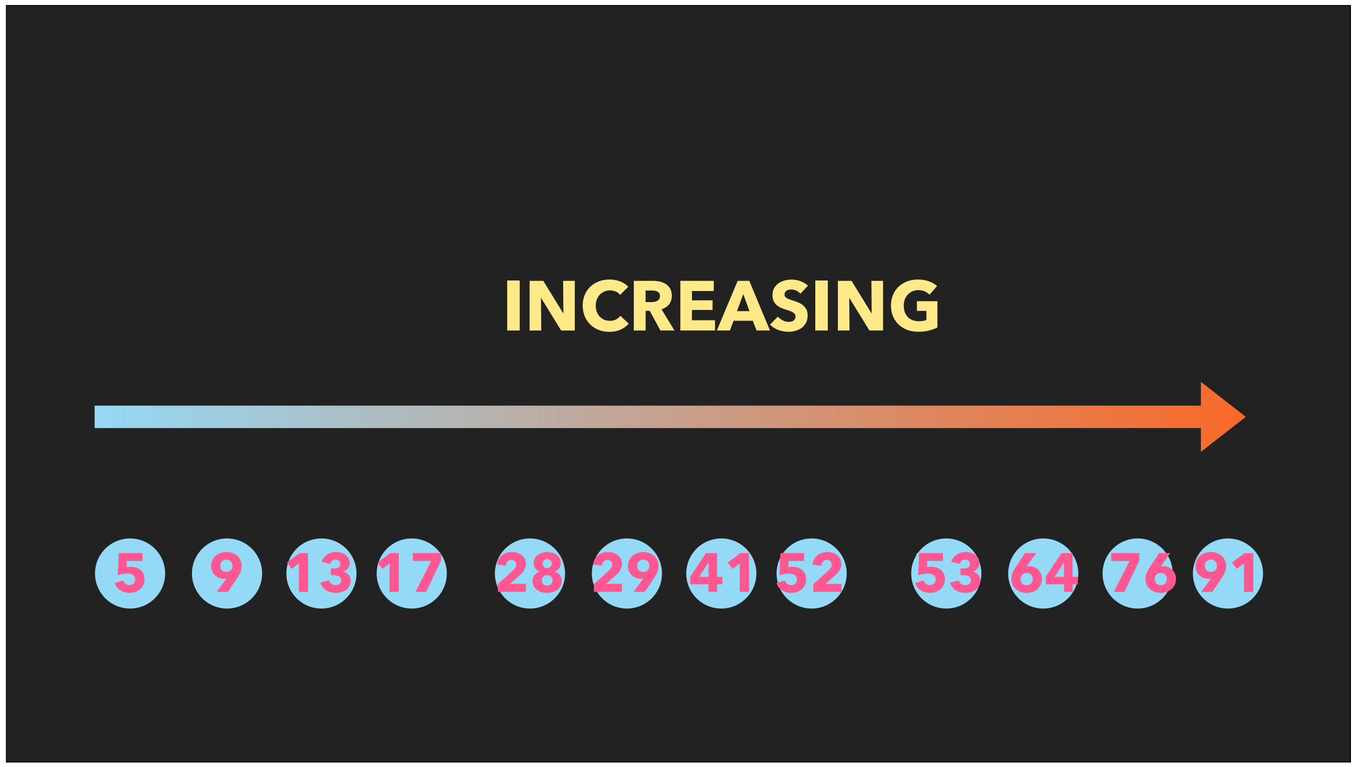
Now in contrast to B-tree-based indexes, which can be used to locate individual records but whose strength is really to quickly discover records associated with a range of keys, the hash map is an index structure whose purpose is to assign **individual** records to, and, later locate in, an array; call it a



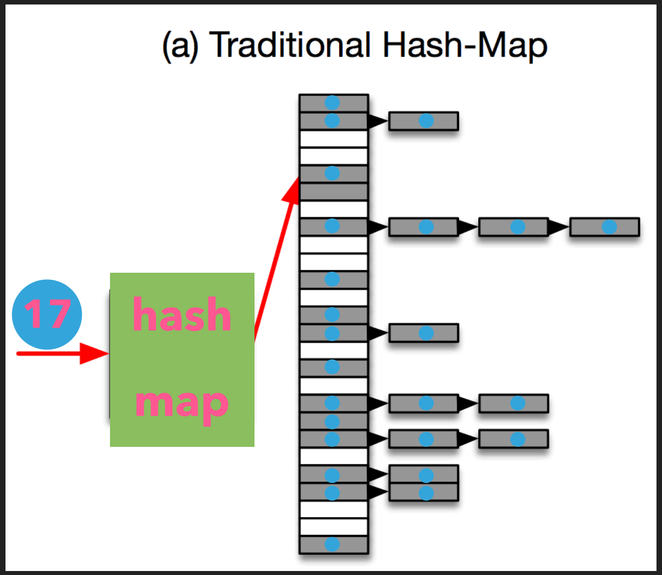
hash map

point index

point index. Viewed as a model, again we have the situation where key goes into the black box and record location comes out, but whereas in the previous case the records were all

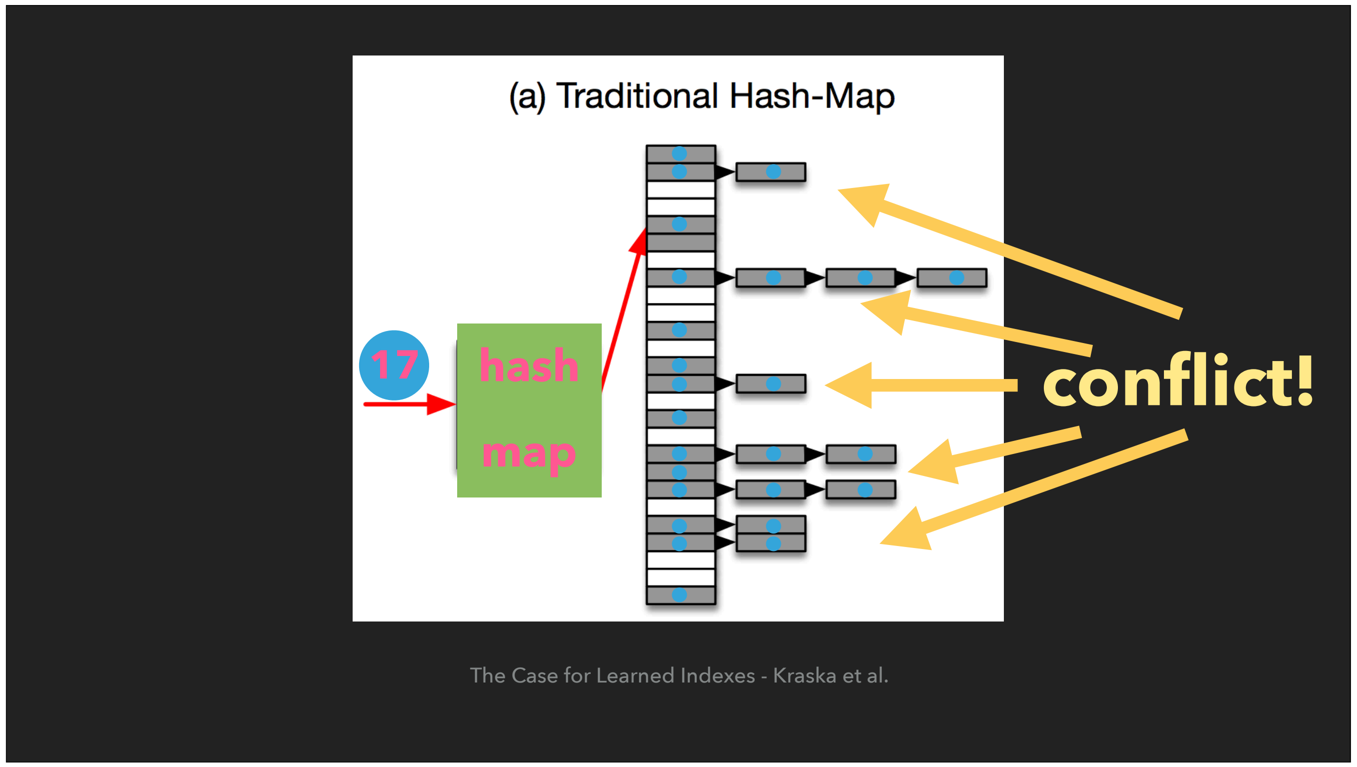


sorted and adjacent to one another, in the point index case the location of the keys in the array



The Case for Learned Indexes - Kraska et al.

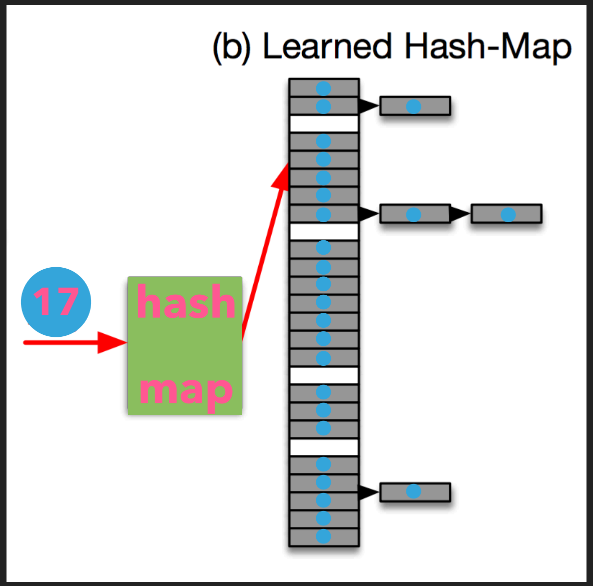
is assigned randomly (albeit via a deterministic hash function). What typically happens is that multiple keys are assigned to the same location, a situation known as a



conflict. Thus what the model points to may not in fact be a record at all, but, say a list of records that needs to be traversed.

Now in an ideal situation, there are no conflicts,

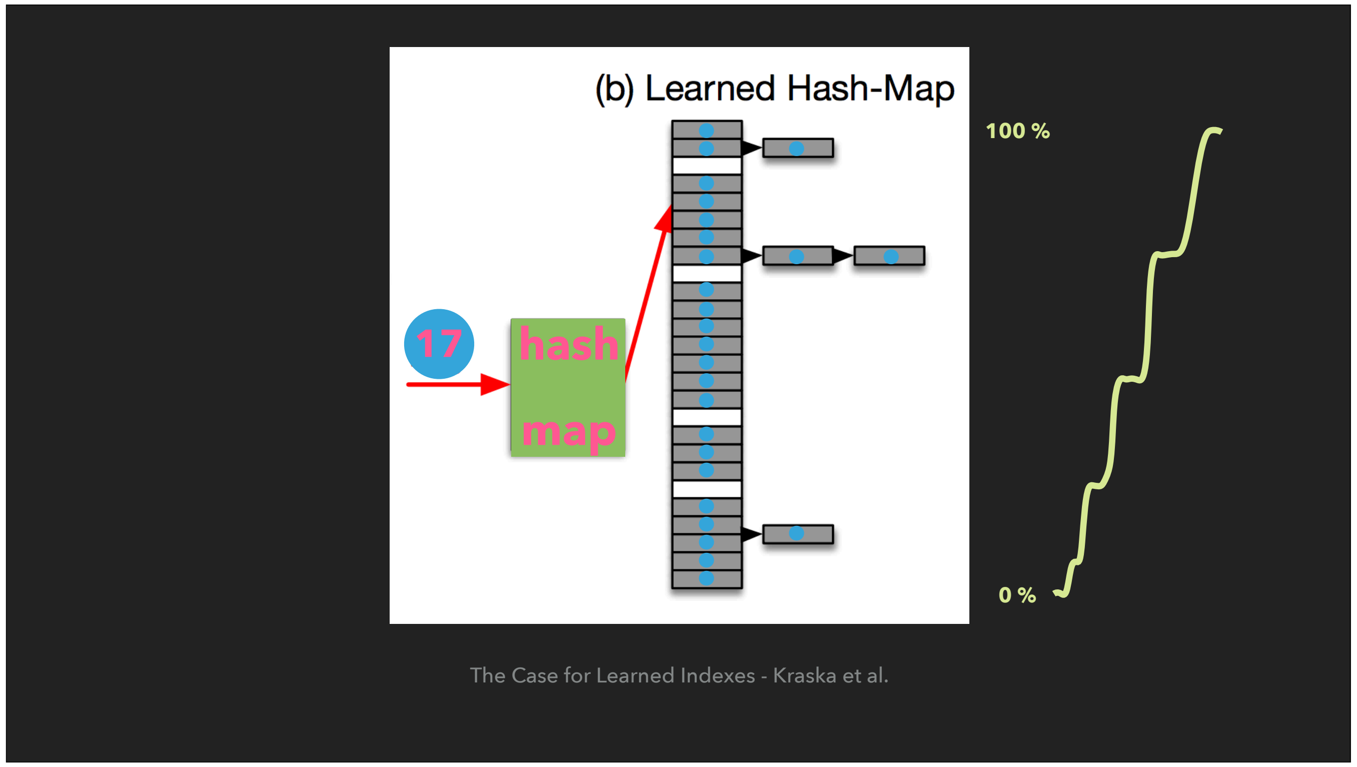
as then lookup becomes a constant-time operation and no extra space needs to be reserved for the overflow. But in the situation when the number of keys equals the number of array slots, because statistics, collisions are unavoidable using naive hashing strategies, and collision avoidance strategies cost either memory or time. So what we would want from our hashing model is



The Case for Learned Indexes - Kraska et al.

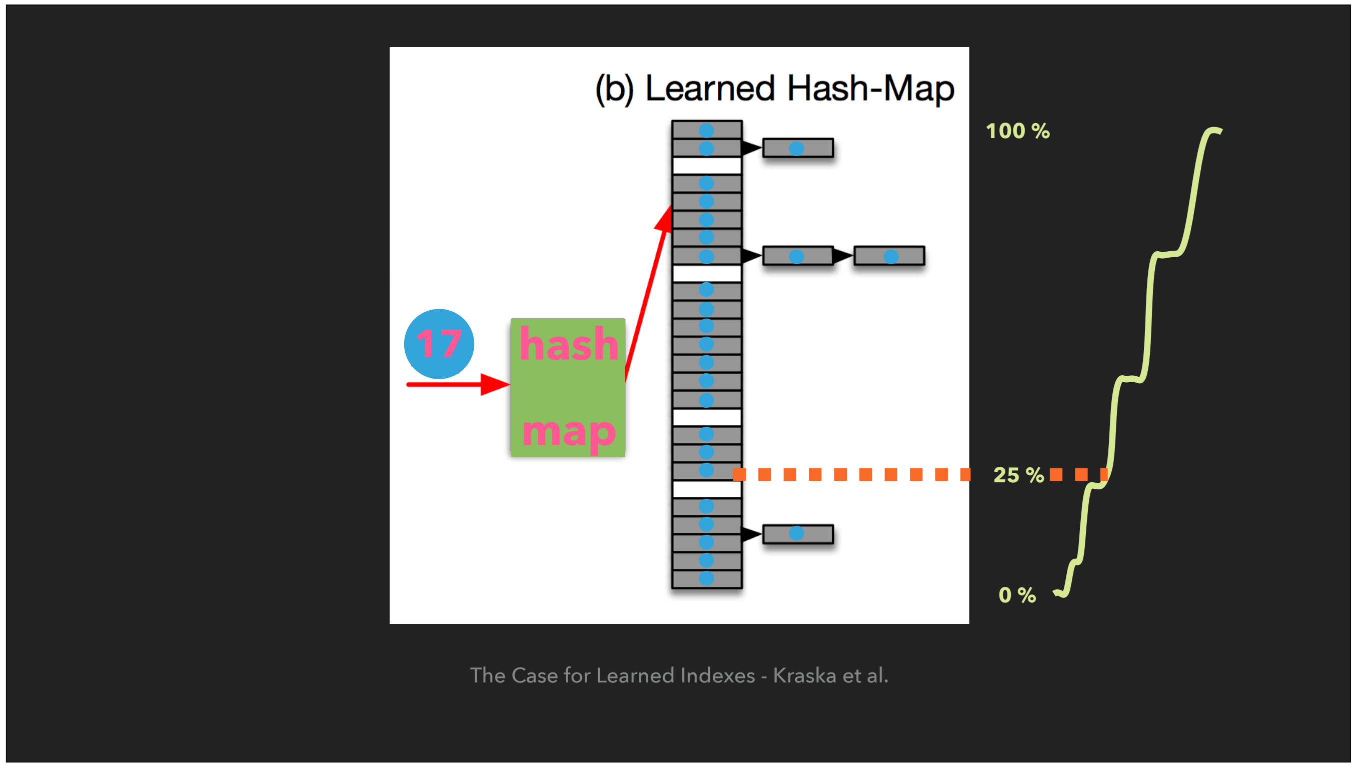
to make location assignments as efficiently as possible, filling every available slot in our array.

To do so, the proposal of Kraska et al is to, again, have the machine learning model learn



The Case for Learned Indexes - Kraska et al.

the cumulative key distribution. That is, say the model predicts that a particular key is

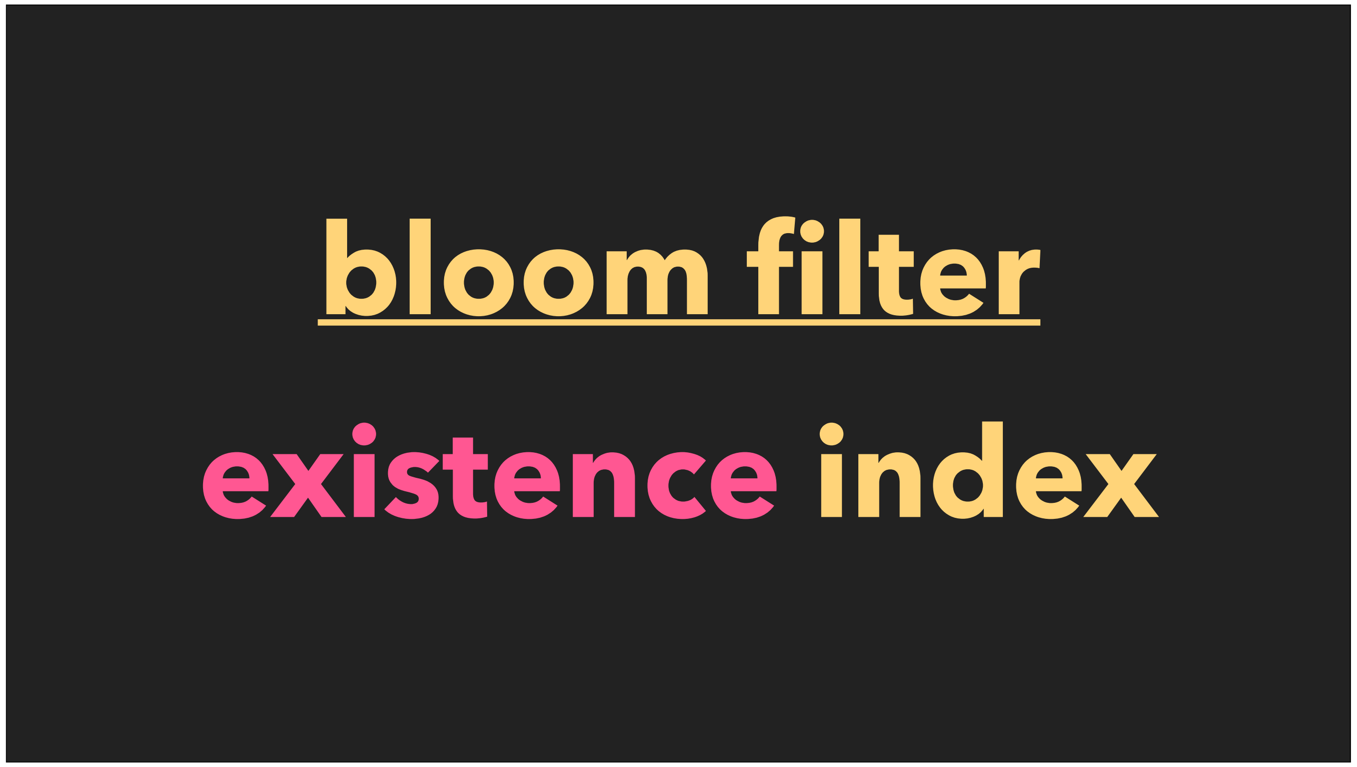


greater than 25% of all keys, then the hashing index tries to insert it 25% of the way along all available array slots. And of course, should there be a collision (which is bound to happen if there are fewer slots than keys), the regular collision resolution techniques could be applied; the point is that by avoiding empty array slots, these costly collision resolution techniques will have to be used less frequently.

That was hash maps.

bloom filter

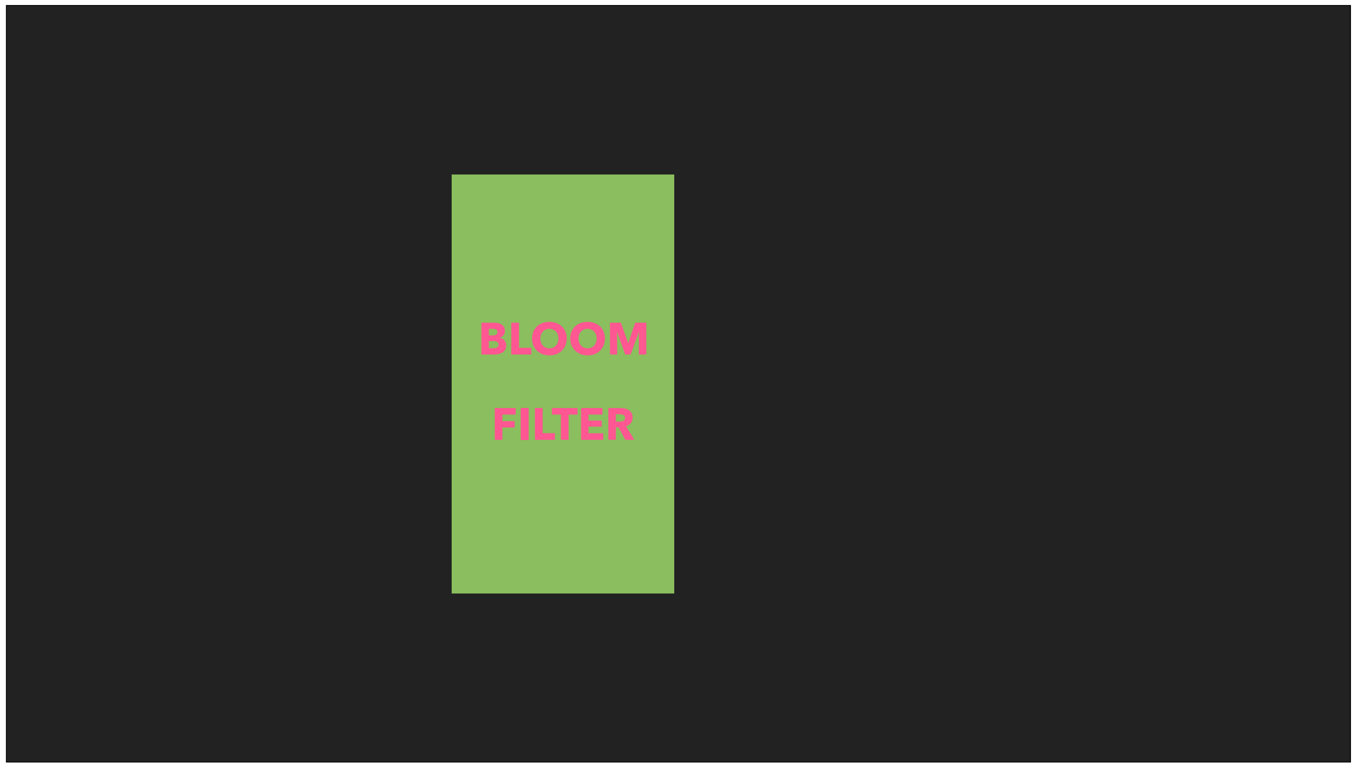
Moving on to Bloom filters, we now are interested in an index structure that indicates



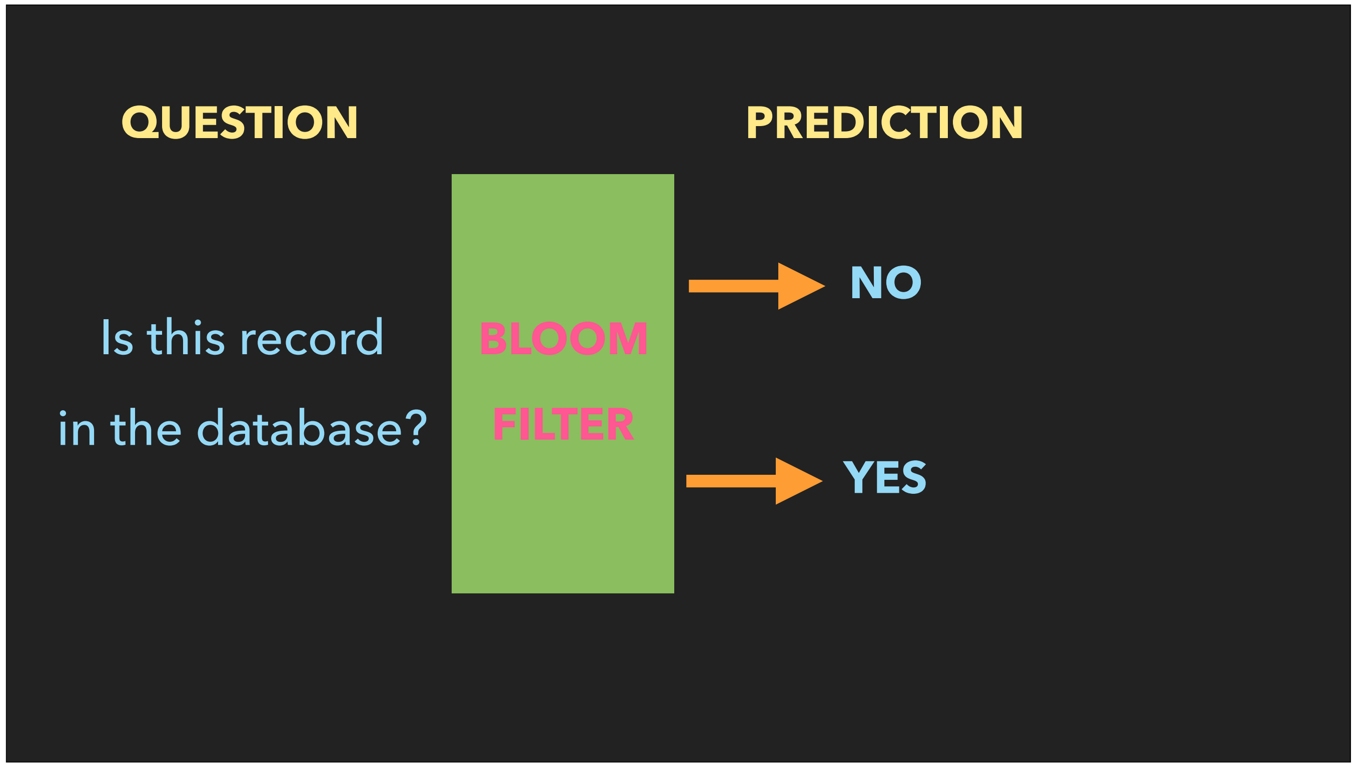
bloom filter

existence index

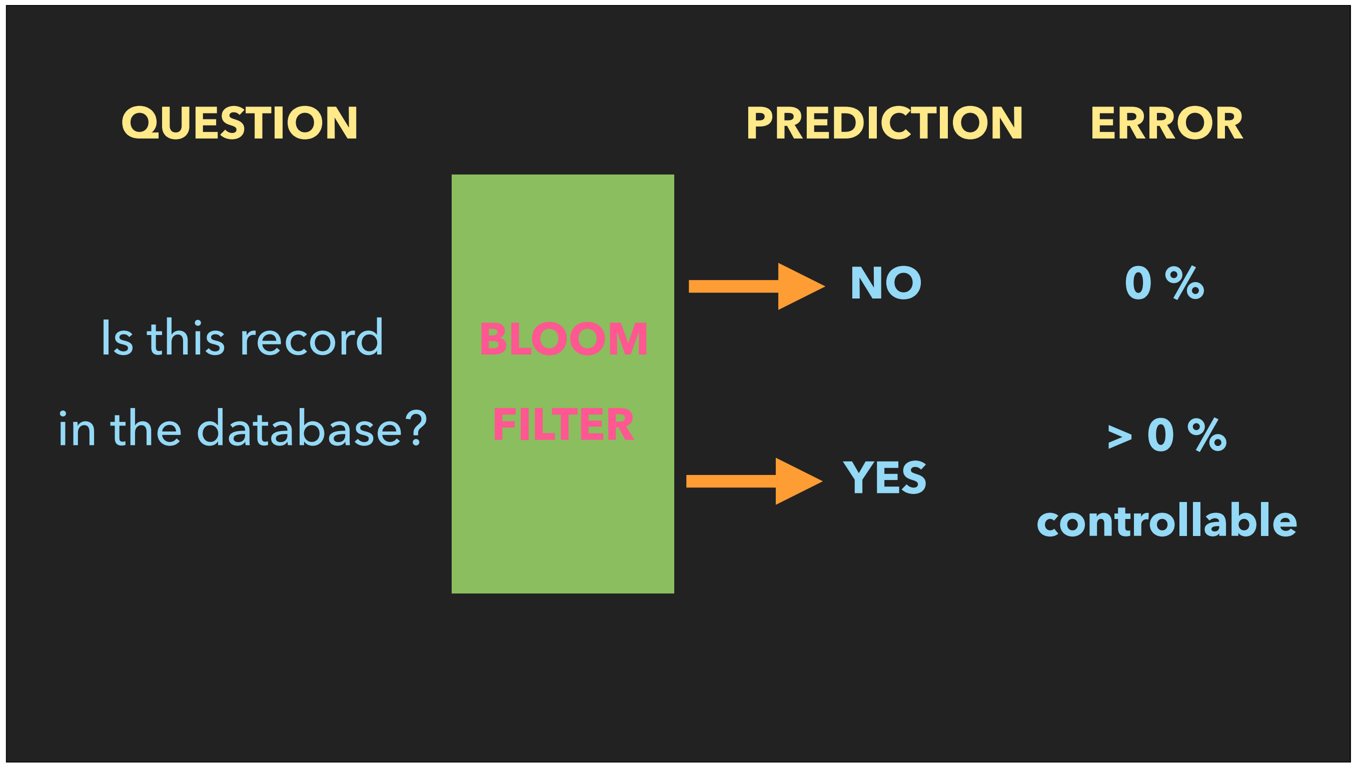
record existence. Specifically, a Bloom filter is a



model that predicts

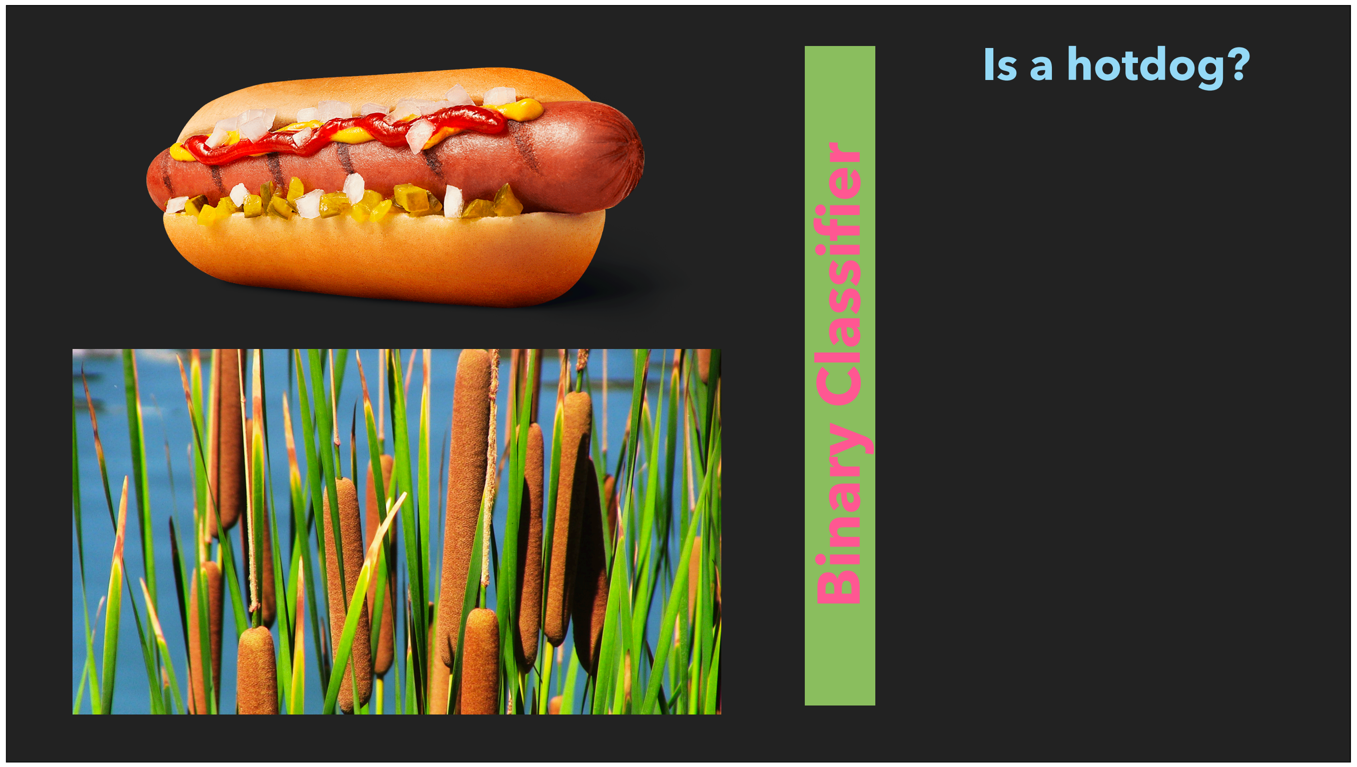


whether or not a particular key is stored in the database, with the additional requirement that

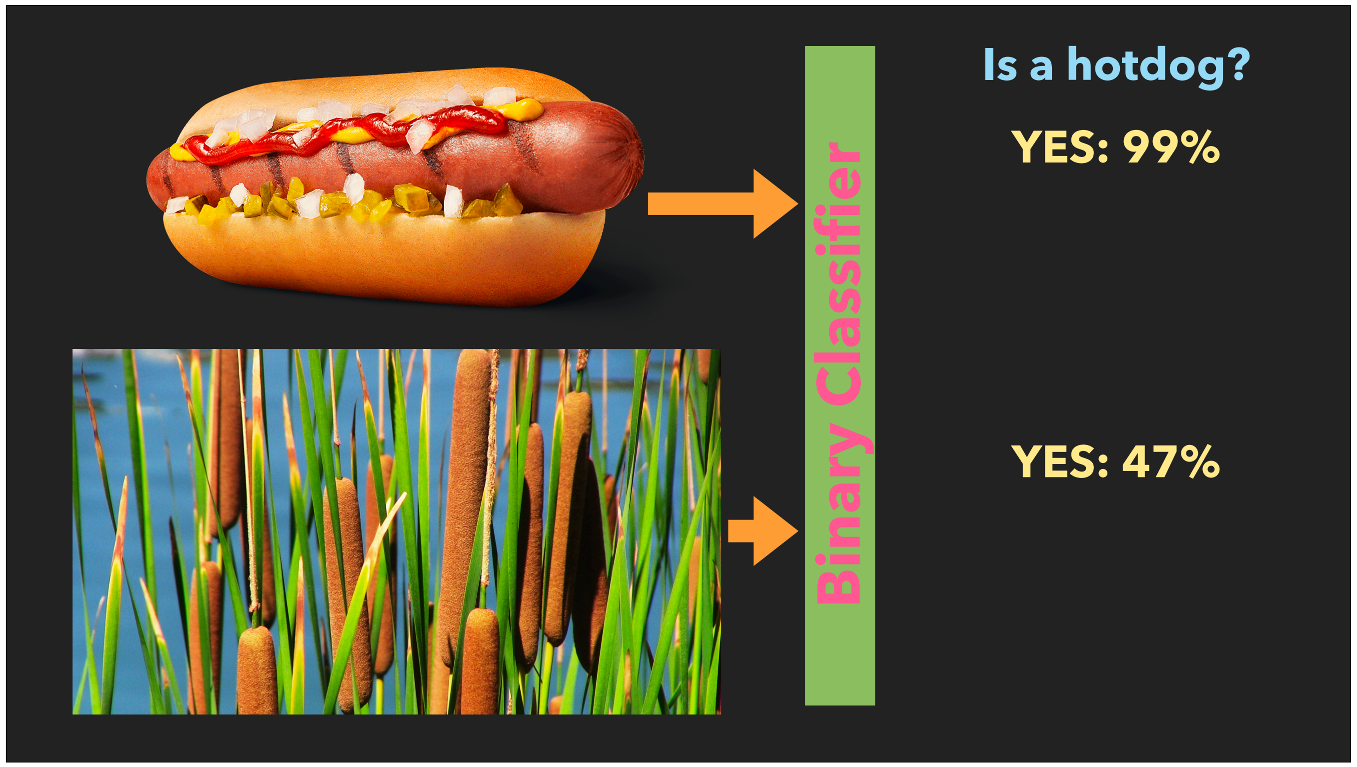


a prediction of "no" have an error of zero and a prediction of "yes" have some error that can be deterministically mitigated, typically by giving the model access to additional compute and memory.

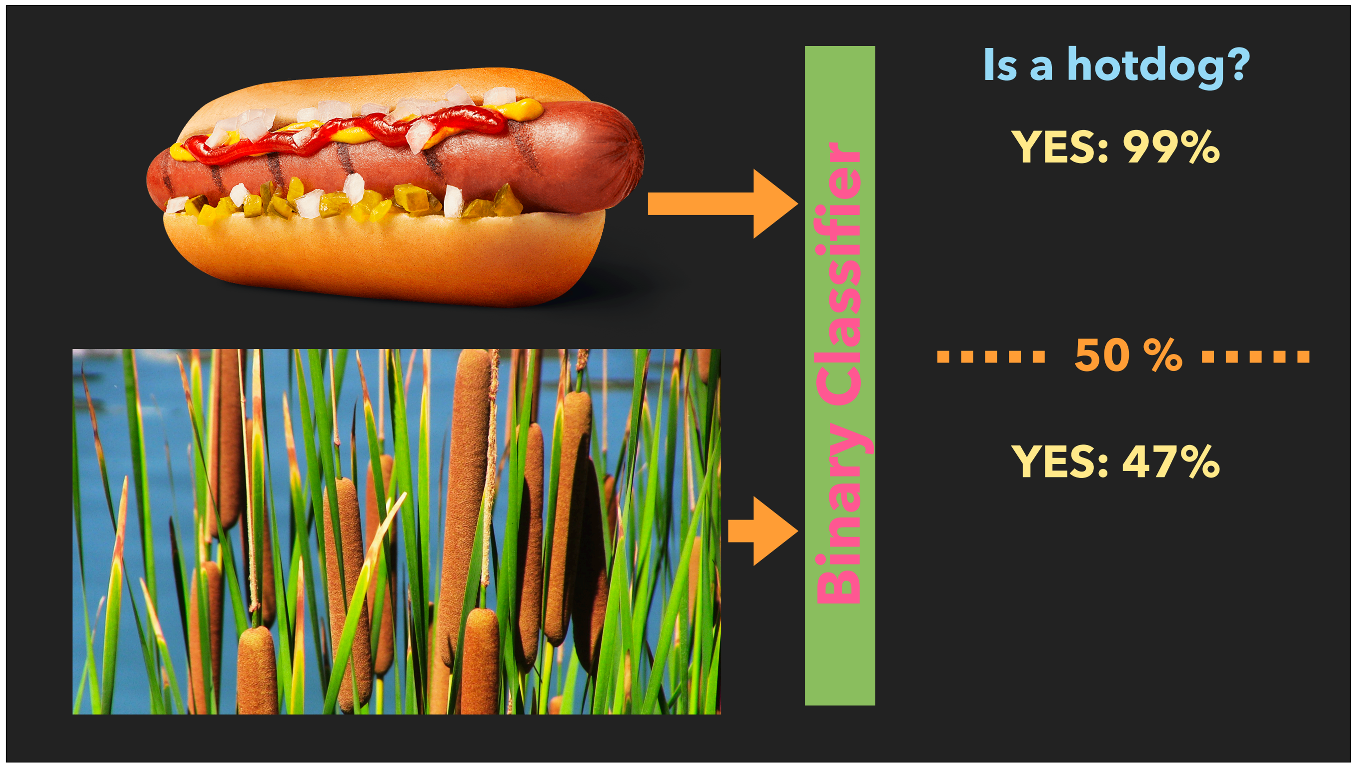
From a machine learning perspective, this seems like a job for a



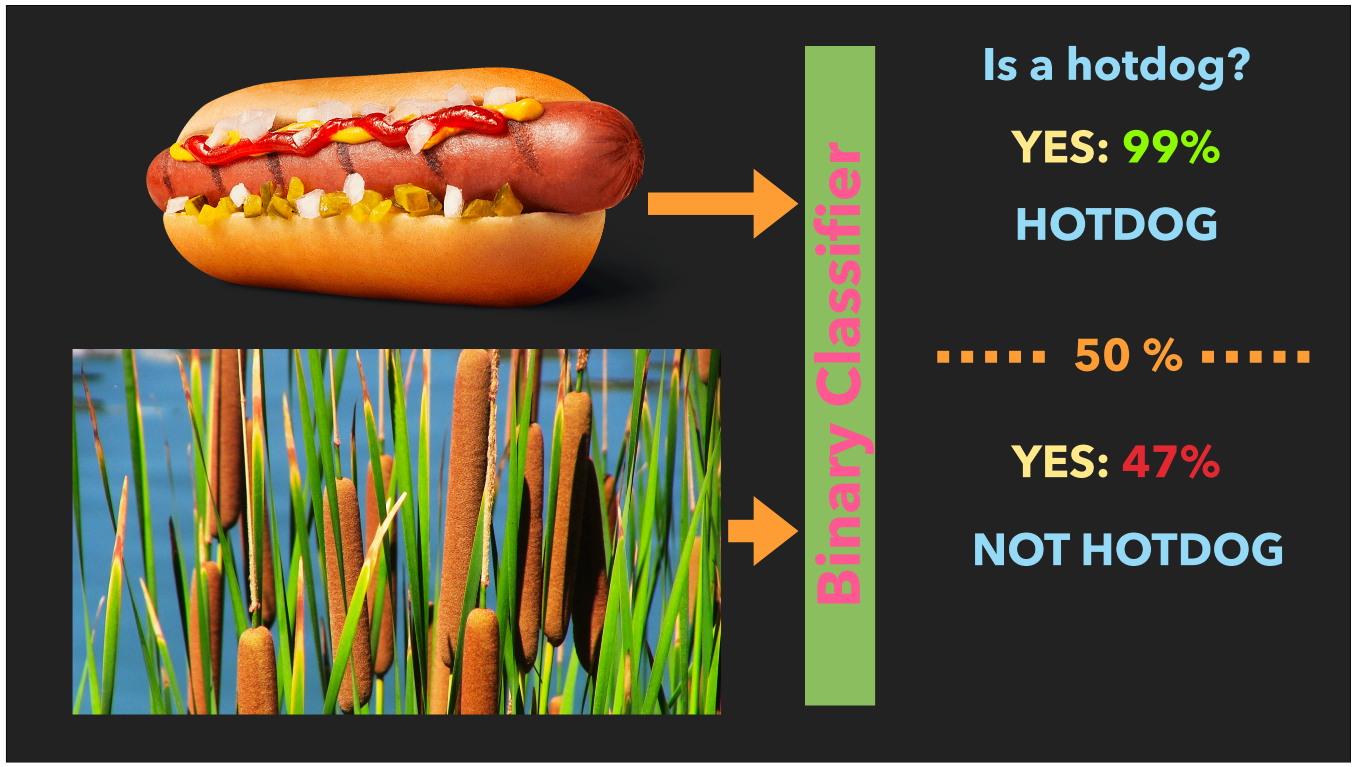
binary classifier, that is, a model which predicts a



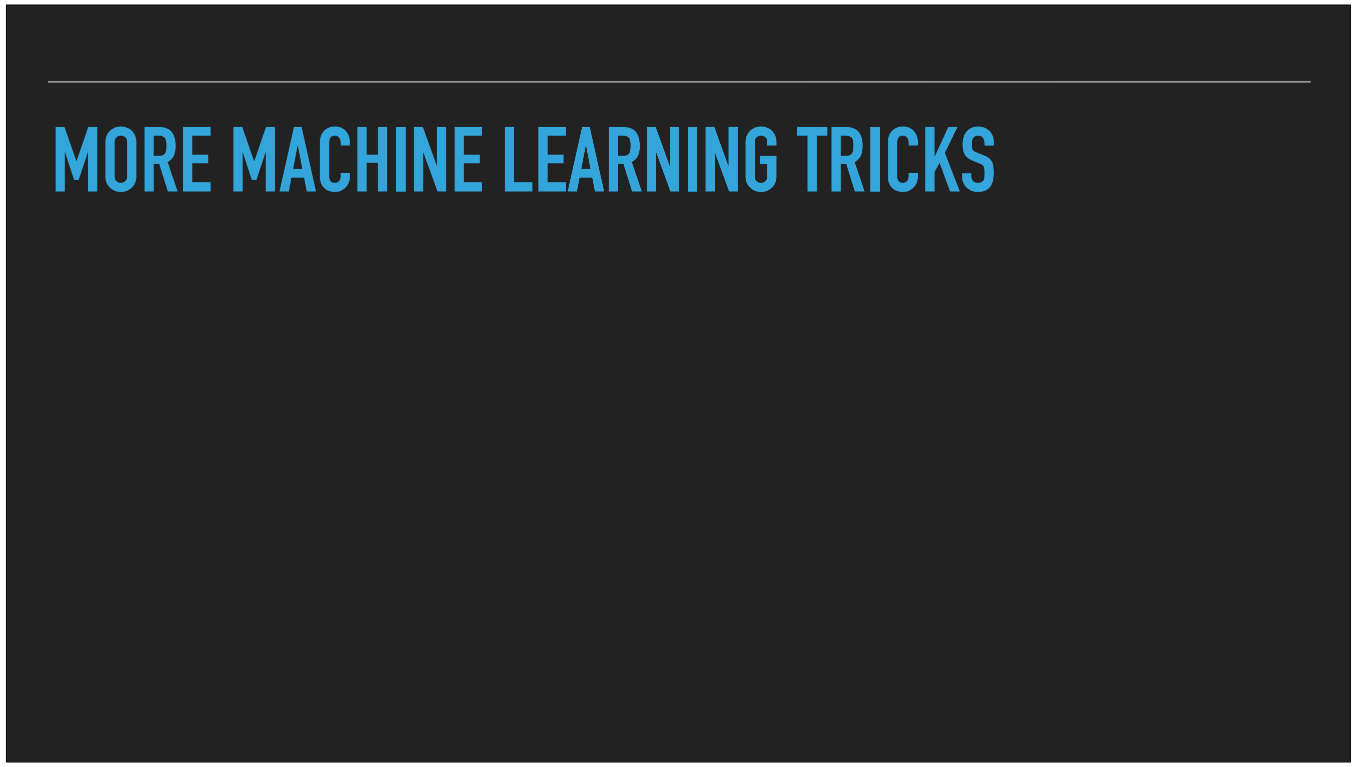
percentage between zero and one hundred, and has a



threshold value such that



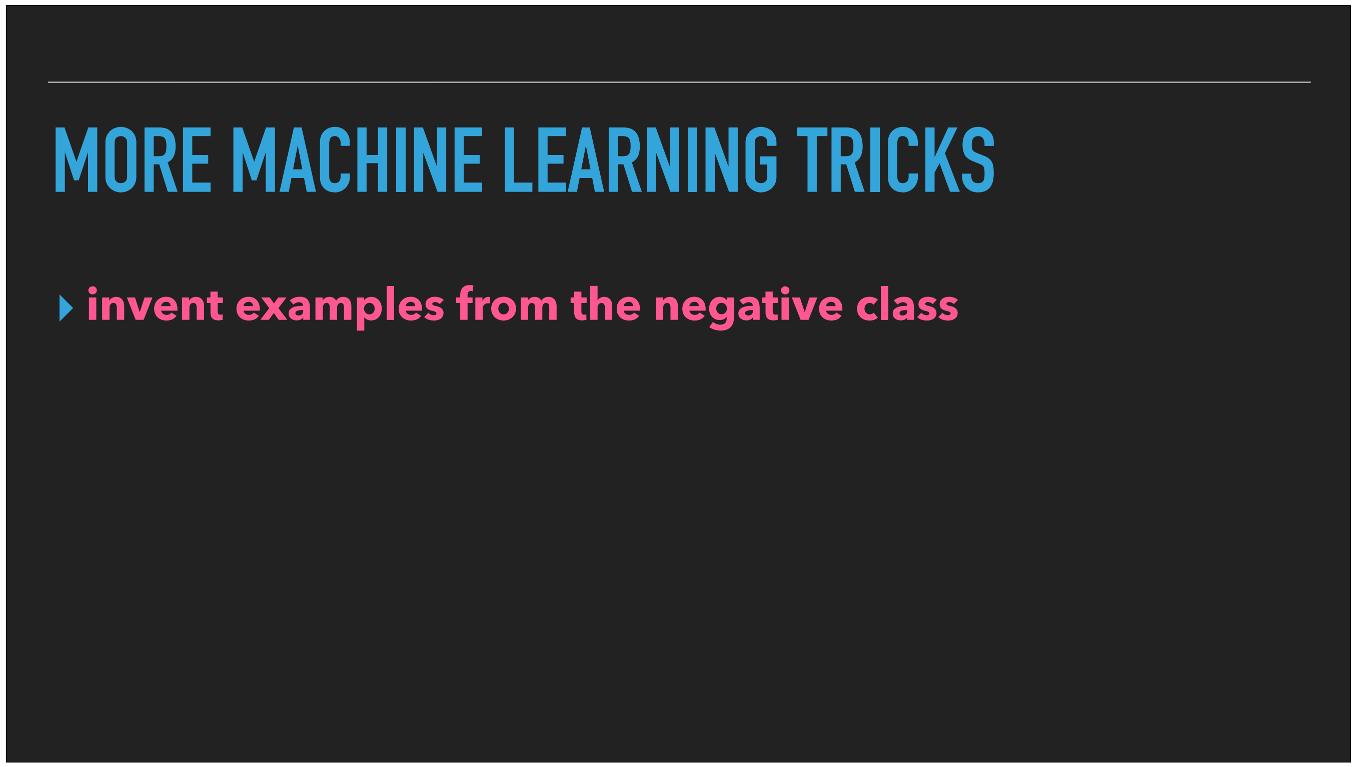
predictions above that number are classified as being in the database and predictions below are classified as not being in the database.



MORE MACHINE LEARNING TRICKS

Just as in the range and point index scenarios, we have to break with standard machine learning methodology, but this time we do it in a different way. Specifically, usually when we train a binary classifier we feed the model examples of both classes, but in this case we have only examples of the positive class, that is, of keys that are actually in our database.

So the first trick we have to use is

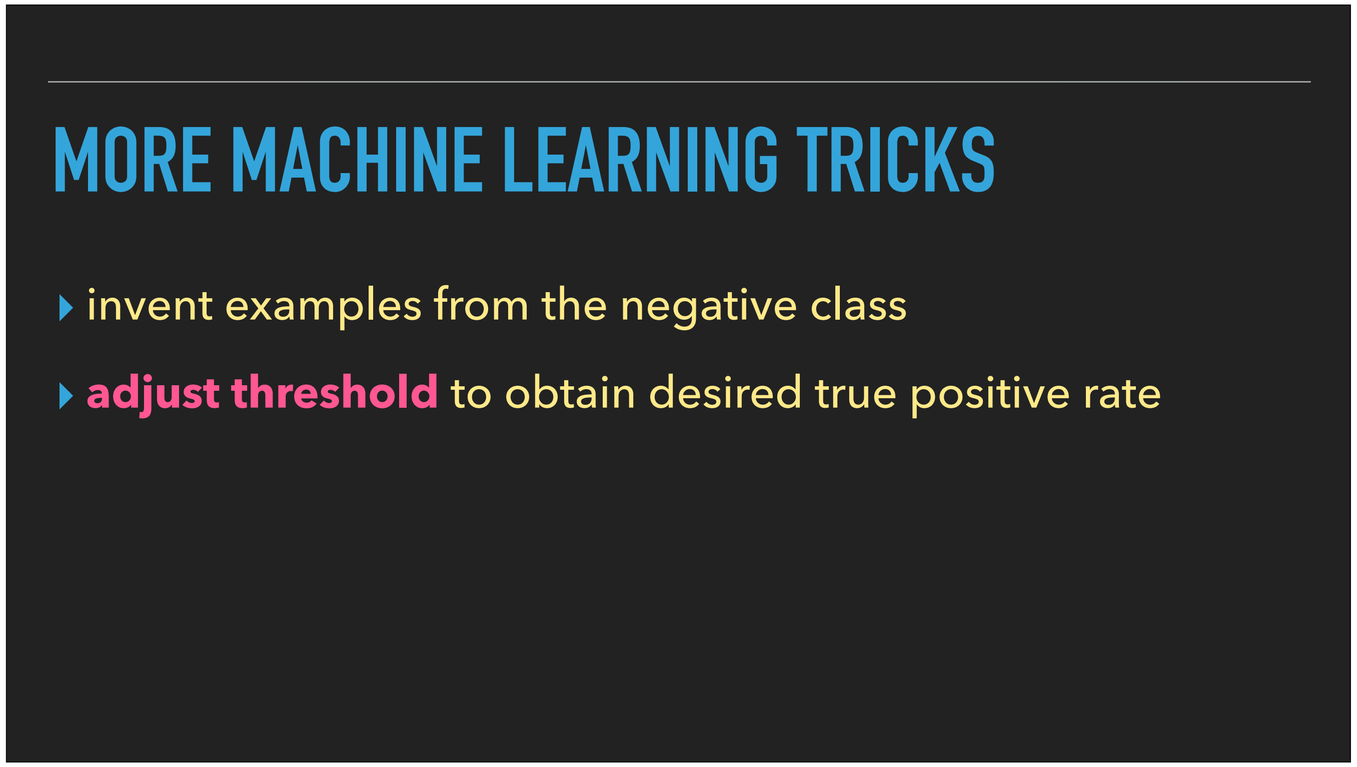


MORE MACHINE LEARNING TRICKS

- invent examples from the negative class

to make up fake keys, that is, values which come from the allowed keyspace but which are not actually used by our records. These "fake keys" we add to the collection of real keys and use to fit our model.

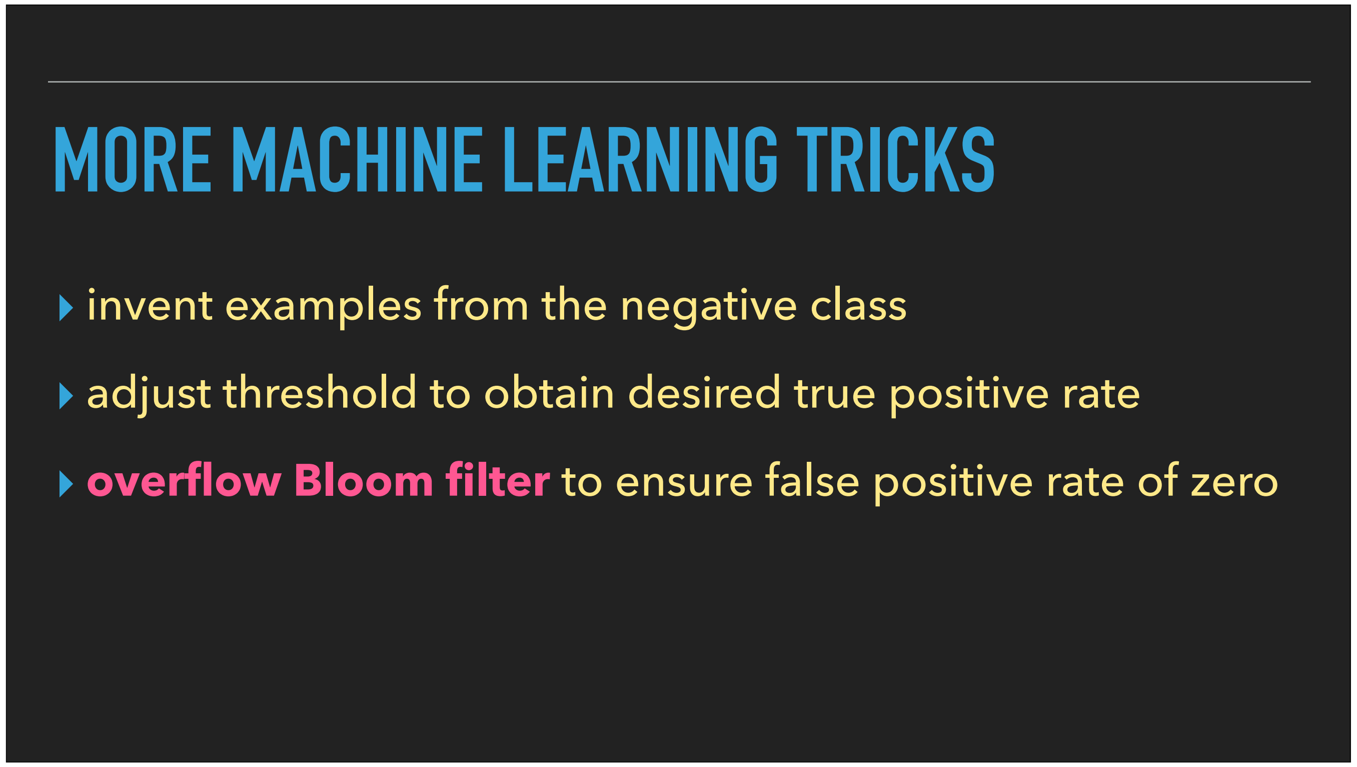
The second trick we use is



MORE MACHINE LEARNING TRICKS

- invent examples from the negative class
- **adjust threshold** to obtain desired true positive rate

to adjust the threshold value to match our desired false positive rate. We'll of course still be left with a false negative rate, which you'll remember we need to get down to zero. So trick three is



MORE MACHINE LEARNING TRICKS

- invent examples from the negative class
- adjust threshold to obtain desired true positive rate
- **overflow Bloom filter** to ensure false positive rate of zero

to actually make a separate Bloom filter, a traditional one, which will be applied to all keys predicted by the machine learning model to belong to the negative class as a double-check. And while this may seem as a bit of a cop-out, we still greatly reduce the resources required to implement our existence index. In particular, because Bloom filters scale linearly with the number of keys they're responsible for, and given that the number of keys our overflow Bloom filter will be responsible for scales with the false negative rate of our machine learning model, even if the binary classifier has a 50% false negative rate, we've managed to reduce the size of the Bloom filter we need by half.

[denouement]

Giovanni Fernandez-Kincade

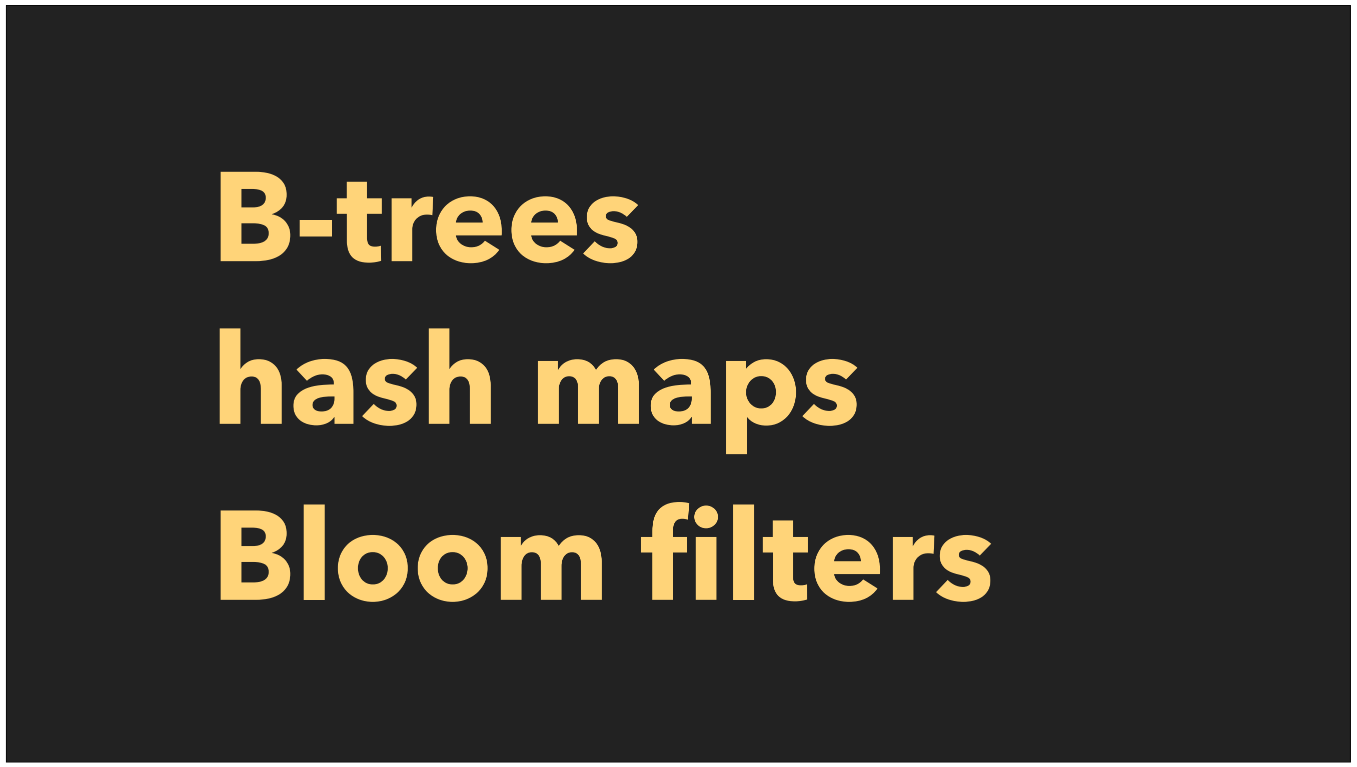
So I've told you about machine learning models could be used to supplant or complement

B-trees

b-trees,

B-trees
hash maps

hash maps,


B-trees
hash maps
Bloom filters

and bloom filters for purposes of range, point, and existence indexing, respectively.

What I **haven't** told you is how well machine learning-based index systems held up against the classical counterparts. So,

results:

how were Kraska et al’s benchmarking results?

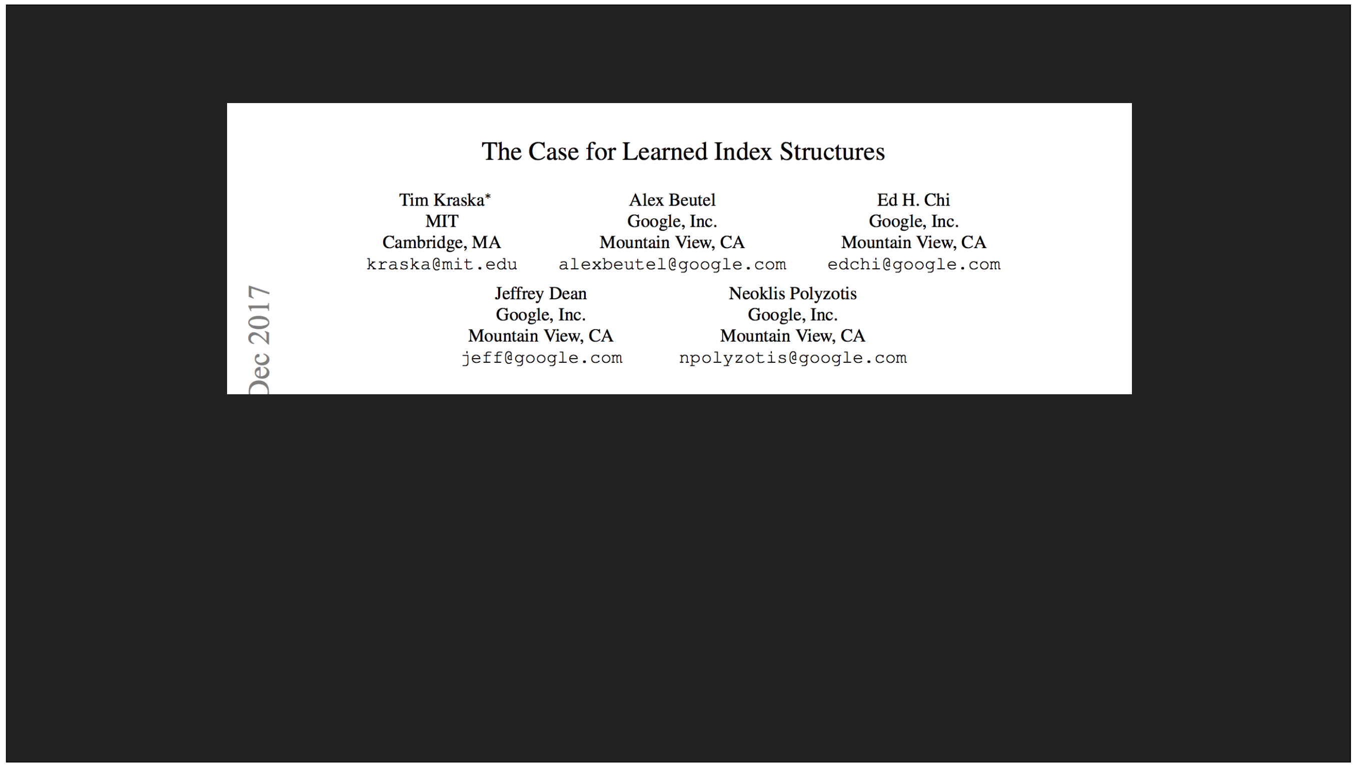
results: good

Well, the results were good. At least,

results: good (according to Google)

according to Kraska et al. And while I don't have time today to go into the details of the benchmarks - and nor am I as a data scientist really on solid-enough footing to be able to evaluate the appropriateness of these tests - I think I can confidently say that this idea has opened up the possibility for new research programs which, especially given the increasing likelihood for the inclusion of machine-learning friendly GPUs and potentially even TPUs in commodity hardware, may very well result in the adoption by future database systems of the ideas I talked about today.

Rounding up, I would like to thank


The Case for Learned Index Structures

Tim Kraska[*]
MIT
Cambridge, MA
kraska@mit.edu

Alex Beutel
Google, Inc.
Mountain View, CA
alexbeutel@google.com

Ed H. Chi
Google, Inc.
Mountain View, CA
edchi@google.com

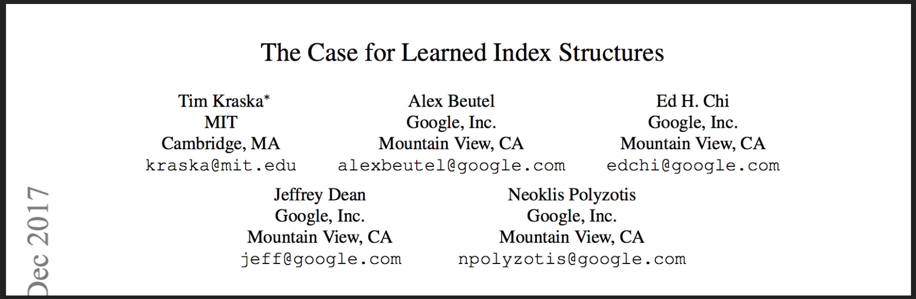
Jeffrey Dean
Google, Inc.
Mountain View, CA
jeff@google.com

Neoklis Polyzotis
Google, Inc.
Mountain View, CA
npolyzotis@google.com

Dec 2017

Tim Kraska, Alex Beutel, Ed Chi, Jeff Dean, and Neoklis Polyzotis for their novel idea;

my employer,

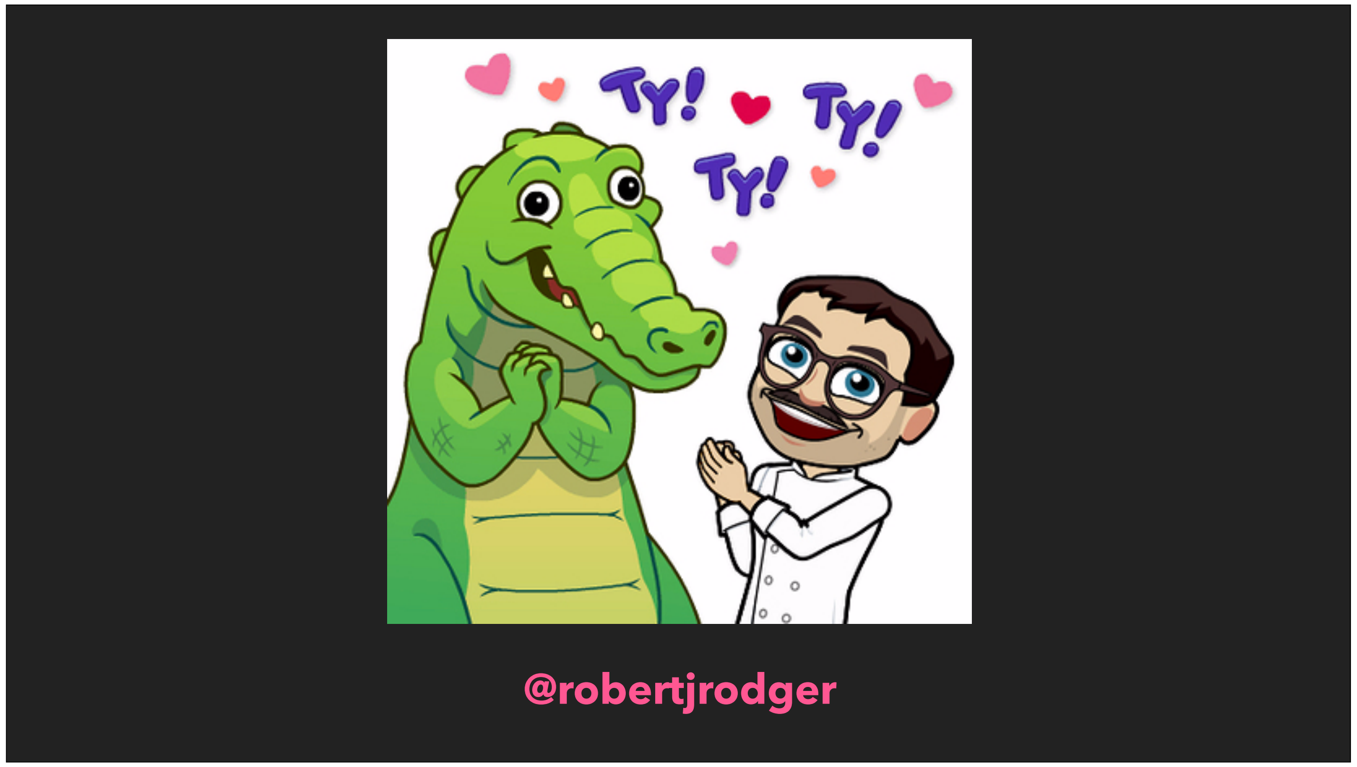

The Case for Learned Index Structures

Tim Kraska* MIT Cambridge, MA kraska@mit.edu
Alex Beutel Google, Inc. Mountain View, CA alexbeutel@google.com
Ed H. Chi Google, Inc. Mountain View, CA edchi@google.com
Jeffrey Dean Google, Inc. Mountain View, CA jeff@google.com
Neoklis Polyzotis Google, Inc. Mountain View, CA npolyzotis@google.com

Dec 2017

GoDataDriven

GoDataDriven, for flying me out here and letting me speak on company time; and



@robertjrodger

you, the audience, for your attention.

And now, if there is any time for questions, I would be happy to field them.