

26 | 数据存储：NoSQL与RDBMS如何取长补短、相辅相成？

2020-05-14 朱晔

Java业务开发常见错误100例

[进入课程 >](#)




讲述：王少泽

时长 20:40 大小 18.93M



你好，我是朱晔。今天，我来和你聊聊数据存储的常见错误。

近几年，各种非关系型数据库，也就是 NoSQL 发展迅猛，在项目中也非常常见。其中不乏一些使用上的极端情况，比如直接把关系型数据库（RDBMS）全部替换为 NoSQL，或是在不合适的场景下错误地使用 NoSQL。


其实，每种 NoSQL 的特点不同，都有其要着重解决的某一方面的问题。因此，我们在使用 NoSQL 的时候，要尽量让它去处理擅长的场景，否则不但发挥不出它的功能和优势， 能会导致性能问题。

NoSQL 一般可以分为缓存数据库、时间序列数据库、全文搜索数据库、文档数据库、图数据库等。今天，我会以缓存数据库 Redis、时间序列数据库 InfluxDB、全文搜索数据库 Elasticsearch 为例，通过一些测试案例，和你聊聊这些常见 NoSQL 的特点，以及它们擅长和不擅长的地方。最后，我也还会和你说说 NoSQL 如何与 RDBMS 相辅相成，来构成一套可以应对高并发的复合数据库体系。

取长补短之 Redis vs MySQL

Redis 是一款设计简洁的缓存数据库，数据都保存在内存中，所以读写单一 Key 的性能非常高。

我们来做一个简单测试，分别填充 10 万条数据到 Redis 和 MySQL 中。MySQL 中的 name 字段做了索引，相当于 Redis 的 Key，data 字段为 100 字节的数据，相当于 Redis 的 Value：

 复制代码

```
1 @SpringBootApplication
2 @Slf4j
3 public class CommonMistakesApplication {
4
5     //模拟10万条数据存到Redis和MySQL
6     public static final int ROWS = 100000;
7     public static final String PAYLOAD = IntStream.rangeClosed(1, 100).mapToObj(
8         @Autowired
9         private StringRedisTemplate stringRedisTemplate;
10        @Autowired
11        private JdbcTemplate jdbcTemplate;
12        @Autowired
13        private StandardEnvironment standardEnvironment;
14
15
16        public static void main(String[] args) {
17            SpringApplication.run(CommonMistakesApplication.class, args);
18        }
19
20        @PostConstruct
21        public void init() {
22            //使用-Dspring.profiles.active=init启动程序进行初始化
23            if (Arrays.stream(standardEnvironment.getActiveProfiles()).anyMatch(s
24                initRedis();
25                initMySQL();
26            }
27        }
28    }
```

```

29 //填充数据到MySQL
30 private void initMySQL() {
31     //删除表
32     jdbcTemplate.execute("DROP TABLE IF EXISTS `r`;");
33     //新建表, name字段做了索引
34     jdbcTemplate.execute("CREATE TABLE `r` (\n" +
35         " `id` bigint(20) NOT NULL AUTO_INCREMENT,\n" +
36         " `data` varchar(2000) NOT NULL,\n" +
37         " `name` varchar(20) NOT NULL,\n" +
38         " PRIMARY KEY (`id`),\n" +
39         " KEY `name` (`name`) USING BTREE\n" +
40         ") ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;");
41
42     //批量插入数据
43     String sql = "INSERT INTO `r` (`data`,`name`) VALUES (?,?)";
44     jdbcTemplate.batchUpdate(sql, new BatchPreparedStatementSetter() {
45         @Override
46         public void setValues(PreparedStatement preparedStatement, int i) {
47             preparedStatement.setString(1, PAYLOAD);
48             preparedStatement.setString(2, "item" + i);
49         }
50
51         @Override
52         public int getBatchSize() {
53             return ROWS;
54         }
55     });
56     log.info("init mysql finished with count {}", jdbcTemplate.queryForObject(
57 }
58
59 //填充数据到Redis
60 private void initRedis() {
61     IntStream.rangeClosed(1, ROWS).forEach(i -> stringRedisTemplate.opsForValue().set("item" + i, PAYLOAD));
62     log.info("init redis finished with count {}", stringRedisTemplate.keys("item*").size());
63 }
64

```

启动程序后，输出了如下日志，数据全部填充完毕：

 复制代码

```

1 [14:22:47.195] [main] [INFO ] [o.g.t.c.n.r.CommonMistakesApplication:80 ] - i
2 [14:22:50.030] [main] [INFO ] [o.g.t.c.n.r.CommonMistakesApplication:74 ] - i

```

然后，比较一下从 MySQL 和 Redis 随机读取单条数据的性能。“公平”起见，像 Redis 那样，我们使用 MySQL 时也根据 Key 来查 Value，也就是根据 name 字段来查 data 字段，并且我们给 name 字段做了索引：

```

1  @Autowired
2  private JdbcTemplate jdbcTemplate;
3  @Autowired
4  private StringRedisTemplate stringRedisTemplate;
5
6  @GetMapping("redis")
7  public void redis() {
8      //使用随机的Key来查询Value, 结果应该等于PAYLOAD
9      Assert.assertTrue(stringRedisTemplate.opsForValue().get("item" + (ThreadLocal
10 })
11
12  @GetMapping("mysql")
13  public void mysql() {
14      //根据随机name来查data, name字段有索引, 结果应该等于PAYLOAD
15      Assert.assertTrue(jdbcTemplate.queryForObject("SELECT data FROM `r` WHERE i
16                      .equals(CommonMistakesApplication.PAYLOAD));
17  }

```

在我的电脑上, 使用 wrk 加 10 个线程 50 个并发连接做压测。可以看到, MySQL 90% 的请求需要 61ms, QPS 为 1460; 而 Redis 90% 的请求在 5ms 左右, QPS 达到了 14008, 几乎是 MySQL 的十倍:

```

→ Downloads wrk -t 10 -c 50 -d 10s http://localhost:45678/redisvsmysql/mysql --latency
Running 10s test @ http://localhost:45678/redisvsmysql/mysql
10 threads and 50 connections
Thread Stats   Avg      Stdev     Max    +/-  Stdev
Latency    37.67ms   32.51ms  385.40ms   91.69%
Req/Sec   146.99    43.90   270.00    65.73%
Latency Distribution
 50%    31.74ms
 75%    43.45ms
 90%    61.51ms
 99%   189.48ms
14680 requests in 10.05s, 1.02MB read
Requests/sec: 1460.50
Transfer/sec: 104.32KB
→ Downloads wrk -t 10 -c 50 -d 10s http://localhost:45678/redisvsmysql/redis --latency
Running 10s test @ http://localhost:45678/redisvsmysql/redis
10 threads and 50 connections
Thread Stats   Avg      Stdev     Max    +/-  Stdev
Latency     4.02ms    3.61ms   52.07ms   95.49%
Req/Sec    1.41k    331.08   2.31k    72.20%
Latency Distribution
 50%     3.34ms
 75%     4.22ms
 90%     5.37ms
 99%    23.89ms
140320 requests in 10.02s, 9.79MB read
Requests/sec: 14008.70
Transfer/sec: 0.98MB

```

但 Redis 薄弱的地方是，不擅长做 Key 的搜索。对 MySQL，我们可以使用 LIKE 操作前匹配走 B+ 树索引实现快速搜索；但对 Redis，我们使用 Keys 命令对 Key 的搜索，其实相当于在 MySQL 里做全表扫描。

我写一段代码来对比一下性能：

复制代码

```
1 @GetMapping("redis2")
2 public void redis2() {
3     Assert.assertTrue(stringRedisTemplate.keys("item71*").size() == 1111);
4 }
5 @GetMapping("mysql2")
6 public void mysql2() {
7     Assert.assertTrue(jdbcTemplate.queryForList("SELECT name FROM `r` WHERE na
8 }
```

可以看到，在 QPS 方面，MySQL 的 QPS 达到了 Redis 的 157 倍；在延迟方面，MySQL 的延迟只有 Redis 的十分之一。

```
→ Downloads wrk -t 10 -c 50 -d 10s http://localhost:45678/redisvsmysql/mysql2 --latency
Running 10s test @ http://localhost:45678/redisvsmysql/mysql2
10 threads and 50 connections
Thread Stats   Avg      Stdev     Max    +/-  Stdev
Latency    53.00ms   14.54ms  207.87ms   78.35%
Req/Sec    94.63     20.86   151.00    69.40%
Latency Distribution
 50%    50.10ms
 75%    59.07ms
 90%    70.47ms
 99%   104.63ms
9483 requests in 10.06s, 676.96KB read
Requests/sec: 942.76
Transfer/sec: 67.30KB

→ Downloads wrk -t 10 -c 50 -d 10s http://localhost:45678/redisvsmysql/redis2 --latency
Running 10s test @ http://localhost:45678/redisvsmysql/redis2
10 threads and 50 connections
Thread Stats   Avg      Stdev     Max    +/-  Stdev
Latency   756.35ms  371.91us  757.07ms   83.33%
Req/Sec     0.62      1.02     4.00    85.71%
Latency Distribution
 50%   756.30ms
 75%   756.35ms
 90%   757.07ms
 99%   757.07ms
60 requests in 10.03s, 4.28KB read
Socket errors: connect 0, read 0, write 0, timeout 54
Requests/sec: 5.98
Transfer/sec: 436.73B
```

Redis 慢的原因有两个：

Redis 的 Keys 命令是 $O(n)$ 时间复杂度。如果数据库中 Key 的数量很多，就会非常慢。

Redis 是单线程的，对于慢的命令如果有并发，串行执行就会非常耗时。

一般而言，我们使用 Redis 都是针对某一个 Key 来使用，而不能在业务代码中使用 Keys 命令从 Redis 中“搜索数据”，因为这不是 Redis 的擅长。对于 Key 的搜索，我们可以先通过关系型数据库进行，然后再从 Redis 存取数据（如果实在需要搜索 Key 可以使用 SCAN 命令）。在生产环境中，我们一般也会配置 Redis 禁用类似 Keys 这种比较危险的命令，你可以 [参考这里](#)。

总结一下，正如“[缓存设计](#)”一讲中提到的，对于业务开发来说，大多数业务场景下 Redis 是作为关系型数据库的辅助用于缓存的，我们一般不会把它当作数据库独立使用。

此外值得一提的是，Redis 提供了丰富的数据结构（Set、SortedSet、Hash、List），并围绕这些数据结构提供了丰富的 API。如果我们好好利用这个特点的话，可以直接在 Redis 中完成一部分服务端计算，避免“读取缓存 -> 计算数据 -> 保存缓存”三部曲中的读取和保存缓存的开销，进一步提高性能。

取长补短之 InfluxDB vs MySQL

InfluxDB 是一款优秀的时序数据库。在“[生产就绪](#)”这一讲中，我们就是使用 InfluxDB 来做的 Metrics 打点。时序数据库的优势，在于处理指标数据的聚合，并且读写效率非常高。

同样的，我们使用一些测试来对比下 InfluxDB 和 MySQL 的性能。

在如下代码中，我们分别填充了 1000 万条数据到 MySQL 和 InfluxDB 中。其中，每条数据只有 ID、时间戳、10000 以内的随机值这 3 列信息，对于 MySQL 我们把时间戳列做了索引：

 复制代码

```
1 @SpringBootApplication
2 @Slf4j
3 public class CommonMistakesApplication {
4
5     public static void main(String[] args) {
6         SpringApplication.run(CommonMistakesApplication.class, args);
7     }
```



```

8
9 //测试数据量
10 public static final int ROWS = 10000000;
11
12 @Autowired
13 private JdbcTemplate jdbcTemplate;
14 @Autowired
15 private StandardEnvironment standardEnvironment;
16
17 @PostConstruct
18 public void init() {
19     //使用-Dspring.profiles.active=init启动程序进行初始化
20     if (Arrays.stream(standardEnvironment.getActiveProfiles()).anyMatch(s →
21         initInfluxDB();
22         initMySQL();
23     })
24 }
25
26 //初始化MySQL
27 private void initMySQL() {
28     long begin = System.currentTimeMillis();
29     jdbcTemplate.execute("DROP TABLE IF EXISTS `m`");
30     //只有ID、值和时间戳三列
31     jdbcTemplate.execute("CREATE TABLE `m` (\n" +
32         " `id` bigint(20) NOT NULL AUTO_INCREMENT,\n" +
33         " `value` bigint NOT NULL,\n" +
34         " `time` timestamp NOT NULL,\n" +
35         " PRIMARY KEY (`id`),\n" +
36         " KEY `time` (`time`) USING BTREE\n" +
37         ") ENGINE=InnoDB DEFAULT CHARSET=utf8mb4");
38
39     String sql = "INSERT INTO `m` (`value`,`time`) VALUES (?,?)";
40     //批量插入数据
41     jdbcTemplate.batchUpdate(sql, new BatchPreparedStatementSetter() {
42         @Override
43         public void setValues(PreparedStatement preparedStatement, int i) {
44             preparedStatement.setLong(1, ThreadLocalRandom.current().nextLong(1000000000L));
45             preparedStatement.setTimestamp(2, Timestamp.valueOf(LocalDate.now().atTime(10, 10, 10)));
46         }
47
48         @Override
49         public int getBatchSize() {
50             return ROWS;
51         }
52     });
53     log.info("init mysql finished with count {} took {}ms", jdbcTemplate.queryForObject("select count(*) from m", Integer.class));
54 }
55
56 //初始化InfluxDB
57 private void initInfluxDB() {
58     long begin = System.currentTimeMillis();
59     OkHttpClient.Builder okHttpClientBuilder = new OkHttpClient().newBuilder();

```

```

60         .connectTimeout(1, TimeUnit.SECONDS)
61         .readTimeout(10, TimeUnit.SECONDS)
62         .writeTimeout(10, TimeUnit.SECONDS);
63     try (InfluxDB influxDB = InfluxDBFactory.connect("http://127.0.0.1:8086",
64         String db = "performance";
65         influxDB.query(new Query("DROP DATABASE " + db));
66         influxDB.query(new Query("CREATE DATABASE " + db));
67         //设置数据库
68         influxDB.setDatabase(db);
69         //批量插入, 10000条数据刷一次, 或1秒刷一次
70         influxDB.enableBatch(BatchOptions.DEFAULTS.actions(10000).flushDuration(1,
71             IntStream.rangeClosed(1, ROWS).mapToObj(i -> Point
72                 .measurement("m")
73                 .addField("value", ThreadLocalRandom.current().nextInt(1000000000L))
74                 .time(LocalDateTime.now().minusSeconds(5 * i).toInstant(ZoneOffset.UTC))
75                 .forEach(influxDB::write);
76         influxDB.flush();
77         log.info("init influxdb finished with count {} took {}ms", influxDB.getRowCount(),
78             time);
79     }
80 }

```

启动后, 程序输出了如下日志:

 复制代码

```

1 [16:08:25.062] [main] [INFO ] [o.g.t.c.n.i.CommonMistakesApplication:104 ] - i
2 [16:11:50.462] [main] [INFO ] [o.g.t.c.n.i.CommonMistakesApplication:80 ] - i

```

InfluxDB 批量插入 1000 万条数据仅用了 54 秒, 相当于每秒插入 18 万条数据, 速度相当快; MySQL 的批量插入, 速度也挺快达到了每秒 4.8 万。

接下来, 我们测试一下。

对这 1000 万数据进行一个统计, 查询最近 60 天的数据, 按照 1 小时的时间粒度聚合, 统计 value 列的最大值、最小值和平均值, 并将统计结果绘制成曲线图:

 复制代码

```

1 @Autowired
2 private JdbcTemplate jdbcTemplate;
3 @GetMapping("mysql")
4 public void mysql() {
5     long begin = System.currentTimeMillis();
6

```




```

7      //使用SQL从MySQL查询, 按照小时分组
8      Object result = jdbcTemplate.queryForList("SELECT date_format(time,'%Y%m%d:
9      log.info("took {} ms result {}", System.currentTimeMillis() - begin, resul
10 }
11
12
13 @GetMapping("influxdb")
14 public void influxdb() {
15     long begin = System.currentTimeMillis();
16     try (InfluxDB influxDB = InfluxDBFactory.connect("http://127.0.0.1:8086",
17         //切换数据库
18         influxDB.setDatabase("performance");
19         //InfluxDB的查询语法InfluxQL类似SQL
20         Object result = influxDB.query(new Query("SELECT MEAN(value),MIN(value)
21         log.info("took {} ms result {}", System.currentTimeMillis() - begin, r
22     }

```

因为数据量非常大, 单次查询就已经很慢, 所以这次我们不进行压测。分别调用两个接口, 可以看到 **MySQL 查询一次耗时 29 秒左右, 而 InfluxDB 耗时 980ms**:

 复制代码

```

1 [16:19:26.562] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.n.i.PerformanceControl
2 [16:20:08.170] [http-nio-45678-exec-6] [INFO ] [o.g.t.c.n.i.PerformanceControl

```

在按照时间区间聚合的案例上, 我们看到了 InfluxDB 的性能优势。但, 我们**肯定不能把 InfluxDB 当作普通数据库**, 原因是:

InfluxDB 不支持数据更新操作, 毕竟时间数据只能随着时间产生新数据, 肯定无法对过去的数据库做修改;

从数据结构上说, 时间序列数据没有单一的主键标识, 必须包含时间戳, 数据只能和时间戳进行关联, 不适合普通业务数据。

此外需要注意, 即便只是使用 InfluxDB 保存和时间相关的指标数据, 我们也要注意不能滥用 tag。

InfluxDB 提供的 tag 功能, 可以为每一个指标设置多个标签, 并且 tag 有索引, 可以对 tag 进行条件搜索或分组。但是, tag 只能保存有限的、可枚举的标签, 不能保存 URL 等信息, 否则可能会出现 [high series cardinality 问题](#), 导致占用大量内存, 甚至是

OOM。你可以点击 [这里](#)，查看 series 和内存占用的关系。对于 InfluxDB，我们无法把 URL 这种原始数据保存到数据库中，只能把数据进行归类，形成有限的 tag 进行保存。

总结一下，对于 MySQL 而言，针对大量的数据使用全表扫描的方式来聚合统计指标数据，性能非常差，一般只能作为临时方案来使用。此时，引入 InfluxDB 之类的时间序列数据库，就很有必要了。时间序列数据库可以作为特定场景（比如监控、统计）的主存储，也可以和关系型数据库搭配使用，作为一个辅助数据源，保存业务系统的指标数据。

取长补短之 Elasticsearch vs MySQL

Elasticsearch（以下简称 ES），是目前非常流行的分布式搜索和分析数据库，独特的倒排索引结构尤其适合进行全文搜索。

简单来讲，倒排索引可以认为是一个 Map，其 Key 是分词之后的关键字，Value 是文档 ID/ 片段 ID 的列表。我们只要输入需要搜索的单词，就可以直接在这个 Map 中得到所有包含这个单词的文档 ID/ 片段 ID 列表，然后再根据其中的文档 ID/ 片段 ID 查询出实际的文档内容。

我们来测试一下，对比下使用 ES 进行关键字全文搜索、在 MySQL 中使用 LIKE 进行搜索的效率差距。

首先，定义一个实体 News，包含新闻分类、标题、内容等字段。这个实体同时会用作 Spring Data JPA 和 Spring Data Elasticsearch 的实体：

 复制代码

```
1 @Entity
2 @Document(indexName = "news", replicas = 0) //@Document注解定义了这是一个ES的索引,
3 @Table(name = "news", indexes = {@Index(columnList = "cateid")}) //@Table注解定
4 @Data
5 @AllArgsConstructor
6 @NoArgsConstructor
7 @DynamicUpdate
8 public class News {
9     @Id
10     private long id;
11     @Field(type = FieldType.Keyword)
12     private String category;//新闻分类名称
13     private int cateid;//新闻分类ID
14     @Column(columnDefinition = "varchar(500)")//@Column注解定义了MySQL中字段, 比
15     @Field(type = FieldType.Text, analyzer = "ik_max_word", searchAnalyzer = "
```

```

16     private String title;//新闻标题
17     @Column(columnDefinition = "text")
18     @Field(type = FieldType.Text, analyzer = "ik_max_word", searchAnalyzer = "
19     private String content;//新闻内容
20 }

```

接下来，我们实现主程序。在启动时，我们会从一个 csv 文件中加载 4000 条新闻数据，然后复制 100 份，拼成 40 万条数据，分别写入 MySQL 和 ES：

 复制代码

```

1  @SpringBootApplication
2  @Slf4j
3  @EnableElasticsearchRepositories(includeFilters = @ComponentScan.Filter(type =
4  @EnableJpaRepositories(excludeFilters = @ComponentScan.Filter(type = FilterType
5  public class CommonMistakesApplication {
6
7      public static void main(String[] args) {
8          Utils.loadPropertySource(CommonMistakesApplication.class, "es.properti
9          SpringApplication.run(CommonMistakesApplication.class, args);
10     }
11
12     @Autowired
13     private StandardEnvironment standardEnvironment;
14     @Autowired
15     private NewsESRepository newsESRepository;
16     @Autowired
17     private NewsMySQLRepository newsMySQLRepository;
18
19     @PostConstruct
20     public void init() {
21         //使用-Dspring.profiles.active=init启动程序进行初始化
22         if (Arrays.stream(standardEnvironment.getActiveProfiles()).anyMatch(s
23             //csv中的原始数据只有4000条
24             List<News> news = loadData();
25             AtomicLong atomicLong = new AtomicLong();
26             news.forEach(item -> item.setTitle("%%" + item.getTitle()));
27             //我们模拟100倍的数据量，也就是40万条
28             IntStream.rangeClosed(1, 100).forEach(repeat -> {
29                 news.forEach(item -> {
30                     //重新设置主键ID
31                     item.setId(atomicLong.incrementAndGet());
32                     //每次复制数据稍微改一下title字段，在前面加上一个数字，代表这是第几次
33                     item.setTitle(item.getTitle().replaceFirst("%%", String.va
34                 });
35                 initMySQL(news, repeat == 1);
36                 log.info("init MySQL finished for {}", repeat);
37                 initES(news, repeat == 1);
38                 log.info("init ES finished for {}", repeat);

```

```

39         });
40
41     }
42 }
43
44 //从news.csv中解析得到原始数据
45 private List<News> loadData() {
46     //使用jackson-dataformat-csv实现csv到POJO的转换
47     CsvMapper csvMapper = new CsvMapper();
48     CsvSchema schema = CsvSchema.emptySchema().withHeader();
49     ObjectReader objectReader = csvMapper.readerFor(News.class).with(schema);
50     ClassLoader classLoader = getClass().getClassLoader();
51     File file = new File(classLoader.getResource("news.csv").getFile());
52     try (Reader reader = new FileReader(file)) {
53         return objectReader.<News>readValues(reader).readAll();
54     } catch (Exception e) {
55         e.printStackTrace();
56     }
57     return null;
58 }
59
60 //把数据保存到ES中
61 private void initES(List<News> news, boolean clear) {
62     if (clear) {
63         //首次调用的时候先删除历史数据
64         newsESRepository.deleteAll();
65     }
66     newsESRepository.saveAll(news);
67 }
68
69 //把数据保存到MySQL中
70 private void initMySQL(List<News> news, boolean clear) {
71     if (clear) {
72         //首次调用的时候先删除历史数据
73         newsMySQLRepository.deleteAll();
74     }
75     newsMySQLRepository.saveAll(news);
76 }
77

```

由于我们使用了 Spring Data，直接定义两个 Repository，然后直接定义查询方法，无需实现任何逻辑即可实现查询，Spring Data 会根据方法名生成相应的 SQL 语句和 ES 查询 DSL，其中 ES 的翻译逻辑 [详见这里](#)。

在这里，我们定义一个 countByCateidAndContentContainingAndContentContaining 方法，代表查询条件是：搜索分类等于 cateid 参数，且内容同时包含关键字 keyword1 和 keyword2，计算符合条件的新闻总数量：

[复制代码](#)

```
1 @Repository
2 public interface NewsMySQLRepository extends JpaRepository<News, Long> {
3     //JPA: 搜索分类等于cateid参数, 且内容同时包含关键字keyword1和keyword2, 计算符合条件的
4     long countByCateidAndContentContainingAndContentContaining(int cateid, Str
5 }
6
7 @Repository
8 public interface NewsESRepository extends ElasticsearchRepository<News, Long> {
9     //ES: 搜索分类等于cateid参数, 且内容同时包含关键字keyword1和keyword2, 计算符合条件的
10    long countByCateidAndContentContainingAndContentContaining(int cateid, Str
11 }
```

对于 ES 和 MySQL，我们使用相同的条件进行搜索，搜索分类是 1，关键字是社会和苹果，然后输出搜索结果和耗时：

[复制代码](#)

```
1 //测试MySQL搜索, 最后输出耗时和结果
2 @GetMapping("mysql")
3 public void mysql(@RequestParam(value = "cateid", defaultValue = "1") int cateid,
4                  @RequestParam(value = "keyword1", defaultValue = "社会") String keyword1,
5                  @RequestParam(value = "keyword2", defaultValue = "苹果") String keyword2) {
6     long begin = System.currentTimeMillis();
7     Object result = newsMySQLRepository.countByCateidAndContentContainingAndContentContaining(cateid, keyword1, keyword2);
8     log.info("took {} ms result {}", System.currentTimeMillis() - begin, result);
9 }
10 //测试ES搜索, 最后输出耗时和结果
11 @GetMapping("es")
12 public void es(@RequestParam(value = "cateid", defaultValue = "1") int cateid,
13               @RequestParam(value = "keyword1", defaultValue = "社会") String keyword1,
14               @RequestParam(value = "keyword2", defaultValue = "苹果") String keyword2) {
15     long begin = System.currentTimeMillis();
16     Object result = newsESRepository.countByCateidAndContentContainingAndContentContaining(cateid, keyword1, keyword2);
17     log.info("took {} ms result {}", System.currentTimeMillis() - begin, result);
18 }
```

分别调用接口可以看到，**ES 耗时仅仅 48ms，MySQL 耗时 6 秒多是 ES 的 100 倍**。很遗憾，虽然新闻分类 ID 已经建了索引，但是这个索引只能起到加速过滤分类 ID 这一单一条件的作用，对于文本内容的全文搜索，B+ 树索引无能为力。


[复制代码](#)

```
1 [22:04:00.951] [http-nio-45678-exec-6] [INFO ] [o.g.t.c.n.esvsmysql.PerformanceTest]
2 Hibernate: select count(news0_.id) as col_0_0_ from news news0_ where news0_.cateid=1 and news0_.content like %社会% and news0_.content like %苹果%
```

但 ES 这种以索引为核心的数据库，也不是万能的，频繁更新就是一个大问题。

MySQL 可以做到仅更新某行数据的某个字段，但 ES 里每次数据字段更新都相当于整个文档索引重建。即便 ES 提供了文档部分更新的功能，但本质上只是节省了提交文档的网络流量，以及减少了更新冲突，其内部实现还是文档删除后重新构建索引。因此，如果要在 ES 中保存一个类似计数器的值，要实现不断更新，其执行效率会非常低。

我们来验证下，分别使用 JdbcTemplate+SQL 语句、ElasticsearchTemplate+ 自定义 UpdateQuery，实现部分更新 MySQL 表和 ES 索引的一个字段，每个方法都是循环更新 1000 次：

 复制代码

```

1  @GetMapping("mysql2")
2  public void mysql2(@RequestParam(value = "id", defaultValue = "400000") long id) {
3      long begin = System.currentTimeMillis();
4      //对于MySQL, 使用JdbcTemplate+SQL语句, 实现直接更新某个category字段, 更新1000次
5      IntStream.rangeClosed(1, 1000).forEach(i -> {
6          jdbcTemplate.update("UPDATE `news` SET category=? WHERE id=?", new Object[] {
7              id, i
8          });
9          log.info("mysql took {} ms result {}", System.currentTimeMillis() - begin, i);
10     });
11 }
12 @GetMapping("es2")
13 public void es(@RequestParam(value = "id", defaultValue = "400000") long id) {
14     long begin = System.currentTimeMillis();
15     IntStream.rangeClosed(1, 1000).forEach(i -> {
16         //对于ES, 通过ElasticsearchTemplate+自定义UpdateQuery, 实现文档的部分更新
17         UpdateQuery updateQuery = null;
18         try {
19             updateQuery = new UpdateQueryBuilder()
20                 .withIndexName("news")
21                 .withId(String.valueOf(id))
22                 .withType("_doc")
23                 .withUpdateRequest(new UpdateRequest().doc(
24                     new JSONObject().startObject()
25                         .field("category", "test" + i)
26                         .endObject())
27                 ).build();
28         } catch (IOException e) {
29             e.printStackTrace();
30         }
31     });
32 }

```



```
32     elasticsearchTemplate.update(updateQuery);
33   });
34   log.info("es took {} ms result {}", System.currentTimeMillis() - begin, new
```

可以看到，MySQL 耗时仅仅 1.5 秒，而 ES 耗时 6.8 秒：

```
[22:23:30.554] [http-nio-45678-exec-4] [INFO ] [o.g.t.c.n.esvsmysql.PerformanceController:84 ] - es took 6860 ms result News(id=400000,
category=test1000, cateid=4, title=1俄战机发动高密度空袭，数十枚空炸弹丢下火光冲天，美军紧急后撤， content=英国广播公司(BBC)9月9日报道，俄军在前几日出动多架轰炸机轰炸叙利亚伊
德利卜省的叛军后，9月8日，俄叙联军又一次对该地区的叛军据点展开了高密度的空袭。英媒称，此次空袭的目标为伊德利卜省的东部和南部地区，此外附近的哈马省部分地区也在空袭范围之内，这里也是最后
一个被叙叛军控制的主要据点。叙利亚人权观察所指出，俄叙联军战机空袭一波接着一波，数十枚空炸弹丢下，爆炸现场火光冲天，叛军的最后据点被这规模空前的空袭轰成废墟，可见，此次空袭取得了十分不错的
成效。据了解，在俄叙联军出动轰炸机消灭叛军主要据点的同时，美军也在伊德利卜省频繁活动。9月7日起，美军组织盟军在叙利亚东部展开了一场打击恐怖组织的演习。美军称，为了避免不必要的冲突，此次演
习的相关信息已经预先向俄罗斯通报。随后，美盟军出动大批战机模拟进攻地面连队，部署行动十分迅速，让人怀疑这是否是一场演习。美盟军在叙利亚东部进行大规模演习的同时，俄叙联军突然发动空袭，也是
将美盟军吓了一跳，其演习也被因之被迫中止，美军慌忙将阵线向后撤离，以免俄军“误伤”。值得一提的是，美国近日已经多次表明，俄叙联军对伊德利卜省发起攻击是不道德的行为，对于空袭表示十分遗憾。日
前，白宫已经批准对叙利亚的新战略，近2000名美军驻扎叙利亚日期将无限延长，军事专家认为，俄叙联军此次规模空前的空袭，是叙军发起地面推进的最后信号，联军十分清楚，如今西方国家插手叙战事的意味
越来越浓厚，如果犹豫不决，很可能错失良机。「注：本文部分图片来源于网络！文章未经授权禁止转载！关注我们，每天阅读更多精彩内容」)
Hibernate: select news0_.id as id1_0_0_, news0_.category as category2_0_0_, news0_.cateid as cateid3_0_0_, news0_.content as content4_0_0_, news0_
.title as title5_0_0_ from news news0_ where news0_.id=?
[22:23:37.324] [http-nio-45678-exec-6] [INFO ] [o.g.t.c.n.esvsmysql.PerformanceController:58 ] - mysql took 1495 ms result Optional[News(id=400000,
category=test1000, cateid=4, title=1俄战机发动高密度空袭，数十枚空炸弹丢下火光冲天，美军紧急后撤， content=英国广播公司(BBC)9月9日报道，俄军在前几日出动多架轰炸机轰炸叙利亚伊
德利卜省的叛军后，9月8日，俄叙联军又一次对该地区的叛军据点展开了高密度的空袭。英媒称，此次空袭的目标为伊德利卜省的东部和南部地区，此外附近的哈马省部分地区也在空袭范围之内，这里也是最后
一个被叙叛军控制的主要据点。叙利亚人权观察所指出，俄叙联军战机空袭一波接着一波，数十枚空炸弹丢下，爆炸现场火光冲天，叛军的最后据点被这规模空前的空袭轰成废墟，可见，此次空袭取得了十分不错的
成效。据了解，在俄叙联军出动轰炸机消灭叛军主要据点的同时，美军也在伊德利卜省频繁活动。9月7日起，美军组织盟军在叙利亚东部展开了一场打击恐怖组织的演习。美军称，为了避免不必要的冲突，此次演
习的相关信息已经预先向俄罗斯通报。随后，美盟军出动大批战机模拟进攻地面连队，部署行动十分迅速，让人怀疑这是否是一场演习。美盟军在叙利亚东部进行大规模演习的同时，俄叙联军突然发动空袭，也是
将美盟军吓了一跳，其演习也被因之被迫中止，美军慌忙将阵线向后撤离，以免俄军“误伤”。值得一提的是，美国近日已经多次表明，俄叙联军对伊德利卜省发起攻击是不道德的行为，对于空袭表示十分遗憾。日
前，白宫已经批准对叙利亚的新战略，近2000名美军驻扎叙利亚日期将无限延长，军事专家认为，俄叙联军此次规模空前的空袭，是叙军发起地面推进的最后信号，联军十分清楚，如今西方国家插手叙战事的意味
越来越浓厚，如果犹豫不决，很可能错失良机。「注：本文部分图片来源于网络！文章未经授权禁止转载！关注我们，每天阅读更多精彩内容」)]
```

ES 是一个分布式的全文搜索数据库，所以与 MySQL 相比的优势在于文本搜索，而且因为其分布式的特性，可以使用一个大 ES 集群处理大规模数据的内容搜索。但，由于 ES 的索引是文档维度的，所以不适用于频繁更新的 OLTP 业务。

一般而言，我们会把 ES 和 MySQL 结合使用，MySQL 直接承担业务系统的增删改操作，而 ES 作为辅助数据库，直接扁平化保存一份业务数据，用于复杂查询、全文搜索和统计。接下来，我也会继续和你分析这一点。

结合 NoSQL 和 MySQL 应对高并发的复合数据库架构

现在，我们通过一些案例看到了 Redis、InfluxDB、ES 这些 NoSQL 数据库，都有擅长和不擅长的场景。那么，有没有全能的数据数据库呢？

我认为没有。每一个存储系统都有其独特的数据结构，数据结构的设计就决定了其擅长和不擅长的场景。

比如，MySQL InnoDB 引擎的 B+ 树对排序和范围查询友好，频繁数据更新的代价不是太大，因此适合 OLTP (On-Line Transaction Processing) 。

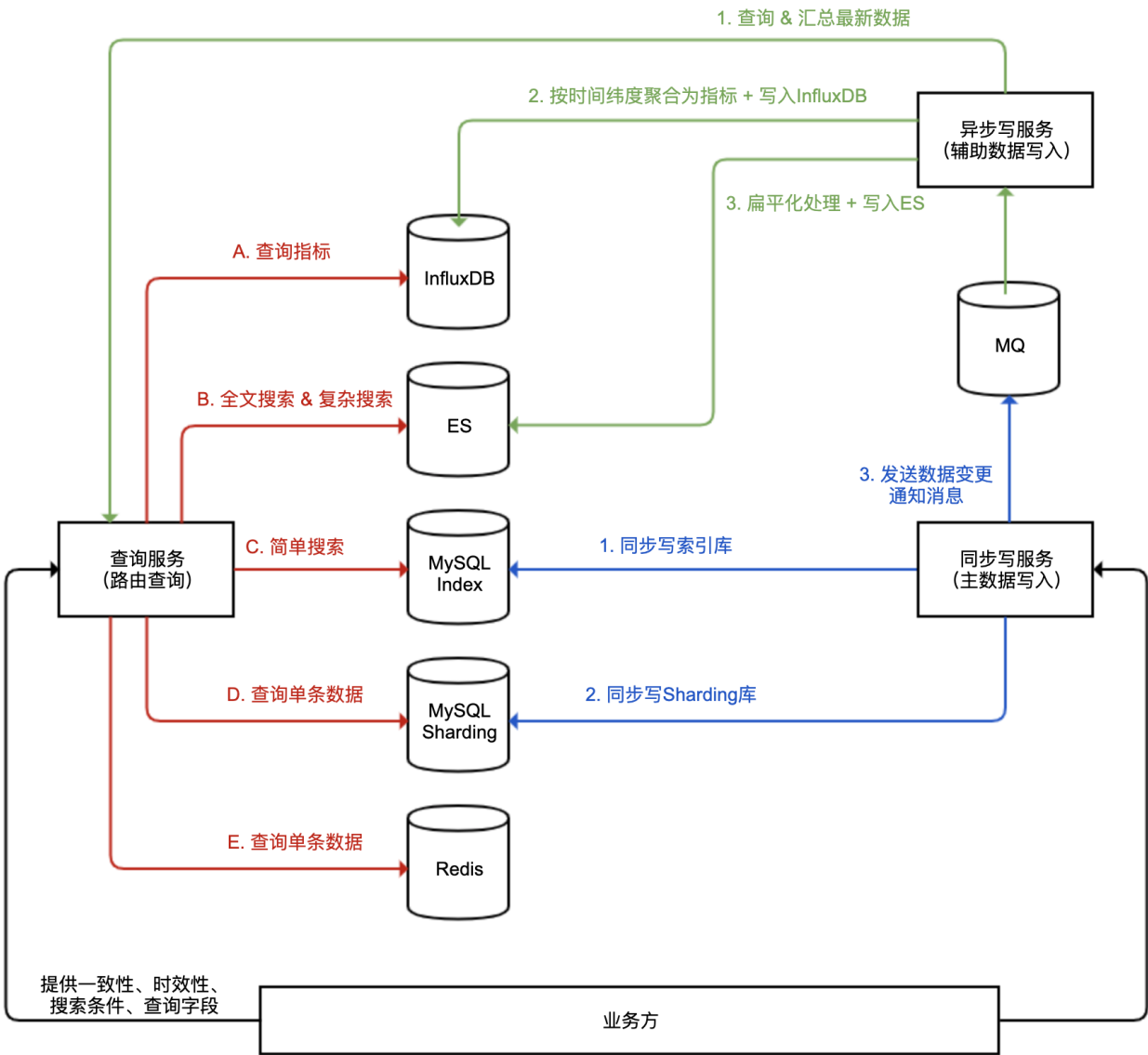
又比如，ES 的 Lucene 采用了 FST (Finite State Transducer) 索引 + 倒排索引，空间效率高，适合对变动不频繁的数据做索引，实现全文搜索。存储系统本身不可能对一份数据使

用多种数据结构保存，因此不可能适用于所有场景。

虽然在大多数业务场景下，MySQL 的性能都不算太差，但对于数据量大、访问量大、业务复杂的互联网应用来说，MySQL 因为实现了 ACID（原子性、一致性、隔离性、持久性）会比较重，而且横向扩展能力较差、功能单一，无法扛下所有数据量和流量，无法应对所有功能需求。因此，我们需要通过架构手段，来组合使用多种存储系统，取长补短，实现 $1+1>2$ 的效果。

我来举个例子。我们设计了一个**包含多个数据库系统的、能应对各种高并发场景的一套数据服务的系统架构**，其中包含了同步写服务、异步写服务和查询服务三部分，分别实现主数据库写入、辅助数据库写入和查询路由。

我们按照服务来依次分析下这个架构。



首先要明确的是，重要的业务主数据只能保存在 MySQL 这样的关系型数据库中，原因有三点：

RDBMS 经过了几十年的验证，已经非常成熟；

RDBMS 的用户数量众多，Bug 修复快、版本稳定、可靠性很高；

RDBMS 强调 ACID，能确保数据完整。

有两种类型的查询任务可以交给 MySQL 来做，性能会比较好，这也是 MySQL 擅长的地方：

按照主键 ID 的查询。直接查询聚簇索引，其性能会很高。但是单表数据量超过亿级后，性能也会衰退，而且单个数据库无法承受超大的查询并发，因此我们可以把数据表进行 Sharding 操作，均匀拆分到多个数据库实例中保存。我们把这套数据库集群称作 Sharding 集群。

按照各种条件进行范围查询，查出主键 ID。对二级索引进行查询得到主键，只需要查询一棵 B+ 树，效率同样很高。但索引的值不宜过大，比如对 varchar(1000) 进行索引不太合适，而索引外键（一般是 int 或 bigint 类型）性能就会比较好。因此，我们可以在 MySQL 中建立一张“索引表”，除了保存主键外，主要是保存各种关联表的外键，以及尽可能少的 varchar 类型的字段。这张索引表的大部分列都可以建上二级索引，用于进行简单搜索，搜索的结果是主键的列表，而不是完整的数据。由于索引表字段轻量并且数量不多（一般控制在 10 个以内），所以即便索引表没有进行 Sharding 拆分，问题也不会很大。

如图上蓝色线所示，写入两种 MySQL 数据表和发送 MQ 消息的这三步，我们用一个**同步写服务**完成了。我在“[异步处理](#)”中提到，所有异步流程都需要补偿，这里的异步流程同样需要。只不过为了简洁，我在这里省略了补偿流程。

然后，如图中绿色线所示，有一个**异步写服务**，监听 MQ 的消息，继续完成辅助数据的更新操作。这里我们选用了 ES 和 InfluxDB 这两种辅助数据库，因此整个异步写数据操作有**三步**：

1. MQ 消息不一定包含完整的数据，甚至可能只包含一个最新数据的主键 ID，我们需要根据 ID 从查询服务查询到完整的数据。

2. 写入 InfluxDB 的数据一般可以按时间间隔进行简单聚合，定时写入 InfluxDB。因此，这里会进行简单的客户端聚合，然后写入 InfluxDB。
3. ES 不适合在各索引之间做连接（Join）操作，适合保存扁平化的数据。比如，我们可以把订单下的用户、商户、商品列表等信息，作为内嵌对象嵌入整个订单 JSON，然后把整个扁平化的 JSON 直接存入 ES。

对于数据写入操作，我们认为操作返回的时候同步数据一定是写入成功的，但是由于各种原因，异步数据写入无法确保立即成功，会有一定延迟，比如：

异步消息丢失的情况，需要补偿处理；

写入 ES 的索引操作本身就会比较慢；

写入 InfluxDB 的数据需要客户端定时聚合。

因此，对于**查询服务**，如图中红色线所示，我们需要根据一定的上下文条件（比如查询一致性要求、时效性要求、搜索的条件、需要返回的数据字段、搜索时间区间等）来把请求路由到合适的数据库，并且做一些聚合处理：

需要根据主键查询单条数据，可以从 MySQL Sharding 集群或 Redis 查询，如果对实时性要求不高也可以从 ES 查询。

按照多个条件搜索订单的场景，可以从 MySQL 索引表查询出主键列表，然后再根据主键从 MySQL Sharding 集群或 Redis 获取数据详情。

各种后台系统需要使用比较复杂的搜索条件，甚至全文搜索来查询订单数据，或是定时分析任务需要一次查询大量数据，这些场景对数据实时性要求都不高，可以到 ES 进行搜索。此外，MySQL 中的数据可以归档，我们可以在 ES 中保留更久的数据，而且查询历史数据一般并发不会很大，可以统一路由到 ES 查询。

监控系统或后台报表系统需要呈现业务监控图表或表格，可以把请求路由到 InfluxDB 查询。

重点回顾

今天，我通过三个案例分别对比了缓存数据库 Redis、时间序列数据库 InfluxDB、搜索数据库 ES 和 MySQL 的性能。我们看到：

Redis 对单条数据的读取性能远远高于 MySQL，但不适合进行范围搜索。

InfluxDB 对于时间序列数据的聚合效率远远高于 MySQL，但因为没有主键，所以不是一个通用数据库。

ES 对关键字的全文搜索能力远远高于 MySQL，但是字段的更新效率较低，不适合保存频繁更新的数据。

最后，我们给出了一个混合使用 MySQL + Redis + InfluxDB + ES 的架构方案，充分发挥了各种数据库的特长，相互配合构成了一个可以应对各种复杂查询，以及高并发读写的存储架构。

主数据由两种 MySQL 数据表构成，其中索引表承担简单条件的搜索来得到主键，Sharding 表承担大并发的主键查询。主数据由同步写服务写入，写入后发出 MQ 消息。

辅助数据可以根据需求选用合适的 NoSQL，由单独一个或多个异步写服务监听 MQ 后异步写入。

由统一的查询服务，对接所有查询需求，根据不同的查询需求路由查询到合适的存储，确保每一个存储系统可以根据场景发挥所长，并分散各数据库系统的查询压力。

今天用到的代码，我都放在了 GitHub 上，你可以点击 [🔗 这个链接](#) 查看。

思考与讨论

1. 我们提到，InfluxDB 不能包含太多 tag。你能写一段测试代码，来模拟这个问题，并观察下 InfluxDB 的内存使用情况吗？
2. 文档数据库 MongoDB，也是一种常用的 NoSQL。你觉得 MongoDB 的优势和劣势是什么呢？它适合用在什么场景下呢？

关于数据存储，你还有其他心得吗？我是朱晔，欢迎在评论区与我留言分享你的想法，也欢迎你把今天的内容分享给你的朋友或同事，一起交流。

6月-7月课表抢先看

充 ¥500 得 ¥580

赠「¥ 118 月球主题 AR 笔记本」



【点击】图片, 立即查看>>>

© 版权归极客邦科技所有, 未经许可不得传播售卖。页面已增加防盗追踪, 如有侵权极客邦将依法追究其法律责任。

上一篇 25 | 异步处理好用, 但非常容易用错

下一篇 27 | 数据源头: 任何客户端的东西都不可信任

精选留言 (12)

写留言



2020-05-15

其实, 具体使用何种数据库, 如果了解其原理, 那就很容易弄懂了。

我就说说ES,

ES, 因为其本身全文索引, 和复杂查询的高性能, 最主要还是依赖于其分词之后简历的映射表。基本可以理解为类似数据库索引一样的存在。索引的弊端, 也就是ES的弊端。

索引, 基本目的是空间换时间, 用一些冗余的节点数据来优化查询性能。但是带来的问...

展开

作者回复: 说的不错



4



那一刻

2020-05-14

我说说个人对于MongoDB的理解，它的优势：

- 1.schema free，不需要额外定义列名，每个文档的key需要一致。
- 2.对于json结构存储和读取方便，比如取json某个key的value比较方便。相对于mysql存储json需要序列化和反序列化操作。
- 3.查询来看，3.0之后的mongo采用了B+树和LSM两种方式，来优化读多写少和写多读...

展开 ▾



2



3



hellojd

2020-05-18

老师没说mongo数据库，我的理解是如果不是acid要求特别高的地方，都可以将mysql替换为mongo.不知道理解对不对。

展开 ▾

作者回复: 非重要数据，并且数据结构不固定的，插入量又很大的原始数据（比如爬虫原始数据）可以考虑Mongo。从个人喜好而言，综合性NOSQL，我更喜欢ES而不是Mongo，Mongo在数据量到TB级别我感觉不稳定，Sharding也不那么好用，一家之言。



2



Demon.Lee

2020-05-14

现在公司的业务对ACID没有那么高的要求，已经全部换成了NoSQL，主要就是MongoDB+Redis+ES，自从用了MongoDB，我感觉解脱了，再也不用去设计一堆关联的表了，字段的扩充或变化都没有什么影响，字段取值的检验全部在应用程序中解决，有点不爽是，写各种复杂查询和统计时没有写rdms的sql那么溜。另外就是，我们使用了Keys去redis查询所有相关的数据，一般数据量不大，但是我一直想优化下，今天老师又提了，必须优...

展开 ▾



2



G小调

2020-05-18

老师 有个疑问？你图里写的mysql索引库 是指表上建立了索引的库吗

作者回复: 是指这个表大多字段都建了索引，主要用于简单搜索，索引库对比Sharding库



1



kyl

2020-05-14

说一下我对mongodb的理解，望老师指正。mongodb因为是文档型数据库以json格式存储，所以可以很方便的存储各种类型的数据，同时mongodb横向扩展只需增加分片比MySQL更加方便，数据量很大的场景下感觉性能优于MySQL。但是mongodb对事务支持比较差，虽然4.0引入了事务，但是可能有坑，另外mongodb不支持表的关联查询，所以还是要根据实际业务场景进行选择。

展开 ∨

作者回复: 基本没错的 mongodb建议用于非重要业务初始数据保存



1



似曾相识

2020-05-14

老师 Influxdb和es 这些数据库做辅助数据库，需要保存全量数据吗？还是根据业务保存部分字段？

作者回复: es一般是保存全量数据，甚至是超全量数据，意思就是比mysql保存的数据还要全。influxdb作为时间序列数据库只能保存加工后的业务或系统指标数据，无法保存实际的业务数据。



1



汝林外史

2020-05-14

又是干货满满，很多新接触的东西，感谢老师。

对于MySQL擅长的地方的第二点不是很理解：

1.不是不建议设置外键吗？

2.专门弄个索引表放主键与外键的关联关系，那岂不是每张表都要配这么一个索引表，这不浪费内存吗？ ...

展开 ∨

作者回复: 1. 我是指逻辑含义上这个字段是外键的作用，不是指要外键关系的绑定

2. 这个方案是一套针对大数据高并发的系统（比如订单系统的）的复合数据引擎方案，不是说普通的业务表都要这么做

3. 不是几倍的关系，索引表的字段是全量字段的子集，索引表不会做Sharding，这点你没有我的意思，可以再详细看一下文中的说明：

对二级索引进行查询得到主键，只需要查询一棵 B+ 树，效率同样很高。但索引的值不宜过大，比如对 varchar(1000) 进行索引不太合适，而索引外键（一般是 int 或 bigint 类型）性能就会比

较好。因此，我们可以在 MySQL 中建立一张“索引表”，除了保存主键外，主要是保存各种关联表的外键，以及尽可能少的 varchar 类型的字段。这张索引表的大部分列都可以建上二级索引，用于进行简单搜索，搜索的结果是主键的列表，而不是完整的数据。由于索引表字段轻量并且数量不多（一般控制在 10 个以内），所以即便索引表没有进行 Sharding 拆分，问题也不会很大。



1



那一刻

2020-05-14

请问老师，文章提到的多数据库系统例子里，redis写入是怎么实现的呢？

作者回复: 取决于缓存的使用策略，比如Cache aside, Read through, Write through，可以根据你需要的方案把写入redis的操作落到读服务或同步写服务去实现。



1



fly12580

2020-05-14

redis还可以用来做分布式并发锁。

展开



1



终结者999号

2020-05-22

老师，我想问一下对于一个高并发的系统，索引库您们保存多长时间呢？没有夜维清理吗？

作者回复: 至少三个月 取决于业务上让用户查多久的订单 更长期的打到es



永夜

2020-05-20

老师你好，我们这边主要做一些交互式数据可视化系统，目前用的主要是postgres数据库，但是一旦数据量比较大，几百万条的数据的一些过滤统计都会比较慢，需要20~30s，我们也需要用到数组，json这样的特殊类型，不知道有没有其它的数据库或者框架能提高这种场景下的效率问题。

展开 ∨

作者回复: 使用ES, 没有Sharding的关系型数据库承载不了你这么大的数据量的统计

