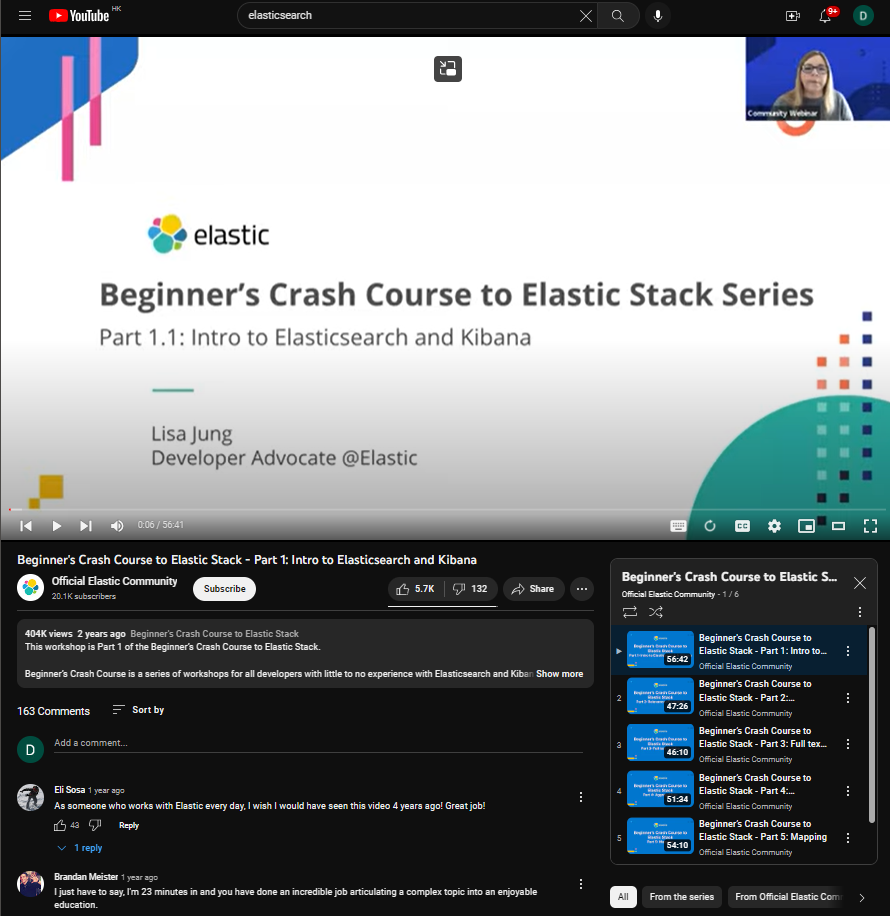
6 Part Videos by the Official Elastic Community - @~50mins each



**A picture containing text, screenshot, font

Description automatically generated**

**Part 1: Intro to Elasticsearch and Kibana**

If you have not used Elastic before, chances are you have used apps that utilized Elastic as part of the search function.

* Index does NOT store documents – it is just for classification documents. The shards DO store the documents.
* Distribute search queries to replica shards

A screenshot of a computer

Description automatically generated with low confidence

1. #Basic syntax format

2. #get \_API/parameter

3.

4. get \_cluster/health

5.

6. get \_nodes/stats

7.

8. PUT favorite\_candy

9.

10. #When indexing a document, both HTTP verbs POST or PUT can be used.

11. #Use POST when you want Elasticsearch to autogenerate an id for your document.

12.

13. POST favorite\_candy/\_doc

14. {

15. "first\_name": "Lisa",

16. "candy": "Sour Skittles"

17. }

18.

19. #Use PUT when you want to assign a specific id to your document(i.e. if your document has a natural identifier - purchase order number, patient id, & etc). For more detailed explanation, check out this documentation from Elastic!

20.

21. PUT favorite\_candy/\_doc/1

22. {

23. "first\_name": "John",

24. "candy": "Starburst"

25. }

26.

27. PUT favorite\_candy/\_doc/2

28. {

29. "first\_name": "Rachel",

30. "candy": "Rolos"

31. }

32.

33. PUT favorite\_candy/\_doc/3

34. {

35. "first\_name": "Tom",

36. "candy": "Sweet Tarts"

37. }

38.

39.

40. PUT favorite\_candy/\_doc/4

41. {

42. "first\_name": "Daryl",

43. "candy": "Cherry Pickers"

44. }

45.

46. #Assigning assigned ID will overwrite the existing with this one

47. PUT favorite\_candy/\_doc/1

48. {

49. "first\_name": "Sally",

50. "candy": "Snickers"

51. }

52.

53. GET favorite\_candy/\_doc/1

54.

55. #\_create Endpoint

56. #When you index a document using an id that already exists, the existing document is overwritten by the new document. If you do not want a existing document to be overwritten, you can use the \_create endpoint!

57.

58. #With the \_create Endpoint, no indexing will occur and you will get a 409 error message.

59.

60. PUT favorite\_candy/\_create/1

61. {

62. "first\_name": "Kevin",

63. "candy": "Turkish Candy"

64. }

65.

66. PUT favorite\_candy/\_create/8

67. {

68. "first\_name": "Kevin",

69. "candy": "Turkish Candy"

70. }

71.

72. #Update a document

73. #If you want to update fields in a document, use the following syntax:

74.

75. POST favorite\_candy/\_update/1

76. {

77. "doc": {

78. "candy": "M&M's"

79. }

80. }

81.

82. #Delete a document

83.

84. DELETE favorite\_candy/\_doc/8

85.

86. GET favorite\_candy/\_doc/8

87.

88. #Take Home Assignment

89. #Create an index called destinations.

90. #Pick five dream travel destinations. For each destination, index a document containing the name and the country.

91. #Read(GET) each document to check the content of the document.

92. #Update a field of a document.

93. #Read(GET) the updated document to ensure that the field has been updated.

94. #Delete a document of one place.

95. #Copy and paste the following request to return all documents from the destinations index. This is a great way to check whether all the CRUD operations you have performed thus far have worked!

96.

97. PUT destinations

98.

99. PUT destinations/\_doc/1

100. {

101. "name": "Aurora Borealis",

102. "country": "UK (Northern Island)"

103. }

104.

105. PUT destinations/\_doc/2

106. {

107. "name": "Fuji Mountain",

108. "country": "Japan"

109. }

110.

111. PUT destinations/\_doc/3

112. {

113. "name": "Labuan Bajo",

114. "country": "Indonesia"

115. }

116.

117. PUT destinations/\_doc/4

118. {

119. "name": "Alpines",

120. "country": "IDK where is it?"

121. }

122.

123. PUT destinations/\_doc/5

124. {

125. "name": "Celtic Stuffs",

126. "country": "Somewhere in Europe"

127. }

128.

129. GET destinations/\_doc/1

130. GET destinations/\_doc/2

131. GET destinations/\_doc/3

132. GET destinations/\_doc/4

133. GET destinations/\_doc/5

134.

135. DELETE destinations/\_doc/5

136.

137. GET destinations/\_doc/5

138.

139. GET destinations/\_search

140. {

141. "query": {

142. "match\_all": {}

143. }

144. }

145.

**Part 2: Relevance of a Search**

- Precision and recall is inversely related

- Term Frequency (TF) vs Inverse Document Frequency (IDF)

1. #Retrieve 10 documents by defualt

2. GET news\_headlines/\_search

3.

4. #hits: 10000 gte. Meaning that >10000

5. #to get exact total number of records, use this command

6.

7. GET news\_headlines/\_search

8. {

9. "track\_total\_hits": true

10. }

OR POST MyIndex/\_count #much faster as it does not execute full query ranking and result fetching

11.

12. get news\_headlines/\_search

13. {

14. "query": {

15. "range": {

16. "date": {

17. "gte": "2015-06-20",

18. "lte": "2015-09-22"

19. }

20. }

21. }

22. }

23.

24. GET news\_headlines/\_search?filter\_path=aggregations

25. {

26. "aggs": {

27. "by\_category": {

28. "terms": {

29. "field": "category",

30. "size": 100 //fetch up to 100 categories if there are more

31. }

32. }

33. }

34. }

35.

36. GET news\_headlines/\_mapping

37.

38. GET news\_headlines/\_search?filter\_path=aggregations

39. {

40. "query": {

41. "match": {

42. "category": "ENTERTAINMENT"

43. }

44. },

45. "aggs": {

46. "popular\_in\_entertainment": {

47. "significant\_text": {

48. "field": "headline"

49. }

50. }

51. }

52. }

53.

54. GET news\_headlines/\_search

55. {

56. "query": {

57. "match": {

58. "headline": {

59. //By default query is OR logic (RECALL FOCUSED)

60. "query": "Khloe Kardashian Kendall Jenner"

61. }

62. }

63. }

64. }

65.

66. GET news\_headlines/\_search

67. {

68. "query": {

69. "match": {

70. "headline": {

71. //Using AND operator (PRECISION FOCUSED)

72. "query": "Khloe Kardashian Kendall Jenner",

73. "operator": "and"

74. }

75. }

76. }

77. }

78.

79. GET news\_headlines/\_search

80. {

81. "query": {

82. "match": {

83. "headline": {

84. //Minimum\_should\_match API strikes the balance between PRECISION and RECALL, m\_s\_m: 3 = 3 out of 4 words have to appear, increasing precision but still give rooms to recall

85. "query": "Khloe Kardashian Kendall Jenner",

86. "minimum\_should\_match": 3

87. }

88. }

89. }

90. }

91.

92.

**Part 3: Full Text Queries**

- match vs match\_phrase (untokenized)

- bool query: must, must\_not, should, filter

1. GET news\_headlines/\_search

2. {

3. "aggregations": {

4. "by\_category": {

5. "terms": {

6. "field": "category",

7. "size": 100

8. }

9. }

10. }

11. }

12.

13. #When the match query is used to search for a phrase, it has high recall but low precision. It pulls up more loosely related documents as it uses "OR" logic by default. It pulls up documents that contains any one of the search terms in the specified field. Moreover, the order and the proximity in which the search terms are found are not taken into account.

14. GET news\_headlines/\_search

15. {

16. "query": {

17. //match is tokenized

18. "match": {

19. "headline": {

20. "query": "Shape of you"

21. }

22. }

23. }

24. }

25.

26. #When the match\_phrase parameter is used, all hits must meet the following criteria:

27.

28. #1. the search terms "Shape", "of", and "you" must appear in the field headline .

29. #2. the terms must appear in that order.

30. #3. the terms must appear next to each other.

31. GET news\_headlines/\_search

32. {

33. "query": {

34. //match\_phrase: word order maintained, next to each other and exactly like the query

35. "match\_phrase": {

36. "headline": {

37. "query": "Shape of you"

38. }

39. }

40. }

41. }

42.

43. #The multi\_match query runs a match query on multiple fields and calculates a score for each field. Then, it assigns the highest score among the fields to the document.

44. GET news\_headlines/\_search

45. {

46. "query": {

47. "multi\_match": {

48. "query": "Michelle Obama",

49. "fields": [

50. "headline",

51. "short\_description",

52. "authors"]

53. }

54. }

55. }

56.

57. #Per-field boosting

58. #Headlines mentioning "Michelle Obama" in the field headline are more likely to be related to our search than the headlines that mention "Michelle Obama" once or twice in the field short\_description.

59.

60. #To improve the precision of your search, you can designate one field to carry more weight than the others.

61.

62. #This can be done by boosting the score of the field headline(per-field boosting). This is notated by adding a carat(^) symbol and number 2 to the desired field as shown below

63.

64. #Multi-match is not match-phrased (they are tokenized)

65. GET news\_headlines/\_search

66. {

67. "query": {

68. "multi\_match": {

69. "query": "Michelle Obama",

70. "fields": [

71. "headline^2",

72. "short\_description"

73. ]

74. }

75. }

76. }

77.

78. #You can improve the precision of a multi\_match query by adding the "type":"phrase" to the query.

79.

80. #The phrase type performs a match\_phrase query on each field and calculates a score for each field. Then, it assigns the highest score among the fields to the document.

81.

82. get news\_headlines/\_search

83. {

84. "query": {

85. "multi\_match": {

86. "query": "party planning",

87. "fields": [

88. "headline^3",

89. "short\_description"],

90. "type": "phrase"

91. }

92. }

93. }

94.

95. #Combined Queries

96. #There will be times when a user asks a multi-faceted question that requires multiple queries to answer.

97.

98. #For example, a user may want to find political headlines about Michelle Obama published before the year 2016.

99.

100. #This search is actually a combination of three queries:

101.

102. #1. Query headlines that contain the search terms "Michelle Obama" in the field headline.

103. #2. Query "Michelle Obama" headlines from the "POLITICS" category.

104. #3. Query "Michelle Obama" headlines published before the year 2016

105. #One of the ways you can combine these queries is through a bool query.

106.

107. #Bool Query

108. #The bool query retrieves documents matching boolean combinations of other queries.

109.

110. #With the bool query, you can combine multiple queries into one request and further specify boolean clauses to narrow down your search results.

111.

112. #There are four clauses to choose from:

113.

114. #must

115. #must\_not

116. #should

117. #filter

118. #You can build combinations of one or more of these clauses. Each clause can contain one or multiple queries that specify the criteria of each clause.

119.

120. #These clauses are optional and can be mixed and matched to cater to your use case. The order in which they appear does not matter either!

121.

122. GET news\_headlines/\_search

123. {

124. "query": {

125. "match\_phrase": {

126. "headline": "Michelle Obama"

127. }

128. },

129. "aggs": {

130. "category\_mentions": {

131. "terms": {

132. "field": "category",

133. "size": 100

134. }

135. }

136. }

137. }

138.

139. #The must clause

140. #The must clause defines all queries(criteria) a document MUST match to be returned as hits. These criteria are expressed in the form of one or multiple queries.

141.

142. #All queries in the must clause must be satisfied for a document to be returned as a hit. As a result, having more queries in the must clause will increase the precision of your query.

143.

144. GET news\_headlines/\_search

145. {

146. "query": {

147. "bool": {

148. "must": [

149. {"match\_phrase": {

150. "headline": "Michelle Obama"

151. }},

152. {

153. "match": {

154. "category": "POLITICS"

155. }

156. }]

157. }

158. }

159. }

160.

161. #The must\_not clause

162. #The must\_not clause defines queries(criteria) a document MUST NOT match to be included in the search results.

163.

164. GET news\_headlines/\_search

165. {

166. "query": {

167. "bool": {

168. "must": [

169. {"match\_phrase": {

170. "headline": "Michelle Obama"

171. }}

172. ],

173. "must\_not": [

174. {"match": {

175. "category": "WEDDINGS"

176. }}

177. ]

178. }

179. }

180. }

181.

182. #The should clause

183. #The should clause adds "nice to have" queries(criteria). The documents do not need to match the "nice to have" queries to be considered as hits. However, the ones that do will be given a higher score so it shows up higher in the search results.

184.

185. #Category of BLACK VOICES will be prioritized (higher score)

186. GET news\_headlines/\_search?filter\_path=hits.hits.\_source.category

187. {

188. "query": {

189. "bool": {

190. "must": [{

191. "match\_phrase": {

192. "headline": "Michelle Obama"

193. }}],

194. "should": [

195. {"match\_phrase": {

196. "category": "BLACK VOICES"

197. }}

198. ]

199. }

200. },

201. "size": 100

202. }

203.

204. #The filter clause

205. #The filter clause contains filter queries that place documents into either "yes" or "no" category.

206.

207. #For example, let's say you are looking for headlines published within a certain time range. Some documents will fall within this range(yes) or do not fall within this range(no).

208.

209. #The filter clause only includes documents that fall into the yes category.

210.

211. #Let's say we wanted to retrieve hits that must include the phrase "Michelle Obama" in the field headline. Among these hits, you want to include documents published within the date range "2014-03-25" and "2016-03-25".

212.

213. #You bool query will look something like this.

214.

215. GET news\_headlines/\_search

216. {

217. "query": {

218. "bool": {

219. "must": [

220. {"match\_phrase": {

221. "headline": "Michelle Obama"

222. }}

223. ],

224. "filter": [

225. {"range": {

226. "date": {

227. "gte": "2014-03-25",

228. "lte": "2016-03-25"

229. }

230. }}

231. ]

232. }

233. }

234. }

235.

236. #Fine-tuning the relevance of bool queries

237. #There are many ways you can fine-tune the relevance of bool queries. One of the ways is to add multiple queries under the should clause.

238.

239. #Adding multiple queries under the should clause

240.

241. #This approach ensures that you maintain a high recall but also offers a way to present more precise search results at the top of your search results.

242.

243. #Let's say you want to run a search for news headlines with the phrase "Michelle Obama" in the field headline. But you want to favor articles that mention her biography "Becoming", and terms like "women" and "empower".

244.

245. #To do this, you can add multiple queries to the should clause.

246.

247. #This will cast a wider net because none of the queries in the should clause need to match. However, the ones that match the queries under the should clause will be given a higher score and placed higher in the search results.

248.

249. #This approach allows you to maintain a high recall but also gives you a way to customize the precision of top hits.

250.

251. GET news\_headlines/\_search

252. {

253. "query": {

254. "bool": {

255. "must": [

256. {"match\_phrase": {

257. "headline": "Michelle Obama"

258. }}

259. ],

260. "should": [

261. {"match": {

262. "headline": "Becoming"

263. }},

264. {"match": {

265. "headline": "women"

266. }},

267. {"match": {

268. "headline": "empower"

269. }}

270. ]

271. }

272. }

273. }

274.

**Part 4: Aggregations**

There are two main ways to search in Elasticsearch:

1. Queries retrieve documents that match the specified criteria.
2. Aggregations present the summary of your data as metrics, statistics, and other analytics.

1. GET ecommnerce\_data/\_search

2.

3. #Metric Aggregations

4. #Metricaggregations are used to compute numeric values based on your dataset. It can be used to calculate the values of sum,min, max, avg, unique count(cardinality) and etc.

5.

6. #Compute the sum of all unit prices in the index

7. GET ecommnerce\_data/\_search

8. {

9. "size": 0, //we are not interested in the hits

10. "aggs": {

11. "sum\_unit\_price": {

12. "sum": {

13. "field": "UnitPrice"

14. }

15. }

16. }

17. }

18.

19. get ecommnerce\_data/\_search

20. {

21. "size": 0,

22. "aggs": {

23. "min\_unit\_price": {

24. "avg": {

25. "field": "UnitPrice"

26. }

27. }

28. }

29. }

30.

31. #Retrieve all aggregations stats

32. get ecommnerce\_data/\_search

33. {

34. "size": 0,

35. "aggs": {

36. "all\_stats\_unit\_price": {

37. "stats": {

38. "field": "UnitPrice"

39. }

40. }

41. }

42. }

43.

44. #Cardinality (unique values) Aggregation

45. GET ecommnerce\_data/\_search

46. {

47. "size": 0,

48. "aggs": {

49. "number\_of\_unique\_customers": {

50. "cardinality": {

51. "field": "CustomerID"

52. }

53. }

54. }

55. }

56.

57. #Limiting the scope of an aggregation

58. #In the previous examples, aggregations were performed on all documents in the ecommerce\_data index. What if you want to run an aggregation on a subset of the documents?

59.

60. #For example, our index contains e-commerce data from multiple countries. Let's say you want to calculate the average unit price of items sold in Germany.

61.

62. #To limit the scope of the aggregation, you can add a query clause to the aggregations request. The query clause defines the subset of documents that aggregations should be performed on.

63.

64. #The combined query and aggregations look like the following.

65.

66. GET ecommnerce\_data/\_search

67. {

68. "size": 0,

69. "query": {

70. "match": {

71. "Country": "Germany"

72. }

73. },

74. "aggs": {

75. "avg\_unit\_price\_germany": {

76. "avg": {

77. "field": "UnitPrice"

78. }

79. }

80. }

81. }

82.

83. #Bucket Aggregations

84. #When you want to aggregate on several subsets of documents, bucket aggregations will come in handy. Bucket aggregations group documents into several sets of documents called buckets. All documents in a bucket share a common criteria.

85.

86. #The following are different types of bucket aggregations.

87.

88. #Date Histogram Aggregation

89. #Histogram Aggregation

90. #Range Aggregation

91. #Terms aggregation

92. #1. Date Histogram Aggregation

93. #When you are looking to group data by time interval, the date\_histogram aggregation will prove very useful!

94.

95. #Our ecommerce\_data index contains transaction data that has been collected over time(from the year 2010 to 2011).

96.

97. #If we are looking to get insights about transactions over time, our first instinct should be to run the date\_histogram aggregation.

98.

99. #There are two ways to define a time interval with date\_histogram aggregation. These are Fixed\_interval and Calendar\_interval.

100.

101. #ixed\_interval With the fixed\_interval, the interval is always constant.

102.

103. #Example: Create a bucket for every 8 hour interval.

104.

105. #This generates error???

106. GET ecommnerce\_data/\_search

107. {

108. "size": 0,

109. "aggs": {

110. "transactions\_by\_8\_hrs": {

111. "date\_histogram": {

112. "field": "InvoiceDate",

113. "calendar\_interval": "1M",

114. "order": {

115. "\_key": "desc"

116. }

117. }

118. }

119. }

120. }

121.

122. GET ecommnerce\_data/\_search

123. {

124. "size": 0,

125. "aggs": {

126. "transactions\_per\_price\_interval": {

127. "histogram": {

128. "field": "UnitPrice",

129. "interval": 10000,

130. "order": {

131. "\_key": "desc"

132. }

133. }

134. }

135. }

136. }

137.

138. #Range Aggregation

139. #The range aggregation is similar to the histogram aggregation in that it can create buckets based on any numerical interval. The difference is that the range aggregation allows you to define intervals of varying sizes so you can customize it to your use case.

140.

141. #For example, what if you wanted to know the number of transactions for items from varying price ranges(between 0 and $50, between $50-$200, and between $200 and up)?

142.

143. GET ecommnerce\_data/\_search

144. {

145. "size": 0,

146. "aggs": {

147. "transactions\_per\_custom\_price\_ranges": {

148. "range": {

149. "field": "UnitPrice",

150. "ranges": [

151. {

152. "to": 50

153. },

154. {

155. "from": 50,

156. "to": 200

157. },

158. {

159. "from": 200,

160. "to": 1000

161. },

162. {

163. "from": 1000,

164. "to": 5000

165. },

166. {

167. "from": 5000

168. }

169. ]

170. }

171. }

172. }

173. }

174.

175. #Terms Aggregation

176. #The terms aggregation creates a new bucket for every unique term it encounters for the specified field. It is often used to find the most frequently found terms in a document.

177.

178. #For example, let's say you want to identify 5 customers with the highest number of transactions(documents).

179.

180. GET ecommnerce\_data/\_search

181. {

182. "size": 0,

183. "aggs": {

184. "top\_5\_customers": {

185. "terms": {

186. //By default, terms aggregate the count of documents, count descending

187. "field": "CustomerID",

188. "size": 5,

189. "order": {

190. "\_count": "asc"

191. }

192. }

193. }

194. }

195. }

196.

197. #Combined Aggregations

198. #So far, we have ran metric aggregations or bucket aggregations to answer simple questions.

199.

200. #There will be times when we will ask more complex questions that require running combinations of these aggregations.

201.

202. #For example, let's say we wanted to know the sum of revenue per day.

203.

204. #To get the answer, we need to first split our data into daily buckets(date\_histogram aggregation).

205.

206. #This does not work???

207. GET ecommnerce\_data/\_search

208. {

209. "size": 0,

210. "aggs": {

211. "transactions\_per\_day": {

212. "date\_histogram": {

213. "field": "InvoiceDate",

214. "calendar\_interval": "day"

215. },

216. "aggs": {

217. "daily\_revenue": {

218. "sum": {

219. "script": {

220. "source": "doc['UnitPrice'].value \* doc['Quantity'].value"

221. }

222. }

223. }

224. }

225. }

226. }

227. }

228.

229. #Calculating multiple metrics per bucket

230. #You can also calculate multiple metrics per bucket.

231.

232. #For example, let's say you wanted to calculate the daily revenue and the number of unique customers per day in one go. To do this, you can add multiple metric aggregations per bucket as shown below!

233.

234. GET ecommerce\_data/\_search

235. {

236. "size": 0,

237. "aggs": {

238. "transactions\_per\_day": {

239. "date\_histogram": {

240. "field": "InvoiceDate",

241. "calendar\_interval": "day"

242. },

243. "aggs": {

244. "daily\_revenue": {

245. "sum": {

246. "script": {

247. "source": "doc['UnitPrice'].value \* doc['Quantity'].value"

248. }

249. }

250. },

251. "number\_of\_unique\_customers\_per\_day": {

252. "cardinality": {

253. "field": "CustomerID"

254. }

255. }

256. }

257. }

258. }

259. }

260.

261. #Sorting by metric value of a sub-aggregation

262.

263. #You do not always need to sort by time interval, numerical interval, or by doc\_count! You can also sort by metric value of sub-aggregations.

264.

265. #Let's take a look at the request below. Within the sub-aggregation, metric values "daily\_revenue" and "number\_of\_unique\_customers\_per\_day" are calculated.

266.

267. #Let's say you wanted to find which day had the highest daily revenue to date!

268.

269. #All you need to do is to add the "order" parameter( and sort buckets based on the metric value of "daily\_revenue" in descending("desc") order!

270.

271. GET ecommerce\_data/\_search

272. {

273. "size": 0,

274. "aggs": {

275. "transactions\_per\_day": {

276. "date\_histogram": {

277. "field": "InvoiceDate",

278. "calendar\_interval": "day",

279. "order": {

280. //sort by daily\_revenue (the sub-aggregation, from high to low (desc)

281. "daily\_revenue": "desc"

282. }

283. },

284. "aggs": {

285. "daily\_revenue": {

286. "sum": {

287. "script": {

288. "source": "doc['UnitPrice'].value \* doc['Quantity'].value"

289. }

290. }

291. },

292. "number\_of\_unique\_customers\_per\_day": {

293. "cardinality": {

294. "field": "CustomerID"

295. }

296. }

297. }

298. }

299. }

300. }

301.

**Part 5: Mapping**

A screenshot of a computer

Description automatically generated with low confidence



When Elasticsearch dynamically creates a mapping for you, it does not know what you want to use a string for so it maps all strings to both field types.

In cases where you do not need both field types, the default setting is wasteful. Since both field types require creating either an inverted index or doc values, creating both field types for unnecessary fields will slow down indexing and take up more disk space.

This is why we define our own mapping as it helps us store and search data more efficiently.

1. Beginner's Crash Course to Elastic Stack Series

2. Part 5: Understanding Mapping with Elasticsearch and Kibana

3. Welcome to the Beginner's Crash Course to Elastic Stack!

4.

5. This repo contains all resources shared during Part 5: Understanding Mapping with Elasticsearch and Kibana.

6.

7. Have you ever encountered the error “Field type is not supported for [whatever you are trying to do with Elasticsearch]”?

8.

9. The most likely culprit of this error is the mapping of your index!

10.

11. Mapping is the process of defining how a document and its fields are indexed and stored. It defines the type and format of the fields in the documents. As a result, mapping can significantly affect how Elasticsearch searches and stores data.

12.

13. Understanding how mapping works will help you define mapping that best serves your use case.

14.

15. By the end of this workshop, you will be able to define what a mapping is and define your own mapping to make indexing and searching more efficient.

16.

17. Resources

18. Table of Contents: Beginner's Crash Course to Elastic Stack:

19.

20. This workshop is a part of the Beginner's Crash Course to Elastic Stack series. Check out this table contents to access all the workshops in the series thus far. This table will continue to get updated as more workshops in the series are released!

21.

22. Free Elastic Cloud Trial

23.

24. Instructions on how to access Elasticsearch and Kibana on Elastic Cloud

25.

26. Instructions for downloading Elasticsearch and Kibana

27.

28. Video recording of the workshop

29.

30. Mini Beginner's Crash Course to Elasticsearch & Kibana playlist

31.

32. Do you prefer learning by watching shorter videos? Check out this playlist to watch short clips of beginner's crash course full length workshops. Part 5 workshop is broken down into episodes 19-22. Season 2 clips will be uploaded here in the future!

33.

34. Presentation Slides

35.

36. What's next? Eager to continue your learning after mastering the concept from this workshop? Move on to Part 6: Troubleshooting Beginner Level Elasticsearch Errors here!

37.

38. What is a Mapping?

39. image

40.

41. Review from Previous Workshops

42. image

43.

44. Indexing a Document

45. The following request will index the following document.

46.

47. Syntax:

48.

49. POST Enter-name-of-the-index/\_doc

50. {

51. "field": "value"

52. }

53. Example:

54.

55. POST temp\_index/\_doc

56. {

57. "name": "Pineapple",

58. "botanical\_name": "Ananas comosus",

59. "produce\_type": "Fruit",

60. "country\_of\_origin": "New Zealand",

61. "date\_purchased": "2020-06-02T12:15:35",

62. "quantity": 200,

63. "unit\_price": 3.11,

64. "description": "a large juicy tropical fruit consisting of aromatic edible yellow flesh surrounded by a tough segmented skin and topped with a tuft of stiff leaves.These pineapples are sourced from New Zealand.",

65. "vendor\_details": {

66. "vendor": "Tropical Fruit Growers of New Zealand",

67. "main\_contact": "Hugh Rose",

68. "vendor\_location": "Whangarei, New Zealand",

69. "preferred\_vendor": true

70. }

71. }

72. Expected response from Elasticsearch:

73.

74. Elasticsearch will confirm that this document has been successfully indexed into the temp\_index. image

75.

76. Mapping Explained

77. Mapping determines how a document and its fields are indexed and stored by defining the type of each field.

78.

79. image

80.

81. It contains a list of the names and types of fields in an index. Depending on its type, each field is indexed and stored differently in Elasticsearch.

82.

83. Dynamic Mapping

84. When a user does not define mapping in advance, Elasticsearch creates or updates the mapping as needed by default. This is known as dynamic mapping.

85.

86. image

87.

88. With dynamic mapping, Elasticsearch looks at each field and tries to infer the data type from the field content. Then, it assigns a type to each field and creates a list of field names and types known as mapping.

89.

90. Depending on the assigned field type, each field is indexed and primed for different types of requests(full text search, aggregations, sorting). This is why mapping plays an important role in how Elasticsearch stores and searches for data.

91.

92. View the Mapping

93. Syntax:

94.

95. GET Enter\_name\_of\_the\_index\_here/\_mapping

96. Example:

97.

98. GET temp\_index/\_mapping

99. Expected response from Elasticsearch:

100.

101. Elasticsearch returns the mapping of the temp\_index. It lists all the fields of the document in an alphabetical order and lists the type of each field(text, keyword, long, float, date, boolean and etc).

102.

103. image image image

104.

105. For the list of all field types, click here!

106.

107. Indexing Strings

108. There are two kinds of string field types:

109.

110. Text

111. Keyword

112. By default, every string gets mapped twice as a text field and as a keyword multi-field. Each field type is primed for different types of requests.

113.

114. Text field type is designed for full-text searches.

115.

116. Keywordfield type is designed for exact searches, aggregations, and sorting.

117.

118. You can customize your mapping by assigning the field type as either text or keyword or both!

119.

120. Text Field Type

121. Text Analysis

122. Ever notice that when you search in Elasticsearch, it is not case sensitive or punctuation does not seem to matter? This is because text analysis occurs when your fields are indexed.

123.

124. By default, strings are analyzed when it is indexed. The string is broken up into individual words also known as tokens. The analyzer further lowercases each token and removes punctuations.

125.

126. image

127.

128. Inverted Index image Once the string is analyzed, the individual tokens are stored in a sorted list known as the inverted index. Each unique token is stored in the inverted index with its associated ID.

129.

130. The same process occurs every time you index a new document.

131.

132. image image image image

133.

134. Keyword Field Type

135. Keyword field type is used for aggregations, sorting, and exact searches. These actions look up the document ID to find the values it has in its fields.

136.

137. Keyword field is suited to perform these actions because it uses a data structure called doc values to store data.

138.

139. For each document, the document id along with the field value(original string) are added to the table. This data structure(doc values) is designed for actions that require looking up the document ID to find the values it has in its fields.

140.

141. image

142.

143. When Elasticsearch dynamically creates a mapping for you, it does not know what you want to use a string for so it maps all strings to both field types.

144.

145. In cases where you do not need both field types, the default setting is wasteful. Since both field types require creating either an inverted index or doc values, creating both field types for unnecessary fields will slow down indexing and take up more disk space.

146.

147. This is why we define our own mapping as it helps us store and search data more efficiently.

148.

149. Mapping Exercise

150. Project: Build an app for a client who manages a produce warehouse

151.

152. This app must enable users to:

153.

154. search for produce name, country of origin and description

155.

156. identify top countries of origin with the most frequent purchase history

157.

158. sort produce by produce type(Fruit or Vegetable)

159.

160. get the summary of monthly expense

161.

162. Sample data

163.

164. {

165. "name": "Pineapple",

166. "botanical\_name": "Ananas comosus",

167. "produce\_type": "Fruit",

168. "country\_of\_origin": "New Zealand",

169. "date\_purchased": "2020-06-02T12:15:35",

170. "quantity": 200,

171. "unit\_price": 3.11,

172. "description": "a large juicy tropical fruit consisting of aromatic edible yellow flesh surrounded by a tough segmented skin and topped with a tuft of stiff leaves.These pineapples are sourced from New Zealand.",

173. "vendor\_details": {

174. "vendor": "Tropical Fruit Growers of New Zealand",

175. "main\_contact": "Hugh Rose",

176. "vendor\_location": "Whangarei, New Zealand",

177. "preferred\_vendor": true

178. }

179. }

180. Plan of Action image image image image image image image

181.

182. Defining your own mapping

183. Rules

184.

185. If you do not define a mapping ahead of time, Elasticsearch dynamically creates the mapping for you.

186. If you do decide to define your own mapping, you can do so at index creation.

187. ONE mapping is defined per index. Once the index has been created, we can only add new fields to a mapping. We CANNOT change the mapping of an existing field.

188. If you must change the type of an existing field, you must create a new index with the desired mapping, then reindex all documents into the new index.

189. Step 1: Index a sample document into a test index.

190.

191. The sample document must contain the fields that you want to define. These fields must also contain values that map closely to the field types you want.

192.

193. Syntax:

194.

195. POST Name-of-test-index/\_doc

196. {

197. "field": "value"

198. }

199. Example:

200.

201. POST test\_index/\_doc

202. {

203. "name": "Pineapple",

204. "botanical\_name": "Ananas comosus",

205. "produce\_type": "Fruit",

206. "country\_of\_origin": "New Zealand",

207. "date\_purchased": "2020-06-02T12:15:35",

208. "quantity": 200,

209. "unit\_price": 3.11,

210. "description": "a large juicy tropical fruit consisting of aromatic edible yellow flesh surrounded by a tough segmented skin and topped with a tuft of stiff leaves.These pineapples are sourced from New Zealand.",

211. "vendor\_details": {

212. "vendor": "Tropical Fruit Growers of New Zealand",

213. "main\_contact": "Hugh Rose",

214. "vendor\_location": "Whangarei, New Zealand",

215. "preferred\_vendor": true

216. }

217. }

218. Expected response from Elasticsearch:

219.

220. The test\_index is successfully created. image

221.

222. Step 2: View the dynamic mapping

223.

224. Syntax:

225.

226. GET Name-the-index-whose-mapping-you-want-to-view/\_mapping

227. Example:

228.

229. GET test\_index/\_mapping

230. Expected response from Elasticsearch:

231.

232. Elasticsearch will display the mapping it has created. It lists the fields in an alphabetical order. This document is identical to the one we indexed into the temp\_index. To save space, the screenshots of the mapping has not been included here.

233.

234. Step 3: Edit the mapping

235.

236. Copy and paste the mapping from step 2 into the Kibana console. From the pasted results, remove the "test\_index" along with its opening and closing brackets. Then, edit the mapping to satisfy the requirements outlined in the figure below.

237.

238. image

239.

240. The optimized mapping should look like the following:

241.

242. {

243. "mappings": {

244. "properties": {

245. "botanical\_name": {

246. "enabled": false

247. },

248. "country\_of\_origin": {

249. "type": "text",

250. "fields": {

251. "keyword": {

252. "type": "keyword"

253. }

254. }

255. },

256. "date\_purchased": {

257. "type": "date"

258. },

259. "description": {

260. "type": "text"

261. },

262. "name": {

263. "type": "text"

264. },

265. "produce\_type": {

266. "type": "keyword"

267. },

268. "quantity": {

269. "type": "long"

270. },

271. "unit\_price": {

272. "type": "float"

273. },

274. "vendor\_details": {

275. "enabled": false

276. }

277. }

278. }

279. }

280. image

281.

282. Step 4: Create a new index with the optimized mapping from step 3.

283.

284. Syntax:

285.

286. PUT Name-of-your-final-index

287. {

288. copy and paste your edited mapping here

289. }

290. Example:

291.

292. PUT produce\_index

293. {

294. "mappings": {

295. "properties": {

296. "botanical\_name": {

297. "enabled": false

298. },

299. "country\_of\_origin": {

300. "type": "text",

301. "fields": {

302. "keyword": {

303. "type": "keyword"

304. }

305. }

306. },

307. "date\_purchased": {

308. "type": "date"

309. },

310. "description": {

311. "type": "text"

312. },

313. "name": {

314. "type": "text"

315. },

316. "produce\_type": {

317. "type": "keyword"

318. },

319. "quantity": {

320. "type": "long"

321. },

322. "unit\_price": {

323. "type": "float"

324. },

325. "vendor\_details": {

326. "enabled": false

327. }

328. }

329. }

330. }

331. Expected response from Elasticsearch:

332.

333. Elasticsearch creates a produce\_index with the customized mapping we defined above!

334.

335. image

336.

337. Step 5: Check the mapping of the new index to make sure the all the fields have been mapped correctly

338.

339. Syntax:

340.

341. GET Name-of-test-index/\_mapping

342. Example:

343.

344. GET produce\_index/\_mapping

345. Expected response from Elasticsearch:

346.

347. Compared to the dynamic mapping, our optimized mapping looks more simple and concise! The current mapping satisfies the requirements that are marked with green check marks.

348.

349. image image image

350.

351. Step 6: Index your dataset into the new index

352.

353. For simplicity's sake, we will index two documents.

354.

355. Index the first document

356.

357. POST produce\_index/\_doc

358. {

359. "name": "Pineapple",

360. "botanical\_name": "Ananas comosus",

361. "produce\_type": "Fruit",

362. "country\_of\_origin": "New Zealand",

363. "date\_purchased": "2020-06-02T12:15:35",

364. "quantity": 200,

365. "unit\_price": 3.11,

366. "description": "a large juicy tropical fruit consisting of aromatic edible yellow flesh surrounded by a tough segmented skin and topped with a tuft of stiff leaves.These pineapples are sourced from New Zealand.",

367. "vendor\_details": {

368. "vendor": "Tropical Fruit Growers of New Zealand",

369. "main\_contact": "Hugh Rose",

370. "vendor\_location": "Whangarei, New Zealand",

371. "preferred\_vendor": true

372. }

373. }

374. Expected response from Elasticsearch:

375.

376. Elasticsearch successfully indexes the first document. image

377.

378. Index the second document

379.

380. The second document has almost identical fields as the first document except that it has an extra field called "organic" set to true!

381.

382. POST produce\_index/\_doc

383. {

384. "name": "Mango",

385. "botanical\_name": "Harum Manis",

386. "produce\_type": "Fruit",

387. "country\_of\_origin": "Indonesia",

388. "organic": true,

389. "date\_purchased": "2020-05-02T07:15:35",

390. "quantity": 500,

391. "unit\_price": 1.5,

392. "description": "Mango Arumanis or Harum Manis is originated from East Java. Arumanis means harum dan manis or fragrant and sweet just like its taste. The ripe Mango Arumanis has dark green skin coated with thin grayish natural wax. The flesh is deep yellow, thick, and soft with little to no fiber. Mango Arumanis is best eaten when ripe.",

393. "vendor\_details": {

394. "vendor": "Ayra Shezan Trading",

395. "main\_contact": "Suharto",

396. "vendor\_location": "Binjai, Indonesia",

397. "preferred\_vendor": true

398. }

399. }

400. Expected response from Elasticsearch:

401.

402. Elasticsearch successfully indexes the second document. image

403.

404. Let's see what happens to the mapping by sending this request below:

405.

406. GET produce\_index/\_mapping

407. Expected response from Elasticsearch:

408.

409. The new field("organic") and its field type(boolean) have been added to the mapping. This is in line with the rules of mapping we discussed earlier since you can add new fields to the mapping. We just cannot change the mapping of an existing field!

410.

411. image image

412.

413. What if you do need to make changes to the mapping of an existing field?

414. Let's say your client changed his mind. He wants to run only full text search on the field "botanical\_name" we disabled earlier.

415.

416. Remember, you CANNOT change the mapping of an existing field. If you do need to make changes to an existing field, you must create a new index with the desired mapping, then reindex all documents into the new index.

417.

418. STEP 1: Create a new index(produce\_v2) with the latest mapping.

419.

420. We removed the "enabled" parameter from the field "botanical\_name" and changed its type to "text".

421.

422. Example:

423.

424. PUT produce\_v2

425. {

426. "mappings": {

427. "properties": {

428. "botanical\_name": {

429. "type": "text"

430. },

431. "country\_of\_origin": {

432. "type": "text",

433. "fields": {

434. "keyword": {

435. "type": "keyword",

436. "ignore\_above": 256

437. }

438. }

439. },

440. "date\_purchased": {

441. "type": "date"

442. },

443. "description": {

444. "type": "text"

445. },

446. "name": {

447. "type": "text"

448. },

449. "organic": {

450. "type": "boolean"

451. },

452. "produce\_type": {

453. "type": "keyword"

454. },

455. "quantity": {

456. "type": "long"

457. },

458. "unit\_price": {

459. "type": "float"

460. },

461. "vendor\_details": {

462. "type": "object",

463. "enabled": false

464. }

465. }

466. }

467. }

468. Expected response from Elasticsearch:

469.

470. Elasticsearch creates a new index(produce\_v2) with the latest mapping. image

471.

472. If you check the mapping, you will see that the filed "botanical\_name" has been typed as text.

473.

474. View the mapping of produce\_v2:

475.

476. GET produce\_v2/\_mapping

477. Expected response from Elasticsearch: image image

478.

479. STEP 2: Reindex the data from the original index(produce\_index) to the one you just created(produce\_v2).

480.

481. POST \_reindex

482. {

483. "source": {

484. "index": "produce\_index"

485. },

486. "dest": {

487. "index": "produce\_v2"

488. }

489. }

490. Expected response from Elasticsearch:

491.

492. This request reindexes data from the produce\_index to the produce\_v2 index. The produce\_v2 index can now be used to run the requests that the client has specified.

493.

494. image

495.

496. Runtime Field

497. image image image

498.

499. Step 1: Create a runtime field and add it to the mapping of the existing index.

500.

501. Syntax:

502.

503. PUT Enter-name-of-index/\_mapping

504. {

505. "runtime": {

506. "Name-your-runtime-field-here": {

507. "type": "Specify-field-type-here",

508. "script": {

509. "source": "Specify the formula you want executed"

510. }

511. }

512. }

513. }

514. Example:

515.

516. PUT produce\_v2/\_mapping

517. {

518. "runtime": {

519. "total": {

520. "type": "double",

521. "script": {

522. "source": "emit(doc['unit\_price'].value\* doc['quantity'].value)"

523. }

524. }

525. }

526. }

527. Expected response from Elasticsearch:

528.

529. Elasticsearch successfully adds the runtime field to the mapping. image

530.

531. Step 2: Check the mapping:

532.

533. GET produce\_v2/\_mapping

534. Expected response from Elasticsearch:

535.

536. Elasticsearch adds a runtime field to the mapping(red box).

537.

538. image

539.

540. Note that the runtime field is not listed under "properties" object which includes the fields in our documents. This is because the runtime field "total" is not indexed!

541.

542. Step 3: Run a request on the runtime field to see it perform its magic!

543.

544. Please note that the following request does not aggregate the monthly expense here. We are running a simple aggregation request to demonstrate how runtime field works!

545.

546. The following request runs a sum aggregation against the runtime field total of all documents in our index.

547.

548. Syntax:

549.

550. GET Enter\_name\_of\_the\_index\_here/\_search

551. {

552. "size": 0,

553. "aggs": {

554. "Name your aggregations here": {

555. "Specify the aggregation type here": {

556. "field": "Name the field you want to aggregate on here"

557. }

558. }

559. }

560. }

561. Example:

562.

563. GET produce\_v2/\_search

564. {

565. "size": 0,

566. "aggs": {

567. "total\_expense": {

568. "sum": {

569. "field": "total"

570. }

571. }

572. }

573. }

574. Expected response from Elasticsearch:

575.

576. When this request is sent, a runtime field called "total" is created and calculated for documents within the scope of our request(entire index). Then, the sum aggregation is ran on the field "total" over all documents in our index.

577.

578. image

579.

580. The runtime field is only created and calculated when a request made on the runtime field is being executed. Runtime fields are not indexed so these do not take up disk space.

581.

582. We also did not have to reindex in order to add a new field to existing documents. For more information on runtime fields, check out this blog!

583.

584. Questions from the workshop

585. Q: If possible please explain the \_meta in mapping which was part of previous video.

586.

587. A: Of course! \_meta in mapping this question is referring to workshop part 4.

588.

589. image

590.

591. I should have just removed the \_meta part before I published this repo. Thank you for submitting a pull request on GitHub @radhakrishnaakamat!

592.

593. So the \_meta field was automatically created by the ml file data visualizer. This is a field where you can store any information regarding the index or the app for developers who are managing it. Think of this field as a place where you can include information regarding the app so developers have info necessary to debug.

594.

595. The \_meta field is optional and deleting the \_meta field will not affect the mapping in any way whatsoever. I am going to delete this field in my part 4 workshop repo as it has been causing a lot of confusion!

596.

597. Q: After you create a new mapping, how do you configure your ingest to use the new mapping?

598.

599. A: I should have asked for clarification as this question can be interpreted in many different ways.

600.

601. If I didn't interpret it correctly, please let me know via Twitter @LisaHJung and I will add the answer to this repo!

602.

603. If you were referring to a situation where you have an old index with outdated mapping that needed to be changed:

604.

605. Remember, we cannot change the mapping of an existing field. Even if you add a new field to a mapping, it only adds the new field to the list of field names and types. It does not add the new field to documents that have been indexed prior to adding a new field to the mapping.

606.

607. You must create a new index with the desired mapping, then reindex documents from the old index to the new one, and direct requests to the new index!

608.

**Part 6: Troubleshooting Errors**

- Troubleshoot errors from Part 1-5

1. GET common\_errors/\_doc/1

2. #404 - Not Found

3.

4. PUT common\_errors/\_doc

5. {

6. "source\_of\_error": "using the wrong syntax for PUT or POST indexing request"

7. }

8. #405 - Method Not Allowed as PUT method should also define ID. If does not want, use POST instead

9.

10. PUT common\_errors/\_doc/1

11. {

12. "source\_of\_error": "this works as now we define \_doc with ID 1"

13. }

14. #201 - Created

15.

16. POST common\_errors/\_update/1

17. {

18. "doc": {

19. "error": "405 Method Not Allowed"

20. "solution": "Look up the syntax of PUT and POST indexing requests and use the correct syntax."

21. }

22. }

23. #400 - Bad Request. No Comma between multiple fields ("error" and "solution")

24.

25. POST common\_errors/\_update/1

26. {

27. "doc": {

28. "error": "405 Method Not Allowed", //comma added

29. "solution": "Look up the syntax of PUT and POST indexing requests and use the correct syntax."

30. }

31. }

32. #200 - OK

33.

34. GET news\_headlines/\_search

35. {

36. "query": {

37. "range": {

38. "date":

39. "gte": "2015-06-20",

40. "lte": "2015-09-22"

41. }

42. }

43. }

44. #400 - Bad Request. Read documentation if unsure what the error is about. "Date" needs curly brackets. (Here the error is misleading)

45.

46. GET news\_headlines/\_search

47. {

48. "query": {

49. "range": {

50. "date": {

51. "gte": "2015-06-20",

52. "lte": "2015-09-22"

53. }

54. }

55. }

56. }

57. #200 - OK

58.

59. GET news\_headlines/\_search

60. {

61. "query": {

62. "multi\_match": {

63. "query": "party planning",

64. "fields": [

65. "headline",

66. "short\_description"

67. ]

68. },

69. "type": "phrase"

70. }

71. }

72. #400 - Bad Request. "type" should be positioned inside "multi\_match" query

73.

74. GET news\_headlines/\_search

75. {

76. "query": {

77. "multi\_match": {

78. "query": "party planning",

79. "type": "phrase",

80. "fields": [

81. "headline",

82. "short\_description"

83. ]

84. }

85. }

86. }

87. #200 - OK

88.

89. GET news\_headlines/\_search

90. {

91. "query": {

92. "match": {

93. "category": "ENTERTAINMENT",

94. "date":"2018-04-12"

95. }

96. }

97. }

98. #400 - Bad Request. Use filter or must clause instead (bool fields).

99.

100. GET news\_headlines/\_search

101. {

102. "query": {

103. "bool": {

104. "must": [

105. {

106. "match": {

107. "category": "ENTERTAINMENT"

108. }},

109. {

110. "match": {

111. "date": "2018-04-12"

112. }}

113. ]

114. }

115. }

116. }

117. #200 - OK (using must)

118.

119. GET news\_headlines/\_search

120. {

121. "query": {

122. "bool": {

123. "filter": [

124. {"match": {

125. "category": "ENTERTAINMENT"

126. }},

127. {

128. "match": {

129. "date": "2018-04-12"

130. }

131. }

132. ]

133. }

134. }

135. }

136. #200 - OK (using filter)

137.

138. GET news\_headlines/\_search

139. {

140. "aggs": {

141. "by\_category": {

142. "terms": {

143. "field": "category"

144. }

145. }

146. }

147. }

148. #200 - OK, but includes 10 hits by defaults and thus aggregations results only start at line 169. Let's try adding the size parameter.

149.

150. GET news\_headlines/\_search

151. {

152. "aggs": {

153. "size": 0,

154. "by\_category": {

155. "terms": {

156. "field": "category"

157. }

158. }

159. }

160. }

161. #400 - Bad Request. Size Parameter should be put OUTSIDE the aggregations (aggs) request.

162.

163. GET news\_headlines/\_search

164. {

165. "size": 0,

166. "aggs": {

167. "by\_category": {

168. "terms": {

169. "field": "category"

170. }

171. }

172. }

173. }

174. #200 - OK

175.

176. GET ecommnerce\_data/\_search

177. {

178. "size": 0,

179. "aggs": {

180. "transactions\_by\_8\_hrs": {

181. "date\_histogram": {

182. "field": "InvoiceDate",

183. "fixed\_interval": "8h"

184. }

185. }

186. }

187. }

188. #400 Bad Request - date\_histogram does not support any fields with "keyword" type. Let's check the mapping then.

189.

190. GET ecommnerce\_data/\_mapping

191. #Can see here that InvoiceDate is of type "keyword". From documentation, date\_histogram can only be either of type date (or date range values). Let's configure the mapping by create a new index with the desired mapping and reindex data from the original index to the new one.

192.

193. PUT ecommnerce\_data\_remapping

194. {

195. "mappings": {

196. "properties": {

197. "Country": {

198. "type": "keyword"

199. },

200. "CustomerID": {

201. "type": "long"

202. },

203. "Description": {

204. "type": "text"

205. },

206. "InvoiceDate": {

207. "type": "date",

208. "format": "M/d/yyyy H:m"

209. },

210. "InvoiceNo": {

211. "type": "keyword"

212. },

213. "Quantity": {

214. "type": "long"

215. },

216. "StockCode": {

217. "type": "keyword"

218. },

219. "UnitPrice": {

220. "type": "double"

221. }

222. }

223. }

224. }

225. #Created a new index with correct type (date) for date\_histogram aggregation. Now need to transfer all the existing data of ecoomnerce\_data to ecommnerce\_data\_remapping

226.

227. POST \_reindex

228. {

229. "source": {

230. "index": "ecommnerce\_data"

231. },

232. "dest": {

233. "index": "ecommnerce\_data\_remapping"

234. }

235. }

236. #Reindex done. Now lets try the date\_histogram aggregation again, using the new ecommnerce\_date\_remapping index.

237.

238. GET ecommnerce\_data\_remapping/\_search

239. {

240. "size": 0,

241. "aggs": {

242. "transactions\_by\_8\_hrs": {

243. "date\_histogram": {

244. "field": "InvoiceDate",

245. "fixed\_interval": "8h"

246. }

247. }

248. }

249. }

250. #200 - OK

251.

252. GET ecommerce\_data/\_search

253. {

254. "size": 0,

255. "aggs": {

256. "transactions\_per\_day": {

257. "date\_histogram": {

258. "field": "InvoiceDate",

259. "calendar\_interval": "day"

260. },

261. "daily\_revenue": {

262. "sum": {

263. "script": {

264. "source": "doc['UnitPrice'].value \* doc['Quantity'].value"

265. }

266. }

267. },

268. "number\_of\_unique\_customers\_per\_day": {

269. "cardinality": {

270. "field": "CustomerID"

271. }

272. }

273. }

274. }

275. }

276. #400 - Bad Request. This is aggregation within aggregation (first bucket is InvoiceDate @8hrs, second bucket is within the first bucket, get the daily\_revenue and cardinality). Hence need to enclose the second aggregation with another aggs parameter.

277.

278. GET ecommnerce\_data\_remapping/\_search

279. {

280. "size": 0,

281. "aggs": {

282. "transactions\_per\_day": {

283. "date\_histogram": {

284. "field": "InvoiceDate",

285. "calendar\_interval": "day"

286. },

287. "aggs": {

288. "daily\_revenue": {

289. "sum": {

290. "script": {

291. "source": "doc['UnitPrice'].value \* doc['Quantity'].value"

292. }

293. }

294. },

295. "number\_of\_unique\_customers\_per\_day": {

296. "cardinality": {

297. "field": "CustomerID"

298. }

299. }

300. }

301. }

302. }

303. }

304. #200 - OK

305.

\*\*\*