MIT CSAIL FastCode Seminar: Professor Michael Bender 11/02/2020

1 00:00:03.689 --> 00:00:10.200
Julian Shun: Alright everyone, welcome to the fast code seminar. So today, I'm very happy to have Michael Bender as our speaker.

2 00:00:10.740 --> 00:00:20.490
Julian Shun: Michael is the David Smith leading scholar of computer science at Stony Brook University and his research spans data structures algorithms.

3 00:00:20.910 --> 00:00:27.360
Julian Shun: Iowa efficient computing parallel computing and scheduling and he's co-authored many papers on these topics.

4 00:00:28.170 --> 00:00:37.770
Julian Shun: He's received many awards for his excellent contributions, both in research and also in teaching, including an R amp D 100 award a test of time award.

5 00:00:38.190 --> 00:00:50.700
Julian Shun: Two best paper award and five awards for graduate and undergraduate teaching Michael was the founder and chief scientist at Tokyo tuck and enterprise database company, which was acquired by

6 00:00:52.980 --> 00:00:58.680
Julian Shun: Michael's also held visiting scientist positions at MIT and King's College London.

7 00:00:59.790 --> 00:01:14.580
Julian Shun: Michael received his bachelor's in applied math from Harvard and add a computer science from Ian s and Leo and France, and he completed his PhD on scheduling algorithms from Harvard.

8 00:01:15.450 --> 00:01:28.020
Julian Shun: And today, Michael's going to tell us about his work on filters and he's going to talk about many filters, including Bloom filters quotient and cuckoo filters. So I'll turn it over to you, Michael.

9 00:01:29.610 --> 00:01:41.130
Michael Bender: Great. Thank you so much. It's really great to be here and also looking at the names of, you know, people coming in and it's really nice to see friends who I haven't spoken to in a long time. And so I'd love to catch up.

10 00:01:41.640 --> 00:01:48.810
Michael Bender: Afterwards, so thanks again. Um, so this is the advertised talk title filters.

11 00:01:50.010 --> 00:01:57.120
Michael Bender: And this is a slightly longer title. Oops, which is time to change your filter.

12 00:01:58.950 --> 00:02:04.980
Michael Bender: And this title is more to the point, but I'll explain why give a little more explanation why later.

13 00:02:06.330 --> 00:02:15.750
Michael Bender: But to begin in order to explain what I mean by a filter, I have to say to say what's the filter. I actually need to say, first of all, what's a dictionary.

14 00:02:17.040 --> 00:02:26.730
Michael Bender: And the dictionary data structure maintain some set es from the university use. So in this picture. The set is elements AMC in the universe is a, b, c, d.

15 00:02:28.110 --> 00:02:31.380
Michael Bender: And the dictionary supports membership queries on the set.

16 00:02:32.130 --> 00:02:43.890
Michael Bender: So if we ask is a member of the set. Well, there's a yes it is a member of the set is be a member of the set know be is not a member of the set is see a member of the set. Yes, he's a member of the set is the number this that know

17 00:02:44.370 --> 00:02:50.820
Michael Bender: The dictionary supports membership queries on some potentially dynamically changing set

18 00:02:51.960 --> 00:03:03.510
Michael Bender: And a filter is just an approximate dictionary. So yeah, so filter data structure. It's that approximate dictionary. And so, it supports approximate membership queries on the set.

19 00:03:04.140 --> 00:03:14.730
Michael Bender: So here we've got is a in the set. Yes, is be in the set. No be is not in the set, you see in the set. Yes, he is in the set is d in the set. Well, D is not in the set.

20 00:03:16.320 --> 00:03:23.820
Michael Bender: But here the filter makes a mistake. It has a false positive. And so it erroneously says yes theist in the set.

21 00:03:24.540 --> 00:03:35.280

Michael Bender: So this is an approximate dictionary, because it's allowed to make some mistakes. It turns out it's only allowed to make false positives as I'll explain later, it doesn't make false positives and false negatives.

22 00:03:35.640 --> 00:03:41.580
Michael Bender: So this is a filter data structure. It's an approximate that membership query data structure on some set S.

23 00:03:43.680 --> 00:03:48.540
Michael Bender: And so just said to a filter guarantees some false positive rate epsilon

24 00:03:50.160 --> 00:03:58.920
Michael Bender: And so what that means is that if you query for some element that is actually in your dictionary, then you have to return. Yes.

25 00:03:59.520 --> 00:04:20.880
Michael Bender: With probability one. So this is a true positive, but if the query element is not in the set. Most of the time you have to say no. So with probability one minus epsilon, you have to say no. And these are negatives but you are allowed to have some bounded probability of having false positives.

26 00:04:22.020 --> 00:04:29.100
Michael Bender: Right, so this is so, so this is the false positive rate. And as I said before, this is, you know, these are one sided errors.

27 00:04:30.270 --> 00:04:38.370
Michael Bender: Okay, so this is why I'm filter guarantees of false or not. Why, but that the filter guarantees some bounded false positive rate epsilon

28 00:04:40.350 --> 00:04:40.740
Okay.

29 00:04:41.760 --> 00:05:00.570
Michael Bender: Um, so the reason why we want this false positive rate epsilon is because we want the data structure to be really compact. So if you're storing an entire dictionary. It could be very large. And this is a picture this whale supposed to be a large dictionary for the space.

30 00:05:02.250 --> 00:05:21.480
Michael Bender: Is sort of omega of end times log of the universe size where the filter. If you have a reasonable false positive rate, then the filter is going to be much tinier. And so that's the advantage. It allows the false positive rate is the false positive rate allows the filter to be more compact.

31 00:05:21.660 --> 00:05:22.680

Michael Bender: So the point, whatever.

32 00:05:22.710 --> 00:05:24.180
Charles Leiserson: This thing question.

33 00:05:24.390 --> 00:05:25.740
Michael Bender: This is Charlie. Oh, good. Yeah.

34 00:05:27.090 --> 00:05:33.360
Charles Leiserson: I didn't quite understand the space over which the randomness is occurring. If you have the same element will it fail.

35 00:05:34.560 --> 00:05:41.250
Charles Leiserson: Twice, or is it something that's random over over you know the particular query or whatever.

36 00:05:42.090 --> 00:05:44.760
Michael Bender: So that is a fantastically good question.

37 00:05:47.100 --> 00:05:50.880
Michael Bender: And so here I'm purposely being a little vague.

38 00:05:51.930 --> 00:05:52.980
Michael Bender: But in fact,

39 00:05:53.040 --> 00:06:09.690
Michael Bender: A couple of years ago, we had a false positive. We had a fox paper, which was basically answering exactly this question I've been going back one slide to the false positive rate where we actually made a data structure that didn't make it where even if you have repeats. You still

40 00:06:11.280 --> 00:06:24.240
Michael Bender: Have the same false false positive rate of epsilon, but with most filters that are not adaptive that's not the case. And so I'm being a little bit vague about this and if you read filter papers.

41 00:06:24.810 --> 00:06:36.180
Michael Bender: People are often a little bit vague about whether this is for random queries or whether it's for a specific query or whether it's like sort of an epsilon false positive rate for every query in the stream.

42 00:06:37.140 --> 00:06:52.560
Michael Bender: And so you can do it for every query in the stream, but I'm not really going to go there. So I'm just saying. For now, let's think about it as one particular query, but that's a

really, really good question. Charles and sort of one of the things that I'm actually most excited about the filters.

43 00:06:54.330 --> 00:06:54.690
So,

44 00:06:55.830 --> 00:07:04.710
Michael Bender: Oh, I should say one more thing, which is I find the lack of personal bank interaction and bought and the lack of actually having body language.

45 00:07:06.690 --> 00:07:09.000
Michael Bender: Visible, I find that very

46 00:07:10.680 --> 00:07:29.640
Michael Bender: Disturbing. So the more questions, and the more interaction. I can have, the better. So like if I tell a joke. And if the joke happens to be funny, then definitely like snap into action and unmute your microphone. LAUGH And then muted again like anything for the interaction.

47 00:07:31.560 --> 00:07:32.520
Michael Bender: Between very grateful.

48 00:07:33.120 --> 00:07:36.180
David Reed: So, so Michael since you invited. This is David Reed.

49 00:07:36.990 --> 00:07:37.710
Michael Bender: Yeah, you bet.

50 00:07:41.370 --> 00:07:46.950
David Reed: You're, you're also assuming, and I don't know whether it's an essential assumption that there is a stream.

51 00:07:48.780 --> 00:07:58.800
David Reed: As opposed to, say, just a one shot query and there will never be one or maybe something that's not as structured as a stream.

52 00:08:01.020 --> 00:08:02.940
David Reed: You know when you said stream. Well, maybe.

53 00:08:04.170 --> 00:08:12.090
David Reed: You know the ordering matters or doesn't matter in terms of, you know, filters, like I assume you're trying to be general but but do you

54 00:08:12.360 --> 00:08:16.920

Michael Bender: So right now I'm sort of finessing the issue in the sense, I'm saying it.

55 00:08:16.980 --> 00:08:18.720
Michael Bender: Could be one query like

56 00:08:19.200 --> 00:08:23.400
Michael Bender: For the, for the moment, it probably makes sense to think about just the single query.

57 00:08:26.130 --> 00:08:32.400
Michael Bender: And and we want these guarantees for a single query, but the point that Charles brings up is

58 00:08:32.910 --> 00:08:41.850
Michael Bender: That like you can imagine that you have an adversary that asks a query. And then based on the results of that query asks another query and based on the results of that query asks another query.

59 00:08:42.810 --> 00:08:50.370
Michael Bender: And so ideally you want to filter that can do all of that. And in fact, you can. And I'll give you the citation. A little bit later in the talk.

60 00:08:50.790 --> 00:09:00.930
Michael Bender: But I think, for the moment, it makes sense to think about just a single query like how do you make the filter so that so that if whatever query the adversary asks you can answer it. Does that make

61 00:09:00.990 --> 00:09:07.470
David Reed: Sense. Yeah, it makes sense. It's just that I'm interested in the order dependent queries.

62 00:09:07.800 --> 00:09:20.340
Michael Bender: Piece. Oh, I'm happy, I'm so, so I'm happy to. So I'm happy to go there. Uh, you know, later and give you much more detail about that because I can wax eloquent about that for hours.

63 00:09:21.060 --> 00:09:24.240
Michael Bender: Right. So, and I'm happy to talk more about that afterwards.

64 00:09:24.450 --> 00:09:30.840
Michael Bender: And if there's time afterwards. Sort of. Sort of wedge in and tell you the piece where the order dependence comes into play.

65 00:09:32.490 --> 00:09:45.330

Michael Bender: So, um, so these are really this is these are not. These are great questions. And it's you've touched upon something that I'm very excited about it wasn't what I the questions that I expected to get but I'm very happy to get them.

66 00:09:46.410 --> 00:09:53.760
Michael Bender: I guess before going on. I should say here with like this is a picture of a filter right here because it's a CNN Money. That's a filter.

67 00:09:54.810 --> 00:10:05.850
Michael Bender: So it's a filter feeder. So originally I had a dictionary. That was a humpback whale, but then that would also be a filter feeders. So I changed it to a sperm whale, which is not that's

68 00:10:06.450 --> 00:10:13.530
Michael Bender: Unnecessary knowledge. Okay, so let's talk about the talk, so far, which is I described the filter data structure. And we also

69 00:10:13.770 --> 00:10:15.000
Michael Bender: Aha, about

70 00:10:15.150 --> 00:10:17.730
Michael Bender: Adaptive look. Thank you.

71 00:10:18.930 --> 00:10:20.070
Michael Bender: Then we talked about

72 00:10:23.490 --> 00:10:26.730
Michael Bender: Adaptive versus non adaptive filter data structures.

73 00:10:27.750 --> 00:10:36.570
Michael Bender: And now I'm going to talk about the Bloom filter, which is by far the most common data structure. Here's a cartoon of a of a filter that I found on the internet.

74 00:10:37.740 --> 00:10:48.420
Michael Bender: Air Conditioner filter, I guess. So this is a classic Bloom filter and a Bloom filter is just a bit array and some number of K hash functions. And here I'm going to say that cash does too.

75 00:10:49.440 --> 00:10:58.710
Michael Bender: And so this is what a Bloom filter looks like when the set is the empty set. And if I add some element a to the set we Bloom filter works is

76 00:11:00.600 --> 00:11:18.660

Michael Bender: A gets hashed with each of these hash functions and in this picture here. The first hash function hash is a two positions 01 and the second hash function hashes eight positions 0123 and so the bits in these two positions gets flipped from zero to one.

77 00:11:20.280 --> 00:11:40.530
Michael Bender: And then just another picture. Say we hash and say we add see to this set S. And so now see gets hash to position 012345 with this first hash function and 01234 the second hash function. And so those bits also get flipped to one. Notice that this a

78 00:11:41.580 --> 00:11:55.080
Michael Bender: Bit in the position 0123 was already at one. And so, it states that one. And so this is how you insert elements into Bloom filter when you've got these hash functions. These K hash functions.

79 00:11:56.790 --> 00:12:03.480
Michael Bender: And but but the key hash functions is just how you build the Bloom filter. So this is ultimately what it looks like it's just a bit array.

80 00:12:04.830 --> 00:12:09.360
Michael Bender: And so now if we want to ask a query. So here, this is a

81 00:12:12.120 --> 00:12:23.520
Michael Bender: You know that the query for whether be is a member of the set. And so you hash be and this first hash function gets hashtag position 012

82 00:12:24.270 --> 00:12:37.680
Michael Bender: Which is zero and the second one gets hashtag position 012345 which is one. But because this is the zero. We know that be is not in the set, because it be we're in the set. Both of these would already be ones.

83 00:12:38.070 --> 00:12:53.310
Michael Bender: And so when you query is being set, then your filter says no, it's not. And again, since you can't really see my face. I decided that I was going to have my talk with a lot of other cartoon faces. So I hope I spent

84 00:12:54.450 --> 00:13:01.950
Michael Bender: I spent an embarrassingly large amount of time drawing eyes and smiley faces. So hopefully you'll appreciate it and this is my cartoon figure

85 00:13:03.420 --> 00:13:03.990
Michael Bender: Right, even

86 00:13:05.220 --> 00:13:09.120
Michael Bender: Sort of afterwards edited the eyes to be to be facing the right direction.

87 00:13:10.410 --> 00:13:12.960
Michael Bender: Okay, so this is a query for

88 00:13:15.540 --> 00:13:27.300
Michael Bender: For some whether be is in the set. So you query whether D is in the set. So now, D. Get attached to position 0123 and position 012 both of these

89 00:13:28.410 --> 00:13:41.610
Michael Bender: Are hash to ones. And so for the answer to the query is is the in the set. Well, the filter says yes, it's in the set, but this is a false positive, because he is not in the set. And in fact, those ones were flipped because of

90 00:13:42.780 --> 00:13:47.670
Michael Bender: Other elements. And so this is why the Bloom filter allows for false positives.

91 00:13:50.670 --> 00:14:09.660
Michael Bender: I should point out that Bloom filters don't support deletes and the issue on delete is it's hard to tell which ones get detrimental because suppose that we decide to delete some elements. See, so C gets hash to

92 00:14:11.370 --> 00:14:21.240
Michael Bender: This position here and this position here if we remove see do we document those back down to zero, or do they have to stay one because there's some other element that's hashing to them.

93 00:14:22.290 --> 00:14:35.940
Michael Bender: So that's why Bloom filters support insertions and queries, but they don't support the leads, because just from looking, you can't tell whether we should be deleting whether we should be documenting those ones or keeping them as ones.

94 00:14:37.410 --> 00:14:39.390
Michael Bender: Okay, so this is what a Bloom filter is

95 00:14:41.130 --> 00:14:55.080
Michael Bender: And so let's talk about the space usage, the space usage of a of a well-tuned Bloom filter is like what, like a little bit more than 1.44 times log of one over the epsilon bits per element.

96 00:14:56.490 --> 00:14:57.060
Michael Bender: And

97 00:14:58.230 --> 00:15:05.670

Michael Bender: You know you can plug in some values of false positive rate and the number of bits that you need. But basically, a common rule of thumb.

98 00:15:06.180 --> 00:15:15.960
Michael Bender: Is that if you're using a filter you kind of want about one bite of space for element that's how much space do you want to dedicate to you to your allocate to your filter.

99 00:15:18.030 --> 00:15:20.880
Michael Bender: Okay, so this is base usage.

100 00:15:22.710 --> 00:15:31.260
Michael Bender: Bloom filters are ubiquitous like they appear all over the place. At first, that when I was making the slide. I was wondering if I should put in citations.

101 00:15:31.920 --> 00:15:41.220
Michael Bender: But then I looked at, you know, the number of citations that there are for Bloom filters and it would. It's like, citing calculus. It's just, it's just one you don't

102 00:15:41.490 --> 00:15:53.040
Michael Bender: Need to put a citation for Calculus. It's just one of the tools that is in every system builders toolbox, along with a relatively small number of other data structures, you know, hash tables balance trees, the trees.

103 00:15:53.310 --> 00:16:01.650
Michael Bender: I would say Bloom filters right up there in terms of importance. It's just one of the most important data structures that you need to know when you're building a system.

104 00:16:03.870 --> 00:16:21.690
Michael Bender: Okay and I, which I guess I should also say, which is why, given how important it is and system building. I always find it a little bit surprising that it is under emphasized and algorithms classes, even though it's just really beautiful data structure and very powerful and very useful.

105 00:16:23.250 --> 00:16:23.610
Michael Bender: Okay.

106 00:16:25.140 --> 00:16:28.020
Michael Bender: So talk so far I've decided

107 00:16:28.200 --> 00:16:29.730
Charles Leiserson: Books and algorithms suck.

108 00:16:35.580 --> 00:16:36.420
Michael Bender: Let's just say

109 00:16:38.130 --> 00:16:46.140
Michael Bender: Um, there's a lot of stuff to do in algorithms textbooks. Like, I can't imagine ever wanting to write one

110 00:16:51.660 --> 00:16:52.680
David Reed: This is David read again.

111 00:16:52.740 --> 00:16:59.280
David Reed: I can say that as as an operating system guy for a large part of my career.

112 00:17:01.050 --> 00:17:06.030
David Reed: I don't know of any operating system that has a Bloom filter and it's Colonel, which is sad.

113 00:17:07.440 --> 00:17:23.340
Michael Bender: Really, I'm actually surprised about that because, you know, when I talk about, you know, a lot of systems, researchers that interact with. They say, oh, I know we can improve performance by putting a filter here doing this and and making the data cache more efficiently. Um, but

114 00:17:25.470 --> 00:17:29.040
Michael Bender: I'll talk about some other places where filters are used in storage systems.

115 00:17:31.320 --> 00:17:37.710
Michael Bender: But okay, so, um, so I've described filters the Bloom filter. And now I'm going to talk about how filters are used.

116 00:17:38.790 --> 00:17:39.450
Michael Bender: And

117 00:17:40.800 --> 00:17:52.530
Michael Bender: I guess the cartoon on the right. This is a filter with coffee runs right through me. It's not exactly opposite to the talk but I enjoyed it enough to paste it in

118 00:17:53.820 --> 00:17:55.680
Michael Bender: So let's talk about how filters are used.

119 00:17:58.020 --> 00:18:01.890
Michael Bender: So the most common use of a filter.

120 00:18:02.940 --> 00:18:16.380

Michael Bender: Is to filter out queries to some large remote dictionary. And so here I've got a picture of some large remote dictionary say it's on you know either across the network or on disk. And then here, we've got the filter that's in RAM.

121 00:18:17.610 --> 00:18:20.580
Michael Bender: And these are a bunch of queries to the dictionary.

122 00:18:21.780 --> 00:18:33.420
Michael Bender: And so you so you query. A the filter says no, it's an addiction or you query be it's not in the dictionary or Corey see it says the filter says yes it is in the dictionary. Turns out the see wasn't the dictionary. So that was, that was a good answer.

123 00:18:33.990 --> 00:18:44.400
Michael Bender: D. It's not an dictionary. Well, there's a false positive efforts, not in the dictionary. So the point is that accessing this large remote dictionary is very, very expensive.

124 00:18:45.150 --> 00:19:05.610
Michael Bender: And if you can store the filter and ram or and just much quickly get filter out all of these negative queries, then you get a big performance boost and this isn't the only use of a filter, but at least in my experience, it's the overwhelming. The most common use of a filter.

125 00:19:07.080 --> 00:19:14.250
Michael Bender: I'm going to say the same thing. Two more times than two different ways. So this is the first way which is I'll talk about the speed up from the filter.

126 00:19:15.030 --> 00:19:24.240
Michael Bender: So, so you've got a workload with P positive queries and and negative quarter. So p queries for elements that are in your set and and negative quarries for elements that are not in your set

127 00:19:25.200 --> 00:19:36.480
Michael Bender: So if you have a dictionary without some sort of filter like without a boom filter, then you need p plus in remote accesses to the dictionary.

128 00:19:36.960 --> 00:19:46.710
Michael Bender: Whereas if you have a filter, then you have what you still need to do all your positive queries, but you only need to do an epsilon fraction of your

129 00:19:47.520 --> 00:19:59.220
Michael Bender: Negative queries and the rest of them just get filtered out. So here I've said the speed up in a different way, which is, it seems not to do anything from positive queries, but it helps negative queries a lot

130 00:20:00.420 --> 00:20:01.320

Michael Bender: But actually,

131 00:20:05.730 --> 00:20:21.030
Michael Bender: Filters are even better than that in a lot of ways. So let me talk about a how filters help queries in a kind of data structure called a log structured merge tree. I'm not going to really say what a log structured merge tree is because for the purpose of this talk. It doesn't matter.

132 00:20:22.050 --> 00:20:28.800
Michael Bender: But so basically a log structured merge streets supports fast, very, very fast insertions.

133 00:20:29.460 --> 00:20:34.980
Michael Bender: And it does this by partitioning one big data structure into a bunch of independent data structures.

134 00:20:35.370 --> 00:20:47.100
Michael Bender: And independent dictionaries and some having one big dictionary that you query, you've got a bunch of little or dictionaries and that's this picture that I've shown you here the dictionaries are exponentially increasing in size.

135 00:20:47.790 --> 00:20:53.520
Michael Bender: And so suppose that I want to query whether some element is is in my set and when a query be

136 00:20:54.540 --> 00:20:55.050
Michael Bender: And so

137 00:20:56.400 --> 00:21:02.460
Michael Bender: You're in luck search and merge tree. You're not just doing one query, but you do need to query for each of these little pieces.

138 00:21:04.920 --> 00:21:14.730
Michael Bender: And so well in this picture you query be the filter says no, you can't be the filter says no, you can't be the filter says yes. Maybe that was a false positive be

139 00:21:15.030 --> 00:21:18.990
Michael Bender: Aquarius know. So the point is that in the log structured merge tree.

140 00:21:19.350 --> 00:21:26.970
Michael Bender: Even if you're querying for an element that is somewhere in this log structured merge tree, the filter still saves on performance.

141 00:21:27.210 --> 00:21:37.470

Michael Bender: Because you've got these smaller dictionaries, where even if you've got one false positive in the big dictionary. You still got a lot of false negatives in these smaller pieces. So filters really

142 00:21:37.950 --> 00:21:42.840
Michael Bender: Are are very powerful and help a lot. And this is just one example with the log structured emerge tree.

143 00:21:44.760 --> 00:21:45.390
Michael Bender: Alright.

144 00:21:47.040 --> 00:21:56.610
Michael Bender: So here's the talk so far. Let's see. I've talked about filters. I've talked about the Bloom filter it said how they are used. And now it's time to motivate

145 00:21:56.970 --> 00:22:12.300
Michael Bender: The title or the second very title of my talk, which is that it's time to change your filter. And here's a cartoon. So there's Darth Vader, he was preparing for a big date and your he's got his air filters and areas is changing and filter.

146 00:22:13.980 --> 00:22:21.180
Michael Bender: So, now is the time to put your put your finger on the mute button and laugh and then unmute again. So I really liked

147 00:22:21.360 --> 00:22:22.770
Michael Bender: It again it's not relevant.

148 00:22:22.830 --> 00:22:23.790
Michael Bender: But I like that to

149 00:22:23.820 --> 00:22:26.400
Michael Bender: That slide too much not to include it.

150 00:22:28.170 --> 00:22:31.020
Michael Bender: Okay. Um, so

151 00:22:33.750 --> 00:22:39.840
Michael Bender: Applications must work around the limited capabilities of a filter.

152 00:22:43.320 --> 00:22:46.410
Michael Bender: So let's talk about some of the limitations. I've already said one of

153 00:22:46.410 --> 00:22:55.860

Michael Bender: Them, which is that you can't delete. And so one of the workarounds is if you need to do a lot of deleting is you need to rebuild your filter from scratch.

154 00:22:56.400 --> 00:23:00.630
Michael Bender: There's also no resizing. And so what that means is you somehow have to guess.

155 00:23:01.530 --> 00:23:15.210
Michael Bender: How many elements, your filters ultimately going to have and you have to rebuild. If you're wrong. And that's why, even a couple slides ago, I talked about this filter that is all zeros. It's not a very you know that that was not a very compact filter.

156 00:23:16.320 --> 00:23:23.730
Michael Bender: So filters also don't support or Bloom filters don't support merging or enumeration developments. So, suppose if that two separate

157 00:23:24.420 --> 00:23:37.320
Michael Bender: Filters, and I want to merge them together into one filter. So a boom filter does not do that efficiently. But that turns out to be a very common and useful operation. And certainly that's something that I've required in my work.

158 00:23:39.540 --> 00:23:43.410
Michael Bender: And finally there no values associated with keys that lots of times you

159 00:23:44.640 --> 00:23:54.690
Michael Bender: You don't just want to have a filter, but you might want to store a small amount of information associated with every element, make some color or something and a filter does a Bloom filter doesn't do that.

160 00:23:55.290 --> 00:24:01.710
Michael Bender: And so you have to combine with some other data structure to get this capability that you want. So the bottom line is that

161 00:24:02.310 --> 00:24:15.060
Michael Bender: Boom filter limitations increase system complexity, they waste space and they slow down application performance and all of these are things that in my own research I've needed and haven't been able to get with a Bloom filter.

162 00:24:17.070 --> 00:24:19.500
Michael Bender: They also have suboptimal ass and tonics

163 00:24:20.850 --> 00:24:27.900
Michael Bender: In the sense that a Bloom filter has like this extra sort of 1.44

164 00:24:29.190 --> 00:24:34.470
Michael Bender: Times, and I'm like, log of one over the false positive rate, but you can do better.

165 00:24:36.600 --> 00:24:44.940
Michael Bender: Bloom filters CPU cost is also suboptimal like it's a function of the false positive rate, but you can get it down to constant

166 00:24:46.320 --> 00:24:53.670
Michael Bender: And finally, Bloom filters don't have good data locality, because these hash functions are hashing all over the

167 00:24:53.970 --> 00:25:02.610
Michael Bender: Place. And if you've got eight, you know, H different hash functions that a different place in your data structure, you don't have good data locality and you and you can get better data locality.

168 00:25:03.120 --> 00:25:14.820
Michael Bender: So again, just repeat Bloom filter limitations increase the system complexity, they waste space they sold on application performance and they're not necessary. And that's what I'll be talking about in the rest of the talk.

169 00:25:16.830 --> 00:25:35.010
Michael Bender: Okay, so there's tons of research on extending or improving a replacing a Bloom filters and like this is this is such a partial list. I just sort of pasted some stuff down, but this is very far from complete and

170 00:25:36.210 --> 00:25:45.360
Michael Bender: The I guess the point is when there's too much stuff growing on top of your filter, like, well, it's time to change your filter. I

171 00:25:47.820 --> 00:25:51.630
Michael Bender: I decided not to put in a picture illustrating this point.

172 00:25:55.620 --> 00:25:58.980
Michael Bender: Which I'm sure. Thank you. Sure that people will

173 00:25:59.190 --> 00:26:02.160
Michael Bender: Appreciate the lack of illustration, there

174 00:26:03.420 --> 00:26:10.080
Michael Bender: Okay, so talk so far, so it's time to change your filter is the talk title. And again, this is another picture.

175 00:26:12.630 --> 00:26:18.570
Michael Bender: Sort of, I see that you change your filter about as often as you change your mind wasn't as good as the Darth

176 00:26:19.740 --> 00:26:23.730
Michael Bender: Vader Darth Vader filter. But it was good enough that I wanted to include it.

177 00:26:24.870 --> 00:26:26.370
Michael Bender: Okay, so

178 00:26:30.270 --> 00:26:37.320
Michael Bender: It took me a while, actually, to figure out what the message of this talk was going to be, and

179 00:26:38.550 --> 00:26:52.050
Michael Bender: In fact, one of the messages that I could have chosen and opted against was the questions, sort of based on Charles and David, which is how do you make a filter. That's actually adaptive and and I'd love to talk to to both of you about that.

180 00:26:53.940 --> 00:26:54.930
Michael Bender: Afterwards more

181 00:26:56.340 --> 00:27:09.180
Michael Bender: So the bottom line is I couldn't find the right talk title. And then I received this email which you know from some company filters fast, which says it's time to change your filter.

182 00:27:10.200 --> 00:27:19.380
Michael Bender: And of course it was email. So I snapped into action and ignored it. But it turns out I was getting a whole bunch of emails.

183 00:27:19.410 --> 00:27:27.960
Michael Bender: Like the whole universe was basically telling me the title of my talk. Have you forgotten to change your filter. Have you forgotten to change your filter. Have you forgotten to change your filter.

184 00:27:28.590 --> 00:27:33.810
Michael Bender: You know, and then you know you want to perform better. You want filters fast and you know for a

185 00:27:35.520 --> 00:27:51.120
Michael Bender: Remote talk is a home filter club. So there's a whole universe was trying to tell me what this talk was about. And at some point, I actually paid attention to it. And that's why, that's why I made decided to make this the message of the talk.

186 00:27:52.260 --> 00:27:52.740
Michael Bender: So,

187 00:27:53.790 --> 00:27:56.880
Michael Bender: So if this is a message of the talk, then

188 00:27:59.310 --> 00:28:10.110
Michael Bender: I also want to talk to have some heroes and so again talk. It's time to change your filter and I kind of want this talk to be a tutorial like introduction to filters.

189 00:28:10.980 --> 00:28:18.870
Michael Bender: As well as general techniques for solving FILTER PROBLEMS. And so, you know, yes. I'm going to talk about my work. But I also want this to be

190 00:28:21.690 --> 00:28:22.140
Michael Bender: Sort of

191 00:28:23.400 --> 00:28:33.540
Michael Bender: explaining something about a field as well. And so these are the three heroes of the talk. So this is the first year, which, and I'll explain all of them. So the sort of three heroes. One of them is fingerprinting

192 00:28:34.170 --> 00:28:40.860
Michael Bender: And one of them is quotient thing. And one of them is collision resolution, and I'll explain what these are, in the next couple slides.

193 00:28:41.520 --> 00:28:48.990
Michael Bender: But just to say something about these icons, which, again, I'm a little bit embarrassed about how much time I spent on them and I'm not gonna let you know.

194 00:28:49.650 --> 00:29:00.930
Michael Bender: But I was sort of proud defined the fingerprinting icon, where I could make the world the nose and so that I worked hard on I guess people that people recognize this face.

195 00:29:08.880 --> 00:29:18.240
Michael Bender: Yeah, so this is Knuth, and because he's he's one of the first people who wrote about and propose this technique that I'll talk about and the people recognize this third face.

196 00:29:26.040 --> 00:29:26.700
David Reed: Muller.

197 00:29:28.110 --> 00:29:29.070
Michael Bender: Yeah, so this is

198 00:29:29.100 --> 00:29:29.730
Muller.

199 00:29:31.380 --> 00:29:37.830
Michael Bender: It turns out this is a mistake in the icon, because this is collision resolution and not collusion resolution.

200 00:29:40.290 --> 00:29:41.730
Michael Bender: So I

201 00:29:43.380 --> 00:29:46.290
Michael Bender: I change the icon. So this is the new icon for collision.

202 00:29:46.290 --> 00:29:47.400
Michael Bender: Resolution. And again,

203 00:29:49.410 --> 00:30:03.420
Michael Bender: I think with these three techniques. It explains a lot of how you go about understanding lots of different filters in in the literature. So to begin, let me talk about the first one and this is fingerprinting

204 00:30:05.340 --> 00:30:08.370
Michael Bender: So in fingerprinting, you have

205 00:30:10.290 --> 00:30:15.330
Michael Bender: So describe how to make a filter from some set. So here's we've got some set S which is the dictionary.

206 00:30:16.620 --> 00:30:23.010
Michael Bender: And in the filter. What you do is you take a fingerprint of every element in the set.

207 00:30:24.150 --> 00:30:35.430
Michael Bender: And because these are Fingerprints of the elements, rather than the original elements. Turns out we can store them more compact Lee than you can with a Bloom filter, and I'll explain why in a couple of slides. But the point here is just that you can

208 00:30:36.870 --> 00:30:54.240
Michael Bender: And so now, since you're storing instead of you're storing the elements in the dictionary and the fingerprints of the elements in the filter. It turns out that the filter is just another dictionary, but a smaller one. Right. So it's just, it's also a dictionary, just a more compact one

209 00:30:55.410 --> 00:30:56.610
Michael Bender: Okay, so now

210 00:30:58.560 --> 00:31:08.730
Michael Bender: If we want to query in the dictionary. So is x, you know, a member of the set. Well, we look in the filter. And if we're storing the fingerprint.

211 00:31:09.750 --> 00:31:15.420
Michael Bender: In the filter than we'd say yes. And if the fingerprint is not in the filter than we say no.

212 00:31:16.590 --> 00:31:27.600
Michael Bender: And this is the only source of false positives. And again, I don't know why I keep repeating this but this can be stored more capacity than a Bloomfield, dark, as I'll explain in a second.

213 00:31:28.740 --> 00:31:34.230
Michael Bender: So again, the way you query is just if the fingerprints there, you say yes, but the fingerprints not there, you say no.

214 00:31:35.490 --> 00:31:35.970
Michael Bender: So,

215 00:31:38.550 --> 00:31:47.280
Michael Bender: Fingerprint collisions are the only source of false positives. So this is just to analyze the false. What I'm going to do next is analyze the

216 00:31:48.900 --> 00:32:03.570
Michael Bender: The error rate and what the source of the false positives is from these fingerprints. But the point is that fingerprint collisions are the only source of false positives. So what I mean by that is if some fingerprint is like if

217 00:32:04.620 --> 00:32:18.840
Michael Bender: The fingerprint of why is stored in our filter. We say yes, but why is it false positive. If sort of why is not in the sad, but there's some other element x that is in the set.

218 00:32:20.010 --> 00:32:22.080
Michael Bender: That shares the same fingerprint with why

219 00:32:23.100 --> 00:32:30.090
Michael Bender: So then we have got a false positive, because why and some element in the second half collided. And this is the only source of false positives.

220 00:32:31.800 --> 00:32:41.340

Michael Bender: Okay, so now what I wanted to do, given that we have this, you know, to explain where the source of false positives. It is that if these hash collisions.

221 00:32:41.970 --> 00:33:02.790
Michael Bender: Um, let's start to analyze how big we need to make the fingerprints. So here we've got some element x and we hash, you know, to get some fingerprint ah of x. And it's going to, we're going to require log n divided by the false positive rate bits.

222 00:33:03.900 --> 00:33:06.450
Michael Bender: And just to do the analysis, the probability that

223 00:33:08.190 --> 00:33:14.850
Michael Bender: Two elements collide, that, you know, some x and y collide is is one half.

224 00:33:16.530 --> 00:33:24.810
Michael Bender: You know, or one over to to the number of bits in the fingerprint, which is epsilon divided by n. So that's the probability of any one collision.

225 00:33:26.010 --> 00:33:40.410
Michael Bender: And so the probability that some element that is not in the set is a false positive, just by the union bound is n, the number of elements in the set and times excellent overhead, which is excellent. So that's where this false positive rate comes from

226 00:33:40.890 --> 00:33:43.620
Michael Bender: And so we see that to get a false positive rate of epsilon

227 00:33:44.790 --> 00:33:48.090
Michael Bender: We want to have log of end divided by epsilon bits.

228 00:33:49.740 --> 00:34:00.150
Michael Bender: Okay. And so now what I need to do is explain that these can be stored compact Lee so naively, we would need log of of

229 00:34:01.050 --> 00:34:19.140
Michael Bender: End divided by epsilon bits per element. But in fact, you can actually store using only log of one divided by epsilon plus like some constant bits per element. So we don't want this end there. We want to turn this end into a one. Bye bye.

230 00:34:21.180 --> 00:34:32.820
Michael Bender: With efficient encoding and that's where the second hero comes in. So we, the first year was fingerprinting with just explained the second year was coaching thing which I'm going to explain now.

231 00:34:34.440 --> 00:34:35.250

Michael Bender: And by the way,

232 00:34:36.660 --> 00:34:45.030
Michael Bender: As much as I enjoy listening to myself talk. I still do want to have questions because I enjoy listening to other people talk more. So

233 00:34:47.010 --> 00:34:47.280
Michael Bender: Yeah.

234 00:34:47.730 --> 00:34:49.890
Julian Shun: I had a question on the previous slide.

235 00:34:51.030 --> 00:34:53.310
Julian Shun: Okay, do you assume here at is

236 00:34:53.340 --> 00:34:58.170
Julian Shun: Fixed or is it is a changing as you do operations on your

237 00:34:58.530 --> 00:34:59.640
Michael Bender: That's a great question.

238 00:35:01.710 --> 00:35:09.150
Michael Bender: I think that we should assume that end is sort of a fixed upper bound on the number of elements.

239 00:35:10.230 --> 00:35:10.950
Julian Shun: But

240 00:35:11.010 --> 00:35:12.690
Michael Bender: I think there's lots of cool

241 00:35:14.310 --> 00:35:19.740
Michael Bender: Work to be done, which I'm sort of excited about about about what happens when n

242 00:35:21.210 --> 00:35:27.360
Michael Bender: Is changing. And there are some recent papers on this, but I still think that it's an open area.

243 00:35:29.520 --> 00:35:29.760
Michael Bender: So,

244 00:35:29.910 --> 00:35:33.720
Michael Bender: But, but, but now it makes sense to think about end is some fixed upper bound

245 00:35:35.220 --> 00:35:38.910
Michael Bender: But, but even, even when you don't know exactly what end is

246 00:35:40.260 --> 00:35:54.660
Michael Bender: Asked me in a couple of slides, why even if you only have some fixed upper bound sort of why this is going still going to be a better scheme, then, then a Bloom filter, even when you don't really have a good sense about what n is. And

247 00:35:54.690 --> 00:36:03.630
Richard Barnes: One more clarifying question on this side. So the total number of bits Union for the hash tables, then and log and over a year and log one over here.

248 00:36:05.100 --> 00:36:05.490
Richard Barnes: Is that

249 00:36:05.580 --> 00:36:06.750
Michael Bender: So the total number

250 00:36:06.900 --> 00:36:25.260
Michael Bender: So if you stored it explicitly without encoding would be n times this quantity here and times log and divided by epsilon bits. That's what you would need with the naive encoding. But now we're going to do better, where we're just going to go to say it's an times this quantity here.

251 00:36:26.970 --> 00:36:30.750
Michael Bender: Right, so we so we want to make this with n times blog one over apps on

252 00:36:32.430 --> 00:36:33.240
Richard Barnes: Cool, or at least

253 00:36:33.270 --> 00:36:43.080
Michael Bender: Order that. Yeah. So yeah. So again, the more questions, the better. Because then I actually feel like I'm interacting with human beings, which I always enjoy

254 00:36:44.730 --> 00:36:46.500
Sarah Scheffler: Okay, one more, since we're here. Sorry.

255 00:36:46.890 --> 00:36:47.640
Sarah Scheffler: I'm so yeah

256 00:36:47.670 --> 00:36:48.210
Michael Bender: Yes, please.

257 00:36:48.420 --> 00:37:02.610
Sarah Scheffler: mbu um so here we're talking about the false the false positive are coming from hash collisions, because we're just hashing every element that said, This is not like the false positives from a Bloom filter. Normally, right, which are about

258 00:37:03.000 --> 00:37:04.860
Michael Bender: That's exactly right.

259 00:37:06.720 --> 00:37:18.090
Michael Bender: So yeah, it's so the only source of false positives is from fingerprint collisions and now we're just going to store the fingerprint. These fingerprints without adding any additional errors. That's right.

260 00:37:22.170 --> 00:37:26.310
Michael Bender: Yeah so much again. I really appreciate the emphasis of that question, sir.

261 00:37:27.120 --> 00:37:42.780
Michael Bender: Um, okay, so now what I want to do is talk about quotient thing. And I'm going to describe how you store these fingerprints in an efficient way so that the space is only order like log of one divided by the ever made bits per element.

262 00:37:44.070 --> 00:38:00.270
Michael Bender: And so again, just the picture from before. Here we've got the element x here we've got our fingerprint there, we're going to divide the fingerprint into the top. The most significant login bits. And then the least significant log of one over epsilon bits.

263 00:38:01.560 --> 00:38:03.960
Michael Bender: So this is our fingerprint. But we're cutting it in half.

264 00:38:05.310 --> 00:38:13.920
Michael Bender: And now we're going to store these bit. So this is going to be called the question and this is going to be called the remainder. So the quotient.

265 00:38:14.790 --> 00:38:28.380
Michael Bender: Is we're going to store implicit the end the remainder. We're going to store explicitly. So the quotient like these q of x, that's going to be stored based on the location in the hash table where you store the remainder

266 00:38:29.460 --> 00:38:38.970
Michael Bender: So in our hash table we store this our effects and based on the address where this is stored. We know the question. And so we know the fingerprint.

267 00:38:40.890 --> 00:38:53.430

Michael Bender: And so, and, you know, this is an old idea that's due to Knuth, and you see it all over the place and certainly an architecture and it's just a it's an old idea. And that turns out to be very powerful.

268 00:38:54.750 --> 00:39:01.920
Michael Bender: But there's the question, which is how to deal with collisions and a hash table because maybe we're storing

269 00:39:02.940 --> 00:39:13.470
Michael Bender: Sort of x. And so we're storing the quotient implicitly, and we're storing the remainder in this place and a hash table. Now we're storing some y

270 00:39:14.040 --> 00:39:22.770
Michael Bender: Which is actually going to be hashed to the same place and a hash table, but we want to store a different remainder. So how do we deal with collisions and a hash table.

271 00:39:26.370 --> 00:39:41.220
Michael Bender: And of course, if this were a an actual class, then I would stop talking and make people feel sufficiently uncomfortable that they need to answer and I'm still trying to figure out whether I should do that. Now, or just give the answer.

272 00:39:44.520 --> 00:39:54.420
Michael Bender: Um, but the main thing is, well, isn't this a solved problem, like, you know, we've all studied hashing. Yeah.

273 00:39:54.750 --> 00:39:56.310
Davin Choo: So what if we just take the song.

274 00:39:57.930 --> 00:40:01.770
Davin Choo: Like in the Bloom filter case you just always set one

275 00:40:04.710 --> 00:40:08.070
Michael Bender: It's an interesting idea somehow to see how to combine but then

276 00:40:10.200 --> 00:40:14.760
Michael Bender: But then you're kind of as soon as you're you're adding things together or

277 00:40:16.440 --> 00:40:18.150
Michael Bender: You're kind of losing information.

278 00:40:20.130 --> 00:40:23.670
Michael Bender: But the thing is, this is just a hash table.

279 00:40:25.620 --> 00:40:30.930

Michael Bender: And so there are lots of ways of dealing with collisions and a hash table. And so the question is, why can't we just use them.

280 00:40:32.970 --> 00:40:42.750
Michael Bender: And so one of the most efficient ways. So changing. It's got problems because pointers are very expensive and waste a lot of space. But what's wrong with just plain old

281 00:40:44.340 --> 00:40:52.380
Michael Bender: You know, standard sort of out of the box linear probing and let me show you what is wrong with it in the next picture in. Next slide.

282 00:40:54.840 --> 00:41:09.930
Michael Bender: So here's a picture of describe what the problem when you've got these hash collisions. When you're doing quotient thing and I'll just give the example of, say, we've got a six bit hash. So this three bits for an address and three bits for data.

283 00:41:11.280 --> 00:41:22.050
Michael Bender: And so like this. So this is just, you know, a table with positions, you know, zero up to seven, and these are empty slots. But let's look at this

284 00:41:23.430 --> 00:41:27.930
Michael Bender: Element right here. So the question, like the remainder is 011

285 00:41:29.580 --> 00:41:40.410
Michael Bender: And it started positions 001 so maybe this element represents a fingerprint that 001011

286 00:41:41.460 --> 00:41:53.790
Michael Bender: But maybe this element here isn't actually in the right place. Maybe this element was actually bumped over because of linear probing and maybe this element is really 000011

287 00:41:55.230 --> 00:42:00.210
Michael Bender: And we can't tell because this hash is stored implicitly based on location.

288 00:42:01.410 --> 00:42:15.030
Michael Bender: And what you're doing with linear probing is you're moving the location. Again, same here. Does this element here represent 100111 or maybe it represents 011111 because they got moved over and we can't tell.

289 00:42:17.610 --> 00:42:25.170
Michael Bender: You the way I've described things now because because when you do linear probing we're losing information.

290 00:42:26.520 --> 00:42:35.190
Michael Bender: And so that's where the third hero of the talk is going to come in. So I've talked about fingerprinting, and I've talked about motion thing. And so now I'm going to talk about collision resolution.

291 00:42:37.500 --> 00:42:38.730
Michael Bender: And so again,

292 00:42:39.750 --> 00:42:53.850
Michael Bender: It took me a while to get to the structure of the talk, but basically the heroes of the talk, are fingerprinting potion thing and collision resolution and you can use these to solve lots and lots and lots of

293 00:42:55.230 --> 00:42:59.610
Michael Bender: You know, different sort of filter problems. Um, but

294 00:43:01.740 --> 00:43:14.760
Michael Bender: You know, but in order to get there. I wanted to explain what I meant by filter how they're used as well as sort of motivating why Bloom filters are pretty awesome, but they're still not the end of the story. Okay, so now I need to talk about collision resolution.

295 00:43:15.900 --> 00:43:26.580
Michael Bender: And there are lots of different ways of doing collision resolution and I'm going to talk about sort of one way, but

296 00:43:27.690 --> 00:43:33.570
Michael Bender: Then I'll talk about a slightly different way. But this is one of the ways that I like best.

297 00:43:34.950 --> 00:43:48.720
Michael Bender: And what we're going to do is used to metadata bits per slot in the hash table. And that's going to let us recover the original location. And so this is one

298 00:43:49.350 --> 00:43:54.210
Michael Bender: Array of metadata bits and and the one is going to mean something is hash to this slot.

299 00:43:55.530 --> 00:43:58.440
Michael Bender: And zero is going to mean that this is an that nothing got

300 00:43:58.440 --> 00:43:59.070
Guy Even: hashed here.

301 00:44:00.450 --> 00:44:04.140

Michael Bender: And then there's going to be another array here that says I'm hashed

302 00:44:05.550 --> 00:44:13.350
Michael Bender: To the same place as the element before me and and when we do the decoding. I'm going to describe this

303 00:44:14.430 --> 00:44:24.300
Michael Bender: In where the decoding is very complicated because the decoding is actually going to start from the beginning of the array and just run through the entire array, but but

304 00:44:24.630 --> 00:44:36.030
Michael Bender: But it can be broken up into smaller localized chunks. But for now, the way I'm describing the decoding, you have to start from the beginning of the array. So let's see. So

305 00:44:36.750 --> 00:44:37.440
David Reed: Actually can I

306 00:44:39.480 --> 00:44:39.930
Michael Bender: Yeah yeah

307 00:44:41.130 --> 00:44:57.150
David Reed: So this may be a dumb question, but it looks to me like you're doing something that's got certain amount of complexity to it and you know I had thought there would be a base case that is like

308 00:44:58.830 --> 00:45:04.230
David Reed: The simplest thing I could imagine, which is replacing the

309 00:45:05.940 --> 00:45:20.850
David Reed: The slot where where there's a collision replacing the slot with a, you know, sort of magic value that says some number of guys have collided here. So you can't you, so you would return a false positive.

310 00:45:23.160 --> 00:45:39.600
David Reed: If anything has the the quotient, you know, no matter what remainder. It has after that. So that seems like the ultimately most simple thing is that makes sense. Yeah.

311 00:45:40.380 --> 00:45:41.970
Michael Bender: It makes sense and

312 00:45:42.000 --> 00:45:42.600
David Reed: In fact,

313 00:45:42.630 --> 00:45:45.210

Michael Bender: I think Kuszmaul is on the talk and he when I

314 00:45:45.420 --> 00:45:49.350
Michael Bender: He and I were recently talking about cool ways that you could try to simplify

315 00:45:50.910 --> 00:45:57.690
Michael Bender: You know some of this may be introducing other sources of false positives. But one of the

316 00:46:00.570 --> 00:46:09.060
Michael Bender: But, but for now, what I really want to do is actually make it so that I really am storing all of the fingerprints. Exactly. And

317 00:46:10.500 --> 00:46:17.550
Michael Bender: This is a fairly complicated data structure to code. And so if you're using this. It's really nice. If you can find a library.

318 00:46:20.430 --> 00:46:21.690
Michael Bender: Is you really enjoy

319 00:46:23.820 --> 00:46:35.730
David Reed: My intuition got stronger when I thought of that because I said oh well you can throw in false positives, you just don't want to throw them in. So they systematically make everything worse to fail fast right

320 00:46:35.790 --> 00:46:36.390
Michael Bender: That's right.

321 00:46:36.600 --> 00:46:37.530
David Reed: And I think there's

322 00:46:37.560 --> 00:46:50.070
Michael Bender: I think that is an exciting thing to think about, about how do you sort of balance, sort of the ease of coding and throwing in some extra false positives. But now I don't want to throw in any extra false positives at all.

323 00:46:52.440 --> 00:47:01.050
Michael Bender: And I just so this slide here where I described the decoding. I'm going to try to say it fast because it's not a slide that

324 00:47:01.920 --> 00:47:10.500
Michael Bender: One absolutely needs to understand to get the rest of the talk, but I actually enjoy it. And so I get pleasure out of, out of explaining this.

325 00:47:11.130 --> 00:47:18.660
Michael Bender: So the variance is going to be the no element is stored before its target position. This is, this is basically going to be

326 00:47:19.200 --> 00:47:28.920
Michael Bender: Built on the kind of hashing called Robin Hood hashing. So if we look at this first slide here, this is zero in slot 001 meeting, nothing has hatched here.

327 00:47:29.850 --> 00:47:42.780
Michael Bender: And so we already know that this first slide is an empty slot. We don't even need an extra bit to let us know that we just know it immediately from that bit here. Then if something is hash to this second slot 01

328 00:47:43.830 --> 00:47:49.440
Michael Bender: And so this remainder is stored the bits are stored in this remainder. And we know what's in the correct slot.

329 00:47:50.040 --> 00:48:00.930
Michael Bender: So now we go to the next thing is we see there's a one in this position here which says I'm hash to the same place as the element before me. So there's a one here, which means that

330 00:48:01.290 --> 00:48:18.900
Michael Bender: This 010 is Tash to the same place. So that means that we know that this second position encodes the fingerprint 001010 again. Same here. This one means that this remainder is hash to the same place as the

331 00:48:20.100 --> 00:48:30.240
Michael Bender: Remainder in the previous position, which was also stored, you know, in the same place here. So that's so here we know that this fingerprint is 001111

332 00:48:31.380 --> 00:48:45.060
Michael Bender: Now we look here, this remainder is mapped to the next array position. And so we look at the next re position is based on this one there. So we know that 010010 is the fingerprint.

333 00:48:45.660 --> 00:48:56.940
Michael Bender: That it belongs and and it really should go there. So now this is sort of what something I think it's cool, which is we look at this one, which does the remainder would map to the next position.

334 00:48:58.110 --> 00:49:10.380

Michael Bender: But recall that no element is stored before its target position and there's no other places that it can be so we know implicitly without even needing any extra bits that this is an empty slot in the array.

335 00:49:12.150 --> 00:49:15.720
Michael Bender: And then again here. There's something hash the disposition

336 00:49:16.950 --> 00:49:26.970
Michael Bender: It's in the correct slot. And so we're in good shape. This is mapped to the same position as the previous remainder. And so, so basically it's somehow you do this on coding where you unzip the array.

337 00:49:27.270 --> 00:49:35.640
Michael Bender: And you can figure out where where every element is stored just based on the zeros and ones. And it's sort of to metadata bits per erase slide.

338 00:49:37.140 --> 00:49:45.150
Michael Bender: Okay, so this is a potion filters and this is one way of doing collision resolution by basically taking the hash table.

339 00:49:45.570 --> 00:49:54.390
Michael Bender: And generalizing it so that it works. Even though we're not explicitly storing the entire fingerprint, but only implicitly storing

340 00:49:54.870 --> 00:50:10.140
Michael Bender: Part of the fingerprint. But we want to do this without losing it losing any information. Okay, so let's talk about closing filter capabilities and just compare this with Bloom filters before I go on to describe other alternatives, so

341 00:50:12.210 --> 00:50:16.230
Michael Bender: You can count. You can delete. There are lots of different ways of counting

342 00:50:18.120 --> 00:50:29.760
Michael Bender: You know, decide depending whether you want to count in unary or binary. I'm happy to talk about that more. After the doc resizing is now much easier because suppose that you

343 00:50:31.740 --> 00:50:42.930
Michael Bender: You have an upper bound of elements, but in fact at the moment you have very few elements. So then you, you just do standard resizing of your hash table where

344 00:50:44.310 --> 00:50:53.070
Michael Bender: You use a much smaller hash table and store more bits of the remainder explicitly and so you're so you're always very space efficient.

345 00:50:53.790 --> 00:51:01.410
Michael Bender: Elements enumeration and filter merging is easy. It's just you're sort of zipping along two different arrays and merging them into a large array.

346 00:51:02.220 --> 00:51:14.340
Michael Bender: And if you want to store some value associated with the elements, just put a value on next to it and just add that value in the array position as well. It's the values are allowed because

347 00:51:16.170 --> 00:51:21.000
Michael Bender: The filter is just a dictionary that is storing fingerprints.

348 00:51:22.440 --> 00:51:26.760
Michael Bender: And so you can add values to the, you know, to the dictionary, the way you would with any other dictionary.

349 00:51:28.110 --> 00:51:32.700
Michael Bender: So this is the filter capabilities in terms of performance. It's

350 00:51:34.140 --> 00:51:50.400
Michael Bender: One plus something tiny times n plus you know times log of one over epsilon. Now we see this log of one over epsilon is the remainder. And then there's also these two additional bits that that we need in order to

351 00:51:51.720 --> 00:51:59.790
Michael Bender: To over, you know, so that we can recover the original position where a fingerprint was hashed the CPU cost is sort of constant expected

352 00:52:00.600 --> 00:52:11.790
Michael Bender: And data locality is much better because it's one probe and one scan. So you've got some practical and theoretical performance advantages from a cushion filter.

353 00:52:13.500 --> 00:52:14.610
Michael Bender: So there's

354 00:52:15.750 --> 00:52:23.490
Michael Bender: Two things I want to do quickly. So I just wanted to remind the general approach, which is sort of fingerprinting

355 00:52:24.690 --> 00:52:30.330
Michael Bender: Quotient thing where we're storing part of the fingerprint implicitly and then collision resolution.

356 00:52:31.110 --> 00:52:46.140
Michael Bender: And what I described in this question filter was based on linear probing and a particular kind of linear probing called Robin Hood hashing where we use these metadata bits to recover each original fingerprint. But there are other ways of doing this.

357 00:52:47.250 --> 00:52:50.400
Michael Bender: You know where you can use some sort of different hash table.

358 00:52:51.960 --> 00:53:03.390
Michael Bender: Maybe more sophisticated maybe less sophisticated or maybe just different, but it's the sort of same general approach of you've got a hash table, but then you need to deal with Collision resolution in some way.

359 00:53:04.530 --> 00:53:19.230
Michael Bender: And so I'll describe one more alternative which is called based cuckoo filters and it's based on using a different kind of hashing scheme and so cuckoo hashing is another beautiful

360 00:53:20.430 --> 00:53:33.240
Michael Bender: Hash table where, again, you've got to hash tables each other to hash functions and each hash function hashes to some particular bucket with some number of slots, you know, for is a reasonable number

361 00:53:34.170 --> 00:53:39.540
Michael Bender: And so here this is some element x, and it could get hashed to any one of the four slots in

362 00:53:40.590 --> 00:53:43.590
Michael Bender: In that position or any one of the four slots and another position.

363 00:53:44.700 --> 00:53:45.270
Michael Bender: And

364 00:53:47.040 --> 00:53:50.940
Michael Bender: You know, if their space to put the element in any one of those buckets. Great.

365 00:53:51.570 --> 00:53:58.830
Michael Bender: But if there's no space in any one of these eight slots, then what you need to do is kick out an element and move it to an alternative location.

366 00:53:59.340 --> 00:54:07.740
Michael Bender: Which. So here, maybe we kick out that position there and now there's room to put backs in because we've kicked out to the other position and and

367 00:54:08.280 --> 00:54:17.490
Michael Bender: That and moving this element may cause other kicks and so on and so part of the beauty of the cuckoo hashing analysis is that, amazingly, it actually works.

368 00:54:18.540 --> 00:54:24.150
Michael Bender: And this is my cartoon of a cuckoo that sort of fits with the style of the rest of the talk.

369 00:54:26.700 --> 00:54:41.610
Michael Bender: Okay, so that's cuckoo hashing. And so the question is, can you make a cuckoo filter based on cuckoo hashing and now we have some other problems and one of them is if you kick out some fingerprint.

370 00:54:42.750 --> 00:54:50.520
Michael Bender: How do you find sort of an alternative like maybe this is one fingerprints and maybe that's the different fingerprints.

371 00:54:51.720 --> 00:54:53.010
Michael Bender: And so how do you

372 00:54:54.240 --> 00:55:00.600
Michael Bender: Sort of moved to an alternative fingerprint. If you're not storing the original element X, you're just storing one of these fingerprints.

373 00:55:01.710 --> 00:55:03.870
Michael Bender: It just, it seems like an impossible problem.

374 00:55:04.890 --> 00:55:13.530
Michael Bender: And I'm just going to say this very briefly, and I'm happy to answer questions after but it turns out you give up on independent hash functions.

375 00:55:14.130 --> 00:55:25.290
Michael Bender: And the alternative location is in fact only going to depend on the remainder bits. So that means that if you have three remainder bits, then you don't have very

376 00:55:26.280 --> 00:55:40.560
Michael Bender: You don't have independent hash functions at all. In fact, you're only going to have eight choices of other bits to go to. So you're going to entirely give up on independent hash functions and the alternative location is going to depend only on a very small number of bits.

377 00:55:41.580 --> 00:55:56.070

Michael Bender: It turns out, when you do this, you also give up on a synthetic correctness. So this is not a data structure that works for arbitrarily large end but the beautiful thing is that it does work for practical and that isn't too large.

378 00:55:57.900 --> 00:55:59.880
Michael Bender: And like like

379 00:56:00.960 --> 00:56:11.610
Michael Bender: When I first read about this. It just seemed like such a not good idea because as some topically it can't work but amazingly, it does work for reasonable values event.

380 00:56:12.600 --> 00:56:27.300
Michael Bender: Now, again, the only final thing I'd like to say is cuckoo hashing seemingly doesn't have these metadata bits because we didn't add any metadata bits, but we're paying for that cost anyway. And that's because since there four slots per cell.

381 00:56:28.860 --> 00:56:38.580
Michael Bender: You need to store more of the fingerprint bits explicitly. And so it's sort of, you don't actually get a space advantage or or really much of a disadvantage.

382 00:56:40.230 --> 00:56:45.990
Michael Bender: I let's see how am I doing on time. Am I really running badly on time or

383
00:56:47.430 --> 00:56:54.210
Julian Shun: As so external clock, but I think you can continue and if people need to leave.

384 00:56:55.380 --> 00:56:55.710
Julian Shun: Thinking

385 00:56:55.740 --> 00:57:02.940
Michael Bender: I could do another three I could do another three minutes to get an A, but if you want me to shut it off. I was doing right to conclusions.

386 00:57:03.990 --> 00:57:04.620
Michael Bender: Another three

387 00:57:05.280 --> 00:57:05.910
Michael Bender: Okay, sounds

388 00:57:05.940 --> 00:57:06.090
Julian Shun: Good.

389 00:57:06.120 --> 00:57:11.880

Michael Bender: Then Julian, please do cut me off when when it feels like I'm being very impolite.

390 00:57:12.840 --> 00:57:13.770
Julian Shun: No Pro. Sure.

391 00:57:14.400 --> 00:57:17.460
Michael Bender: Yes, okay. So in terms of comparisons.

392 00:57:17.850 --> 00:57:18.570
Julian Shun: Sort of, I

393 00:57:18.630 --> 00:57:24.570
Michael Bender: Don't want to do a whole table with this, they're both pretty good space, one is a little better locality. Then the other

394 00:57:25.110 --> 00:57:35.490
Michael Bender: Cuckoo filters degraded higher load factors, they're little, they're much easier to to code. They've got pre synthetic guarantees, but they fail one and as large enough, but

395 00:57:36.120 --> 00:57:45.600
Michael Bender: The point isn't isn't really to compare these data structures because it's just sort of one or two data structures in a in a

396 00:57:46.830 --> 00:57:47.880
Michael Bender: In a large field.

397 00:57:49.320 --> 00:58:00.870
Michael Bender: This is this slide. The only reason I'm not skipping it is is this is going back to the question that Charles asked, originally, which is turns out there is an optimal filter.

398 00:58:01.350 --> 00:58:11.490
Michael Bender: With space that's optimal to lower terms and it has sort of an error rate that's always epsilon, no matter what the query is even when you repeat queries.

399 00:58:12.390 --> 00:58:27.030
Michael Bender: And even when you have sort of an adaptive adversary that repeats queries that knows mistakes. We're all operations are constant in search query delete with high probability. I should have written them with high probability there and that's above

400 00:58:28.140 --> 00:58:30.780
Michael Bender: Okay. So just to conclude with

401 00:58:32.010 --> 00:58:42.180

Michael Bender: One slide on some empirical performance, which is that like there are lots of different filter data structures that people implement

402 00:58:43.530 --> 00:58:52.020
Michael Bender: In general, the space among all of them is pretty similar, but they also have very different performance guarantees so

403 00:58:53.700 --> 00:59:00.300
Michael Bender: So if you look at. So again, the x axis is load factor which is how full the data structure is

404 00:59:01.110 --> 00:59:08.250
Michael Bender: The y axis is throughput, which is just so up is good and down is bad. This is to put for insertions so up is good and down is bad.

405 00:59:08.700 --> 00:59:18.420
Michael Bender: And this is helpful X axis is helpful, the data structure is and you can see. So the cuckoo filters a data structure that's really, really awesome when it's not too full.

406 00:59:19.290 --> 00:59:34.800
Michael Bender: But it performs much worse as the data structure gets more and more full and you can kind of see why that happens because as it gets more full there are more and more kicks, where one element gets kicked to a different element to a different location.

407 00:59:35.880 --> 00:59:37.080
Michael Bender: And that causes

408 00:59:38.190 --> 00:59:39.600
Michael Bender: Like poor data locality.

409 00:59:41.490 --> 00:59:42.390
Michael Bender: There.

410 00:59:43.680 --> 00:59:56.820
Michael Bender: So here, the main thing about this slide is to report on the data structure that does some things including minimum of two choice and and and sort of underlying vector operations, because

411 00:59:57.510 --> 01:00:09.120
Michael Bender: In order to get good performance. And again, this was a comment. I think that David, you asked earlier, which is like, isn't this kind of a CPU intensive data structure to implement

412 01:00:09.690 --> 01:00:20.820

Michael Bender: And the answer is yes, it is. And so, so this is showing how with, you know, some algorithms and some vector operations you can you can make this decoding.

413 01:00:21.810 --> 01:00:31.530
Michael Bender: Much faster. And I think what I want to do is I think I want to conclude, because I don't want to go over. Does anybody know what this slide is

414 01:00:32.550 --> 01:00:33.060
Michael Bender: It.

415 01:00:34.080 --> 01:00:40.260
Michael Bender: This is not the first time I've used this slide in a talk, but I'm curious if anybody recognizes what this slide is

416 01:00:44.550 --> 01:00:47.640
Michael Bender: I guess I'll just say this is my summary slide.

417 01:00:52.260 --> 01:00:52.830
Michael Bender: And

418 01:00:54.120 --> 01:00:58.080
Michael Bender: And in terms of morals of the talk. The first moral is well

419 01:00:58.140 --> 01:00:59.850
Julian Shun: It's time to change your filter, which was the

420 01:00:59.850 --> 01:01:00.270
Title

421 01:01:01.800 --> 01:01:15.690
Michael Bender: The heroes of the talk, we're fingerprinting and quotient and inclusion resolution and the thing that I think is so cool is that these are the unifying the heroes of the talk, both in theory, and in practice.

422 01:01:17.700 --> 01:01:34.920
Michael Bender: And, you know, regardless of whether the goal is to implement the fastest filter you can or come up with the best synthetics, you can. There's still the same heroes, and I guess to be philosophical applications should demand a richer set of applications from their film for

423 01:01:34.950 --> 01:01:36.180
Julian Shun: A set of operations from their

424 01:01:36.180 --> 01:01:44.100

Michael Bender: Filter because the filter is can you know can meet the demand so applications should use use

425 01:01:45.210 --> 01:01:47.490
Michael Bender: filters that that can do more things.

426 01:01:48.660 --> 01:01:54.960
Michael Bender: And I guess I'll conclude with one more thing, which is it turns out Stony Brook is hiring in theory.

427 01:01:56.010 --> 01:01:56.580
Michael Bender: So,

428 01:01:57.780 --> 01:02:07.590
Michael Bender: One position is earmarked for quantum computing and one is for a general theory position and like this is sort of a strange

429 01:02:08.220 --> 01:02:25.800
Michael Bender: Time when I imagine hiring may be getting tight but apparently we are in fact hiring and so please spread the word and also talk to me privately if you might be interested. So I'll stop there. I'll go back to the moral of the talk and answer any questions.

430 01:02:27.060 --> 01:02:28.980
Julian Shun: Great. Thanks so much, Michael.

431 01:02:30.180 --> 01:02:32.070
Julian Shun: Yeah, very, very, very top.

432 01:02:33.480 --> 01:02:39.360
Julian Shun: Does anyone have any questions if you have any questions please feel free to speak up.

433 01:02:43.680 --> 01:02:45.630
Michael Bender: Yeah, the more the better. As far as I'm concerned.

434 01:02:46.710 --> 01:02:51.300
Julian Shun: Yeah, I actually had one other question. Do you know if there's been

435 01:02:52.590 --> 01:03:01.260
Julian Shun: Any work on concurrent data structures for some of these like more complicated filters, like the Morton filter enclosure filter.

436 01:03:02.040 --> 01:03:03.600
Michael Bender: Um, there is some

437 01:03:05.220 --> 01:03:07.080
Michael Bender: In some cases, the

438 01:03:08.400 --> 01:03:17.880
Michael Bender: The concurrency comes nearly for free because, you know, if you have a data structure that's based on linear probing

439 01:03:19.350 --> 01:03:19.920
Michael Bender: Then

440 01:03:21.450 --> 01:03:28.110
Michael Bender: You know, you're basically doing very local changes to your, your data structure. And so you can kind of lock.

441 01:03:29.250 --> 01:03:34.950
Michael Bender: Small segments. In some cases, it's a little bit harder to achieve.

442 01:03:37.500 --> 01:03:47.130
Michael Bender: But so i guess i they so it's a yes, there has been some work on this, but I also think that

443 01:03:51.330 --> 01:03:54.420
Michael Bender: The more deeply you dive into a topic that seems closed.

444 01:03:55.620 --> 01:04:04.830
Michael Bender: Well, the more open like the more cool new problems do you discover so yes there's stuff that's done, but it doesn't mean it's not anything. It's not worth thinking about. Right.

445 01:04:05.430 --> 01:04:06.420
Julian Shun: Great, thank you.

446 01:04:10.290 --> 01:04:14.640
Tim Kaler: I had a question. Sorry, I can't, I can't seem to put my video on anymore.

447 01:04:15.660 --> 01:04:18.000
Tim Kaler: I think the host turned it off earlier but

448 01:04:19.410 --> 01:04:19.650
Michael Bender: Did I

449 01:04:20.670 --> 01:04:21.180
Tim Kaler: No,

450 01:04:21.330 --> 01:04:21.780
I don't know.

451 01:04:23.880 --> 01:04:26.760
Tim Kaler: So, um, you mentioned. Oh, there we go. Thank you.

452 01:04:28.530 --> 01:04:30.000
Tim Kaler: And sorry for having my video on before

453 01:04:31.980 --> 01:04:41.850
Tim Kaler: I'm curious kind of what the theoretical trade off is in terms of like space and computation between the some of these other filters and kind of

454 01:04:43.080 --> 01:04:49.650
Tim Kaler: generalizations of traditional Bloom filters. For example, there are scalable Bloom filters, where you kind of

455 01:04:50.730 --> 01:05:00.540
Tim Kaler: Have a have a chain of different filters of increasing size then you kind of tighten the false positive rate as you add additional

456 01:05:02.250 --> 01:05:08.490
Tim Kaler: Blocks to your Bloom filter and you kind of get it for you kind of get a curve of, you know,

457 01:05:09.930 --> 01:05:19.140
Tim Kaler: In exchange for that additional flexibility you have to pay with more hash functions and and a little bit more space.

458 01:05:20.760 --> 01:05:27.450
Tim Kaler: Do you have a sense of kind of a weird question filters kind of fall on that curve in terms of computation and space trade off.

459 01:05:29.430 --> 01:05:31.500
Michael Bender: So again,

460 01:05:32.550 --> 01:05:34.680
Michael Bender: The meat in all of these

461 01:05:35.850 --> 01:05:45.930
Michael Bender: Yeah, I can answer this sort of both in theory, and in practice were sort of, in theory, you really can get the best of

462 01:05:47.610 --> 01:05:59.910
Michael Bender: Sort of, if you use kind of a rare model where where you're doing word operations which you have to do anyway in in hash table. You can you can basically do everything in constant time with high probability

463 01:06:02.610 --> 01:06:13.620
Michael Bender: You know, using sort of the technique of fingerprinting and coaching thing and collision detection. And so in that sense I don't make theoretically

464 01:06:14.820 --> 01:06:16.950
Michael Bender: It's hard to imagine doing much better than that.

465
01:06:18.930 --> 01:06:20.250
Michael Bender: In terms of practice.

466 01:06:21.720 --> 01:06:22.320
Michael Bender: I'm

467 01:06:24.720 --> 01:06:27.120
Michael Bender: Sort of these coaching techniques are

468 01:06:28.440 --> 01:06:31.410
Michael Bender: There certainly computationally intensive and they're a pain to right

469 01:06:33.930 --> 01:06:45.030
Michael Bender: So there's so many cool techniques. I don't like dissing anything because I because there's so many cool approaches in general, I sort of feel like the

470 01:06:46.320 --> 01:06:52.680
Michael Bender: Like doing something based on filtering in to me it's, it, it feels like a more promising.

471 01:06:53.790 --> 01:06:58.770
Michael Bender: And more powerful approach, but that could also be a part as an answer.

472 01:06:59.370 --> 01:07:09.810
William Henry Kuszmaul: Well, to clarify mine, my interpretation of Tim's question is that he's the questions kind of about dynamic resizing, what if you don't know. Yeah, is our priority.

473 01:07:10.500 --> 01:07:20.190
William Henry Kuszmaul: And the point being that if you if you if you underestimated the size you can compensate after the fact, by building a second bigger filter that has epsilon

474 01:07:20.550 --> 01:07:33.660
William Henry Kuszmaul: About a factor of two smaller and then another if that again was wrong, then you again double the size and both yet another filter who's epsilon is another factor of two smaller in each time you kind of add one extra bit to your

475 01:07:34.980 --> 01:07:38.850
William Henry Kuszmaul: To your fingerprints to kind of compensate for the fact that, well, you're actually

476 01:07:38.910 --> 01:07:39.450
Michael Bender: That's right.

477 01:07:39.540 --> 01:07:43.320
William Henry Kuszmaul: Having more and more data structures and so I guess the question is

478 01:07:44.340 --> 01:07:53.820
William Henry Kuszmaul: Um, I mean, you can do the same trick with a question filter. It's kind of agnostic to which data structure, you're expanding, but is there a better way to dynamically resize a question filter.

479 01:07:54.780 --> 01:08:00.690
William Henry Kuszmaul: Like it just do question filters are kind of the days, for she described. Do they work well with dynamic resizing

480 01:08:02.040 --> 01:08:16.410
Michael Bender: Oh, very nice. So, Bill, you know, thank you. Um, I guess I would say this, which is one of the things that I did early on. If you look at, do I have this the slide where I say where filters are used.

481 01:08:17.970 --> 01:08:21.330
Michael Bender: Yeah. So this slide here where I talk about where filters are used.

482 01:08:23.430 --> 01:08:35.250
Michael Bender: When you think about how to do dynamic resizing, I feel that the important question that you need to model is when do you have access, just to the

483 01:08:36.000 --> 01:08:47.100
Michael Bender: The fingerprints or or Bloom filter or when do you have access to the encoded lossy information and when do you have access to be able to probe the dictionary.

484 01:08:47.640 --> 01:08:57.450
Michael Bender: And re get the original element so that you can decide, oh, I know I have the original element. So now I realized I need more fingerprints or I need to encode something

485 01:08:57.930 --> 01:09:10.470
Michael Bender: And and make the filter, a little better. And so one of the reasons why I did this sort of lengthy introduction where I actually say how the filter is used is because it sort of

486 01:09:10.830 --> 01:09:25.860
Michael Bender: It sort of makes it a little clearer when do I have access to an element in order to decide to like sort of to add more bits to a fingerprint and and sort of what I think the right answer is, whenever you're doing a false positive.

487 01:09:26.940 --> 01:09:37.320
Michael Bender: Which is or true positive, which causes you to access the dictionary, then you can also sort of access some of the original elements in the dictionary.

488 01:09:37.860 --> 01:09:50.310
Michael Bender: And then go back and re encode them with more bits into your filter. And so I actually think that that the way to think about the filter also is sort of in conjunction with some dictionary that you're backing up

489 01:09:51.210 --> 01:09:52.080
Michael Bender: It's better know whether

490 01:09:52.320 --> 01:09:53.100
William Henry Kuszmaul: You’re claiming that

491 01:09:53.160 --> 01:09:53.670
Michael Bender: Yeah, go ahead.

492 01:09:54.990 --> 01:10:04.590
William Henry Kuszmaul: Are you saying, so you're saying, okay, every time we every time that the data structure says something is present, then we're going to go check the dictionary. Anyway, the real yeah

493 01:10:05.610 --> 01:10:09.360
William Henry Kuszmaul: And we're using that as an opportunity to do a little bit of rebuilding on our filter.

494 01:10:10.410 --> 01:10:18.090
William Henry Kuszmaul: Are you saying that if you do that, right. You can also support dynamic resizing, kind of, you can allow for your filter to resize over time, or is it

495 01:10:19.650 --> 01:10:19.860
William Henry Kuszmaul: Is

496 01:10:20.070 --> 01:10:22.020

Michael Bender: Like to what degree. So that's one of that.

497 01:10:22.350 --> 01:10:24.810
Michael Bender: So that's one of the next papers that I would want to

498 01:10:24.810 --> 01:10:29.880
Michael Bender: Write it hasn't been written yet but that, but I do think that that is correct.

499 01:10:30.420 --> 01:10:42.180
Michael Bender: That that sort of when you've got false positive. Yes, you're accessing the dictionary. So you're getting more you have access to access the original elements. So you've got more bits that so you can use that to

500 01:10:42.450 --> 01:10:53.970
Michael Bender: Say, Oh, they're more than there. They're not elements. There are two elements. There are five times as many elements as I thought I would ever have. So that's an opportunity to to add more false positive, more, more fingerprint bits.

501 01:10:54.360 --> 01:11:01.620
Michael Bender: And reduce the the and keep a good false positive rate. So, so that's that's what I'm saying. But

502 01:11:03.960 --> 01:11:14.520
Michael Bender: But even if the point is, even if you don't do that. And even if you don't even if you don't do that at all. The nice thing about keeping fingerprints is

503 01:11:16.980 --> 01:11:17.700
Michael Bender: Is

504 01:11:18.840 --> 01:11:30.450
Michael Bender: Like say it is that you get to decide how many of the fingerprint bits you want to store explicitly and how many of the fingerprint bits you want to store implicitly

505 01:11:31.350 --> 01:11:44.160
Michael Bender: And that is going to change as you increase the size of your hash table. So even without doing anything fancy that I described before fingerprints work really well with the resizing because

506 01:11:47.970 --> 01:11:58.350
Michael Bender: You know with like if you have your hash table you store. One more, one more bit explicitly if you double it, you store one bit of your fingerprint.

507 01:11:59.460 --> 01:12:00.060
Implicitly

508 01:12:01.710 --> 01:12:01.890
Right.

509 01:12:06.960 --> 01:12:12.990
Tim Kaler: Yeah, well I just started thinking about this when we were talking when there was that discussion about implementation complexity as it kind of

510 01:12:14.010 --> 01:12:28.230
Tim Kaler: When I when I kind of think about some of the enhancements to Bloom filters. I think kind of one of the reasons I kind of discount them sometimes is because, you know, they're more complex and often Bloom filters one of its main virtues is simplicity

511
01:12:29.250 --> 01:12:37.830
Tim Kaler: So I was just, Yeah, it'd be very interesting to see what the trade off is when you kind of consider the generalizations of Bloom filters in terms of like space provided

512 01:12:38.160 --> 01:12:46.890
Tim Kaler: Even if it's just an esoteric curiosity. Because, as you said, kind of for practical purposes you you do quite well for you know reasonable set sizes.

513 01:12:48.420 --> 01:12:49.800
Michael Bender: That's right. Totally agree.

514 01:12:51.870 --> 01:12:52.380
Michael Bender: And those are

515 01:12:52.740 --> 01:12:56.520
Michael Bender: You know well like these are amazing. Like all of the questions that people asked, are things

516 01:12:56.520 --> 01:13:00.510
Michael Bender: That I'm excited about it either have done or want to do or

517 01:13:01.710 --> 01:13:01.950
Michael Bender: Just

518 01:13:03.480 --> 01:13:04.650
Michael Bender: Generally excited about.

519 01:13:11.550 --> 01:13:12.510
Julian Shun: Great, thanks a lot.

520 01:13:13.980 --> 01:13:15.660
Julian Shun: Any other questions?

521 01:13:22.110 --> 01:13:29.130
Michael Bender: And you know i'm also available. Like, like, you know, I am happy to stay on if people want to talk. And it's also really nice to see old friends.

522 01:13:30.780 --> 01:13:34.110
Michael Bender: As I said you know the names of people that they haven't spoken to in a long time.

523 01:13:45.900 --> 01:13:46.320
Julian Shun: Well,

524 01:13:47.640 --> 01:13:50.010
Julian Shun: It seems like there are no questions at the moment.

525 01:13:51.720 --> 01:14:01.860
Julian Shun: But I'm going to just leave the zoom room open in case anyone wants to chat, assuming that you're, you don't have to run, Michael.

526 01:14:03.180 --> 01:14:05.130
Michael Bender: Yeah, I'm available. Yeah.

527 01:14:08.760 --> 01:14:15.990
Julian Shun: Yeah, I actually have a three o'clock meeting, I need to get to. But I'll just leave this on.

528 01:14:17.550 --> 01:14:23.760
Julian Shun: Yeah, I'm listening. Yeah. Great. And thanks for all the jokes to make the talk more fun.

529 01:14:26.220 --> 01:14:27.960
Michael Bender: It's really, it's really great to be here.

530 01:14:29.850 --> 01:14:30.480
Michael Bender: Virtually

531 01:14:31.080 --> 01:14:31.890
Julian Shun: Yeah yeah

532 01:14:34.770 --> 01:14:42.360
Julian Shun: Alright, thanks. So I'm going to log off but Linda's still going to be logged on so the meetings, not going to close and

533 01:14:43.980 --> 01:14:51.270
Julian Shun: Yeah. Anyone who wants to chat, please feel free to chat and Michael, when you have to leave you can you can just sign off.