**LinkedIn Learning – Elasticsearch in Depth**

**Edward Capriolo**

**Introduction**

Much more than full-text searches

- [Narrator] Elasticsearch is one of the most popular search engines, but many may not know that Elastic can do more than full tech search. Elasticsearch is a document database that offers the ability to store documents while indexing specific fields inside the documents. These indexes are distributed and replicated across clusters of tens or even hundreds of server computers. My name is Edward Capriolo. I'm a technologist who has been lucky enough to take on a number of big data challenges. I have designed and implement systems that store and analyze terabytes of data using a variety of so-called NoSQL systems. If you're ready to pull up your sleeves and learn how to utilize advance in-depth features of Elasticsearch, join me in my LinkedIn learning course.

**Getting Started**

The Elasticsearch stack

A picture containing text, font, screenshot, white

Description automatically generated

A picture containing text, font, screenshot, white

Description automatically generated

A screenshot of a document database

Description automatically generated with medium confidence

A picture containing text, screenshot, font

Description automatically generated

A diagram of a cluster of nodes

Description automatically generated with low confidenceA picture containing text, screenshot, font, line

Description automatically generated

Installing Elasticsearch and Kibana Locally

*[Skipped to save space (need 100GB+)]*

Executing basic CRUD-like data operations

- [Instructor] In this section, we are going to demonstrate how to do basic create, read, update and delete or CRUD operations. Let's take note of some important aspects in the design of Elasticsearch. First, the protocol is HTTP based. This makes it easy to use when compared to other databases that may only have a binary interface. Even a user with a browser, JavaScript, cURL or Postman has full ability to execute every operation the database supports. Throughout the course, we use **Kibana dev tools** which is a simple tool to execute commands. But it could as easily be done with tools, like we mentioned above, or a high level API written in a programming language like Java. Next, the Elastic API leverages concept from restful web services, like using **HTTP verbs**. A user wishing to add a document uses an http PUT verb or retrieving a document uses a GET verb. In the cases where the requests are complex, Elastic uses structured JSON in the body of requests. This is called the Querydsl. In the case of Elastic, the DSL or domain specific language exposes the unique features that Elastic offers. So here we're going to type. The C in CRUD stands for create. And we could start with that. We start with the PUT verb, which is appropriate since we are creating data. The URL next contains the name of an index, bowled over, which will hold the data. In this case, we append underscore doc, which has a special meaning to Elastic. And the final path of the URL is the ID for the document. Even though this looks like a number, it's actually a string. For the body, we are using a JSON document and including three fields: first name, last name and average. If the insertion is successful, the results will look like that on the right, or the successful count is one. Let's insert a second user. To do this, we copy and paste it. But notice we changed the ID in the URL. If we didn't, we would accidentally overwrite the first document. Let's remove the data we just created. For this, we'll use the DELETE verb and specify the same URL. In the right, we could see what the status result should look like when we match a document and delete it. Quickly, let's look at what some of the endpoints return when you attempt to delete a document that does not exist. Here we showed the status of what happens when you attempt to read and delete a document that doesn't exist in the index. Now let's look at another use case. What if we want to add additional fields to an already existing document or update it? For this, we do not need to know the entire source, only the fields you wish to add. In this example, let's give bowler number one a nickname of lincoln. To do this, we start with a POST verb. In the first part of the URL, we put the index name. In the second, we put update. In the third path, we put the document ID. We're also adding the refresh equals true so that Elastic will re-index this data right away. To update the doc, we use a JSON body. We put a doc object. And inside it, all the fields inside the doc become part of the original document. We could fetch the document back to see if it's been added. Now the fun part. The R in CRUD is read. It is Elasticsearch after all. So let's explore some ways to search our data. If we don't know the idea of a document, we could search it out using the search endpoint. We start it with the GET verb. Again, we mentioned the index name. And this time, we referenced the underscore search end point. Instead of an ID, we added a q parameter to the query and specified a value. Notice it even included a star. The star is like a wild card match. Using q searches across all documents and all fields. If we're only interested in the document data and not the extraneous data Elastic adds to the response, we could use the filter path parameter. Let's give it a try. Since we added hits dot hits dot underscore source, you could see, on the right, we only get the body of the match documents. Let's do a more advanced search. Let's find averages between a minimum and a maximum using range queries. This could be done using the q parameter as well. We will specify the field average and includes beginning and end values. While we've mentioned that Elastic uses REST whenever possible, larger, more complex queries become hard to express even in a URL, even using query parameters. We'll do a few examples that use a query body. We could transplant the queue into the query string object. Notice, this particular query object is very sensitive to white space, so be careful. Adding a single space here makes the query return no results. The rest of the course will cover more complex aspects of Elasticsearch. We will show how to update or delete by query and how to do more complex searching, like nested bool queries. For now, we're going to only give you the most basic match all query, so you could get an idea of what other features may be out there as the auto-complete scroll by. As we could see, there's match, match all, match phrase, match prefix. We won't have time to show all of these features. But you could poke around and try the data yourself. For now, we're just going to do the most basic match all, which returns all documents. Let's add Bob back to see if we could find Bob. As you can see on the right, match all returned all documents.

1. //Elasticsearch uses REST wherever possible

2.

3. //add a document of index bowled\_over with ID 1

4. put bowled\_over/\_doc/1

5. {

6. "first\_name": "Edward",

7. "last\_name": "Capriolo",

8. "average": 190.0

9. }

10.

11. //add a document with ID 2

12. put bowled\_over/\_doc/2

13. {

14. "first\_name": "Bob",

15. "last\_name": "Smith",

16. "average": 204.5

17. }

18.

19. //delete document with ID 2

20. delete bowled\_over/\_doc/2

21.

22. //retrieve the document with id 2, will throw 404 as it is deleted already

23. get bowled\_over/\_doc/2

24.

25. //?refresh=true to make sure elastic index it right away

26. post bowled\_over/\_update/1?refresh=true

27. {

28. //JSON body to update the document

29. "doc": {

30. "nickname": "lincoln"

31. }

32. }

33.

34. //retrieve the \_source element only of document with id 1

35. get bowled\_over/\_source/1

36.

37. //if dont know the ID of the document, can use the search endpoint

38.

39. //\* as the wildcard

40. get bowled\_over/\_search?q=edwar\*

41.

42. //narrow down retrieved result hits.hits.\_source element

43. get bowled\_over/\_search?q=edwar\*&filter\_path=hits.hits.\_source

44.

45. //search for all docs with average between 190 and 200

46. get bowled\_over/\_search?q=average(>190 AND <220)&filter\_path=hits.hits.\_source

47.

48. //transplant the 'q' to query string object

49. get /bowled\_over/\_search

50. {

51. "query": {

52. "query\_string": {

53. "query": "average(>190 AND <220)"

54. }

55. }

56. }

57.

58. //returns all documents

59. get /bowled\_over/\_search

60. {

61. "query": {

62. "match\_all": {}

63. }

64. }

65.

Using Elastic SQL

**Using Elastic SQL**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] In this section, we are going to use Elastic SQL to query data. Elastic is a document database that stores JSON documents, and it has a Query DSL that actually uses JSON as the primary language of interacting with the database. That being said, SQL is a widely popular declarative language to query databases. **Users with SQL experience could leverage Elastic SQL to do many types of queries.** Let's get started using a data set of user-generated content for blogs or poetry. We're going to insert two documents. Each has an author, a date, a title, and a package, and we'll let Elastic create the schema dynamically. Now, to search documents, the most common way is to use the Query DSL. We could put together a simple match query looking for some words in the passage, and we could find some matches. Here, we constructed a simple match query that's looking in the passage fields for bowling, dinner, cheer, or award. This matched some documents. Again, it takes some time to ramp up and learn Query DSL, and since SQL interface is so popular, it could answer a lot of basic queries. So let's search the author field using SQL, and see if we could find the same poem. Let's start by running the most basic query. Here, we'll hit the SQL endpoint, setting the format to text, and we'll make a body feeding one parameter, a query, which is running Select Star. This returns two results, one by Nathaniel Bowlmorn, and another by Whiph Onpin. Let's use a filter to query the results down. Here, we used a where clause in the same way in the Query DSL we would use a filter query. While it is not possible to do every type of SQL query in Elastic SQL, the language is powerful. Let's construct a group and count query. This query has a few sections. In the first, we're selecting the author field and an aggregate field, which is the count. Then we're using a from section to reference our index. We're using a where section to filter the data, and we're grouping by author. Also, notice we use the three quotes for our query. This allows us to do multi-line queries fairly easily. If we remove the format equals text, we could see that the result is returned as JSON, and also, importantly, it has a cursor, which would allow us to retrieve larger result sets and page through them. The endpoint has a translate feature. This takes the SQL query and converts it to a Query DSL query. This is a really great way to learn the Query DSL language. As you could see on the right, the query's been broken down into two parts. The filter is done as a wild card query on the author field, and the grouping is done in an aggregation query.

1. PUT /bowled\_over\_artist\_guild/\_doc/1

2. {

3. "Author": "Nathaniel Bowlmorn",

4. "date" : "2020-01-01",

5. "title" : "It's down to the last frame",

6. "passage" : """As I distinct remember,

7. I was bowling in december,

8. and every separate strikeball,

9. left a defiant 10 pin standing on the deck

10. early i wish to bowl tommorow"""

11. }

12.

13. PUT /bowled\_over\_artist\_guild/\_doc/2

14. {

15. "Author": "Whiph Onpin",

16. "date" : "2020-01-01",

17. "title" : "o Captain, team captain",

18. "passage" : """O Captain! team captain! our painful season is done;

19. Our team has weather'd every split, the place we sought is won;

20. The bowling dinner is near, the awards I hear, the people all cheering,

21. """

22. }

23.

24. GET bowled\_over\_artist\_guild/\_search

25. {

26. "query": {

27. "match": {

28. "passage": "bowl dinner cheer award" //either one of these words

29. }

30. }

31. }

32.

33. //using SQL to find the same poem

34. POST \_sql?format=txt

35. {

36. "query": "select \* from bowled\_over\_artist\_guild" //pure SQL query

37. }

38.

39. //similar to filter query in query DSL

40. POST \_sql?format=txt

41. {

42. "query": "select \* from bowled\_over\_artist\_guild where Author like '%Whiph%'"

43. }

44.

45. //""" three quotes for multiline texts

46. POST \_sql

47. {

48. "query": """

49. SELECT Author,count(1)

50. FROM bowled\_over\_artist\_guild

51. WHERE Author like '%Whiph%'

52. GROUP BY Author"""

53. }

54.

55. //the /translate converts SQL query to Query DSL query (excellent way to learn Query DSL!)

56. POST \_sql/translate

57. {

58. "query": """

59. SELECT Author,count(1)

60. FROM bowled\_over\_artist\_guild

61. WHERE Author like '%Whiph%'

62. GROUP BY Author"""

63. }

64.

A screenshot of a computer

Description automatically generated with medium confidence

Basic Elastic schema and data design

**Basic Elastic schema and data design**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] In this section we'll discuss and show examples of schema and data design in Elastic. Use cases of Elastic vary. Some may use it as a primary database meaning it is possibly the only place data is stored. Other times Elastic is used to supplement the primary database. This is common because many relational databases are well suited for dealing with discreet rows and tables or columns of fixed width while Elasticsearch can store documents with megabytes of text and provide sophisticated searching inside these documents such as fuzzy matching or boosting. When using Elasticsearch as a supplemental database, it is possible not to store the initial source document. This might be a little bit hard to wrap your head around but think about it this way. **When indexing data like text for searching, the data is tokenized and normalized to make the searching more efficient.**Thus, if you only need the data in Elastic for that case, storing the original document isn't strictly necessary. This is a very important design decision because if the source data is not stored in Elastic, it can be hard to debug how something is working and features like re-indexing can't work. **Elastic generally advises that you should store the data and I would say you should as well**. Let's go through an example where we choose not to store the source data. Let's quickly demonstrate setting no source. On line three, we've created an index definition. We're starting with mappings and we're using the underscore source field, and we're setting the enabled attribute equal to false. Here we're inserting a simple document. And here we've defined a search. In this case, we're searching where the author keyword is Nathaniel Boulmore. Let's see if we find the data. In this case, we do. Remember the source isn't saved so all we know is that document ID matches this search. Let's give an example of how this can be difficult. Here are then another search, which seems to be exactly the same, but something is different. I'm searching not in the author keyword field but the author field, which has been tokenized. Notice how without the source data, it's very hard to understand why this query would work and this query wouldn't work. In Elastic, every document has an ID. During insertion, if the user does not specify an ID, Elastic will create one dynamically. If the user does select an ID, Elastic will use that. This is implications depending on if the action is an insert, an upsert, or an update. And take note that when specifying an ID, Elastic does have to do some more work on the right path. This is an important design that affects how your application will work. For example, if you're doing blind rights and you don't use an ID, you could potentially introduce duplicates in a failure scenario. This might be okay. Maybe you'll remove the duplicates in your query phase later, or some other background process will do it. But you may want tighter control and you may want to use updates to make sure documents could only exist once. To illustrate this, we're going to create a simple data set. Notice on line 47, we're using the put verb and we're specifying an ID of one. We could also change this and use the post verb. In this case, we don't have to specify a document ID. This worked. Notice the ID created is returned on the right. We could use this to search the data. Again, remember, note that this approach could easily create data duplication. Each time this request is resent, I'm getting a new ID. There's now 29 documents of this than the database. Elastic is often described as a near realtime search engine. While many operations are atomic like put or get, other operations like fuzzy matching rely on indexes to be rebuilt to reflect the latest data. Rebuilding an index is expensive, especially for high data volumes, so it's usually only done periodically. In order to show how the refresh interval works, we've put together this script. We're going to create an index, set the settings for the refresh time to be 30 seconds. We're going to insert some data and then retrieve it using the get API. Then we're going to look for it to search API. If we do this quickly, the data won't be found, but then if we search 30 or 31 seconds later, it should be found. Let's give it a go. The data is found here as we could see on the right. Notice the data is not found. We'll wait a little while. La de da, waiting for some data. Waiting, waiting, waiting, waiting, waiting waiting. Hm, da da. Oh, there it is. So this shows you the effect of the refresh interval, and again, this setting is in place because built a new index is an expensive process.

1. //here source isnt saved

2. PUT no\_source\_example

3. {

4. "mappings": {

5. "\_source": {

6. "enabled": false

7. }

8. }

9. }

10.

11. PUT /no\_source\_example/\_doc/1

12. {

13. "Author": "Nathaniel Bowlmorn",

14. "date" : "2020-01-01",

15. "title" : "It's down to the last frame",

16. "passage" : """As I distinct remember,

17. I was bowling in december,

18. and every separate strikeball,

19. left a defiant 10 pin standing on the deck

20. early i wish to bowl tommorow"""

21. }

22.

23. //data is successfully retrieved (elastic acknowledged Nathaniel Bowlmorn as ID 1)

24. GET /no\_source\_example/\_search

25. {

26. "query": {

27. "term": {

28. "Author.keyword": {

29. "value": "Nathaniel Bowlmorn"

30. }

31. }

32. }

33. }

34.

35. //data CANNOT be retrieved and the reason is unclear since we do not save the \_source element. (the only diff in this query is that "Author" is missing ".keyword", meaning that "Nathaniel Bowlmorn" has been tokenized and thus no "Author" found for "Nathaniel Bowlmorn")

36. GET /no\_source\_example/\_search

37. {

38. "query": {

39. "term": {

40. "Author": {

41. "value": "Nathaniel Bowlmorn"

42. }

43. }

44. }

45. }

46.

47. #user supplied id example

48. DELETE bowled\_over\_artist\_guild

49. PUT /bowled\_over\_artist\_guild/\_doc/1

50. {

51. "Author": "Nathaniel Bowlmorn",

52. "date" : "2020-01-01",

53. "title" : "It's down to the last frame",

54. "passage" : """As I distinct remember,

55. I was bowling in december,

56. and every separate strikeball,

57. left a defiant 10 pin standing on the deck

58. early i wish to bowl tommorow"""

59. }

60.

61. //POST verb not need ID, ES will dynamically assign ID

62. POST /bowled\_over\_artist\_guild/\_doc

63. {

64. "Author": "Nathaniel Bowlmorn",

65. "date" : "2020-01-01",

66. "title" : "It's down to the last frame",

67. "passage" : """As I distinct remember,

68. I was bowling in december,

69. and every separate strikeball,

70. left a defiant 10 pin standing on the deck

71. early i wish to bowl tommorow"""

72. }

73.

74. //each time the POST is run, it assigns new ID (hence can create multiple duplicates)

75. POST /bowled\_over\_artist\_guild/\_doc

76. {

77. "Author": "Nathaniel Bowlmorn",

78. "date" : "2020-01-01",

79. "title" : "It's down to the last frame",

80. "passage" : """As I distinct remember,

81. I was bowling in december,

82. and every separate strikeball,

83. left a defiant 10 pin standing on the deck

84. early i wish to bowl tommorow"""

85. }

86.

87. GET /bowled\_over\_artist\_guild/\_source/kpAFmYgBFdgHJSLmRWij

88.

89. GET /bowled\_over\_artist\_guild/\_doc/kpAFmYgBFdgHJSLmRWij

90. GET /bowled\_over\_artist\_guild/\_source/kpAFmYgBFdgHJSLmRWij

91.

92. # refresh settings

93. PUT refresh\_example

94.

95. //indexing is only done every 30s

96. PUT refresh\_example/\_settings

97. {

98. "index" : {

99. "refresh\_interval" : "30s"

100. }

101. }

102.

103. PUT /refresh\_example/\_doc/1

104. {

105. "Author": "Nathaniel Bowlmorn",

106. "date" : "2020-01-01",

107. "title" : "It's down to the last frame",

108. "passage" : """As I distinct remember,

109. I was bowling in december,

110. and every separate strikeball,

111. left a defiant 10 pin standing on the deck

112. early i wish to bowl tommorow"""

113. }

114.

115.

116. GET /refresh\_example/\_doc/1

117.

118. //need to wait >30s as indexing is only done every 30s

119. GET /refresh\_example/\_search

120. {

121. "query": { "match\_all": {}}

122. }

123.

Document database and schema topics

A picture containing text, screenshot, font

Description automatically generated

A picture containing text, screenshot, font

Description automatically generated

**Document database schema topics**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] In this section we're going to cover document database schema topics including types, mapping, and dynamic schemas. Before we get started, however, I'd like to take a minute to compare document databases to relational databases. Relational databases are very common and thus there's lots of information about how to design them. Typically, databases have one-to-one relationships, one to many relationships, many to many relationships, and there's volumes of material on how to create a database correctly either in third normal form or fourth normal form or even fifth normal form. Document databases are different. There's not as much theory on them. Many of them fall into the category of noSQL or newSQL stores and people have to come up with new patterns. Typically in a document database, you do normalizing because joins and other relationships aren't as easy to model. Then the question becomes how do you divide your data? Do you put it in one very large document? Does it have nested documents inside of it or subsections? That takes some specialized learning and, quite frankly, some trial and error. So what we're going to do is we're going to just start with a simple schema and get more complicated as we explain some of these things. So the first things many people think is they may believe that Elastic Search is schemaless. You could come to the web interface of Elastic and you could create a document with a date and age and a nickname and say insert this into an index called Simple Schema. And Elastic accepts it even though we haven't set up any schema at all. So they might believe it's schemaless, but that isn't true. A mapping was created dynamically that we could take a look at by hitting the mapping endpoint. As we could see on the right here, Elastic Search auto detected the schema, and in this case correctly, or at least we think it's correctly. Let's insert another more complex document. Here notice the nickname is a single field, but here it's an array. We've added two new objects. One is called Complex Type and one is an array called Complex Data 2, which has Complex objects inside of it. Elastic Search allowed us to insert this data. Let's see what the mapping looks like. As we can see, the mapping is getting much larger. We have Complex Data Defined, Complex Data 2. What's going on here? Well, what's happening is Elastic Search is a document database, so unless the document violates the schema, it will allow it to go in. So let's make a schema violation. Let's change this 40 years to a number like 40 I guess That works. But, why does it work? It only works because Elastic is converting this number to a string before inserting it. But it wouldn't do it the other way. If we had defined this data as a number first, it wouldn't take a string, it would throw an exception when we tried to insert it and that would be pretty bad. So as a rule of thumb, and the way you should design every application not just a rule of thumb, you should create a schema. It keeps things consistent. So we're going to do that here. We're going to get a fresh start. So what we're going to do is we're going to delete what we've created. **I've already captured the mapping and I'm going to add something to the mapping, which is dynamic equals false. Notice we could still insert the same data and let's try to search it all. The data is still here. And let's look at the mapping. Now, the only difference in this case is we could still insert dynamic documents, however, since we've set strict equal to false, we won't change the mapping. Let me show you what I mean. We've here edit fields to the document, but they aren't present in this mapping. And this is dynamic set to false meaning when new data comes in we aren't going to create a field definition for it, even though we'll accept it in the documents.** Now this is still problematic later on if we did want to index this field and the data wasn't in a consistent way, it wouldn't work and be useful. So this is a trade off with document databases. You have to decide what you want to allow and how strongly typed you want everything to be.

1. DELETE simple \_schema

2.

3. //ES is NOT schemeless, prove that we can get the schema while retrieving the mapping API

4. PUT simple\_schema/\_doc/1

5. {

6. "date": "2020-01-01",

7. "age": 40,

8. "nickname": "butch"

9. }

10.

11. PUT simple\_schema/\_doc/2

12. {

13. "date": "2020-01-01",

14. "аgе": 30,

15. "nickname": ["flash", "derek"],

16. "complex data1": {

17. "key": "value",

18. "check": false

19. },

20. "complex data2": [

21. {

22. "year": "2020",

23. "features": ["aircondition", "sunroof"]

24. },

25. {

26. "happy": "joy"

27. }

28. ]

29. }

30.

31. GET simple\_schema/\_search

32. //mapping is created dynamically

33. GET simple\_schema/\_mapping

34.

35. //create a schema

36. DELETE simple\_schema

37.

38. PUT simple\_schema

39. {

40. "mappings": {

41. "dynamic": false,

42. "properties": {

43. "age" : {

44. "type": "long"

45. },

46. "complex\_data1": {

47. "properties": {

48. "check": {

49. "type": "boolean"

50. },

51. "key": {

52. "type": "text",

53. "fields": {

54. "keyword": {

55. "type": "keyword",

56. "ignore\_above": 256

57. }

58. }

59. }

60. }

61. }

62. }

63. }

64. }

65.

**Specialized Data Type and Query Support**

Finding objects with Geoshape

- [Instructor] In this section, we're going to demonstrate how to use geospatial indexing support. To get started, we will create an index. In the mappings section, we will create a field called pin, which is an object. Inside it, we will nest a field called location. Note, it's important that the type of this object will be geo\_point. This is a specialized type that represents a point on the earth, which is a sphere. I feel the need to tell a flat earth joke, but I'll avoid that. Next, we'll populate a small database of interesting things. Let's imagine that "Bowled Over" has propelled our bowling career and on the Pro bowling tour, we're traveling around the world and while we're doing that, we wish to do some sightseeing. For our first document, we're adding the Statue of Liberty to our database. The field attraction\_type is set to monument and we've given it a human-readable name. For the pin coordinates, we specified longitude and latitude, which I researched offline. We also have created two other documents. One is a theme park I often visit in Rye, New York called Playland. The other is the location of The Alamo, which is in Texas. Note that the Statue of Liberty and Rye Playland are very close together, while Texas is fairly far away. Now that this data is inserted, let's do some searching. We're starting with a basic template to query data. You could see it on line 56. Here, we're starting with a Boolean query and we're doing a match\_all and we're boosting results where the type is theme park. We can see that all three documents were returned, but the Playland theme park has a much higher score, which you could see on line six in the right pane. Now, we're going to modify this base query and have it take into account some geospatial data. To do this, we've added a filter clause to the query. Recall that filters must be true for results to be returned. Inside the filter, we're adding a geo\_distance query object. This has some mandatory and optional properties. An important one here is distance. Here, we specified the distance as 50 miles. Next, we're setting the pin.location to somewhere in New York. As expected, we could see that only two of the results were returned, because they're within 50 miles of this pin point. Another neat thing we can do, instead of filtering by distance, is boost by distance. This allows us to create queries where we can give locations closer to our point of reference a higher score. To do this, we add a should clause and we use the distance\_feature query object. We set the fields we want to reference, in this case pin.location, and we set the pivot field to 10 miles. This effectively adjusts the score by 10 mile increments, saying that as things get 10 miles farther away, the score gets lowered. Finally, we put the origin. Here, I use the array syntax. (chuckles) Be very careful here. I accidentally assumed that when using array syntax the values are latitude, longitude, but actually they're longitude, latitude and inverting this location New York actually placed it somewhere in Antarctica. After running this query, we could see that the scores have been adjusted and the score of Playland is higher than the score of Statue of Liberty just because we've boosted it and it's closer to our reference location. Also note, Elastic has some very cool visualizations in Kibana. Let's look at how maps could be rendered. Here, we navigate to the Maps interface and we add a layer. We could take documents from Elasticsearch. We specify our index using an index pattern, which is location\*, and then we add a layer. As you could see here, the locations appear on our map. Again, Kibana is a very cool and very functional web interface.

1. //geospatial indexing support

2. //specialized type that represents a point on the earth

3.

4. DELETE locations

5.

6. //create a mapping that uses the geopoint type

7. PUT /locations

8. {

9. "mappings": {

10. "properties": {

11. "pin": {

12. "properties": {

13. "location": {

14. "type": "geo\_point"

15. }

16. }

17. }

18. }

19. }

20. }

21.

22. // create some attractions

23. PUT /locations/\_doc/1

24. {

25. "attraction\_type": "monument",

26. "name": "statue of liberty",

27. "pin": {

28. "location": {

29. "lat": 40.6892,

30. "lon": -74.044502

31. }

32. }

33. }

34.

35. PUT /locations/\_doc/2

36. {

37. "attraction\_type": "theme park",

38. "name": "Playland",

39. "pin": {

40. "location": {

41. "lat": 40.981613,

42. "lon": -73.691925

43. }

44. }

45. }

46.

47. PUT /locations/\_doc/3

48. {

49. "attraction\_type": "monument",

50. "name": "The Alamo",

51. "pin": {

52. "location": {

53. "lat": 29.425171,

54. "lon": -98.494614

55. }

56. }

57. }

58.

59. GET /locations/\_search?filter\_path=hits.hits.\_source, hits.hits.\_score

60. {

61. "query": {

62. "bool": {

63. "must": {

64. "match\_all": {}

65. },

66. "should": {

67. "match": {"attraction\_type": {"query": "theme park", "boost": 2}}

68. }

69.

70.

71. }

72. }

73.

74. }

75.

76.

77. // search for locations within 50 miles of whie plains

78. GET /locations/\_search?filter\_path=hits.hits.\_source, hits.hits.\_score

79. {

80. "query": {

81. "bool": {

82. "must": {

83. "match\_all": {}

84. },

85. "should": {

86. "match": {"attraction\_type": {"query": "theme park", "boost": 2}}

87. },

88.

89. "filter": {

90. "geo\_distance": {

91.

92. "distance": "50mi",

93. "pin.location": {

94. "lat": 41.033,

95. "lon": -73.762909

96. }

97. }

98. }

99. }

100. }

101.

102. }

103.

104. //boost closer results

105. GET /locations/\_search

106. {

107. "query": {

108. "bool": {

109. "must": {

110. "match\_all": {}

111. },

112. "should": {

113. "distance\_feature": {

114. "field": "pin.location",

115. "pivot": "10mi",

116. //longitude, latitude - NOT vice versa

117. "origin": [-73.7629,41.0340],

118. "boost": 2.0

119. }

120. },

121.

122. "filter": {

123. "geo\_distance": {

124.

125. "distance": "100mi",

126. "pin.location": {

127. "lat": 41.0340,

128. "lon": -73.7629

129. }

130. }

131. }

132. }

133. }

134.

135. }

136.

Storing data and searching complex nested data [UNCLEAR]

- [Instructor] This section is going to describe how to store, index and search complicated nested elements inside JSON documents. Here on line three we've created a document that describes the scores of two teams bowling against each other. Notice on line six we have a scores element which is actually an array of multiple games. Notice each bowler has bowled two games. Let's insert the document. In this case, Elastic search has created the mapping for us. Let's take a look at it. On the right, on line 18 we can see scores doesn't have a type. Internally, it's type is object. There's implication that comes with the object storage and we'll explain them right now. Let's do a search for a bowler named John on the Strike Force team. Notice we found a match but what this means is that there is a document with a bowler named John and a team named Strike Force. But importantly, this design does not allow us to match only a single team with a bowler named John. If we scroll up to our original document we could see that the bowler named John on line 34 is on the Bowling Turkeys. There's no one named John on the Strike Force team. Let's delete the index and recreate it using the nested type instead of the object type for the scores array. Internally, what this does is each nested object becomes an internal document linked to the parent document. Note here that this index has the same mappings as the original one. The only difference is on line 95 we've used the type nested. Let's go back and reinsert our original document. Now, to search this document for the criteria we want we're going to use a special search type called nested. On line 127, we've started a search. On line 130, you could see that our query contains an element called nested supplying a path. In this case, the path is scores, the element, that's the array. This type of query will unroll the scores into multiple documents, allowing us to search them, and in the query we've included our original search to look for a bowler named Strike Force. Let's see if we get any matches. There is indeed a bowler named Bob on the Strike Force team as we could see on the hits on the right on line 30. Let's do our original search and see if it incorrectly finds a bowler named John. Here on line 153, we've done the same search. We've just replaced the name Bob with John. As expected, as you can see on the right, there's no matches here. There's no hits. That's because we're only searching nested documents. A quick note about nesting is that every nested internal document creates a separate document in Elastic so it's not the ideal to use when you may have arrays that have hundreds or even thousands of internal documents. There may be a better way to design and we'll talk about some later.

1. #This section will describe how to store index and search

2. #complex nested elements inside json documents

3. # here we have modeled a document that tracks the scores of two

4. #teams bowling against each other. notice scores is an array of objects

5. # each team has two bowlers who have bowled 2 games

6. DELETE bowled\_over\_team\_scores

7. PUT /bowled\_over\_team\_scores/\_doc/1

8. {

9. "league\_name": "friday night lights",

10. "date" : "2020-01-01",

11. "scores" : [

12. {

13. "game": 1,

14. "team\_name" : "strike force",

15. "bowler": "bob",

16. "score": 203

17. },

18. {

19. "game": 2,

20. "team\_name" : "strike force",

21. "bowler": "bob",

22. "score": 204

23. },

24. {

25. "game": 1,

26. "team\_name" : "strike force",

27. "bowler": "sara",

28. "score": 177

29. },

30. { "game": 2,

31. "team\_name" : "strike force",

32. "bowler": "sara",

33. "score": 176

34. },

35. {

36. "game": 1,

37. "team\_name" : "bowling turkeys",

38. "bowler": "john",

39. "score": 160

40. },

41. {

42. "game": 2,

43. "team\_name" : "bowling turkeys",

44. "bowler": "john",

45. "score": 164

46. },

47. {

48. "game": 1,

49. "team\_name" : "bowling turkeys",

50. "bowler": "ted",

51. "score": 121

52. },

53. {

54. "game": 2,

55. "team\_name" : "bowling turkeys",

56. "bowler": "ted",

57. "score": 241

58. }

59. ]

60. }

61. //get mapping

62. # We inserted this document so elastic created the mapping for us dynamically

63. # notice that for scores the mapping is not specifically define so it defaults

64. # to type object. This has implications when it comes to searching that we will see

65.

66. GET /bowled\_over\_team\_scores/\_mapping

67.

68. #Lets do a searach for a bowler named john on the strike force team

69. #notice we found a match, but this means there is a document with

70. # a bowler named john and a team named strike force.

71. # but IMPORTANTLY this design using object doesnt allow us to match

72. # only a team with a bowler named john

73. #why

74. GET bowled\_over\_team\_scores/\_search

75. {

76. "query": {

77. "bool": {

78. "must": [

79. { "match": { "scores.team\_name": "strike force" }},

80. { "match": { "scores.bowler": "john" }}

81. ]

82. }

83. }

84. }

85.

86. GET bowled\_over\_team\_scores/\_search

87. {

88. "query": {

89. "bool": {

90. "must": [

91. { "match": { "scores.team\_name": "strike force" }},

92. { "match": { "scores.bowler": "j" }}

93. ]

94. }

95. }

96. }

97. #lets delete the index and recreate it using the nested

98. #type instead of the object type for the scroes array

99. # internally what this does is each nested object becomes an internal document

100. # linked to the parent document

101. DELETE bowled\_over\_team\_scores

102.

103. PUT bowled\_over\_team\_scores

104. {

105.

106. "mappings" : {

107. "properties" : {

108. "date" : {

109. "type" : "date"

110. },

111. "league\_name" : {

112. "type" : "text",

113. "fields" : {

114. "keyword" : {

115. "type" : "keyword",

116. "ignore\_above" : 256

117. }

118. }

119. },

120. "scores" : {

121. "type": "nested", //note the nested type here

122. "properties" : {

123. "bowler" : {

124. "type" : "text",

125. "fields" : {

126. "keyword" : {

127. "type" : "keyword",

128. "ignore\_above" : 256

129. }

130. }

131. },

132. "game" : {

133. "type" : "long"

134. },

135. "score" : {

136. "type" : "long"

137. },

138. "team\_name" : {

139. "type" : "text",

140. "fields" : {

141. "keyword" : {

142. "type" : "keyword",

143. "ignore\_above" : 256

144. }

145. }

146. }

147. }

148. }

149. }

150. }

151. }

152.

153. # Now we can use a special query type named nested which searches across

154. # nested documents this matches a single score with the criteria

155. GET bowled\_over\_team\_scores/\_search

156. {

157. "query": {

158. "nested": {

159. "path": "scores",

160. "query": {

161. "bool": {

162. "must": [

163. { "match": { "scores.team\_name": "strike force" }},

164. { "match": { "scores.bowler": "bob" }}

165. ]

166. }

167. }

168. }

169. }

170. }

171.

172. # this search doesn not match because a single element in the array does not match

173. GET bowled\_over\_team\_scores/\_search

174. {

175. "query": {

176. "nested": {

177. "path": "scores",

178. "query": {

179. "bool": {

180. "must": [

181. { "match": { "scores.team\_name": "strike force" }},

182. { "match": { "scores.bowler": "john" }}

183. ]

184. }

185. }

186. }

187. }

188. }

189.

Creating parent/child relationships between documents

**Creating parent/child relationships between documents**

Selecting transcript lines in this section will navigate to timestamp in the video

- In this section, we're going to show how to relate documents together using the parent-child features. So before we get into using this feature, we're going to start off with a few caveats. Elasticsearch is not a relational database. It is a document oriented database. And this type of data store favors de normalizing data, which generally means copying data instead of trying to link documents together. However, this feature is here and it is kind of useful and interesting and we'll explain how it works. So the best use case of this feature is when a parent has many children. So the canonical example given is if you have a store with items and you have lots of orders of those items, that would be a perfect time to use it. Where a bad time to use it would be hundreds of millions of items related to hundreds of millions of other items. There's a few other caveats such as documents can only have one parent. But let's not get too deep into that. Let's just give an example and go with it. So for this example, we're going to use the idea of a bowling alley, having arcade machines. So we're going to model the bowling alley in an index called "lanes" and we're also going to model the arcade machines inside separate documents. To do this we're starting with the mapping. Notice on line nine, we have a type called "join\_field." No, we have a type name join\_field, and it's type is join and we've listed relations, biz name, arcade machine. In this relation, the biz name is the parent and the arcade machine is the child. So we're going to create our first document here. This will be a bowling alley called "White Plains Bowl." And if you notice the join field, the type is biz name. So it's the parent side of this relationship. Now we're going to say that this bowling alley has two arcade machines. So we're going to create documents for this. Pay special attention here. We're creating a document in the same index. Its ID is three, but the routing is one. This is very important. The documents that represent the arcades or the children have to be in the same chart as the parent. Otherwise this won't work. So this is where the routing is used. It tells Elastic how to route this document. This document represents an arcade machine of "Street Fighter Two." And as you can see, it's joint field is an arcade machine and it is Parent is ID one. Now we're going to create another arcade machine, kind of the grabby claw thing where you almost pick up the toy but you never get it. This is also in the same bowling alley. Awesome. So now let's see how we can use these features. The most simple thing to do is say, well, I'm going to go to a bowling alley and I know which one I'm going to go to. Before I go, I want to get a feel for what arcade machines are there. So can issue a query where we ask for the parent one, what are arcade machines or what children are of that parent? Awesome. We could see here that "Street Fighter Two" and the Grabby Claw thing is available in the arcade. So I want to go out and ask the Bowled on social network where I could find this arcade machine. So here, we're using a different query. starting on line 66, you could see the query. We're using the has child's relationship and we're asking who has a child of type arcade machine whose name happens to be "Street fighter." And as you could see, Elastic correctly told us that "White Plains Bowl" has this arcade machine.

1. DELETE lanes

2. PUT lanes

3. {

4. "mappings": {

5. "properties" : {

6. "id" : {

7. "type": "keyword"

8. },

9. "join\_field": {

10. "type": "join",

11. "relations" : {

12. "biz\_name": "arcade\_machine"

13. }

14. }

15. }

16. }

17. }

18.

19. PUT lanes/\_doc/1

20. {

21. "id": "1",

22. "text": "White Plains Post Bowl",

23. "join\_field": {

24. "name": "biz\_name"

25. }

26. }

27.

28. //route document to its parent (id=1, need to be the same id as the parent shard)

29. PUT lanes/\_doc/3?routing=1

30. {

31. "id": "3",

32. "text": "Street fighter 2",

33. "join\_field": {

34. "name": "arcade\_machine",

35. "parent": "1"

36. }

37.

38. }

39.

40. PUT lanes/\_doc/4?routing=1

41. {

42. "id": "4",

43. "text": "The grabby claw thing",

44. "join\_field": {

45. "name": "arcade\_machine",

46. "parent": "1"

47. }

48.

49. }

50.

51. #arcade machines in parent

52. GET lanes/\_search

53. {

54. "query": {

55. "parent\_id": {

56. "type": "arcade\_machine",

57. "id": "1"

58.

59. }

60.

61. }

62. }

63.

64. #arcade machines in parent

65. GET lanes/\_search

66. {

67. "query": {

68. "has\_child": {

69. "type": "arcade\_machine",

70. "query": {

71. "term": {

72. "text.keyword": {

73. "value": "Street fighter 2"

74. }

75. }

76. }

77. }

78.

79. }

80. }

81.

Combining multiple fields to simplify searching

**Combining multiple fields to simplify searching**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] In this section, we are going to demonstrate how to combine fields for searching. For end users, we want searching to be as simple as possible. In some cases, this can be complicated. An index could have multiple fields and for a Elastic search performance is generally better when searching fewer fields. To solve for this we are going to create an amalgamation field which combines data from multiple fields into a single field. For this example, we are going to model a small data set with user information for our social network for bowlers, Bowled Over. On line three are we are using the put verb. The target URL is the Bowled Over endpoint and the JSON body has mappings for two fields a first name and a last name. Notice we used keyword for the fields instead of text. This is implications we will solve later on in the lesson. (mouse clicking) Next, we are going to insert two small documents which simulate users. Notice for this example, in the second document Edward is an upper case. Let's imagine someone using our social network. When they join, they want to look for their friends. They've heard that Edward is a member so they want to go look for him. To do this, we use a get verb and we're specifying a URL containing the Bowled Over index. We use the underscore search endpoint and we use the simple Q equals Edward parameter. This search didn't find Edward and the reason is the first name field is defined as a keyword, meaning only exact matches find the data. If we change this to uppercase Edward, the record is found. One way to solve this problem is by using the copy to feature. What this feature does is at insertion time when data is added values of some fields are copied to others. To accomplish this, we'll add a field called monikers which is a fancy synonym for name. Next, for the existing fields, first name and last name, we will change them and include copy to settings. You can see on line 37, when we overwrite an existing mapping, we use a put and we specify the underscore mapping in the URL and we don't include the mapping as the top level element of the body. For this feature to work we actually have to re-insert the data. Take note that changing the mapping does not automatically cause a re-index. Let's go back to our original searches and try them again, starting with the first name that was not the same case as the inserted data. In the right result pain you could see that the data is found. This is 'cause behind the scenes the query is searching all fields including the new moniker field. Let's look at another quick example. Often people have trouble spelling names they're unfamiliar with. In particular, my last name ends in an O but people sometimes believe it ends in an A. We can use the fuzziness feature to attempt to match spellings that are close. To do this, we're creating a match query and we're using the fuzziness parameter. Notice this found me, even though the last name was specified as Capriola in the query but in the data set, it's Capriolo.

1. DELETE bowled\_over

2.

3. //automation field - combine data from multiple fields to one field

4. PUT bowled\_over

5. {

6. "mappings": {

7. "properties": {

8. "first\_name": {

9. "type": "keyword"

10. },

11. "last\_name": {

12. "type": "keyword"

13. }

14. }

15. }

16. }

17.

18. PUT bowled\_over/\_doc/1

19. {

20. "first\_name": "bob",

21. "last\_name": "smith",

22. "average": 204.5

23. }

24.

25. //the 'E' is uppercase

26. PUT bowled\_over/\_doc/2

27. {

28. "first\_name": "Edward",

29. "last\_name": "capriolo",

30. "average": 192.4

31. }

32.

33. //the 'e' is lowercase - didn't find edward as type="keyword" is case-sensitive

34. GET /bowled\_over/\_search?q=edward

35.

36. //the 'E' is lowercase - edward found

37. GET /bowled\_over/\_search?q=Edward

38.

39.

40. PUT bowled\_over/\_mapping

41. {

42.

43. "properties": {

44. "first\_name": {

45. "type": "keyword",

46. "copy\_to": "monikers" //fancy term for 'name', type="text" allows case-insensitive searching

47. },

48. "last\_name": {

49. "type": "keyword",

50. "copy\_to": "monikers"

51. },

52. "monikers": {

53. "type": "text"

54. }

55.

56. }

57. }

58.

59. //fuzziness allows mispeled to be matched [0,1,2], max is fuzziness=2

60. GET bowled\_over/\_search

61. {

62. "query": {

63. "match" : {

64. "monikers": {

65. "query": "capriola",

66. "fuzziness": 1

67. }

68. }

69. }

70. }

71.

Using Elasticsearch pipelines to enhance data

- [Instructor] In this section, we're going to learn about a data processing feature in Elastic known as ingest pipelines. Pipelines provide the ability to do data processing inside the Elasticsearch database. This allows us to alter data, add or remove fields, or even prevent a record from being added if it doesn't meet requirements. First, we are going to create an index with a mapping. This is a modified version of the schema we use throughout the course representing users in our fictional social network for bowlers named bowled\_over. This schema simply defines a few fields to store the user's name. A common ask is to record when a document is inserted. Having the insert time is good for diagnostics. And let's say that our social network, we may find it useful to boost searches for new users. And for that, we wish to automatically attach the time the data was indexed. We could use the pipeline for this. If you notice on line 22, we've created a statement which will create a pipeline known as add\_index\_time. These could have descriptions, but the most interesting part is the processor section. Here we've defined the set processor. And this element has two parts, a field and a value. You can notice in the field, we're using the \_source, which means the source document being inserted and index time is the fields you wish to add. The value uses templating. And again, the ingest object is provided automatically, as well as a timestamp. So we don't need very much logic here. Running the PUT will insert, and we'll create the index. And then by default, we want to have the pipeline run when we insert into our bowled\_over index. So we'll set it as the default pipeline for bowled\_over. Now that we've defined our pipeline and made it the default pipeline for our bowled\_over index, let's do an insert and see what happens. Notice here, we're inserting a document with only two fields, the first name and the last name. The insert was successful. Let's see what happens when we read it back. Awesome. We can see that the index time has been automatically added to this document. Let's look at another built-in processor. It is very common to use IP addresses for logging web traffic. Elasticsearch will download GeoIP databases, and it provides a processor, that given a source IP will automatically attach GeoIP information to a document as it's inserted. On line 50, you can see we're defining another ingest pipeline known as geoip. For the processor here, we're setting it to be geoip, and we're telling it which fields we expect the data to be in, which in this case is the ip field. If you remember, when we defined the bowled\_over index, we gave it a default pipeline. So in this case, we could override the pipeline by providing an argument in our PUT statement. Here, we're setting an IP and we're inserting data. Let's see what happens when we read it back. Great. We can see here that the document has GeoIP data, which has been automatically added, including the country, latitude, and longitude. It is also possible to use conditional logic in a pipeline. For this example, let's say we wish to use a bowler's average to decide which pipeline to run. To do this, if you look at line 73, we'll define a new pipeline called conditional\_pipeline. Notice this one has two processors. Each processor is activated by an IF statement. In this case, the context, which is the document being inserted, needs a field named average. And if the field is less than 200, the add\_index\_time pipeline will be activated. And if the field is more than 200, the geoip pipeline will be activated. Again, when we're putting the document, we could override the default index and select this one. So let's see what happens when we use the conditional pipeline. If we read back the data, we can see that the geographic GeoIP information has been added. This is because the user's average was greater than 200.

1. DELETE /bowled\_over

2.

3. PUT /bowled\_over

4.

5. PUT bowled\_over/\_mapping

6. {

7.

8. "properties": {

9. "first\_name": {

10. "type": "keyword",

11. "copy\_to": "monikers"

12. },

13. "last\_name": {

14. "type": "keyword",

15. "copy\_to": "monikers"

16. },

17. "monikers": {

18. "type": "text"

19. }

20.

21. }

22. }

23.

24. PUT \_ingest/pipeline/add\_index\_time

25. {

26. "description": "Adds indexed\_at timestamp to documents",

27. "processors": [

28. {

29. "set": {

30. "field": "\_source.indexed\_time",

31. "value": "{{\_ingest.timestamp}}"

32. }

33. }

34. ]

35. }

36.

37. PUT bowled\_over/\_settings

38. {

39. "index.default\_pipeline": "add\_index\_time"

40. }

41.

42. PUT bowled\_over/\_doc/1

43. {

44. "first\_name": "Bob",

45. "last\_name": "Smith"

46. }

47.

48. GET /bowled\_over/\_doc/1

49.

50. PUT \_ingest/pipeline/geoip

51. {

52. "description": "Add geoip info",

53. "processors": [

54. {

55. "geoip": {

56. "field": "ip"

57. }

58. }

59. ]

60. }

61.

62. PUT /bowled\_over/\_doc/1?pipeline=geoip

63. {

64. "first\_name": "Bob",

65. "last\_name": "Smith",

66. "ip": "209.191.139.200"

67. }

68.

69. GET /bowled\_over/\_doc/1

70.

71. DELETE /bowled\_over/\_doc/1

72.

73. PUT \_ingest/pipeline/conditional\_pipeline

74. {

75. "processors": [

76. {

77. "pipeline": {

78. "if": "ctx.avg < 200", //ctx stands for context

79. "name": "add\_index\_time"

80. }

81. },

82.

83. {

84. "pipeline": {

85. "if": "ctx.avg >= 200",

86. "name": "geoip"

87. }

88. }

89. ]

90. }

91.

92. PUT /bowled\_over/\_doc/1?pipeline=conditional\_pipeline

93. {

94. "first\_name": "Bob",

95. "last\_name": "Smith",

96. "ip": "209.191.139.200",

97. "avg": 200

98. }

99.

100. //will evaluate to geoip pipeline as avg is >= 200

101. GET /bowled\_over/\_doc/1

102.

Storing data with varying structures using flatten

- [Instructor] This section demonstrates the flattened data type. This data type is best used for documents that have a large number of unknown fields. To illustrate when this feature is useful, imagine our social network bold over starts distributing an app to employees that allows 'em to send small notes to each other. These notes are just defined as documents with arbitrary keys and values. Let's imagine some notes that users may send back and forth to each other. We've created these three documents over here. They're very simple. One is from Ted, reminding himself to fix the pin setter. A second is a document from Ted to Mindy saying "The pin setter can't be fixed so we need to order a new one." Notice that these two messages already have different fields. This one has a from, a date and a message. While the second message has a to and an order. Let's insert one more document. This is a document of Mindy reminding herself. she has to find new investors because bold over social network has driven their revenues high and there's more bowlers, causing the pin setters to break. So this design looks like it's working great. Let's take a look at the mapping inside elastic. Now, as we could see on our right every field that's ever been inserted, the from, the message, the order, the reminder, the to and the why have created mappings. This design won't work over the long run because each index has a maximum number of mappings. And even if you could define many of them, they're not very useful in this format and the index would be very bloated. So let's look at a better design using flattening. What we're going to do here is create a new mapping. We're going to set the dynamic attribute on the top level object to strict. This won't allow people to create arbitrary fields in the top level object. And we're going to pull out some of the critical things from our note systems, a date, a from, and a to. We're going to assume all messages have these three things. Now the question is what to do with all the other fields, but we could use is the flatten type. And as you can see on line 56, we've created an element called unstructured which type is flattened. What this will allow us to do is all the fields that are new have to be put in this object but they won't ever have any schema. Let's give it a try. So here's the message 3 that Mindy had sent to herself. We've just restructured it slightly where all of the extra fields are now in this unstructured section, and we'll insert it. Let's take a look at the mapping that was generated from this insert. As we can see, these fields, reminder and why aren't present in our schema on the right here. This is because they have been flattened. Let's do some searches of the flattened data type. On line 78 we've created a simple search and we're using the term, search, and we're looking for values that are said find new investors. So here, as you can see, we've been able to match on the value. Very good. What you could also do is you could pick specific fields in the unstructured and match only on those. In this case, unstructured remainder just by using dot notation, that found the same document. Now, this approach obviously has some limits. The benefit of using the flattened type is it stops the mapping explosions. However, the downside is all the data is really internally sorted into single field, and thus it's one type. So if your data is a mix of numbers and strings and dates, it will only sort in a lexigraphical way. Still, it's better than trying to maintain the metadata for every single field inserted in a very dynamic index like this.

1. #This section demonstrates the flattened data type, this type is best used for documents that have a large number of unknown fields

2.

3. # to illustrate when this feature is useful image our social network boled over starts distributing an app to employees that

4. # allows them to send notes to each other. The notes are defined as documents with arbitrary fields and values.

5.

6. #Lets create a few notes that users will send to each other

7.

8. DELETE bowled\_over\_notes

9. PUT /bowled\_over\_notes/\_doc/1

10. {

11. "from": "ted",

12. "date" : "2020-01-01",

13. "message" : "Dont forget to fix the pinsetter on lane 2"

14. }

15.

16. PUT /bowled\_over\_notes/\_doc/2

17. {

18. "from": "ted",

19. "to": "mindy",

20. "date" : "2020-01-02",

21. "order" : "1 new pinsetters"

22. }

23.

24. PUT /bowled\_over\_notes/\_doc/3

25. {

26. "from": "mindy",

27. "to": "mindy",

28. "date" : "2020-01-03",

29. "reminder" : "find new investors",

30. "why": "operational costs higher due to overwhelming successed of bowled\_over social network. equipment mean tme to failuer is higher"

31. }

32.

33. #lets take a look at the mapping discuss mapping explosion here

34.

35. GET /bowled\_over\_notes/\_mapping

36.

37.

38.

39.

40. DELETE bowled\_over\_notes

41.

42. # decribe moving a few fields to defined high level objects nad introducing the unstrutored flattened field

43. PUT /bowled\_over\_notes

44. {

45. "mappings" : {

46. "dynamic": "strict",

47. "properties" : {

48. "date" : {

49. "type" : "date"

50. },

51. "from" : {

52. "type" : "text",

53. "fields" : {

54. "keyword" : {

55. "type" : "keyword",

56. "ignore\_above" : 256

57. }

58. }

59. },

60. "to" : {

61. "type" : "text",

62. "fields" : {

63. "keyword" : {

64. "type" : "keyword",

65. "ignore\_above" : 256

66. }

67. }

68. },

69. "unstructured": {

70. "type": "flattened"

71. }

72. }

73. }

74. }

75.

76.

77.

78. //benefits of flattened data: prevents unstructured data explosion. cons: limited to 1 type (if a mixture of numbers, letters, etc, and say you want to sort, will be sorted lexicograpically only)

79. PUT /bowled\_over\_notes/\_doc/3

80. {

81. "from": "mindy",

82. "to": "mindy",

83. "date" : "2020-01-03",

84. "unstructured" : {

85. "reminder" : "find new investors",

86. "why": "operational costs higher due to overwhelming successed of bowled\_over social network. equipment mean tme to failuer is higher"

87. }

88. }

89.

90.

91. GET /bowled\_over\_notes/\_mapping

92.

93.

94. GET bowled\_over\_notes/\_search

95. {

96. "query": {

97. "term": {"unstructured": "find new investors"}

98. }

99. }

100.

101. GET bowled\_over\_notes/\_search

102. {

103. "query": {

104. "term": {"unstructured.reminder": "find new investors"}

105. }

106. }

107.

**Advanced Queries**

Filtering, matching and scoring explained

**Filtering, matching, and scoring explained**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] In this section, we are going to cover filtering, matching, and scoring. To do this, we're using our dataset which is a sample of user contributed data which we've called the bowled\_over\_artist\_guild. Effectively, it's like almost a blog system where users could create articles with a title, and a passage, which is a text like a poem. We have two examples of this in our pane. We've already inserted this data. You notice the passage is a large body of text. The author title and data are marked smaller discrete values. We've used the mapping endpoint, so we can see what it looks like. What elastic's doing here is for each string field that's considering it, a text, but also adding a keyword field assuming the data is below a certain size that's what the signal ignore above does. So let's do a basic search. So basic search provides it \_search endpoint. And the query that's very simple is the match all query. Now notice this query has returned both results. Obviously we're trying to match all of them, but I think to start noticing for this section is results have a score. Match inquiries always have a score, and what that means is some documents match more than others, and this relates to how elastic internally stores documents, and how it scores words that are more frequent than others. Let's look at another example. In this example, starting on line 36, we're trying another search, and this time we're trying to use a term query on the author field. Notice this didn't work, and that's because the author field is a text field, so this term can't be found. Text fields are turned into inverted indexes, and finding individual terms is difficult. However, in our schema, all of the fields are also available as a keyword. So in this case, we've mentioned the author keyword column, and looked for the same turn. Indeed, this time we actually find the passage we're looking for. So that's a little quirk of how text and keyword fields work. Let's look at some work complex examples. Starting here, line 59, we're doing a query, and we're starting off with a bool query. Bool means boolean, and it allows us to have multiple sections. A must, a must not, a should, and a should not. It is one more section called the filter. Everything in the filter will be returned from the query, and has no effect on scoring, let's see what we mean. Here, we've done a term query, and we're looking for a specific author. Notice here that the score is zero because filtering ignores scores. It is a true or false match. The thing is either here or it is not. Let's look at a more complex example. In this case, starting on line 73, we're starting with the bool query, and we're using the should condition. In this context, a should is like an or, if either of the clauses below it match will return the result. So here what we've done is we've done one match, and we're looking for the word "I", a very common word inside a passage, we'll find many passages with "I" and then we're matching another passage looking for the word bowl. Notice in this case, we've applied a boost on line 87. What this does is it matches documents higher if they have bowl, than if they would have "I", since "I" is a very common word. So we'll subtract the filter here and see what this does. As you can see, both of our articles are matched because this article has the word "I", and this article has the word "I". However, bowl is matched over here, but this article is scored higher than the other one because we're boosting the word bowling. Now, just because you're using a keyword field doesn't mean you only have to search the entire word. The prefix filter works on keywords, and can match the front half of them. So here we've done a filter query, and we've used the prefix query option, and we're looking for half of the author's name. This still works. Here's an example of what's known as an exist query. This checks to see if the document has a field called author cuz not every document has to have every field. Both of our documents have an author column, so they're both returned. And of course, bool queries can also be nested. In this example, starting on line 167 we have an outer bool query with a should, and an inner bool query with a must. Also notice in this case, we're using the fuzzy operator. Fuzzy means it's matching based on expressions, and in this case we have a star, so words ending in "iel". Again here, notice the score, based on the matching conditions, the score may go up and down. Finally, I just wanted to give an example of using the match all query. In some cases, you want to match all the data, and then filter out specific things. So you could use the match all syntax, and in this case we're using a term query on a field that is a date. This works fine, you could specify the date in multiple ways.

1. DELETE bowled\_over\_artist\_guild

2. PUT /bowled\_over\_artist\_guild/\_doc/1

3. {

4. "Author": "Nathaniel Bowlmorn",

5. "date" : "2020-01-01",

6. "title" : "It's down to the last frame",

7. "passage" : """As I distinct remember,

8. I was bowling in december,

9. and every separate strikeball,

10. left a defiant 10 pin standing on the deck

11. early i wish to bowl tommorow"""

12. }

13.

14. PUT /bowled\_over\_artist\_guild/\_doc/2

15. {

16. "Author": "Whiph Onpin",

17. "date" : "2020-01-02",

18. "title" : "o Captain, team captain",

19. "passage" : """O Captain! team captain! our painful season is done;

20. Our team has weather'd every split, the place we sought is won;

21. The bowling dinner is near, the awards I hear, the people all cheering,

22. """

23. }

24.

25. GET /bowled\_over\_artist\_guild/\_mapping

26.

27. //matching always has a score, meaning more matches will be prioritized and have higher scores

28. GET bowled\_over\_artist\_guild/\_search

29. {

30. "query": {

31. "match\_all": {}

32. }

33. }

34.

35. # term queries work only on "keyword" (if only Author will not work)

36. GET /bowled\_over\_artist\_guild/\_search

37. {

38. "query": {

39. "term": {

40. "Author": {

41. "value": "Nathaniel Bowlmorn"

42. }

43. }

44. }

45. }

46.

47. # notice there is no score here

48. //filtering ignores scores, either its found or not

49. GET /bowled\_over\_artist\_guild/\_search

50. {

51. "query": {

52. "bool": {

53. "filter": [

54. {

55. "term": {

56. "Author.keyword": "Nathaniel Bowlmorn"

57. }

58. }

59. ]

60. }

61. }

62. }

63.

64. # the score for second is lower as only one matches

65. //should = OR, must = AND

66. GET /bowled\_over\_artist\_guild/\_search

67. {

68. "query": {

69. "bool": {

70. "should": [

71. {

72. "match": {

73. "passage": "I"

74. }

75. },

76. {

77. "match": {

78. "passage": {

79. "query": "bowl",

80. "boost": 1.8 //bowl gets more scores as it is boosted

81. }

82. }

83. }

84. ]

85. }

86. }

87. }

88.

89. # If you know the front half of a keyword you can use a

90. # prefix query

91. GET /bowled\_over\_artist\_guild/\_search

92. {

93. "query": {

94. "bool": {

95. "filter": [

96. {

97. "exists": {

98. "field": "Author"

99. }

100. }

101. ]

102. }

103. }

104. }

105.

106. # boolean can be nested

107. GET /bowled\_over\_artist\_guild/\_search

108. {

109. "query": {

110. "bool": {

111. "should": [

112. {

113. "bool": {

114. "must": [

115. {

116. "match": {

117. "title": "last"

118. }

119. }

120. ]

121. }

122. },

123. {

124. "fuzzy": {

125. "Author": {

126. "value": "\*iel"

127. }

128. }

129. }

130. ]

131. }

132. }

133. }

134.

135. GET /bowled\_over\_artist\_guild/\_search

136. {

137. "query": {

138. "bool": {

139. "must": [

140. {

141. "match\_all": {}}

142. ],

143. "filter": [

144. {

145. "term": {

146. "date": "2020-01-02"

147. }

148. }

149. ]

150. }

151. }

152. }

153.

Retrieving large result sets using the Scroll API

- [Instructor] In this section we're going to cover retrieving large result sets using the **point in time or PIT feature**. To understand this feature, let's look at an example case. You could imagine that an Elastic index may have millions or billions of documents and you want to move your way through them. Well, imagine that index was changing as you worked with it. Scrolling through the data could produce unpredictable results. PITs freeze the data and ensure that whatever operation you're doing has a consistent view of the index. To illustrate how PIT works, we've created a simple index with two documents. To freeze the index or create a point in time, we're going to use the post verb. We're going to mention the index name and then we're going to use the PIT endpoint. Here we're specifying a keep a live time of 10 minutes. What that means is if the PIT isn't interacted with within 10 minutes, it will close on its own. Notice this PIT query has returned an ID. This ID needs to be used in subsequent calls to reference the PIT. Here we've created a basic search. Notice we're only trying to return one result and the query we're doing is a match off. We've referenced our PIT query and a new keep alive time and an important thing. We're sorting the results here based on the shard doc. This is an internal Elastic identifier that is guaranteed to be there. You could search on anything as long as you could move through it and the appropriate tiebreakers are in place. Let's run the first query. Notice this query had two hits but since we requested a size of only one, one was returned. Important. Notice there's this sort field here, and the data is one. This is returning the Elastic ID of the data. We need to use this column in our next query. So we've created another PIT query here. All the criteria is the same. We've updated the ID of the PIT and notice there's a search after. Again, since this query returned one we'll put search after one and run the query. In this case, we've returned the second query of the set. So this is how to use PIT. Obviously, this was a very small data set, only two columns but this would work over large column, over large data sets as well. When you're done using the PIT, you could delete it. Simply use the delete verb, reference the PIT endpoint and set the appropriate ID. It's good to delete your PITs as soon as possible because Elastic behind the scenes is merging and rebuilding the internal document structure all the time and PITs force those documents to be open for much longer so they're usable, so it's good to clean up so Elastic could free the associated resources.

1. DELETE bowled\_over\_artist\_guild

2. PUT /bowled\_over\_artist\_guild/\_doc/1

3. {

4. "Author": "Nathaniel Bowlmorn",

5. "date" : "2020-01-01",

6. "title" : "It's down to the last frame",

7. "passage" : """As I distinct remember,

8. I was bowling in december,

9. and every separate strikeball,

10. left a defiant 10 pin standing on the deck

11. early i wish to bowl tommorow"""

12. }

13.

14. PUT /bowled\_over\_artist\_guild/\_doc/2

15. {

16. "Author": "Whiph Onpin",

17. "date" : "2020-01-02",

18. "title" : "o Captain, team captain",

19. "passage" : """O Captain! team captain! our painful season is done;

20. Our team has weather'd every split, the place we sought is won;

21. The bowling dinner is near, the awards I hear, the people all cheering,

22. """

23. }

24.

25. //pit endpoint for 10mins, if idle for >10m then PIT will close on its own

26. POST bowled\_over\_artist\_guild/\_pit?keep\_alive=10m

27. POST \_search

28. {

29. "size": 1,

30. "query": {

31. "match\_all": {}

32. },

33. "pit": {

34. "id": "45XtAwEYYm93bGVkX292ZXJfYXJ0aXN0X2d1aWxkFkNTNnlqOVJlU2pXeFg3dHN6LVhRcHcAFlpjSGFZZFdJUXUtc1NPMkE5alNoV3cAAAAAAAADYPsWTjNXNUF5bjlRMmkxZU9HS1hjUEZrUQABFkNTNnlqOVJlU2pXeFg3dHN6LVhRcHcAAA=="

35. ,"keep\_alive": "10m"

36. },

37. "sort": [

38. {"\_shard\_doc": "desc"}

39. ]

40. }

41.

42. POST \_search

43. {

44. "size": 1,

45. "query": {

46. "match\_all": {}

47. },

48. "pit": {

49. "id": "45XtAwEYYm93bGVkX292ZXJfYXJ0aXN0X2d1aWxkFkNTNnlqOVJlU2pXeFg3dHN6LVhRcHcAFlpjSGFZZFdJUXUtc1NPMkE5alNoV3cAAAAAAAADYPsWTjNXNUF5bjlRMmkxZU9HS1hjUEZrUQABFkNTNnlqOVJlU2pXeFg3dHN6LVhRcHcAAA=="

50. },

51. "search\_after": [

52. 1

53. ],

54. "sort": [

55. {"\_shard\_doc": "desc"}

56. ]

57. }

58.

59. DELETE /\_pit

60. {

61. "id": "45XtAwEYYm93bGVkX292ZXJfYXJ0aXN0X2d1aWxkFkNTNnlqOVJlU2pXeFg3dHN6LVhRcHcAFlpjSGFZZFdJUXUtc1NPMkE5alNoV3cAAAAAAAADYPsWTjNXNUF5bjlRMmkxZU9HS1hjUEZrUQABFkNTNnlqOVJlU2pXeFg3dHN6LVhRcHcAAA=="

62. }

63.

Performing aggregations

**Performing aggregations**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] In this section, we're going to be performing aggregations to summarize data. These type of aggregations include grouping and counting, getting distinct elements, or even generating histograms. The data set we're going to use for this example is from our nested data section. The index is called bowled\_over\_team\_scores and this document models the scores in a bowling league. Starting on line 54, we could see this data. We have a league name and a date, and then an array of scores. Each score has a game, a team name, a bowler, and a score. And in this data set there's actually two teams, each with two bowlers. We included a second document where the same two teams bowl a week later. Let's take a look at our first aggregation query. On line 167, we're issuing a search. We're using the nested feature because we want to look at individual bowlers, so we've reached in and we're only trying to match the data from the bowler Bob on the Strikeforce team and we're generating two aggregations. One is the unique IDs or unique dates that Bob has bowled. And the other, interestingly, is the average date. While we can sum the date, it won't produce any meaningful data. Let's just see what it would look like. Starting here, we can see the value of aggregations. Here's the average date, but more interestingly here's the unique IDs. As we could see, this bowler has actually bowled on two different dates. Let's do some other fun aggregations. Let's say we wanted to give a booby prize to the person who bowled the lowest game. We could figure that out by using the min query on the scores field. Getting used to writing these aggregations can be a little bit tricky, so I have a simple tip. I use the SQL interface and I prototype some of my queries there. So here I'm doing an SQL group in count, where I'm selecting a league name and account, and grouping by the league name. You could take this data and look at in a tabular format and once you have a query you like, you could translate it like so. On the right now, Elastic has written the query DSL query for us so we don't have to figure out how to do this. We're using it here inside a search and we get back the same data. Another neat way to summarize data is using a histogram. Starting on line 259, we build an aggregate and we unroll the scores as we've done in other examples. And then inside the aggregation column, we're using the histogram function and telling it to use the score field. We could set different intervals here and we could see what the results look like. On the right, we could see the intervals are broken by groups of 20 and we could see how many people have bowled a 120, how many games have bowled a 160, how many people bowled a 240. There are a lot of aggregation queries and options in Elasticsearch, and we can't cover them all, but this gives you a good jumpstart to give you some ideas of what you can do.

1. DELETE bowled\_over\_team\_scores

2.

3. PUT bowled\_over\_team\_scores

4. {

5.

6. "mappings" : {

7. "properties" : {

8. "date" : {

9. "type" : "date"

10. },

11. "league\_name" : {

12. "type" : "text",

13. "fields" : {

14. "keyword" : {

15. "type" : "keyword",

16. "ignore\_above" : 256

17. }

18. }

19. },

20. "scores" : {

21. "type": "nested",

22. "properties" : {

23. "bowler" : {

24. "type" : "text",

25. "fields" : {

26. "keyword" : {

27. "type" : "keyword",

28. "ignore\_above" : 256

29. }

30. }

31. },

32. "game" : {

33. "type" : "long"

34. },

35. "score" : {

36. "type" : "long"

37. },

38. "team\_name" : {

39. "type" : "text",

40. "fields" : {

41. "keyword" : {

42. "type" : "keyword",

43. "ignore\_above" : 256

44. }

45. }

46. }

47. }

48. }

49. }

50. }

51. }

52.

53. PUT /bowled\_over\_team\_scores/\_doc/1

54. {

55. "league\_name": "friday night lights",

56. "date" : "2020-01-01",

57. "scores" : [

58. {

59. "game": 1,

60. "team\_name" : "strike force",

61. "bowler": "bob",

62. "score": 203

63. },

64. {

65. "game": 2,

66. "team\_name" : "strike force",

67. "bowler": "bob",

68. "score": 204

69. },

70. {

71. "game": 1,

72. "team\_name" : "strike force",

73. "bowler": "sara",

74. "score": 177

75. },

76. { "game": 2,

77. "team\_name" : "strike force",

78. "bowler": "sara",

79. "score": 176

80. },

81. {

82. "game": 1,

83. "team\_name" : "bowling turkeys",

84. "bowler": "john",

85. "score": 160

86. },

87. {

88. "game": 2,

89. "team\_name" : "bowling turkeys",

90. "bowler": "john",

91. "score": 164

92. },

93. {

94. "game": 1,

95. "team\_name" : "bowling turkeys",

96. "bowler": "ted",

97. "score": 121

98. },

99. {

100. "game": 2,

101. "team\_name" : "bowling turkeys",

102. "bowler": "ted",

103. "score": 241

104. }

105. ]

106. }

107.

108.

109. PUT /bowled\_over\_team\_scores/\_doc/2

110. {

111. "league\_name": "friday night lights",

112. "date" : "2020-01-08",

113. "scores" : [

114. {

115. "game": 1,

116. "team\_name" : "strike force",

117. "bowler": "bob",

118. "score": 233

119. },

120. {

121. "game": 2,

122. "team\_name" : "strike force",

123. "bowler": "bob",

124. "score": 234

125. },

126. {

127. "game": 1,

128. "team\_name" : "strike force",

129. "bowler": "sara",

130. "score": 170

131. },

132. { "game": 2,

133. "team\_name" : "strike force",

134. "bowler": "sara",

135. "score": 199

136. },

137. {

138. "game": 1,

139. "team\_name" : "bowling turkeys",

140. "bowler": "john",

141. "score": 170

142. },

143. {

144. "game": 2,

145. "team\_name" : "bowling turkeys",

146. "bowler": "john",

147. "score": 174

148. },

149. {

150. "game": 1,

151. "team\_name" : "bowling turkeys",

152. "bowler": "ted",

153. "score": 120

154. },

155. {

156. "game": 2,

157. "team\_name" : "bowling turkeys",

158. "bowler": "ted",

159. "score": 200

160. }

161. ]

162. }

163.

164.

165.

166.

167. GET bowled\_over\_team\_scores/\_search

168. {

169. "query": {

170. "nested": {

171. "path": "scores",

172. "query": {

173. "bool": {

174. "must": [

175. { "match": { "scores.team\_name": "strike force" }},

176. { "match": { "scores.bowler": "bob" }}

177. ]

178. }

179. }

180. }

181. }

182. ,

183. "aggs": {

184. "unique\_ids": {

185. "terms": {

186. "field" : "date"

187. }

188. },

189. "the\_aveage" : {

190. "avg": {

191. "field": "date"

192. }

193. }

194. }

195. , "\_source": false

196. }

197.

198. GET bowled\_over\_team\_scores/\_search

199. {

200. "aggs": {

201. "scores": {

202. "nested": {

203. "path": "scores"

204. },

205. "aggs": {

206. "min\_score": {

207. "min": {

208. "field": "scores.score"

209. }

210. }

211. }

212. }

213. }

214. , "\_source": false

215. }

216.

217. //much easier to just do SQL query

218. GET \_sql?format=txt

219. {

220. "query": """

221. SELECT league\_name,count(1) FROM "bowled\_over\_team\_scores" group by league\_name

222. """

223. }

224.

225. GET \_sql/translate

226. {

227. "query": """

228. SELECT league\_name,count(1) FROM "bowled\_over\_team\_scores" group by league\_name

229. """

230. }

231.

232.

233.

234. GET bowled\_over\_team\_scores/\_search

235. {

236. "size": 0,

237. "\_source": false,

238. "track\_total\_hits": -1,

239. "aggregations": {

240. "groupby": {

241. "composite": {

242. "size": 1000,

243. "sources": [

244. {

245. "c5a30fe4": {

246. "terms": {

247. "field": "league\_name.keyword",

248. "missing\_bucket": true,

249. "order": "asc"

250. }

251. }

252. }

253. ]

254. }

255. }

256. }

257. }

258.

259.

260. GET bowled\_over\_team\_scores/\_search

261. {

262. "aggs": {

263. "scores": {

264. "nested": {

265. "path": "scores"

266. },

267. "aggs": {

268. "thename": {

269. "histogram": {

270. "field": "scores.score",

271. "interval": 20

272. }

273. }

274. }

275. }

276. }

277. , "\_source": false

278. }

279.

Indexing templates to define index properties

**Indexing templates to define index properties**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] In this section, we are going to use index templates to define index properties. Throughout the course, we've created a number of indexes to hold data. Usually, we have let the database create the mapping dynamically or we've created a mapping file per index. Index templates allow you to create templates with properties that match index names on creation. That's a mouthful, but as we use it I think it'll make more sense. Let's start by creating an index and looking at the settings and mappings. Here we'll use a data set of user contributed writings and poetry we've been calling the Bowled over Artist Guild. Let's take a look at the settings that were created when we inserted the data. We can do this by using the settings endpoint. Notice here that the number of shards is one. We could also look at the mapping of this index and this is what we've seen many times. The mapping holds the data types and their indexing settings. Now to understand index templates, let's start with a concept called table by day or in elastic sense, index by day. Generally, this pattern is useful when data can be grouped by time. It makes it easy to find data and maybe even delete the data because it all would fit into the same bucket based on the time it was created. To accomplish this, we're going to do inserts and we're going to add a date to the name of the index. Notice here, the index has a date component in the name itself. Let's insert a second document for a second day. We could easily search over these two indexes. Notice we've used a star here which searches over all of the indexes we have. Let's use an index template here to make sure all indexes created have the same settings and schema. Here we've created a template that matches the index pattern, Bowled over Artist Guild star and we've adjusted the number of shards to three. Let's also include some mapping so that every document has the same indexing. Awesome. Now we could delete the data and insert it again and we'll see if the settings take effect. Notice the delete endpoint can take a wild card expression too, just like the get or put. Obviously, you want to be very careful when deleting with a wild card like a star. All right. Let's insert the data again. Now, let's review the settings and see if they match what we would expect. As we can see, the number of shards is three. So index templates can be used a number of ways. They could define settings for the entire application or a specific index. They could also be ordered by number, controlling which rules get applied if multiple templates match the expression.

1. DELETE bowled\_over\_artist\_guild

2. PUT /bowled\_over\_artist\_guild/\_doc/1

3. {

4. "Author": "Nathaniel Bowlmorn",

5. "date" : "2020-01-01",

6. "title" : "It's down to the last frame",

7. "passage" : """As I distinct remember,

8. I was bowling in december,

9. and every separate strikeball,

10. left a defiant 10 pin standing on the deck

11. early i wish to bowl tommorow"""

12. }

13.

14.

15. GET bowled\_over\_artist\_guild/\_settings

16. GET bowled\_over\_artist\_guild/\_mapping

17.

18. //the index has a date component in the name itself

19. PUT /bowled\_over\_artist\_guild\_2020-01-01/\_doc/1

20. {

21. "Author": "Nathaniel Bowlmorn",

22. "date" : "2020-01-01",

23. "title" : "It's down to the last frame",

24. "passage" : """As I distinct remember,

25. I was bowling in december,

26. and every separate strikeball,

27. left a defiant 10 pin standing on the deck

28. early i wish to bowl tommorow"""

29. }

30.

31. PUT /bowled\_over\_artist\_guild\_2020-01-02/\_doc/2

32. {

33. "Author": "Whiph Onpin",

34. "date" : "2020-01-02",

35. "title" : "o Captain, team captain",

36. "passage" : """O Captain! team captain! our painful season is done;

37. Our team has weather'd every split, the place we sought is won;

38. The bowling dinner is near, the awards I hear, the people all cheering,

39. """

40. }

41.

42. GET bowled\_over\_artist\_guild\_\*/\_search

43. {

44. "query": {

45. "match\_all": {}

46. }

47. }

48.

49. //create templates for indices

50. PUT \_template/bowled\_over\_artist\_guild

51. {

52. "index\_patterns": "bowled\_over\_artist\_guild\_\*",

53. "settings": {

54. "number\_of\_shards": 3

55. },

56. "mappings" : {

57. "properties" : {

58. "Author" : {

59. "type" : "text",

60. "fields" : {

61. "keyword" : {

62. "type" : "keyword",

63. "ignore\_above" : 256

64. }

65. }

66. },

67. "date" : {

68. "type" : "date"

69. },

70. "passage" : {

71. "type" : "text",

72. "fields" : {

73. "keyword" : {

74. "type" : "keyword",

75. "ignore\_above" : 256

76. }

77. }

78. },

79. "title" : {

80. "type" : "text",

81. "fields" : {

82. "keyword" : {

83. "type" : "keyword",

84. "ignore\_above" : 256

85. }

86. }

87. }

88. }

89. }

90. }

91.

92. //be VERY CAREFUL while deleting indices with wildcare (\*)

93. DELETE bowled\_over\_artist\_guild\*

94.

95. //shards count are now 3

96. GET bowled\_over\_artist\_guild\_2020-01-01/\_settings

97.

Implementing expiring data with lifecyle management

**Implementing expiring data with lifecycle management**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] In this section, we're going to demonstrate how to do expiring, or what is sometimes called TTL-based data, using internal lifecycle management**. Internal lifecycle management, or ILM, is a feature in Elastic that automatically changes the characteristics of indexes based on criteria.**A common use case for ILM is when implementing a hot, warm, cold architecture. Data moves between hardware classes as it's searched less often. These type of operations can be done manually by scripting or Chron or some kind of scheduling tool, but Elasticsearch has its own inbuilt ILM. Let's take a look at how ILM works. Here we've used the put verb to create a policy. We've called it fast TTL policy. The policy has two phases, hot and delete. As you can see, the hot data rolls over at a one minute rate and the delete data deletes things. This is not the kind of setup you'd want to use in production but here we want to expire the indexes so we could see this work quickly. The next phase of ILM is that it doesn't work well with standard indexes. Usually it's used as what's called index streams. These look like an index, in that they have an index template and the index template matches a pattern, except an important note is, as you can see on line 57, is the data stream is specified. So here, any data written to an index named fast TTL star won't be written to a standard index. It'll be written to a data stream. And note, we've attached the index to a lifecycle, the same policy we've mentioned above, fast TTL policy. This next setting here causes the Elastic to examine the ILM policy every 15 seconds. You might want to raise or lower this based on your requirements, but it makes sense to keep it at the default unless you know what you're doing. So now the interesting part of ILM. Documents written to ILM don't post to this exact index. They post to a data stream, which behind the scenes is a series of indexes. Let's take a look. One inserted document, and then another. Notice here, on line three you could actually see that the index returned is a hidden index. You could see this 'cause it starts with a dot. What does this mean? Here, we're using the indexes, the cat endpoint to show the indexes. And as you can see, there's actually an index here with two documents inside it. But we could search this index by using the friendly name we defined above. As you can see, the data is found. If we wait a while and retry this query, our policy should automatically delete this data. So we'll go out and get a coffee or something and come back and see what happens. Nope, still there. We could see a new index has rolled over. Oh, look at that. The data's now gone. And the indexes themselves have rolled over. So that's how internal lifecycle management works. Behind the scenes, Elastic manages indexes and the user sees them as data streams.

1. PUT \_ilm/policy/fast-ttl-policy

2. {

3. "policy": {

4. "phases": {

5. "hot": {

6. "actions": {

7. "rollover": {

8. "max\_age": "1m"

9.

10. },

11. "set\_priority": {

12. "priority": 100

13. }

14. },

15. "min\_age": "0ms"

16. },

17. "delete": {

18. "actions": {

19. "delete": {}

20. }

21. }

22. }

23. }

24. }

25.

26. PUT \_index\_template/fast-ttl-template

27. {

28. "template": {

29. "settings": {

30. "index": {

31. "lifecycle": {

32. "name": "fast-ttl-policy"

33. }

34. }

35. },

36. "mappings": {

37. "dynamic": true,

38. "numeric\_detection": false,

39. "date\_detection": true,

40. "dynamic\_date\_formats": [

41. "strict\_date\_optional\_time",

42. "yyyy/MM/dd HH:mm:ss Z||yyyy/MM/dd Z"

43. ],

44. "\_source": {

45. "enabled": true,

46. "includes": [],

47. "excludes": []

48. },

49. "\_routing": {

50. "required": false

51. }

52. }

53. },

54. "index\_patterns": [

55. "fast-ttl-\*"

56. ],

57. "data\_stream": {}

58. }

59.

60.

61. PUT \_cluster/settings

62. {

63. "transient": {

64. "indices.lifecycle.poll\_interval": "15s"

65. }

66. }

67.

68.

69. POST fast-ttl-ilm/\_doc

70. {

71. "@timestamp": "2021-01-11",

72. "x": "something"

73. }

74.

75. POST fast-ttl-ilm/\_doc

76. {

77. "@timestamp": "2021-01-11",

78. "x": "something else"

79. }

80.

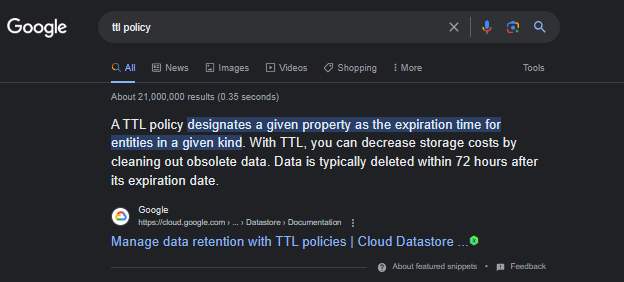
81. //indices' name start with a dot: hidden indices according to TTL policy(?)

82. GET \_cat/indices/fast-ttl-ilm?v

83.

84. GET fast-ttl-ilm/\_search

85.



Working with index aliases

**Working with index aliases**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] This section is going to show how to use index aliases to simplify data management. To understand this, let's think of a few examples. In some other section, we've approached the concept of index by day, where your data is actually divided across multiple indexes. In this case, we've created an index called Bold Over Artist Guild, and it's actually divided by day, where some documents exist in one day and some documents exist in another day. This is a common theme to deal with time series data and eventually you may have to roll up or modify these indexes and aliases are very helpful with this. Let me explain. In Elasticsearch, we could use wild cards or list multiple indexes to do a query. Here we're doing a match query that searches over two indexes. As expected, this is returning two results, as you could see on line 13. Now, these indexes over time, we may decide to change them. Like we may decide we want to add another field or possibly change how we're indexing in existing fields. Elasticsearch has an API for this. It's called the re-index API. On line 34, you could see an example. We're making a post to re-index and we're specifying a source index, which in this case, we're using the data from the first of the month, shown on line 37, and we're creating a new index where we just appended A to this name. You would normally do this if you change the index definition or the index template and you want to re-index. So we've done that here. We've re-indexed our data, but now there's a problem. If we search for the Bold Over Artist Guild, instead of finding two documents, we'll find three. And that's because there's actually three indexes now behind the scenes. We have the same document choice in Bold Over Artist Guild and Bold Over Artist Guild A. So we could use index aliases to fix this problem. How would we do that? Elastic has an endpoint called aliases, and it supports one or more actions that could be specified in an array. So the first thing we're going to do here, is we're going to remove the old index and we're going to edit index alias. Or if you feel safe, we could put them in the reverse order**. It doesn't really matter because the entire operation happens completely or not at all**. So let's execute this action. It's acknowledged here. And now let's go run our search again. Now we have the correct number of results, which is two. And as you could see, one datum is coming from the second index, and this is our updated index, and the original index is now gone. So there's many ways you could use index templates. You could do perhaps index by day, and then make an alias which includes day one, day two, day three. Index aliases could also use wild cards inside them or other patterns.

1. DELETE bowled\_over\_artist\_guild\_\*

2. PUT /bowled\_over\_artist\_guild\_2020-01-01/\_doc/1

3. {

4. "Author": "Nathaniel Bowlmorn",

5. "date" : "2020-01-01",

6. "title" : "It's down to the last frame",

7. "passage" : """As I distinct remember,

8. I was bowling in december,

9. and every separate strikeball,

10. left a defiant 10 pin standing on the deck

11. early i wish to bowl tommorow"""

12. }

13.

14. PUT /bowled\_over\_artist\_guild\_2020-01-02/\_doc/2

15. {

16. "Author": "Whiph Onpin",

17. "date" : "2020-01-02",

18. "title" : "o Captain, team captain",

19. "passage" : """O Captain! team captain! our painful season is done;

20. Our team has weather'd every split, the place we sought is won;

21. The bowling dinner is near, the awards I hear, the people all cheering,

22. """

23. }

24.

25. //list 2 indices simultaneously

26. GET bowled\_over\_artist\_guild\_2020-01-01,bowled\_over\_artist\_guild\_2020-01-02/\_search

27. {

28. "query": {

29. "match\_all": {}

30. }

31. }

32.

33.

34. POST \_reindex

35. {

36. "source": {

37. "index": "bowled\_over\_artist\_guild\_2020-01-01"

38. },

39. "dest": {

40. "index": "bowled\_over\_artist\_guild\_2020-01-01\_a"

41. }

42. }

43.

44. //returns 3 indices instead of 2 (duplicates for 2020-01-01 and 2020-01-01\_a)

45. GET bowled\_over\_artist\_guild\_\*/\_search

46. {

47. "query": {

48. "match\_all": {}

49. }

50. }

51.

52. //after running POST \_aliases, original indice is now gone

53. POST \_aliases

54. {

55. "actions": [

56.

57. {

58. "add": {

59. "index": "bowled\_over\_artist\_guild\_2020-01-01\_a",

60. "alias": "bowled\_over\_artist\_guild\_2020-01-01"

61. }

62. },

63. {

64. "remove\_index": {

65.

66. "index": "bowled\_over\_artist\_guild\_2020-01-01"

67. }

68. }

69. ]

70. }

71.

**Modifying Data**

Updating documents using scripting

**Updating documents using scripting**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] In this section we're going to update documents using scripting. This technique is useful in a variety of cases, one being if you only want to update part of a document or two, if you want to update a document based on its current values. Let's get started. Throughout this course we have envisioned a social network called "Bowled Over." A social network for bowlers. In this case we'll add a new index that records what a bowler does in their league play. In this case, the league has, starting on line 3 you could see we've created a new index called bowled over games, and we have a date and a league name, a bowler name, and a score. We're going to insert this document and let Elastic create the index mapping automatically. As you can see on the right, this was inserted. Let's look at the mapping by using the bowled over games mapping. You can note on the right here on line 27 that the scores type is long. Elastic search in this case doesn't care whether scores is a single value or an array value. It just needs to know the field is long. Let's see what the data looks like we inserted. Great, on the right we could see there's scores as an array and there's only one score. Let's imagine we finished our second game and bowled a measly 115. Rather than inserting the whole document we could use the update syntax and scripting to add a score to this array. Starting on line 15, you could see we're using a post and we're using the underscore update endpoint. Inside the body of the post, we have a script. The script has a few components. First, it has a source. This is actual code and the code itself is defined by the language which in this case is the painless language. So this source code says inside the context, that's the request. And inside the source document being inserted take the scores field and add an element to it. Which is the elements? The element is defined in the params score which is almost like another document inside this script. So we're trying to add a 151 to this array. Let's see if it works. We use the get endpoint to view the results and in this case, we could see on the right on line 8, 151 has been added to the current document. Now, after bowling a 151 we might realize it's time to get a soda and take a little break, see if we can do better in our third and final game. So what we'll do is we'll introduce a concept for storing a bar tab, simply an item count. So the script below here is different. Rather than a pending to an existing field as we did before, we're going to create a new field. So as you can see we're starting a post on line 28 to the same document. And again, we're using a script, in this case on line 31 you could see we use the multi-line syntax. So we use three quotes to specify a script that's multiple lines. And what we're doing here is instead of appending to an array, we are actually updating a field. And in this case, we have an if statement to cover what would happen if it doesn't exist. So our main goal here is to put this parameter in the bar tab if the field doesn't exist or add it to the bar tab if it doesn't exist. That's covered by both the cases here on line 33 and line 35. Let's run this example. Great. We could read back the results and see if the bar tab was added. Yes, in this case we could see we still have the two scores from the bowler and the new bar tab field has been introduced. Now, obviously this isn't the most efficient way to model a system of keeping a bar tab. Usually we want to track the items but let's just say we are going with this setup here. What would happen if we accidentally sent the same update twice or even three times? Well, now we have a problem because the bar tab is set three times and we only ordered a single soda. To deal with this type of problem Elasticsearch offers optimistic concurrency controls meaning it's a conditional update. So what we'll do here is we will append this field which says only execute this script if the document is at sequence number 6. Let's see if this works. Ah, we've got an exception here, and that's because the sequence isn't at 6 it's currently at 4. So the only way we could update this document is if we specify the correct sequence number. So let's set the sequence number to what we expect, 4. And then let's run this insert again. This time it was accepted and the sequence number is now 5 and the primary term is 1. Let's see what the results look like. Yes, as you can see, the scores are updated and the bar tab has been updated appropriately.

1. DELETE bowled\_over\_games

2.

3. PUT /bowled\_over\_games/\_doc/1

4. {

5. "date": "2021-02-04",

6. "league": "Friday Night Rockstars",

7. "bowler\_name": "Ron California",

8. "scores":[211]

9. }

10.

11. GET /bowled\_over\_games/\_mapping

12.

13. GET /bowled\_over\_games/\_source/1

14.

15. POST /bowled\_over\_games/\_update/1

16. {

17. "script": {

18. "source": "ctx.\_source.scores.add(params.score)",

19. "lang": "painless",

20. //add element to scores on \_source

21. "params": {

22. "score": 151

23. }

24. }

25. }

26.

27. //score of 151 added to the 'scores' array

28. GET /bowled\_over\_games/\_source/1

29.

30. POST /bowled\_over\_games/\_update/1

31. {

32. "script" : {

33. "source": """

34. if (ctx.\_source.bartab == null) {

35. ctx.\_source.bartab = params.cost

36. } else {

37. ctx.\_source.bartab += params.cost

38. }""",

39. "lang": "painless",

40. "params": {

41. "cost": 3.99

42. }

43. }

44. }

45.

46. //bartab field is introduced

47. GET bowled\_over\_games/\_doc/1

48.

49. POST /bowled\_over\_games/\_update/1

50. {

51. "script" : {

52. "source": "if (ctx.\_source.bartab == null) { ctx.\_source.bartab= params.cost } else {ctx.\_source.bartab += params.cost }",

53. "lang": "painless",

54. "params": {

55. "cost": 3.99

56. }

57. }

58. }

59.

60. GET bowled\_over\_games/\_doc/1

61.

62. //multiple if conditions

63. POST /bowled\_over\_games/\_update/1

64. {

65. "script" : {

66. "source": """

67. if (ctx.\_source.bartab == null) {

68. ctx.\_source.bartab = params.cost

69.

70. } else {

71. ctx.\_source.bartab += params.cost

72. }

73. """,

74. "lang": "painless",

75. "params": {

76. "cost": 3.99

77. }

78. },

79. "if\_seq\_no" : 5,

80. "if\_primary\_term": 1

81. }

82.

83. POST /bowled\_over\_games/\_update/1

84. {

85. "script" : {

86. "source": "if (ctx.\_source.bartab == null) { ctx.\_source.bartab= params.cost } else {ctx.\_source.bartab += params.cost }",

87. "lang": "painless",

88. "params": {

89. "cost": 3.99

90. }

91. },

92. "if\_seq\_no" : 6,

93. "if\_primary\_term": 1

94. }

95.

Saving scripts for reuse

**Saving scripts for reuse**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] In this section, we're going to save and reuse a script. There's a few good reasons to do this. The first is that scripts have application logic and it's better if those are specified in only one place. And the second is compiling the scripts is performance intensive. Elastic search puts a cap on how many scripts you could compile in a given time period. This is the same document and script that was used earlier in the chapter, so I'm not going to explain it again fully but let's just go through it quickly. We have a document that represents a bowler in a league and we have another document which describes how we want to append the bar tab field to it. Let's run them both. In this case, we could see that our document was created and we used the scripting logic to append a bar tab field or create it if it doesn't exist. Now, as I mentioned we don't want to recreate this script in every single request. It could be error prone, so now we're going to save that script to use it later. Starting on line 30, you could see how this is done. On line 30, we use a post to the scripts endpoint and we name the script update bar tab. We took the same scripting above. We simply created the script elements, specified the language and specified the source. Again, this is just a script that appends the fields if they exist or create it if it does not. We could save this, and now the script is saved in the elastic search server and is reusable. If we want to check to see if it's there we could use the get verb and look for the script. Great. As we can see, the script is there, so let's move on. In our example above, we mentioned we did an update and we specified parameters and we inlined the script. In this case we'll just reference the script saved on the server. So if you look at this example down here starting on line 147, we simply use the same script element but instead of using the source, we use the ID. This ID matches the name above of the script we inserted, and we supplied the parameters as before. Now we could run this example and see what happens. As you can see we've updated the bar tab using the script that was saved on the server.

1. DELETE bowled\_over\_games

2. # create a simple document

3. PUT /bowled\_over\_games/\_doc/1

4. {

5. "date": "2021-02-04",

6. "league": "Friday Night Rockstars",

7. "bowler\_name": "Ron California",

8. "scores":[211]

9. }

10.

11. //scripting logic to append/increment bartab

12. POST /bowled\_over\_games/\_update/1

13. {

14. "script" : {

15. "source": """

16. if (ctx.\_source.bartab == null) {

17. ctx.\_source.bartab = params.cost

18. } else {

19. ctx.\_source.bartab += params.cost

20. }""",

21. "lang": "painless",

22. "params": {

23. "cost": 3.99

24. }

25. }

26. }

27.

28. GET /bowled\_over\_games/\_source/1

29.

30. //save the script using \_scripts endpoint

31. POST \_scripts/update-bartab

32. {

33. "script": {

34. "lang": "painless",

35. "source": """

36. if (ctx.\_source.bartab == null) {

37. ctx.\_source.bartab = params.cost

38. } else {

39. ctx.\_source.bartab += params.cost

40. }

41. """

42. }

43. }

44.

45. //check the saved script(s)

46. GET \_scripts/update-bartab

47.

48. //reusing the script by simply stipulating the script ID

49. POST /bowled\_over\_games/\_update/1

50. {

51. "script" : {

52. "id": "update-bartab",

53. "params": {

54. "cost": 3.99

55. }

56. }

57. }

58.

Updating by query

**Updating by query**

Selecting transcript lines in this section will navigate to timestamp in the video

- Update by Query is a feature in Elasticsearch that allows us to select one or more documents using a filtered DSL or a Lucene query syntax and modify them using a variety of methods. This is of course useful when you know the IDs of a document but there's many of them. Or, you don't know the IDs and you want to change numerous messages. So to get started, we're going to recreate a document which we've used previously in this section. It stores bowlers, their scores and a bar tab, which shows how much money they've spent at the bar and have yet to pay back. So, interestingly enough, let's say this feature goes over very well and many people bowl but forget to pay their bar tab. Well, this creates a problem but let's turn this into a solution. Someone in marketing came up with a clever promotion. Why don't we take everybody who owes a bar tab and tell them if they bowl a game higher than 200 in the month of February, we'll pay their bar tab for free. Interesting concept. How do we locate those people? As you can see on line 13, we've composed a search query. We're using a bool query at the top level on line 17 and then we're using multiple must clauses. We're specifying a range, people who have bowled over 200. A range of time, dates between February 1st and February 29th and a range. We only want to forgive bar tabs that are less than $10. Let's see if this matches anybody. Great. We could see on the right, that this one bowler has been matched. So how do we go about forgiving how much money they owe? Here we've composed a script query on line 48. Notice the source of the script. We're going to set their bar tab equal to zero, as we've done on line 53. And we're also going to leave a message that says, "Hey we've paid your bar tab." Now in this case, we're doing a conditional update query. So we use the post verb, the name of the index, bowled\_over\_games

1. PUT /bowled\_over\_games/\_doc/1

2. {

3. "date": "2021-02-04",

4. "league": "Friday Night Rockstars",

5. "bowler\_name": "Ron California",

6. "scores":[211],

7. "bartab": 9.25

8. }

9.

10. //search for people who have bartab over 10 dollars that has a score >= 200 in February 2021

11. GET bowled\_over\_games/\_search

12. {

13.

14. "query": {

15. "bool": {

16. "must": [

17. {

18. "range": {

19. "scores": {

20. "gte": 200

21. }

22. }

23. },

24. {

25. "range": {

26. "date": {

27. "gte": "2021-02-01",

28. "lte": "2021-02-29"

29. }

30. }

31. },

32. {

33. "range": {

34. "bartab": {

35. "lte": 10

36. }

37. }

38. }

39. ]

40. }

41. }

42.

43. }

44.

45. //conditional update query

46. POST bowled\_over\_games/\_update\_by\_query

47. {

48. "script": {

49. "source": """

50. ctx.\_source.message = 'we payed your tab of ' + ctx.\_source.bartab;

51. ctx.\_source.bartab=0;""",

52. "lang": "painless"

53. },

54. "query": {

55. "bool": {

56. "must": [

57. {

58. "range": {

59. "scores": {

60. "gte": 200

61. }

62. }

63. },

64. {

65. "range": {

66. "date": {

67. "gte": "2021-02-01",

68. "lte": "2021-02-29"

69. }

70. }

71. },

72. {

73. "range": {

74. "bartab": {

75. "lte": 10

76. }

77. }

78. }

79. ]

80. }

81. }

82. }

83.

84. GET /bowled\_over\_games/\_source/1

85.

Using document versioning

**Using document versioning**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] In this section, we are going to demonstrate how to use versioning support to make conditional updates to documents. Before we show how to do it, let's take a moment to come up with a case where we might need this feature. Transactions and data stores take many forms. Many databases are categorized as online transaction processing, or OLTP, and have strict transaction locking and serialization features, often grouped into a category of things called ACID transactions. Taking this to another level, sometimes transactions are needed between two systems or distributed transactions. My head hurts a little bit. Let's get back to Elastic. **Elastic has what is called optimistic concurrency control using versioning. This is exposed as a compare and swap. Apply this operation only if the document is at the expected version.** This approach has some upsides and some downsides. If a system holds many documents, each of them rarely being updated, the chance of a conflict and retry is very low. However, if there are many users attempting to update a document at roughly the same time, retries cause delay, and at a certain point, if something is retrying too much, the overall availability of the system is low. Applications usually have to give up retrying, and ultimately, that looks like a failure. One more important thing to note. Generally, even systems that store and process transactions, such as the movement of monetary funds, typically are designed using logs or ledgers and not fields or tables where moving money involves large multi-table or multi-row transactions. To understand how Elastic optimistic concurrency control works, let's take a use case. Let's say our bowling season is over, and it's time to plan an award dinner. Each user is given a link, a one-time link, and asked to vote on a local restaurant over the next few days. It's expected that a few people will vote, and very few will vote at the same time. So Elastic's optimistic concurrency control seems like a good fit. To do this, we are going to create an index called votes, and the administrator will create a first document. The document simply uses keys as the names of restaurants, Hamburger Heaven, Pizzas are Us, and Holy Wings, and sets the vote to zero. Now it's time for people to vote. We definitely want to make sure that the same transaction doesn't cause a double vote. So this is where Elastic's versioning comes in. If you notice on line 12, we're using the post verb. The first part is the votes index, and then we're using the update endpoint. We're specifying the document id, but notice the query parameters, we're using two. If sequence number equals zero, and if primary term equals one. Let's look on the right. Currently, we could see that the sequence number is zero, and the primary term is one. So we could expect this update to succeed. Let's try it. This did. Let's take a look at what the data looks like. The data has correctly registered one vote for Hamburger Heaven. Now the version is two, and the sequence number is one. Version is a field that we don't touch directly. We always interact through the primary term in the sequence number. Let's try to accidentally make the same vote again. As we can see, this was rejected. Elastic has given us a reason on line 13. There is a version conflict. Sequence number and primary term don't match. Let's make another vote for Hamburger Heaven using the right values. This post succeeded, and we could see the document value is expected. Using this, we could also do scripted updates or manipulate the field using counters or the painless query language.

1. DELETE votes

2. PUT votes/\_doc/1

3. {

4. "hamburger\_heaven": 0,

5. "pizza\_r\_us": 0,

6. "unholy\_wings": 0

7. }

8.

9. GET votes/\_doc/1

10.

11. //will reject if seq\_no and primary\_term don't match (version conflict)

12. POST votes/\_update/1?if\_seq\_no=0&if\_primary\_term=1

13. {

14. "doc": {

15. "hamburger\_heaven": 1

16. }

17. }

18.

19.

20. GET votes/\_doc/1

21.

22.

23. POST votes/\_update/1?if\_seq\_no=1&if\_primary\_term=1

24. {

25. "doc": {

26. "hamburger\_heaven": 2

27. }

28. }

29.

Managing long-running operations using Task API

**Managing long-running operations using Task API**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] This section is going to explain how to use the task feature in Elastic, which manages long-running queries. If you remember from our previous section, we introduced a bowled over games index and we added a field to that index, which is shown on line seven on the left, which is the bar tab. And then if you remember, we created a query that would locate bar tabs based on criteria and forgive them. We won't go through this whole query but suffice it to say it searches our data set looking for bar tabs and it replaces them. Let's run this query. Notice we're using wait for completion equals true on line 11. On the right, you could see the response we got. We have a certain number of records updated and our request completed successfully. But if you could imagine our Elastic search database may have billions of documents and brontobytes of data, so not all queries could be run in real time. For example, imagine this request matched millions of results. We probably wouldn't be able to change them all instantaneously so we could simulate how that would work. In a case like that, we would set wait for completion equal to false. So let's run this same query again. Now, in this case, we've been given back the task id which is an ID used to manage the task. I've copied this task ID here and let's see how we would use it. Well firstly, we could list all the tasks. On the right, you'll see here there's some tasks that are actually being made by Elasticsearch itself so they're not all relevant to things that users are doing. We could seek out a specific task. So let's look at this task. We could see this task on the right is completed and we have the same information, how many rows were updated, and that's basically it. Because this task was already complete, there's nothing we could do with it, but if it wasn't complete we could cancel it using this particular endpoint. So we could say, tasks cancel. We won't click that because it won't work. But an important part is when we create a task like this we do have to delete it ourselves and the syntax for that is here. So we would use the delete verb and say dot tasks slash task and give the name. As you can see on the right this results in the task being deleted. So this is how we create background or asynchronous tasks and how we manage them. Beyond that there's some other things you could look forward to. For example, if a task is very large you could actually split the task into parts and paralyze it. We'll get into that in a later section when we talk about paging.

1. PUT /bowled\_over\_games/\_doc/1

2. {

3. "date": "2021-02-04",

4. "league": "Friday Night Rockstars",

5. "bowler\_name": "Ron California",

6. "scores":[211],

7. "bartab": 9.25

8. }

9.

10. //wait for completion = false meaning that This setting specifies whether or not the request should return immediately or wait for the operation to complete before returning

11. POST bowled\_over\_games/\_update\_by\_query?wait\_for\_completion=false

12. {

13. "script": {

14. "source": """

15. ctx.\_source.message = 'we payed your tab of ' + ctx.\_source.bartab;

16. ctx.\_source.bartab=0;""",

17. "lang": "painless"

18. },

19. "query": {

20. "bool": {

21. "must": [

22. {

23. "range": {

24. "scores": {

25. "gte": 200

26. }

27. }

28. },

29. {

30. "range": {

31. "date": {

32. "gte": "2021-02-01",

33. "lte": "2021-02-29"

34. }

35. }

36. },

37. {

38. "range": {

39. "bartab": {

40. "lte": 10

41. }

42. }

43. }

44. ]

45. }

46. }

47. }

48.

49. #get all tasks

50. GET \_tasks/

51.

52. #get a specific task

53. //can check if the task is completed or not (i.e. still running)

54. GET \_tasks/ZcHaYdWIQu-sSO2A9jShWw:3187041

55.

56. #cancel a running task

57. POST \_tasks/ZcHaYdWIQu-sSO2A9jShWw:31870411/\_cancel

58.

59.

60. #delete a task

61. DELETE .tasks/task/ZcHaYdWIQu-sSO2A9jShWw:3187041g:512501

62.

**Conclusion**

Next steps

A picture containing text, screenshot, line, font

Description automatically generated

A screen shot of a computer

Description automatically generated with low confidence

A picture containing text, screenshot, font, design

Description automatically generated