NoSQL 这个词在近些年正变得随处可见. 但是到底 “NoSQL” 指的是什么? 它是如何并且为什么这么有用? 在本文, 我们将会通过纯 Python (我比较喜欢叫它, “轻结构化的伪代码”) 写一个 NoSQL 数据库来回答这些问题.

**OldSQL**

很多情况下, SQL 已经成为 “数据库” (database) 的一个同义词. 实际上, **SQL** 是 *Strctured Query Language* 的首字母缩写, 而并非指数据库技术本身. 更确切地说, 它所指的是从 **RDBMS** (关系型数据库管理系统, *Relational Database Management System* ) 中检索数据的一门语言. MySQL, MS SQL Server 和 Oracle 都属于 RDBMS 的其中一员.

RDBMS 中的 R, 即 “Relational” (有关系，关联的), 是其中内容最丰富的部分. 数据通过 *表 (table)* 进行组织, 每张表都是一些由 *类型 (type)* 相关联的 *列 (column)* 构成. 所有表, 列及其类的类型被称为数据库的 *schema* (架构或模式). schema 通过每张表的描述信息完整刻画了数据库的结构. 比如, 一张叫做 Car 的表可能有以下一些列:

* Make: a string
* Model: a string
* Year: a four-digit number; alternatively, a date
* Color: a string
* VIN(Vehicle Identification Number): a string

在一张表中, 每个单一的条目叫做一 *行 (row)*, 或者一条 *记录 (record)*. 为了区分每条记录, 通常会定义一个 *主键 (primary key)*. 表中的 *主键* 是其中一列 , 它能够唯一标识每一行. 在表 Car 中, VIN 是一个天然的主键选择, 因为它能够保证每辆车具有唯一的标识. 两个不同的行可能会在 Make, Model, Year 和 Color 列上有相同的值, 但是对于不同的车而言, 肯定会有不同的 VIN. 反之， 只要两行拥有同一个 VIN, 我们不必去检查其他列就可以认为这两行指的的就是同一辆车.

**Querying**

SQL 能够让我们通过对数据库进行 *query (查询)* 来获取有用的信息. *查询* 简单来说, 查询就是用一个结构化语言向 RDBMS 提问, 并将其返回的行解释为问题的答案. 假设数据库表示了美国所有的注册车辆, 为了获取 *所有的* 记录, 我们可以通过在数据库上进行如下的 *SQL 查询* :

Python



|  |  |
| --- | --- |
| 1 | SELECT Make, Model FROM Car; |

将 SQL 大致翻译成中文:

* “SELECT”: “向我展示”
* “Make, Model”: “Make 和 Model 的值”
* “FROM Car”: “对表 Car 中的每一行”

也就是, “向我展示表 Car 每一行中 Make 和 Model 的值”. 执行查询后， 我们将会得到一些查询的结果, 其中每个都是 Make 和 Model. 如果我们仅关心在 1994 年注册的车的颜色, 那么可以:

Python



|  |  |
| --- | --- |
| 1 | SELECT Color FROM Car WHERE Year = 1994; |

此时, 我们会得到一个类似如下的列表:

Python



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | Black  Red  Red  White  Blue  Black  White  Yellow |

最后, 我们可以通过使用表的 *(primary key) 主键* , 这里就是 VIN 来指定查询一辆车:

Python



|  |  |
| --- | --- |
| 1 | SELECT \* FROM Car WHERE VIN = '2134AFGER245267' |

上面这条查询语句会返回所指定车辆的属性信息.

主键被定义为唯一不可重复的. 也就是说, 带有某一指定 VIN 的车辆在表中至多只能出现一次. 这一点非常重要，为什么? 来看一个例子:

**Relations**

假设我们正在经营一个汽车修理的业务. 除了其他一些必要的事情, 我们还需要追踪一辆车的服务历史, 即在该辆车上所有的修整记录. 那么我们可能会创建包含以下一些列的 ServiceHistory 表:

VIN | Make | Model | Year | Color | Service Performed | Mechanic | Price | Date

这样, 每次当车辆维修以后, 我们就在表中添加新的一行, 并写入该次服务我们做了一些什么事情, 是哪位维修工, 花费多少和服务时间等.

但是等一下, 我们都知道，对于同一辆车而言，所有车辆自身信息有关的列是不变的。 也就是说，如果把我的 Black 2014 Lexus RX 350 修整 10 次的话， 那么即使 Make, Model, Year 和 Color 这些信息并不会改变，每一次仍然重复记录了这些信息. 与无效的重复记录相比， 一个更合理的做法是对此类信息只存储一次， 并在有需要的时候进行查询。

那么该怎么做呢？ 我们可以创建第二张表： Vehicle , 它有如下一些列:

VIN | Make | Model | Year | Color

这样一来， 对于 ServiceHistory 表， 我们可以精简为如下一些列:

VIN | Service Performed | Mechanic | Price | Date

你可能会问，为什么 VIN 会在两张表中同时出现？ 因为我们需要有一个方式来确认在 ServiceHistory 表的 *这* 辆车指的就是 Vehicle 表中的 *那* 辆车, 也就是需要确认两张表中的两条记录所表示的是同一辆车。 这样的话,我们仅需要为每辆车的自身信息存储一次即可. 每次当车辆过来维修的时候， 我们就在 ServiceHistory 表中创建新的一行， 而不必在 Vehicle 表中添加新的记录。 毕竟， 它们指的是同一辆车。

我们可以通过 SQL 查询语句来展开 Vehicle 与 ServiceHistory 两张表中包含的隐式关系:

Python



|  |  |
| --- | --- |
| 1 | SELECT Vehicle.Model, Vehicle.Year FROM Vehicle, ServiceHistory WHERE Vehicle.VIN = ServiceHistory.VIN AND ServiceHistory.Price > 75.00; |

该查询旨在查找维修费用大于 $75.00 的所有车辆的 Model 和 Year. 注意到我们是通过匹配 Vehicle 与 ServiceHistory 表中的 VIN 值来筛选满足条件的记录. 返回的将是两张表中符合条件的一些记录, 而 “Vehicle.Model” 与 “Vehicle.Year” , 表示我们只想要 Vehicle表中的这两列.

如果我们的数据库没有 *索引 (indexes)* (正确的应该是 indices), 上面的查询就需要执行 *表扫描 (table scan)* 来定位匹配查询要求的行。 table scan 是按照顺序对表中的每一行进行依次检查， 而这通常会非常的慢。 实际上, table scan 实际上是所有查询中最慢的。

可以通过对列加索引来避免扫描表。 我们可以把索引看做一种数据结构， 它能够通过预排序让我们在被索引的列上快速地找到一个指定的值 (或指定范围内的一些值). 也就是说， 如果我们在 Price 列上有一个索引， 那么就不需要一行一行地对整个表进行扫描来判断其价格是否大于 75.00, 而是只需要使用包含在索引中的信息 “跳” 到第一个价格高于 75.00 的那一行， 并返回随后的每一行（由于索引是有序的， 因此这些行的价格至少是 75.00）。

当应对大量的数据时， 索引是提高查询速度不可或缺的一个工具。当然， 跟所有的事情一样，有得必有失， 使用索引会导致一些额外的消耗： 索引的数据结构会消耗内存，而这些内存本可用于数据库中存储数据。这就需要我们权衡其利弊，寻求一个折中的办法， 但是为经常查询的列加索引是 *非常* 常见的做法。

**The Clear Box**

得益于数据库能够检查一张表的 *schema* (描述了每列包含了什么类型的数据), 像索引这样的高级特性才能够实现， 并且能够基于数据做出一个合理的决策。 也就是说, 对于一个数据库而言， 一张表其实是一个 “黑盒” (或者说透明的盒子) 的反义词？

当我们谈到 NoSQL 数据库的时候要牢牢记住这一点。 当涉及 *query* 不同类型数据库引擎的能力时， 这也是其中非常重要的一部分。

**Schemas**

我们已经知道, 一张表的 *schema* , 描述了列的名字及其所包含数据的类型。它还包括了其他一些信息， 比如哪些列可以为空， 哪些列不允许有重复值， 以及其他对表中列的所有限制信息。 在任意时刻一张表只能有一个 schema, 并且 *表中的所有行必须遵守 schema 的规定* 。

这是一个非常重要的约束条件。 假设你有一张数据库的表， 里面有数以百万计的消费者信息。 你的销售团队想要添加额外的一些信息 （比如， 用户的年龄）, 以期提高他们邮件营销算法的准确度。 这就需要来 *alter (更改)* 现有的表 — 添加新的一列。 我们还需要决定是否表中的每一行都要求该列必须有一个值。 通常情况下， 让一个列有值是十分有道理的， 但是这么做的话可能会需要一些我们无法轻易获得的信息（比如数据库中每个用户的年龄）。因此在这个层面上，也需要有些权衡之策。

此外，对一个大型数据库做一些改变通常并不是一件小事。为了以防出现错误，有一个回滚方案非常重要。但即使是如此，一旦当 schema 做出改变后，我们也并不总是能够撤销这些变动。 schema 的维护可能是 DBA 工作中最困难的部分之一。

**Key/Value Stores**

在 “NoSQL” 这个词存在前， 像 memcached 这样的 *键/值 数据存储 (Key/Value Data Stores)* 无须 table schema 也可提供数据存储的功能。 实际上， 在 K/V 存储时， 根本没有 “表 （table）” 的概念。 只有 *键 (keys)* 与 *值 （values）* . 如果键值存储听起来比较熟悉的话， 那可能是因为这个概念的构建原则与 Python 的 dict 与 set 相一致: 使用 hash table （哈希表） 来提供基于键的快速数据查询。 一个基于 Python 的最原始的 NoSQL 数据库, 简单来说就是一个大的字典 (dictionary) .

为了理解它的工作原理，亲自动手写一个吧！ 首先来看一下一些简单的设计想法：

* 一个 Python 的 dict 作为主要的数据存储
* 仅支持 string 类型作为键 (key)
* 支持存储 integer, string 和 list
* 一个使用 ASCLL string 的简单 TCP/IP 服务器用来传递消息
* 一些像 INCREMENT, DELETE , APPEND 和 STATS 这样的高级命令 (command)

有一个基于 ASCII 的 TCP/IP 接口的数据存储有一个好处， 那就是我们使用简单的 telnet程序即可与服务器进行交互, 并不需要特殊的客户端 (尽管这是一个非常好的练习并且只需要 15 行代码即可完成)。

对于我们发送到服务器及其它的返回信息，我们需要一个 “有线格式”。下面是一个简单的说明：

**Commands Supported**

* PUT
  + 参数： Key, Value
  + 目的： 向数据库中插入一条新的条目 (entry)
* GET
  + 参数： Key
  + 目的： 从数据库中检索一个已存储的值
* PUTLIST
  + 参数： Key, Value
  + 目的： 向数据库中插入一个新的列表条目
* APPEND
  + 参数： Key, Value
  + 目的： 向数据库中一个已有的列表添加一个新的元素
* INCREMENT
  + 参数： key
  + 目的： 增长数据库的中一个整型值
* DELETE
  + 参数： Key
  + 目的： 从数据库中删除一个条目
* STATS
  + 参数： 无 (N/A)
  + 目的： 请求每个执行命令的 成功/失败 的统计信息

现在我们来定义消息的自身结构。

**Message Structure**

**Request Messages**

一条 *请求消息 (Request Message)* 包含了一个命令（command），一个键 (key), 一个值 (value), 一个值的类型（type）. 后三个取决于消息类型，是可选项, 非必须。; 被用作是分隔符。即使并没有包含上述可选项， 但是在消息中仍然必须有三个 ; 字符。

Python



|  |  |
| --- | --- |
| 1 | COMMAND; [KEY]; [VALUE]; [VALUE TYPE] |

* **COMMAND** 是上面列表中的命令之一
* **KEY** 是一个可以用作数据库 key 的 string （可选）
* **VALUE** 是数据库中的一个 integer, list 或 string (可选)
  + list 可以被表示为一个用逗号分隔的一串 string, 比如说, “red, green, blue”
* **VALUE TYPE** 描述了 **VALUE** 应该被解释为什么类型
  + 可能的类型值有：INT, STRING, LIST

**Examples**

* "PUT; foo; 1; INT"
* "GET; foo;;"
* "PUTLIST; bar; a,b,c ; LIST"
* "APPEND; bar; d; STRING"
* "GETLIST; bar; ;"
* STATS; ;;
* INCREMENT; foo;;
* DELETE; foo;;

**Reponse Messages**

一个 *响应消息 (Reponse Message)* 包含了两个部分， 通过 ; 进行分隔。第一个部分总是 True|False , 它取决于所执行的命令是否成功。 第二个部分是命令消息 (command message), 当出现错误时，便会显示错误信息。对于那些执行成功的命令，如果我们不想要默认的返回值（比如 PUT）, 就会出现成功的信息。 如果我们返回成功命令的值 (比如 GET), 那么第二个部分就会是自身值。

**Examples**

* True; Key [foo] set to [1]
* True; 1
* True; Key [bar] set to [['a', 'b', 'c']]
* True; Key [bar] had value [d] appended
* True; ['a', 'b', 'c', 'd']
* True; {'PUTLIST': {'success': 1, 'error': 0}, 'STATS': {'success': 0, 'error': 0}, 'INCREMENT': {'success': 0, 'error': 0}, 'GET': {'success': 0, 'error': 0}, 'PUT': {'success': 0, 'error': 0}, 'GETLIST': {'success': 1, 'error': 0}, 'APPEND': {'success': 1, 'error': 0}, 'DELETE': {'success': 0, 'error': 0}}

**Show Me The Code!**

我将会以块状摘要的形式来展示全部代码。 整个代码不过 180 行，读起来也不会花费很长时间。

**Set Up**

下面是我们服务器所需的一些样板代码：

Python



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | """NoSQL database written in Python"""    # Standard library imports  import socket    HOST = 'localhost'  PORT = 50505  SOCKET = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  STATS = {      'PUT': {'success': 0, 'error': 0},      'GET': {'success': 0, 'error': 0},      'GETLIST': {'success': 0, 'error': 0},      'PUTLIST': {'success': 0, 'error': 0},      'INCREMENT': {'success': 0, 'error': 0},      'APPEND': {'success': 0, 'error': 0},      'DELETE': {'success': 0, 'error': 0},      'STATS': {'success': 0, 'error': 0},      } |

很容易看到， 上面的只是一个包的导入和一些数据的初始化。

**Set up(Cont’d)**

接下来我会跳过一些代码， 以便能够继续展示上面准备部分剩余的代码。 注意它涉及到了一些尚不存在的一些函数， 不过没关系， 我们会在后面涉及。 在完整版（将会呈现在最后）中， 所有内容都会被有序编排。 这里是剩余的安装代码：

Python



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | COMMAND\_HANDERS = {      'PUT': handle\_put,      'GET': handle\_get,      'GETLIST': handle\_getlist,      'PUTLIST': handle\_putlist,      'INCREMENT': handle\_increment,      'APPEND': handle\_append,      'DELETE': handle\_delete,      'STATS': handle\_stats,  }    DATA = {}      def main():      """Main entry point for script"""      SOCKET.bind(HOST, PORT)      SOCKET.listen(1)      while 1:          connection, address = SOCKET.accept()          print('New connection from [{}]'.format(address))          data = connection.recv(4096).decode()          command, key, value = parse\_message(data)          if command == 'STATS':              response = handle\_stats()          elif command in ('GET', 'GETLIST', 'INCREMENT', 'DELETE'):              response = COMMAND\_HANDERS[command](key)          elif command in (                  'PUT',                  'PUTLIST',                  'APPEND', ):              response = COMMAND\_HANDERS[command](key, value)          else:              response = (False, 'Unknown command type {}'.format(command))          update\_stats(command, response[0])          connection.sandall('{};{}'.format(response[0], response[1]))          connection.close()      if \_\_name\_\_ == '\_\_main\_\_':      main() |

我们创建了 COMMAND\_HANDLERS, 它常被称为是一个 *查找表 (look-up table)* . COMMAND\_HANDLERS 的工作是将命令与用于处理该命令的函数进行关联起来。 比如说， 如果我们收到一个 GET 命令， COMMAND\_HANDLERS[command](key) 就等同于说 handle\_get(key) . 记住，在 Python 中, 函数可以被认为是一个值，并且可以像其他任何值一样被存储在一个 dict 中。

在上面的代码中， 虽然有些命令请求的参数相同，但是我仍决定分开处理每个命令。 尽管可以简单粗暴地强制所有的 handle\_ 函数接受一个 key 和一个 value , 但是我希望这些处理函数条理能够更加有条理， 更加容易测试，同时减少出现错误的可能性。

注意 socket 相关的代码已是十分极简。 虽然整个服务器基于 TCP/IP 通信， 但是并没有太多底层的网络交互代码。

最后还须需要注意的一小点: DATA 字典， 因为这个点并不十分重要， 因而你很可能会遗漏它。 DATA 就是实际用来存储的 key-value pair, 正是它们实际构成了我们的数据库。

**Command Parser**

下面来看一些 *命令解析器 (command parser)* , 它负责解释接收到的消息：

Python



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | def parse\_message(data):      """Return a tuple containing the command, the key, and (optionally) the      value cast to the appropriate type."""      command, key, value, value\_type = data.strip().split(';')      if value\_type:          if value\_type == 'LIST':              value = value.split(',')          elif value\_type == 'INT':              value = int(value)          else:              value = str(value)      else:          value = None      return command, key, value |

这里我们可以看到发生了类型转换 (type conversion). 如果希望值是一个 list, 我们可以通过对 string 调用 str.split(',') 来得到我们想要的值。 对于 int, 我们可以简单地使用参数为 string 的 int() 即可。 对于字符串与 str() 也是同样的道理。

**Command Handlers**

下面是命令处理器 (command handler) 的代码. 它们都十分直观，易于理解。 注意到虽然有很多的错误检查， 但是也并不是面面俱到, 十分庞杂。 在你阅读的过程中，如果发现有任何错误请移步 [这里](https://github.com/liuchengxu/blog-cn/issues) 进行讨论.

Python



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78 | def update\_stats(command, success):      """Update the STATS dict with info about if executing \*command\* was a      \*success\*"""      if success:          STATS[command]['success'] += 1      else:          STATS[command]['error'] += 1      def handle\_put(key, value):      """Return a tuple containing True and the message to send back to the      client."""      DATA[key] = value      return (True, 'key [{}] set to [{}]'.format(key, value))      def handle\_get(key):      """Return a tuple containing True if the key exists and the message to send      back to the client"""      if key not in DATA:          return (False, 'Error: Key [{}] not found'.format(key))      else:          return (True, DATA[key])      def handle\_putlist(key, value):      """Return a tuple containing True if the command succeeded and the message      to send back to the client."""      return handle\_put(key, value)      def handle\_putlist(key, value):      """Return a tuple containing True if the command succeeded and the message      to send back to the client"""      return handle\_put(key, value)      def handle\_getlist(key):      """Return a tuple containing True if the key contained a list and the      message to send back to the client."""      return\_value = exists, value = handle\_get(key)      if not exists:          return return\_value      elif not isinstance(value, list):          return (False, 'ERROR: Key [{}] contains non-list value ([{}])'.format(              key, value))      else:          return return\_value      def handle\_increment(key):      """Return a tuple containing True if the key's value could be incremented      and the message to send back to the client."""      return\_value = exists, value = handle\_get(key)      if not exists:          return return\_value      elif not isinstance(list\_value, list):          return (False, 'ERROR: Key [{}] contains non-list value ([{}])'.format(              key, value))      else:          DATA[key].append(value)          return (True, 'Key [{}] had value [{}] appended'.format(key, value))      def handle\_delete(key):      """Return a tuple containing True if the key could be deleted and the      message to send back to the client."""      if key not in DATA:          return (              False,              'ERROR: Key [{}] not found and could not be deleted.'.format(key))      else:          del DATA[key]      def handle\_stats():      """Return a tuple containing True and the contents of the STATS dict."""      return (True, str(STATS)) |

有两点需要注意： *多重赋值 (multiple assignment)* 和代码重用. 有些函数仅仅是为了更加有逻辑性而对已有函数的简单包装而已， 比如 handle\_get 和 handle\_getlist . 由于我们有时仅仅是需要一个已有函数的返回值，而其他时候却需要检查该函数到底返回了什么内容， 这时候就会使用 *多重赋值* 。

来看一下 handle\_append . 如果我们尝试调用 handle\_get 但是 key 并不存在时， 那么我们简单地返回 handle\_get 所返回的内容。 此外， 我们还希望能够将 handle\_get 返回的 tuple 作为一个单独的返回值进行引用。 那么当 key 不存在的时候， 我们就可以简单地使用 return return\_value .

如果它 *确实存在* ， 那么我们需要检查该返回值。并且， 我们也希望能够将 handle\_get 的返回值作为单独的变量进行引用。 为了能够处理上述两种情况，同时考虑需要分开处理结果的情形，我们使用了多重赋值。 如此一来， 就不必书写多行代码， 同时能够保持代码清晰。 return\_value = exists, list\_value = handle\_get(key) 能够显式地表明我们将要以至少两种不同的方式引用 handle\_get 的返回值。

**How Is This a Database?**

上面的程序显然并非一个 RDBMS, 但却绝对称得上是一个 NoSQL 数据库。它如此易于创建的原因是我们并没有任何与 *数据 (data)* 的实际交互。 我们只是做了极简的类型检查，存储用户所发送的任何内容。 如果需要存储更加结构化的数据， 我们可能需要针对数据库创建一个 schema 用于存储和检索数据。

既然 NoSQL 数据库更容易写， 更容易维护，更容易实现， 那么我们为什么不是只使用 mongoDB 就好了？ 当然是有原因的， 还是那句话，有得必有失， 我们需要在 NoSQL 数据库所提供的数据灵活性 (data flexibility) 基础上权衡数据库的可搜索性 (searchability).

**Querying Data**

假如我们上面的 NoSQL 数据库来存储早前的 Car 数据。 那么我们可能会使用 VIN 作为 key, 使用一个列表作为每列的值， 也就是说, 2134AFGER245267 = ['Lexus', 'RX350', 2013, Black] . 当然了， 我们已经丢掉了列表中每个索引的 *涵义 (meaning)* . 我们只需要知道在某个地方索引 1 存储了汽车的 Model , 索引 2 存储了 Year.

糟糕的事情来了， 当我们想要执行先前的查询语句时会发生什么？ 找到 1994 年所有车的颜色将会变得噩梦一般。 我们必须遍历 DATA 中的 *每一个值* 来确认这个值是否存储了 car 数据亦或根本是其他不相关的数据， 比如说检查索引 2, 看索引 2 的值是否等于 1994，接着再继续取索引 3 的值. 这比 table scan 还要糟糕，因为它不仅要扫描每一行数据，还需要应用一些复杂的规则来回答查询。

NoSQL 数据库的作者当然也意识到了这些问题，(鉴于查询是一个非常有用的 feature) 他们也想出了一些方法来使得查询变得不那么 “遥不可及”。一个方法是结构化所使用的数据，比如 JSON, 允许引用其他行来表示关系。 同时， 大部分 NoSQL 数据库都有名字空间 (namespace) 的概念， 单一类型的数据可以被存储在数据库中该类型所独有的 “section” 中，这使得查询引擎能够利用所要查询数据的 “shape” 信息。

当然了，尽管为了增强可查询性已经存在 （并且实现了）了一些更加复杂的方法， 但是在存储更少量的 schema 与增强可查询性之间做出妥协始终是一个不可逃避的问题。 本例中我们的数据库仅支持通过 key 进行查询。 如果我们需要支持更加丰富的查询， 那么事情就会变得复杂的多了。

**Summary**

至此, 希望 “NoSQL” 这个概念已然十分清晰。 我们学习了一点 SQL, 并且了解了 RDBMS 是如何工作的。 我们看到了如何从一个 RDBMS 中检索数据 （使用 SQL *查询 (query)*）. 通过搭建了一个玩具级别的 NoSQL 数据库, 了解了在可查询性与简洁性之间面临的一些问题， 还讨论了一些数据库作者应对这些问题时所采用的一些方法。

学习资料可以咨询若兰老师QQ:2463598866