目录

§1 大数据时代的技术 hive

(1) hive 是基于 Hadoop 的一个数据仓库工具，可以将结构化的数据文件映射为一张数据库表，并提供完整的 sql 查询功能，可以将 sql 语句转换为 MapReduce 任务进行运行。其优点是学习成本低，可以通过类 SQL 语句快速实现简单的 MapReduce 统计，不必开发专门的 MapReduce 应用，十分适合数据仓库的统计分析。

(2) Hive 是建立在 Hadoop 上的数据仓库基础构架。它提供了一系列的工具，可以用来进行数据提取转化加载（ETL），这是一种可以存储、查询和分析存储在 Hadoop 中的大规模数据的机制。Hive 定义了简单的类 SQL 查询语言，称为 HQL，允许熟悉 SQL 的用户查询数据。同时，这个语言也允许熟悉 MapReduce 的开发者开发自定义的 mapper 和 reducer 函数来处理内建的 mapper 和 reducer 无法完成的复杂的分析工作。

使用 hive 的命令行接口，感觉很像操作关系数据库，但是 hive 和关系数据库还是有很大的不同，下面比较下 hive 与关系数据库的区别，具体如下:

(1) hive 和关系数据库存储文件的系统不同，hive 使用的是 hadoop 的 HDFS（hadoop 的分布式文件系统），关系数据库则是服务器本地的文件系统;

(2) hive 使用的计算模型是 mapreduce，而关系数据库则是自己设计的计算模型，一般是基于连接（Join）操作，索引（B^+ 树上的查询);

(3) 关系数据库都是为实时查询的业务进行设计的，而 hive 则是为海量数据做数据挖掘设计的，实时性很差；实时性的区别导致 hive 的应用场景和关系数据库有很大的不同，因此，hive 一般是用于统计分析的，商业推荐系统中，实时性的要求没那么高。

(4) Hive 很容易扩展自己的存储能力和计算能力，这个是继承 hadoop 的，而关系数据库在这个方面要比数据库差很多。

§1.1 hive 的技术架构

由图 1-1 可知，hdfs 和 mapreduce 是 hive 架构的根基。Hive 架构包括如下组件：CLI（command line interface）、JDBC/ODBC、Thrift Server、WEB GUI、metastore 和 Driver(Complier、Optimizer 和 Executor)，这些组件可以分为两大类：服务端组件和客户端组件。

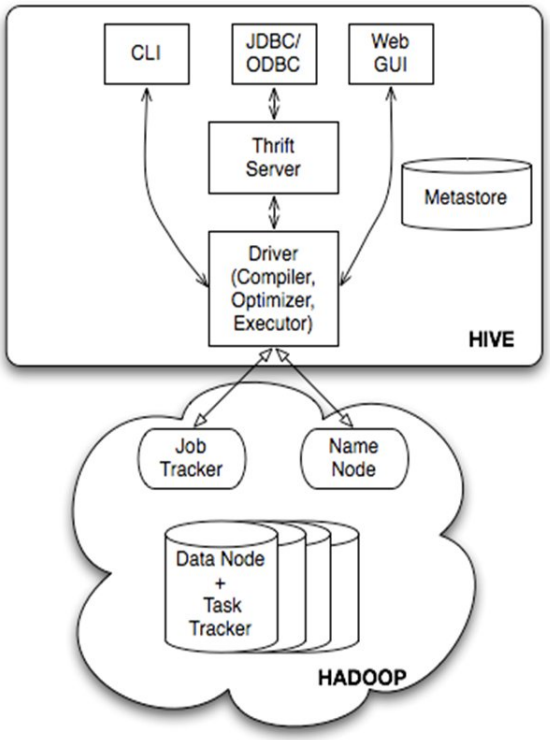

图 1-1 hive 基础架构图

Driver 组件：该组件包括 Complier、Optimizer 和 Executor，它的作用是将我们写的 HiveQL（类 SQL）语句进行解析、编译优化，生成执行计划，然后调用底层的 mapreduce 计算框架。

Metastore 组件：元数据服务组件，这个组件存储 hive 的元数据，hive 的元数据存储在关系数据库里，hive 支持的关系数据库有 derby、mysql。元数据对于 hive 十分重要，因此 hive 支持把 metastore 服务独立出来，安装到远程的服务器集群里，从而解耦 hive 服务和 metastore 服务，保证 hive 运行的健壮性。

Thrift 服务：thrift 是 facebook 开发的一个软件框架，它用来进行可扩展且跨语言的服务的开发，hive 集成了该服务，能让不同的编程语言调用 hive 的接口。客户端组件：

CLI：command line interface，命令行接口。

Thrift 客户端：上面的架构图里没有写上 Thrift 客户端，但是 hive 架构的许多客户端接口是建立在 thrift 客户端之上，包括 JDBC 和 ODBC 接口。

WEBGUI：hive 客户端提供了一种通过网页的方式访问 hive 所提供的服务。这个接口对应 hive 的 hwi 组件（hive web interface），使用前要启动 hwi 服务。

Hive 的 metastore 组件是 hive 元数据集中存放地。Metastore 组件包

括两个部分：metastore 服务和后台数据的存储。后台数据存储的介质就是关系数据库，例如 hive 默认的嵌入式磁盘数据库 derby，还有 mysql 数据库。Metastore 服务是建立在后台数据存储介质之上，并且可以和 hive 服务进行交互的服务组件，默认情况下，metastore 服务和 hive 服务是安装在一起的，运行在同一个进程当中。也可以把 metastore 服务从 hive 服务里剥离出来，metastore 独立安装在一个集群里，hive 远程调用 metastore 服务，这样可以把元数据这一层放到防火墙之后，客户端访问 hive 服务，就可以连接到元数据这一层，从而提供了更好的管理性和安全保障。使用远程的 metastore 服务，可以让 metastore 服务和 hive 服务运行在不同的进程里，这样也保证了 hive 的稳定性，提升了 hive 服务的效率。

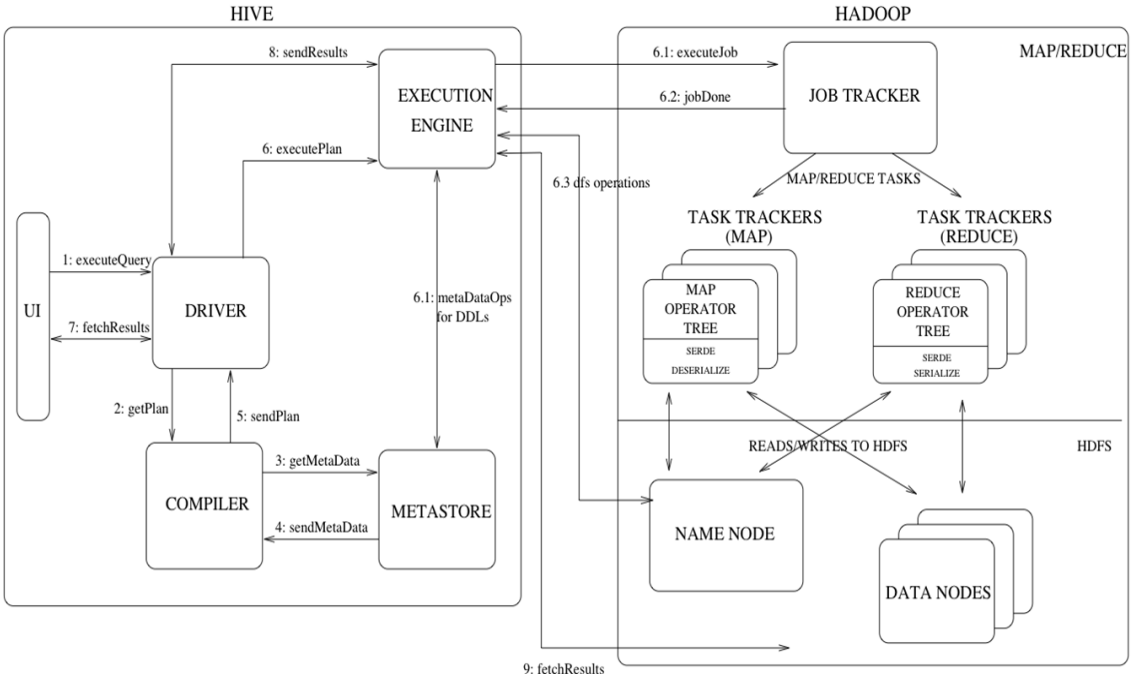

图 1-2 hive 执行流程

§1.2 hive 与关系数据库的区别

关系数据库里，表的加载模式是在数据加载时候强制确定的（表的加载模式是指数据库存储数据的文件格式），如果加载数据时候发现加载的数据不符合模式，关系数据库则会拒绝加载数据，这个就叫“写时模式”，写时模式会在数据加载时候对数据模式进行检查校验的操作。Hive 在加载数据时候和关系数据库不同，hive 在加载数据时候不会对数据进行检查，也不会更改被加载的数据文件，而检查数据格式的操作是在查询操作时候执行，这种模式叫“读时模式”。在实际应用中，写时模式在加载数据时候会对列进行索引，对数据进行压缩，因此加载数据的速度很慢，但是当数据加载好了，我们去查询数据的时候，速度很快。但是

当我们的数据是非结构化，存储模式也是未知时候，关系数据操作这种场景就麻烦多了，这时候 hive 就会发挥它的优势。

关系数据库一个重要的特点是可以对某一行或某些行的数据进行更新、删除操作，hive 不支持对某个具体行的操作，hive 对数据的操作只支持覆盖原数据和追加数据。Hive 也不支持事务和索引。更新、事务和索引都是关系数据库的特征，这些 hive 都不支持，也不打算支持，原因是 hive 的设计是海量数据进行处理，全数据的扫描是它的工作常态，针对某些具体数据进行操作的效率是很差的，对于数据的添加操作，hive 是通过查询将原表的数据进行转化最后存储在新表里，这和传统数据库的更新操作有很大不同。

Hive 也可以在 hadoop 做实时查询上做一份自己的贡献，那就是和 hbase 集成，hbase 可以进行快速查询，但是 hbase 不支持类 SQL 的语句，那么此时 hive 可以给 hbase 提供 sql 语法解析的外壳，可以用类 sql 语句操作 hbase 数据库。今天的 hive 就写到这里，关于 hive 我打算一共写三篇文章，这是第一篇，下一篇主要讲 hive 支持的数据模型，例如：数据库（database）、表（table）、分区（partition）和桶（bucket），还有 hive 文件存储的格式，还有 hive 支持的数据类型。第三篇文章就会讲到 hiveQL 的使用、以及结合 mapreduce 查询优化的技术和自定义函数，以及我们现在在公司项目里运用 hive 的实例。马云在退休的时候说互联网现在进入了大数据时代，大数据是现在互联网的趋势，而 hadoop 就是大数据时代里的核心技术，但是 hadoop 和 mapreduce 操作专业型太强，所以 facebook 在这些的基础上开发了 hive 框架，毕竟世界上会 sql 的人比会 java 的人多的多，hive 是可以说是学习 hadoop 相关技术的一个突破口，哪些自立于投身 hadoop 技术开发的童鞋们，可以先从 hive 开始哦。

§1.3 hive 的 join 类型简介

作为数据分析中经常进行的 join 操作，传统 DBMS 数据库已经将各种算法优化到了极致，而对于 hadoop 使用的 mapreduce 所进行的 join 操作，去年开始也是有各种不同的算法论文出现，讨论各种算法的适用场景和取舍条件，本文讨论 hive 中出现的几种 join 优化，然后讨论其他算法实现，希望能给使用 hadoop 做数据分析的开发人员提供一点帮助。

Facebook 今年在 yahoo 的 hadoop summit 大会上做了一个关于最近两个版本的 hive 上所做的一些 join 的优化，其中主要涉及到 hive 的几个关键特性：值分区,hash 分区,map join,index。

§1.3.1 Common Join

最为普通的 join 策略，不受数据量的大小影响，也可以叫做 reduce side join，最没效率的一种 join 方式. 它由一个 mapreduce job 完成. 首先将大表和小表分别进行 map 操作，在 map shuffle 的阶段每一个 **map output key** 变成了 **table-name-tag-prefix + join-column-value**，但是在进行 partition 的时候它仍然只使用 **join-column-value** 进行 hash.

每一个 reduce 接受所有的 map 传过来的 split，在 reducce 的 shuffle 阶段，它将 **map output key** 前面的 **table-name-tag-prefix** 给舍弃掉进行比较. 因为 reduce 的个数可以由小表的大小进行决定，所以对于每一个节点的 reduce 一定可以将小表的 split 放入内存变成 hashtable，然后将大表的每一条记录进行一条一条的比较.

§1.3.2 Map Join

Map Join 的计算步骤分两步，将小表的数据变成 hashtable，广播到所有的 map 端，将大表的数据进行合理的切分，然后在 map 阶段的时候用大表的数据一行一行的去探测 (probe) 小表的 hashtable. 如果 join key 相等，就写入 HDFS.Map Join 之所以叫做 Map Join 是因为它所有的工作都在 map 端进行计算.

hive 在 Map Join 上做了几个优化: hive 0.6 的时候默认认为写在 select 后面的是大表，前面的是小表，或者使用 **+mapjoin(map_table)** 提示进行设定.hive 0.7 的时候这个计算是自动化的，它首先会自动判断哪个是小表，哪个是大表，这个参数由（**hive.auto.convert.join=true**）来控制。小表的大小由（**hive.smalltable.filesize**）参数控制（默认是 25M），当小表超过这个大小，hive 会默认转化成 Common Join。

首先小表的 Map 阶段它会将自己转化成 MapReduce Local Task，然后从 HDFS 上取小表的所有数据，将自己转化成 Hashtable file 并压缩打包放入 DistributedCache 里面.

目前 hive 的 map join 有几个限制，一个是它打算用 BloomFilter 来实现 hashtable，BloomFilter 大概比 hashtable 省 8-10 倍的内存，但是 BloomFilter 的大小比较难控制. 现在 DistributedCache 里面 hashtable 默认的复制是 3 份，对于一个有 1000 个 map 的大表来说，这个数字太小，大多数 map 操作都等着 DistributedCache 复制.

§1.3.3 Bucket Map Join

hive 建表的时候支持 hash 分区通过指定 clustered by(col_name,xxx) into number_buckets，当连接的两个表的 join key 就是 bucket column 的时候，就可以通过 **hive.optimize.bucketmapjoin=true** 来控制 hive 执行 **bucket map join** 了，需要注意的是小表的 number_buckets 必须是大表的倍数．无论多少个表进行连接这个条件都必须满足。（其实如果都按照 2 的指数倍来分 bucket，大表也可以是小表的倍数，不过这中间需要多计算一次，对 int 有效，long 和 string 不清楚）。

Bucket Map Join 执行计划分两步，第一步先将小表做 map 操作变成 hashtable，然后广播到所有大表的 map 端，大表的 map 端接受了 number_buckets 个小表的 hashtable 并不需要合成一个大的 hashtable，直接可以进行 map 操作，map 操作会产生 number_buckets 个 split，每个 split 的标记跟小表的 hashtable 标记是一样的，在执行 projection 操作的时候，只需要将小表的一个 hashtable 放入内存即可，然后将大表的对应的 split 拿出来进行判断，所以其内存限制为小表中最大的那个 hashtable 的大小．

Bucket Map Join 同时也是 **Map Side Join** 的一种实现，所有计算都在 Map 端完成，没有 Reduce 的都被叫做 **Map Side Join**，**Bucket** 只是 **hive** 的一种 **hash partition** 的实现，另外一种当然是值分区**:create table a (xxx) partition by (col_name)**

一般 hive 中两个表不一定会有同一个 partition key，即使有也不一定会是 join key．所以 hive 没有这种基于值的 map side join，hive 中的 list partition 主要是用来过滤数据的而不是分区．两个主要参数为 (hive.optimize.cp = true 和 hive.optimize.pruner=true)．

hadoop 源代码中默认提供 **Map Side Join** 的实现，可以在 hadoop 源码的 src/contrib/data_join/src 目录下找到相关的几个类．其中 TaggedMapOutput 即可以用来实现 hash 也可以实现 list，看你自己决定怎么分区． Hadoop Definitive Guide 第 8 章关于 **Map Side Join** 和 **side data distribution** 章节也有一个例子示例怎样实现值分区的 map side join．

§1.3.4 Where 条件查询

当使用 hive 进行 where 查询时，例如*'select * from cable.testData where cardId=80000364;'* 由于 hive 默认的，每个 MapTask 的输入块大小为 256MB，因此对 14664.9MB 的文件，大概需要 57 个 MapTask，由于 where

条件查询，不需要进行 reduce 运算，因此 ReduceTask 数目为 0，此时可以通过设置 *set mapred.max.split.size=128000000;*，对上述例子，可生成 112 个 MapTask。在两亿多条记录中（共 2,3235,9109 条记录）查询，耗费时间 67S（57 个 MapTask 时，112 个 MapTask 耗费时间约 79S）。

§1.4 Hive 提供的索引功能

Hive 提供有限的索引功能，这不像传统的关系型数据库那样有“键 (key)”的概念，用户可以在某些列上创建索引来加速某些操作，给一个表创建的索引数据被保存在另外的表中。Hive 的索引功能现在还相对较晚，提供的选项还较少。但是，索引被设计为可使用内置的可插拔的 java 代码来定制，用户可以扩展这个功能来满足自己的需求。

当然不是所有的查询都会受惠于 Hive 索引。用户可以使用 EXPLAIN 语法来分析 HiveQL 语句是否可以使用索引来提升用户查询的性能。像 RDBMS 中的索引一样，需要评估索引创建的是否合理，毕竟，索引需要更多的 hdfs 磁盘空间，并且创建维护索引也会有一定的代价。用户必须要权衡从索引得到的好处和代价。先建立一个临时表，避免覆盖我精心准备的表，*create table cable.TempTable as select cardId,Date,Time as text from cable.testData;* 在 TempTable 的 cardId 列上，建立索引，耗费时间：200.727S，下面在 cardId 列上建立一个索引：

```
create index TempTable_index on table TempTable(cardId)
as 'org.apache.hadoop.hive.ql.index.compact.CompactIndexHandler'
with deferred rebuild;
alter index TempTable_index on TempTable rebuild;
--现在建立了索引表TempTable_index
--通过下述命令，可以查询TempTable_index内容
select * from cable.cable__temptable_temptable_index__ limit 10;
-- 下面是使用索引的过程,/tmp/table02_index_data为hdfs上目录;
Insert overwrite directory "/tmp/table02_index_data"
select `_bucketname`,`_offsets` from cable.cable__temptable_temptable_index__
where cardId = 80000000;
Set hive.index.compact.file=/tmp/table02_index_data;
Set hive.optimize.index.filter=false;
Set hive.input.format =
org.apache.hadoop.hive.ql.index.compact.HiveCompactIndexInputFormat;
select * from cable.TempTable where cardId=80000000;
```

使用后，查询耗时约 13.246S，而不使用索引时，耗时约 61.907S，但执行 *select `_bucketname`,`_offsets` from cable.cable__temptable_temptable_index__ where id = 80000000;* 这一步耗时约 51.635S，总体上得不偿失。在本示例中，在 employees 上创建了名为 employees_index 的索引，索引数据存放在 employees_index_table 索引表中，WITH DEFERRED REBUILD 表明创建一个空索引，可以在之后使用如下语句创建索引数据：

```
ALTER INDEX employees_index ON TABLE employees
PARTITION(country = 'US') REBUILD;
```

PARTITIONED BY 表明只对某个分区创建索引，若没有该选项则表示对所有分区都创建索引，另外要注意的是 index 的分区索引默认是和表的分区一致的，也不能对视图 VIEW 创建索引。AS 'org.apache.hadoop.hive.ql.index.compact.CompactInde 表示使用 Apache 的 org.apache.hadoop.hive.ql.index.compact.CompactIndexHandler 作为创建索引的 handler，当然也可以使用第三方的实现类或其他的实现类。当然也可以对其他字段创建索引。

Bitmap 位图索引:Hive v0.80 添加了一个内置的 bitmap 位图索引。Bitmap 位图索引通常适用于只有少数不同值的列。现在我们修改上一个索引示例为位图索引:

```
CREATE INDEX employees_index
ON TABLE employees (country)
AS 'BITMAP'
WITH DEFERRED REBUILD
IDXPROPERTIES ('creator' = 'me','created_at' = 'some_time')
IN TABLE employees_index_table
PARTITIONED BY (country,name)
COMMENT 'Employees indexed by country and name.';
```

重建索引：如果用户在创建索引时指定 WITH DEFERRED REBUILD 关键字，那么开始时是一个空索引。我们在任何时候使用 ALTER INDEX 语句来创建或重建索引:

```
ALTER INDEX employees_index ON TABLE employees
PARTITION(country = 'US')
REBUILD;
```

Hive 没有提供一个内置的在数据变更时自动触发创建索引的机制，故用户需要自己通过 ALTER INDEX 语句来创建索引；另外，重建索引操作是一个原子操作，因此当 rebuild 失败时，已构建的索引也无法使用。查看索引：用户可以查看某个表上的所有的索引:

```
SHOW FORMATTED INDEX ON employees;
SHOW FORMATTED INDEXES ON employees;
```

删除索引：在删除一个索引的时候，也会同时删除索引数据所在的表，如:

```
DROP INDEX IF EXISTS employees_index ON TABLE employees;
```

§2 ElasticSearch

Elasticsearch 是个开源分布式搜索引擎，它的特点有：分布式，零配置，自动发现，索引自动分片，索引副本机制，Restful 风格接口，多数据源，自动搜索负载等。

§2.1 What's a mapping?

A mapping not only tells ES what is in a field…it tells ES what terms are indexed and searchable.

A mapping is composed of one or more ‘analyzers’, which are in turn built with one or more ‘filters’. When ES indexes your document, it passes the field into each specified analyzer, which pass the field to individual filters.

Filters are easy to understand: a filter is a function that transforms data. Given a string, it returns another string (after some modifications). A function that converts strings to lowercase is a good example of a filter.

An analyzer is simply a group of filters that are executed in-order. So an analyzer may first apply a lowercase filter, then apply a stop-word removal filter. Once the analyzer is run, the remaining terms are indexed and stored in ES.

Which means a mapping is simply a series of instructions on how to transform your input data into searchable, indexed terms.

使用 **curl -X PUT http://localhost:9200/test/item/1 -d '{"name":"zach", "description": "A Pretty cool guy."}'** 时，ES 会生成如下的 mapping:

```
mappings: {
   item: {
      properties: {
         description: { type: string }
         name: { type: string }
      }
   }
}
```

Above, ES guessed “string” as the mapping for ``description''. When ES implicitly creates a “string” mapping, it applies the

default global analyzer. Unless you' ve changed it, the default analyzer is the Standard Analyzer. This analyzer will apply the standard token filter, lowercase filter and stop token filter to your field. We lost a word ("A") capitalization and punctuation. Importantly, even though ES continues to store the original document in it' s original form, the only parts that are searchable are the parts that have been run through an analyzer.So, don' t think of mappings as data-types , think of them as instructions on how you will eventually search your data. If you care about stop-words like "a" , you need to change the analyzer so that they aren' t removed.

§2.2 Constructing more complicated mapping

Setting up proper analyzers in ES is all about thinking about the search query. You have to provide instructions to ES about the appropriate transformations so you can search intelligently.

The first thing that happens to an input query is tokenization , breaking an input query into smaller chunks called tokens. There are several tokenizers available, which you should explore on your own when you get a chance. The Standard tokenizer is being used in this example, which is a pretty good tokenizer for most English-language search problems. You can query ES to see how it tokenizes a sample sentence:

curl -X GET "http://localhost:9200/test/_analyze?tokenizer=standard &pretty=true" -d 'The quick brown fox is jumping over the lazy dog.'

Ok, so our input query has been turned into tokens. Referring back to the mapping, the next step is to apply filters to these tokens. In order, these filters are applied to each token: Standard Token Filter, Lowercase Filter, ASCII Folding Filter.

curl -X GET "http://localhost:9200/test/_analyze?filter=standard &pretty=true" -d 'The quick brown fox is jumping over the lazy dog.'

```
"partial":{
     "search_analyzer":"full_name",
     "index_analyzer":"partial_name",
     "type":"string"
```

```
}
```

As you can see, we specify both a search and index analyzer. Huh? These two separate analyzers instruct ES what to do when it is indexing a field, or searching a field. But why are these needed?

The index analyzer is easy to understand. We want to break up our input fields into various tokens so we can later search it. So we instruct ES to use the new partial_name analyzer that we built, so that it can create nGrams for us.

The search analyzer is a little trickier to understand, but crucial to getting good relevance. Imagine querying for "Race". We want that query to match "race", "races" and "racecar". When searching, we want to make sure ES eventually searches with the token "race". The full_name analyzer will give us the needed token to search with.

If, however, we used the partial_name nGram analyzer, we would generate a list of nGrams as our search query. The search query "Race" would turn into ["ra", "rac", "race"].Those tokens are then used to search the index. As you might guess, "ra" and "rac" will match a lot of things you don't want, such as "racket" or "ratify" or "rapport".

So specifying different index and search analyzers is critical when working with things like ngrams. Make sure you always double check what you expect to query ES with…and what is actually being passed to ES.

§2.3 depth into ElasticSearch

当使用如下命令搜索时:

```
curl -XPOST "http://namenode:9200/_search" -d'
{
    "query": {
      "match_all" : {}
    },
    "filter" : {
      "term" : { "director" : "Francis Ford Coppola"}
    }
}'
```

没有任何结果，而使用

```
curl -XPOST "http://namenode:9200/_search" -d'
{
    "query": {
      "match_all" : {}
    },
    "filter" : {
      "term" : { "year" : "1962"}
    }
}'
```

却能输出结果，What's going on here? We've obviously indexed two movies with "Francis Ford Coppola" as director and that's what we see in search results as well. Well, while ES has a JSON object with that data that it returns to us in search results in the form of the _source property .

When we index a document with ElasticSearch it (simplified) does two things: it stores the original data untouched for later retrieval in the form of _source and it indexes each JSON property into one or more fields in a Lucene index. During the indexing it processes each field according to how the field is mapped. If it isn't mapped , default mappings depending on the fields type (string, number etc) is used.

As we haven't supplied any mappings for our index, Elastic-Search uses the default mappings for the director field. This means that in the index the director fields value isn't **"Francis Ford Coppola"**. Instead it's something like **["francis", "ford", "coppola"]**. We can verify that by modifying our filter to instead match "francis" (or "ford" or "coppola").

So, what to do if we want to filter by the exact name of the director? We modify how it's mapped. There are a number of ways to add mappings to ElasticSearch, through a configuration file, as part of a HTTP request that creates and index and by calling the _mapping endpoint. Using the last approach, we could fix the above issue by adding a mapping for the "director" field , to instruct ElasticSearch not to analyze (tokenize etc.) the field at all.

```
curl -X PUT namenode:9200/movies/movie/_mapping -d'
{
   "movie": {
      "properties": {
         "director": {
            "type": "string",
```

```
            "index": "not_analyzed"
          }
        }
    }
}'
```

In many cases it's not possible to modify existing mappings. Often the easiest work around for that is to create a new index with the desired mappings and re-index all of the data into the new index. The second problem is that, even if we could add it, we would have limited our ability to search in the director field. That is, while a search for the exact value in the field would match we wouldn't be able to search for single words in the field.

Luckily, there's a simple solution to our problem. We add a mapping that upgrades the field to a multi field. What that means is that we'll map the field multiple times for indexing. Given that one of the ways we map it match the existing mapping both by name and settings that will work fine and we won't have to create a new index.

```
curl -XPUT "namenode:9200/movies/movie/_mapping" -d'
{
    "movie": {
        "properties": {
            "director": {
                "type": "multi_field",
                "fields": {
                    "director": {"type": "string's"},
                    "original": {"type" : "string", "index" : "not_analyzed"}
                }
            }
        }
    }
}'
```

§2.4 ElasticSearch 集群设置

This will print the number of open files the process can open on startup. Alternatively, you can retrieve the max_file_descriptors for each node using the Nodes Info API, with:

curl -XGET 'namenode:9200/_nodes/process?pretty=true'，下面是删除 ES 上名为 jdbc 的 index 的 Restfule 命令:

curl -XDELETE 'http://namenode:9200/jdbc/'

配置文件在 **{$ES_HOME}/config** 文件夹下，**elasticsearch.yml** 和 **logging.yml**，修改 **elasticsearch.yml** 文件中的 cluster.name，当集群名称

相同时，每个 ES 节点将会搜索它的伙伴节点，因此必须保证集群内每个节点的 cluster.name 相同，下面是关闭 ES 集群的 Restful 命令：

```
# 关闭集群内的某个ES节点'_local'
$ curl -XPOST 'http://namenode:9200/_cluster/nodes/_local/_shutdown'
# 关闭集群内的全部ES节点
$ curl -XPOST 'http://namenode:9200/_shutdown'
```

注意，如果一台机器上不止一个 ES 在运行，那么通过**./bin/elastic-search** 开启的 ES 的 http_address 将会使用 9200 以上的接口（形如 9201,9202,···），而相应的 transport_address 也递增（形如: 9301,9302,···），因此，为使用 9200 端口，可使用上述命令关闭其它 ES 进程，可通过 conf 目录下的 log 文件来查看某些端口是否被占用。**elasticsearch.yml** 文件存在如下配置信息：

(1) node.master: true,node.data: true，允许节点存储数据，同时作为主节点；

(2) node.master: true,node.data: false，节点不存储数据，但作为集群的协调者；

(3) node.master: false,node.data: true，允许节点存储数据，但不作为主节点；

(4) node.master: false,node.data: false，节点不存储数据，也不作为协调者，但作为搜索任务的一个承担者；

(5) cluster.name: HadoopSearch，node.name: "ES-Slave-02",HadoopSearch 必须相同，但 node.name 每个节点可以自由设置；

如想将 ES 作为一个服务，需要从 github 上下载 elasticsearch-servicewrapper，然后调用 chkconfig，将其添加到/etc/rc[0~6].d/中。

```
curl -L https://github.com/elasticsearch/elasticsearch-servicewrapper/
archive/master.zip > master.zip
unzip master.zip
cd elasticsearch-servicewrapper-master/
mv service /opt/elasticsearch/bin
/opt/elasticsearch/bin/service/elasticsearch install
## 如果想卸载该服务调用:
/opt/elasticsearch/bin/service/elasticsearch remove
## 如果想让ES开机启动
chkconfig elasticsearch on
## 如果想现在开启ES服务
service elasticsearch start
```

配置完后，可通过 *curl -X GET 'http://192.168.50.75:9200/_cluster/nodes?pretty'* 命令，查询集群下的节点信息。

为连接 hive 与 ES，运行 hive 后，在 hive 命令行内执行 **add jar /opt/elasticsearch-hadoop-1.3.0/dist/elasticsearch-hadoop-1.3.0.jar;** 或者 hdfs 上的 jar 包:**add jar hdfs://namenode:9000/elasticsearch-hadoop-1.3.0.jar** 可加载 elasticsearch-hadoop 插件，使用该插件的具体操作如下:

```
DROP TABLE IF EXISTS artist_1;
CREATE EXTERNAL TABLE artists_1 (
  cardid STRING, date STRING, time STRING)
STORED BY 'org.elasticsearch.hadoop.hive.ESStorageHandler'
TBLPROPERTIES('es.resource' = 'liubo/artists/',
              'es.host' = '192.168.50.75',
              'es.mapping.names' = 'text:time'
);
-- 集群下应使用'192.168.50.75'，而非'localhost'(es-hadoop的默认值)
-- insert data to Elasticsearch from another hive table
INSERT OVERWRITE TABLE artists_1
SELECT * FROM cable.temptable;
```

下面的代码是将 Mysql 中的表导入到 ES 中，建立名为 jdbc 的 index，表名称为 jiangsu。

```
curl -XPUT 'localhost:9200/_river/jiangsu/_meta' -d '{
    "type" : "jdbc",
    "jdbc" : {
        "driver" : "com.mysql.jdbc.Driver",
        "url" : "jdbc:mysql://192.168.50.75:3306/jsyx",
        "user" : "root",
        "password" : "123456",
        "sql" : "select * from jiangsu"
    },
    "index" : {
        "index" : "jdbc",
        "type" : "jiangsu"
    }
}'
```

§2.5 ES 性能优化

一个 Elasticsearch 节点会有多个线程池，但重要的是下面四个:

- 索引（index）: 主要是索引数据和删除数据操作（默认是 cached 类型）;
- 搜索（search）: 主要是获取，统计和搜索操作（默认是 cached 类型）;
- 批量操作（bulk）: 对索引的批量操作，现在尚且不清楚它是不是原子性的，如果是原子的，则放在 MapReduce 里是没有问题的;

- 更新（refresh）：主要是更新操作，如当一个文档被索引后，何时能够通过搜索看到该文档；

在生成索引的过程中，需要修改如下配置参数：

- index.store.type： mmapfs。因为内存映射文件机制能更好地利用 OS 缓存；
- indices.memory.index_buffer_size： 30% 默认值为 10%，表示 10% 的内存作为 indexing buffer；
- index.translog.flush_threshold_ops： 50000，当写日志数达到 50000 时，做一次同步；
- index.refresh_interval:30s，默认值为 1s，新建的索引记录将在 1 秒钟后查询到；

```
curl -XPUT 'http://namenode:9200/hivetest/?pretty' -d '{
    "settings" : {
       "index" : {
       "refresh_interval" : "30s",
       "index.store.type": "mmapfs",
       "indices.memory.index_buffer_size": "30%",
       "index.translog.flush_threshold_ops": "50000"
        }
    }
}'
```