系　理

世界有很多重要的　　系：博客帖子有一些　，　行　有多次交易　，客　有多个　行　，有多个　明　，文件目　有多个文件和子目　。

系型数据　被明　　—　不意外—用来　行　　系管理：

- 个　体（或 行，在　系世界中）可以被 主　唯一　。
- 体　　化　　（　式）。唯一　体的数据只存　一次，而相　体只存　它的主　。只能在一个具体位置修改　个　体的数据。
- 体可以　行　　，可以跨　体搜索。
- 个　体的　化是 原子的，　一致的，　隔　的，　和 持久的。（可以在 *ACID Transactions* 中看更多　。）
- 大多数　系数据　支持跨多个　体的 ACID 事　。

但是　系型数据　有其局限性，包括　全文　索有限的支持能力。　　体　　消耗是很昂　的，　的越多，消耗就越昂　。特　是跨服　器　行　体　　成本　其昂　，基本不可用。　　但　个的服器上又存在数据量的限制。

Elasticsearch　，和大多数　NoSQL　数据　似，是扁平化的。索引是独立文　的集合体。　文　是否匹配搜索　求取决于它是否包含所有的所需信息。

Elasticsearch 中　个文　的数据　更是 ACIDic 的，　而　及多个文　的事　不是。当一个事　部分失　，无法回　索引数据到前一个状　。

扁平化有以下　：

- 索引　程是快速和无　的。
- 搜索　程是快速和无　的。
- 因　个文　相互都是独立的，大　模数据可以在多个　点上　行分布。

但　系　然非常重要。某些　候，我　需要　小扁平化和　世界　系模型的差　。以下四　常用的方法，用来在 Elasticsearch 中　行　系型数据的管理：

- Application-side joins
- Data denormalization
- Nested objects
- Parent/child relationships

通常都需要　合其中的某几个方法来得到最　的解决方案。

用　接

我　通　在我　的　用程序中　　接可以（部分）模　系数据　。　例如，比方　我　正在　用　和他　的博客文章　行索引。在　系世界中，我　会　来操作：

```
PUT /my_index/user/1 ①
{
  "name":     "John Smith",
  "email":    "john@smith.com",
  "dob":      "1970/10/24"
}

PUT /my_index/blogpost/2 ①
{
  "title":    "Relationships",
  "body":     "It's complicated...",
  "user":     1 ②
}
```

① 个文 的 index, type, 和 id 一起 造成主 。

② blogpost 通 用 的 id 接到用 。index 和 type 并不需要因 在我 的 用程序中已 硬 。

通 用 的 ID 1 可以很容易的 到博客帖子。

```
GET /my_index/blogpost/_search
{
  "query": {
    "filtered": {
      "filter": {
        "term": { "user": 1 }
      }
    }
  }
}
```

了 到用 叫做 John 的博客帖子，我 需要 行 次 ： 第一次会 所有叫做 John 的用 从而取他 的 ID 集合，接着第二次会将 些 ID 集合放到 似于前面一个例子的 ：

```
GET /my_index/user/_search
{
  "query": {
    "match": {
      "name": "John"
    }
  }
}

GET /my_index/blogpost/_search
{
  "query": {
    "filtered": {
      "filter": {
        "terms": { "user": [1] }  ①
      }
    }
  }
}
```

① 行第一个 得到的 果将填充到 terms 器中。

用 接的主要 点是可以 数据 行 准化 理。只能在 user 文 中修改用 的名称。 点是，了在搜索 接文 ，必 行 外的 。

在 个例子中，只有一个用 匹配我 的第一个 ，但在 世界中，我 可以很 易的遇到数以百万的叫做 John 的用 。包含所有 些用 的 IDs 会 生一个非常大的 ， 是一个数百万 的 。

方法 用于第一个 体（例如，在 个例子中 user ）只有少量的文 的情况，并且最好它很少改 。 将允 用程序 果 行 存，并避免 常 行第一次 。

非 化 的数据

使用 Elasticsearch 得到最好的搜索性能的方法是有目的的通 在索引 行非 化 denormalizing 。 个文 保持一定数量的冗余副本可以在需要 避免 行 。

如果我 希望能 通 某个用 姓名 到他写的博客文章，可以在博客文 中包含 个用 的姓名：

```
PUT /my_index/user/1
{
  "name":     "John Smith",
  "email":    "john@smith.com",
  "dob":      "1970/10/24"
}

PUT /my_index/blogpost/2
{
  "title":    "Relationships",
  "body":     "It's complicated...",
  "user":     {
    "id":       1,
    "name":     "John Smith" ①
  }
}
```

① 部分用 的字段数据已被冗余到 blogpost 文 中。

在，我 通 次 就能 通 relationships 到用 John 的博客文章。

```
GET /my_index/blogpost/_search
{
  "query": {
    "bool": {
      "must": [
        { "match": { "title":     "relationships" }},
        { "match": { "user.name": "John"          }}
      ]
    }
  }
}
```

数据非 化的 点是速度快。因 个文 都包含了所需的所有信息，当 些信息需要在 行匹配，并不需要 行昂 的 接操作。

字段折

一个普遍的需求是需要通 特定字段 行分 。例如我 需要按照用 名称 分 返回最相 的博客文章。按照用 名分 意味着 行 terms 聚合。 能 按照用 整体 名称 行分 ，名称字段 保持 not_analyzed 的形式， 具体 明参考 [aggregations-and-analysis]：

```
PUT /my_index/_mapping/blogpost
{
  "properties": {
    "user": {
      "properties": {
        "name": { ①
          "type": "string",
          "fields": {
            "raw": { ②
              "type":  "string",
              "index": "not_analyzed"
            }
          }
        }
      }
    }
  }
}
```

① `user.name` 字段将用来 行全文 索。

② `user.name.raw` 字段将用来通 `terms` 聚合 行分 。

然后添加一些数据:

```
PUT /my_index/user/1
{
  "name": "John Smith",
  "email": "john@smith.com",
  "dob": "1970/10/24"
}

PUT /my_index/blogpost/2
{
  "title": "Relationships",
  "body": "It's complicated...",
  "user": {
    "id": 1,
    "name": "John Smith"
  }
}

PUT /my_index/user/3
{
  "name": "Alice John",
  "email": "alice@john.com",
  "dob": "1979/01/04"
}

PUT /my_index/blogpost/4
{
  "title": "Relationships are cool",
  "body": "It's not complicated at all...",
  "user": {
    "id": 3,
    "name": "Alice John"
  }
}
```

在我 来 包含 relationships 并且作者名包含 John 的博客, 果再按作者名分 , 感{ref}/search-aggregations-metrics-top-hits-aggregation.html[top_hits aggregation] 提供了按照用 行分 的功能：

```
GET /my_index/blogpost/_search
{
  "size" : 0, ①
  "query": { ②
    "bool": {
      "must": [
        { "match": { "title":     "relationships" }},
        { "match": { "user.name": "John"          }}
      ]
    }
  },
  "aggs": {
    "users": {
      "terms": {
        "field":   "user.name.raw",      ③
        "order": { "top_score": "desc" } ④
      },
      "aggs": {
        "top_score": { "max":      { "script":  "_score"           }}, ④
        "blogposts": { "top_hits": { "_source": "title", "size": 5 }}  ⑤
      }
    }
  }
}
```

① 我 感 趣的博客文章是通 blogposts 聚合返回的，所以我 可以通 将 size 置成 0 来禁止 hits 常 搜索。

② query 返回通 relationships 名称 John 的用 的博客文章。

③ terms 聚合 一个 user.name.raw 建一个桶。

④ top_score 聚合 通 users 聚合得到的 一个桶按照文 分 行排序。

⑤ top_hits 聚合 个用 返回五个最相 的博客文章的 title 字段。

里 示 短 果：

```
...
"hits": {
  "total":     2,
  "max_score": 0,
  "hits":      [] ①
},
"aggregations": {
  "users": {
     "buckets": [
        {
           "key":       "John Smith", ②
           "doc_count": 1,
           "blogposts": {
              "hits": { ③
                 "total":     1,
                 "max_score": 0.35258877,
                 "hits": [
                    {
                       "_index": "my_index",
                       "_type":  "blogpost",
                       "_id":    "2",
                       "_score": 0.35258877,
                       "_source": {
                          "title": "Relationships"
                       }
                    }
                 ]
              }
           },
           "top_score": { ④
              "value": 0.3525887727737427
           }
        },
...
```

① 因 我 置 size 0，所以 hits 数 是空的。

② 在 果中出 的 一个用 都会有一个 的桶。

③ 在 个用 桶下面都会有一个 blogposts.hits 数 包含 个用 的 果。

④ 用 桶按照 个用 最相 的博客文章 行排序。

使用 top_hits 聚合等效 行一个 返回 些用 的名字和他 最相 的博客文章，然后 一个用 行相同的 ，以 得最好的博客。但前者的效率要好很多。

一个桶返回的 命中 果是基于最初主 行的一个 量 迷 果集。 个迷 提供了一些 期望的常用特性，例如高亮 示以及分 功能。

非　　化和并

当然，数据非　　化也有弊端。　　第一个　点是索引会更大因　　个博客文章文　的　　`_source` 将会更大，并且　里有很多的索引字段。　通常不是一个大　　。数据写到磁　将会被高度　　，而且磁　已　很廉　了。Elasticsearch 可以愉快地　付　些　外的数据。

更重要的　　是，如果用　改　了他的名字，他所有的博客文章也需要更新了。幸　的是，用　不　常更改名称。即使他　做了，　用　也不可能写超　几千篇博客文章，所以更新博客文章通　`scroll` 和 `bulk` APIs 大概耗　不到一秒。

然而，　我　考　一个更　　的　景，其中的　化很常　，影　深　，而且非常重要，并　。

在　个例子中，我　将在 Elasticsearch 模　一个文件系　的目　　，非常　似 Linux 文件系　：根目　是 `/`，　个目　可以包含文件和子目　。

我　希望能　搜索到一个特定目　下的文件，等效于：

```
grep "some text" /clinton/projects/elasticsearch/*
```

就要求我　索引文件所在目　的路径：

```
PUT /fs/file/1
{
  "name":     "README.txt", ①
  "path":     "/clinton/projects/elasticsearch", ②
  "contents": "Starting a new Elasticsearch project is easy..."
}
```

① 文件名

② 文件所在目　的全路径

NOTE	事　上，我　也　当索引 `directory`　文　，如此我　可以在目　内列出所有的文件和子目　，但　了　　，我　将忽略　个需求。

我　也希望能　搜索到一个特定目　下的目　　包含的的任何文件，相当于此：

```
grep -r "some text" /clinton
```

了支持　一点，我　需要　路径　次　　行索引：

- `/clinton`
- `/clinton/projects`
- `/clinton/projects/elasticsearch`

　　次　　能　通　　`path`　字段使用　{ref}/analysis-pathhierarchy-tokenizer.html[`path_hierarchy` tokenizer] 自　生成：

```
PUT /fs
{
  "settings": {
    "analysis": {
      "analyzer": {
        "paths": { ①
          "tokenizer": "path_hierarchy"
        }
      }
    }
  }
}
```

① 自定 的 paths 分析器在 置中使用 {ref}/analysis-pathhierarchy-tokenizer.html[path_hierarchy tokenizer]。

file 型的映射看起来如下所示：

```
PUT /fs/_mapping/file
{
  "properties": {
    "name": { ①
      "type":  "string",
      "index": "not_analyzed"
    },
    "path": { ②
      "type":  "string",
      "index": "not_analyzed",
      "fields": {
        "tree": { ②
          "type":     "string",
          "analyzer": "paths"
        }
      }
    }
  }
}
```

① name 字段将包含 切名称。

② path 字段将包含 切的目 名称，而 path.tree 字段将包含路径 次 。

一旦索引建立并且文件已被 入索引，我 可以 行一个搜索，在 /clinton/projects/elasticsearch 目中包含 elasticsearch 的文件，如下所示：

```
GET /fs/file/_search
{
  "query": {
    "filtered": {
      "query": {
        "match": {
          "contents": "elasticsearch"
        }
      },
      "filter": {
        "term": { ①
          "path": "/clinton/projects/elasticsearch"
        }
      }
    }
  }
}
```

① 在 目 中 文件。

所有在 /clinton 下面的任何子目 存放的文件将在 path.tree 字段中包含 /clinton 。所以我 能 搜索 /clinton 的任何子目 中的所有文件，如下所示：

```
GET /fs/file/_search
{
  "query": {
    "filtered": {
      "query": {
        "match": {
          "contents": "elasticsearch"
        }
      },
      "filter": {
        "term": { ①
          "path.tree": "/clinton"
        }
      }
    }
  }
}
```

① 在 个目 或其下任何子目 中 文件。

重命名文件和目

到目前 止一切 利。 重命名一个文件很容易—所需要的只是一个 的 update 或 index 求。 甚至可以使用 optimistic concurrency control 保 的 化不会与其他用 的 化 生冲突：

```
PUT /fs/file/1?version=2 ①
{
  "name":     "README.asciidoc",
  "path":     "/clinton/projects/elasticsearch",
  "contents": "Starting a new Elasticsearch project is easy..."
}
```

① version 号 保 更改 用于 索引中具有此相同的版本号的文 。

我 甚至可以重命名一个目 ，但 意味着更新所有存在于 目 下路径 次 中的所有文件。 可能快速或 慢，取决于有多少文件需要更新。我 所需要做的就是使用 scroll 来 索所有的文件，以及 bulk API 来更新它 。 个 程不是原子的，但是所有的文件将会迅速 移到他 的新存放位置。

解决并

当我 允 多个人 同 重命名文件或目 ， 就来了。 想一下， 正在 一个包含了成百上千文件的目 /clinton 行重命名操作。 同 ， 一个用 个目 下的 个文件 /clinton/projects/elasticsearch/README.txt 行重命名操作。 个用 的修改操作，尽管在 的操作后 始，但可能会更快的完成。

以下有 情况可能出 ：

- 决定使用 version （版本）号，在 情况下，当与 README.txt 文件重命名的版本号 生冲突 ，的批量重命名操作将会失 。
- 没有使用版本控制， 的 更将覆 其他用 的 更。

 的原因是 Elasticsearch 不支持 ACID 事 。 个文件的 更是 ACIDic 的，但包含多个文 的更不支持。

如果 的主要数据存 是 系数据 ，并且 Elasticsearch 作 一个搜索引 或一 提升性能的方法，可以首先在数据 中 行 更 作，然后在完成后将 些 更 制到 Elasticsearch。通 方式， 将受益于数据 ACID 事 支持，并且在 Elasticsearch 中以正 的 序 生 更。 并在 系数据 中得到了 理。

如果 不使用 系型存 ， 些并 就需要在 Elasticsearch 的事 水准 行 理。 以下是三个切 可行的使用 Elasticsearch 的解决方案，它 都 及某 形式的 ：

- 全局
- 文
-

TIP	当使用一个外部系 替代 Elasticsearch ，本 中所描述的解决方案可以通 相同的原来 。

全局

通 在任何 只允 一个 程来 行 更 作，我 可以完全避免并 。 大多数的 更只

及少量文件，会很快完成。一个　　目　的重命名操作会　其他　更造成　　　的阻塞，但可能很少做。

因　在 Elasticsearch　文　　的　更支持 ACIDic，我　可以使用一个文　是否存在的状　作　一个全局　。　了　求得到　，我　　create 全局　文　：

```
PUT /fs/lock/global/_create
{}
```

如果　个　 create　　求因冲突　常而失　，　明　一个　程已被授予全局　，我　将不得不　后再　。如果　求成功了，我　自豪的成　全局　的主人，然后可以　　完成我　的　更。一旦完成，我　就必　通　　除全局　文　来　放　：

```
DELETE /fs/lock/global
```

根据　更的　繁程度以及　　消耗，一个全局　能　系　造成大幅度的性能限制。　　我　可以通　　我　的　更　粒度的方式来　加并行度。

文

我　可以使用前面描述相同的方法技　来　定个体文　，而不是　定整个文件系　。我　可以使用 scrolled search　索所有的文　，　些文　会被　更影　因此　一个文　都　建了一个　文件：

```
PUT /fs/lock/_bulk
{ "create": { "_id": 1}} ①
{ "process_id": 123    } ②
{ "create": { "_id": 2}}
{ "process_id": 123    }
```

① lock 文　的 ID 将与　被　定的文件的 ID 相同。

② process_id 代表要　行　更　程的唯一 ID。

如果一些文件已被　定，部分的 bulk　求将失　，我　将不得不再次　　。

当然，如果我　　再次　定　所有　的文件，　　我　前面使用的　 create　　句将会失　，因所有文件都已被我　　定！　我　需要一个　update　　求　　upsert　参数以及下面　个　 script，而不是一个　　的 create　句：

```
if ( ctx._source.process_id != process_id ) { ①
  assert false;  ②
}
ctx.op = 'noop'; ③
```

① process_id 是　　到脚本的一个参数。

② assert false 将引　　常，　致更新失　。

③ 将 op 从 update 更新到 noop 防止更新 求作出任何改 ，但 返回成功。

完整的 update 求如下所示：

```
POST /fs/lock/1/_update
{
  "upsert": { "process_id": 123 },
  "script": "if ( ctx._source.process_id != process_id )
  { assert false }; ctx.op = 'noop';"
  "params": {
    "process_id": 123
  }
}
```

如果文 并不存在， upsert 文 将会被 入—和前面 create 求相同。 但是，如果 文件 存在， 脚本会 看存 在文 上的 process_id 。 如果 process_id 匹配，更新不会 行（ noop ）但脚本会返回成功。 如果 者并不匹配， assert false 出一个 常， 也知道了 取 的 已失 。

一旦所有 已成功 建， 就可以 行 的 更。

之后， 必 放所有的 ，通 索所有的 文 并 行批量 除，可以完成 的 放：

```
POST /fs/_refresh ①

GET /fs/lock/_search?scroll=1m ②
{
    "sort" : ["_doc"],
    "query": {
        "match" : {
            "process_id" : 123
        }
    }
}

PUT /fs/lock/_bulk
{ "delete": { "_id": 1}}
{ "delete": { "_id": 2}}
```

① refresh 用 保所有 lock 文 搜索 求可 。

② 当 需要在 次搜索 求返回大量的 索 果集 ， 可以使用 scroll 。

文 可以 粒度的 控制，但是 数百万文 建 文件 也很大。 在某些情况下，可以用少得多的工作量 粒度的 定，如以下目 景中所示。

在前面的例子中，我 可以 定的目 的一部分，而不是 定 一个 及的文 。 我 将需要独占

我 要重命名的文件或目 ，它可以通 独占 文 来 ：

```
{ "lock_type": "exclusive" }
```

同 我 需要共享 定所有的父目 ，通 共享 文 ：

```
{
  "lock_type":  "shared",
  "lock_count": 1 ①
}
```

① lock_count 持有共享 程的数量。

/clinton/projects/elasticsearch/README.txt 行重命名的 程需要在 个文件上有 独占 ，以及在 /clinton 、 /clinton/projects 和 /clinton/projects/elasticsearch 目 有 共享 。

一个 的 create 求将 足独占 的要求，但共享 需要脚本的更新来 一些 外的 ：

```
if (ctx._source.lock_type == 'exclusive') {
  assert false; ①
}
ctx._source.lock_count++ ②
```

① 如果 lock_type 是 exclusive （独占）的，assert 句将 出一个 常， 致更新 求失 。

② 否 ，我 lock_count 行 量 理。

个脚本 理了 lock 文 已 存在的情况，但我 需要一个用来 理的文 不存在情况的 upsert 文 。完整的更新 求如下：

```
POST /fs/lock/%2Fclinton/_update ①
{
  "upsert": { ②
    "lock_type":  "shared",
    "lock_count": 1
  },
  "script": "if (ctx._source.lock_type == 'exclusive')
  { assert false }; ctx._source.lock_count++"
}
```

① 文 的 ID 是 /clinton ， URL 后成 %2fclinton 。

② upsert 文 如果不存在， 会被 入。

一旦我 成功地在所有的父目 中 得一个共享 ，我 在文件本身 create 一个独占 ：

```
PUT /fs/lock/%2Fclinton%2fprojects%2felasticsearch%2fREADME.txt/_create
{ "lock_type": "exclusive" }
```

在，如果有其他人想要重新命名 /clinton 目 ，他 将不得不在 条路径上 得一个独占 ：

```
PUT /fs/lock/%2Fclinton/_create
{ "lock_type": "exclusive" }
```

个 求将失 ，因 一个具有相同 ID 的 lock 文 已 存在。 一个用 将不得不等待我的操作完成以及 放我 的 。独占 只能 被 除：

```
DELETE /fs/lock/%2Fclinton%2fprojects%2felasticsearch%2fREADME.txt
```

共享 需要 一个脚本 lock_count ，如果 数下降到零， 除 lock 文 ：

```
if (--ctx._source.lock_count == 0) {
  ctx.op = 'delete' ①
}
```

① 一旦 lock_count 到0， ctx.op 会从 update 被修改成 delete 。

此更新 求将 父目 由下至上的 行，从最 路径到最短路径：

```
POST /fs/lock/%2Fclinton%2fprojects%2felasticsearch/_update
{
  "script": "if (--ctx._source.lock_count == 0) { ctx.op = 'delete' } "
}
```

用最小的代 提供了 粒度的并 控制。当然，它不 用于所有的情况—数据模型必 有 似于目的 序 路径才能使用。

NOTE

三个方案—全局、文 或 —都没有 理 最棘手的 ：如果持有 的 程死了 ？

一个 程的意外死亡 我 留下了2个 ：

- 我 如何知道我 可以 放的死亡 程中所持有的 ？
- 我 如何清理死去的 程没有完成的 更？

些主 超出了本 的 ，但是如果 决定使用 ， 需要 他 行一些思考。

当非 化成 很多 目的一个很好的 ，采用 方案的需求会 来 的 。 作替代方案，Elasticsearch 提供 个模型 助我 理相 的 体：嵌套的 象 和 父子 系 。