# GitHub 平台的社交网络属性挖掘：数据驱动的关系型与非关系型数据库比较

傅尔正，田翔宇，赵予珩，朱秦　复旦大学计算机学院
{18307130163, 18307110203, 18307130071, 18307130092}@fudan.edu.cn

## 摘要

GitHub 是目前全世界最流行的源代码托管服务平台之一。全球各地的用户对 GitHub 上代码仓库的各类不同行为数据形成了一张巨大的交互网络。为了探究这张交互网络中社交网络的性质，我们从 GH Archive 上获取了 GitHub 上的用户行为数据，将其分别存储在 MySQL， Redis 和 Neo4j 三种数据库中，并对 GitHub 的社交特征进行挖掘。在这个过程中，我们对比了不同数据库的存储性能，并通过针对不同特征的查询操作对比了这三种数据库的性能效率，分析了这三种数据库的表现以及适用场景，并利用这些数据和数据库实现了一个简单的 GitHub 社交网络分析平台 GitHub Search。

## 关键字

GitHub，社交网络，数据库，性能对比，MySQL，Neo4j，Redis

## Abstract

GitHub is currently one of the most popular source code hosting service platforms, where a bunch of operations on different repositories triggered by users all over the world form a gigantic interaction graph. In order to figure out the basic properties of this interaction graph and excavate the social-network features lying under it, we obtain massive user behavior data on GH Archive, and create their storage in different databases including MySQL, Redis and Neo4j. Through this process, we also make a comparison among the storage performance of these databases, design diverse query operations with certain aims to match these three databases with best efficiency, and analyze the performance as well as their applicable scenarios of these three databases. What is more, we also implement a simple GitHub social network analysis platform, GitHub Search, utilizing data, databases and analysis from our work.

## Keywords

GitHub, Social Network, Database, Performance，MySQL，Neo4j，Redis

# 1 引言

随着包括计算机网络技术、数据库技术以及其他众多在内的计算机技术的不断迭新和增速发展，以互联网为代表的计算机应用为越来越多的用户提供了各式各样的服务，而对等地，用户也对这些应用提出了纷繁复杂的需求。其中，社交媒体的出现和发展也与频繁而又高效的用户到用户交互、用户到服务器交互以及服务器到服务器交互相互促进和发展。以用户社交为目的创建的社交媒体，如 Facebook, Twitter, Instagram 等，已形成一定规模的社交网络，并被广泛研究[1, 2]。然而，对于另外一些社交性并不太强，或是主要功能基点不设立在社交上的应用，在如今互联网的背景下，也或多或少地加入了用户交互功能。例如，在某网络购物平台，用户间的相互推荐也构成了一张关系网络[3]，它也能为整个平台的信息流向提供研究的基点，帮助后台发现更多潜在的用户需求。但是我们发现，对于非社交网络服务上的一些社交特征研究尚为欠缺。基于上述背景，我们着眼于 GitHub，一个目前全世界最大的源代码托管服务平台以及合作平台之一，来开展我们的研究。尽管 GitHub 的主要功能是代码协作，但它同时为用户提供了一系列用户交互操作(i.e 事件, event)，例如，评论（Issue Comment），收藏（Star），关注（Watch）等等。这些事件同样在 GitHub 平台构建出一张“隐性”的交互网络，其内嵌的社交网络性质有待我们的进一步发掘。

另一方面，网络服务其数据存储规模随着互联网的发展也在呈指数型增长。为了应对各色的应用请求以及分析需求，服务商需要不断提高其服务质量，例如不断更新机器以及服务器架构，使得其性能能够不成为全局的瓶颈，或者是运用高效算法，例如图算法和最新的深度学习算法来更有效地解决实际问题。但往往被忽视的是，操作数据的存储方式，也深刻影响着计算的效能，而根据需求的不同，不同的存储方式也为需求的实现赋能或减分[4]。例如，要求高安全性的银行系统，往往选择依赖性较好的关系型数据库存储用户数据；而对于社交网络上的一些分析需求，图数据库则有可能能更好表现出图网络上的某些特征[5, 6]。尽管之前已有一系列关系型与非关系型数据库的性能对比的论文[7, 8, 9, 10, 11]，但是据我们所知，目前还没有人在 GitHub 的交互数据集上做过相关的研究。基于此，我们以在第一段提出的发掘 GitHub 社交网络性质为需求，探寻各类需求适用的数据存储模式，以此来优化这些需求的实现。

从以上两个方面出发，本文的贡献有三：

第一，我们充分挖掘了 GitHub 这一代码托管平台上的社交网络特征。我们将特征分为三点：单点行为、社群特征以及全局分析。单点行为中我们着眼于整个社交网络的局部特征，从某个用户（actor）或某个代码仓库（repo）的视角观察其产生的关联交互特征，包括查询某个用户在一段时间内发起的事件信息、某个时间段中操作过同一个仓库的用户信息、以及用户发起事件的活跃时间段等等；对于社群特征，我们关心用户在社交网络处于的结构位置，即一个用户在事务发起语境下所建立的交互图的“邻居”、所在连通块的大小、以及其聚集系数等；在全局分析上，我们在更大范围的观察整个平台的特征，并提供有效信息给相关研究，包括全局最受关注的仓库（类似于“微博热搜”的概念）、最活跃的用户群体、以及被发起次数最多的事件类型排序等等。我们从微观到宏观不同尺度下整合分析整个网络的交互信息。

第二，我们分别选择 MySQL, Redis 和 Neo4j 三个互不相同的数据库，实现上面提出的请求，并做相关的性能对比和分析。我们首先将获取的 GitHub 上的交互数据集以不同形式导入上述三种数据库，获得不同的效率呈现。而对于上面提到的三类特征，我们通过各类不同的查询操作来实现：对于单点行为，我们通过一系列简单查询

来实现；对于社群特征，Neo4j 可以发挥其图数据库的特长，而其余两个数据库则需要利用其它方法来实现；而全局分析，则对数据库的遍历以及排序等功能提出了要求。在上面各类查询中，三种不同的数据库各在不同的方面体现出了效率上的优势与劣势，但除了效率上的呈现，在实现这些查询的过程中，效率低的数据库也可能展现出其他方面的优势，可能在其他的情境下会更明显地展现。我们将在正文部分对这部分数据库的表现进行详细阐述和分析。

第三，我们实现了一个简单的服务请求平台 GitHub Search，将上述研究成果落地。利用 Vue 的 Web 框架，我们向使用者提供了实现第一部分部分操作的接口，并在后端与我们在第二部分找到的最适合的数据库相链接。当用户发送请求时，平台能通过最佳的数据库快速处理，并将结果返回到客户端并进行渲染。例如，用户可以通过此平台查询用户数据和行为分析，以及针对仓库的分析，还可以查询最受欢迎的仓库和用户等。

综上所述，在本文中，我们从 GitHub 出发，通过挖掘其内在的社交网络属性，面向不同的请求测试了在不同类数据库下的反应效率，对比分析了在不同情境下的数据存储需要，以及最终具体实现了这些请求。

# 2 数据集及获取

基于对 GitHub 内在社交网络属性挖掘的需求，我们选取 GH Archive[1]作为本次项目使用的数据集。GH Archive 是一个对 GitHub 上公开的活动进行记录与归类的项目，其利用 GitHub 开放的 API 对全世界数千万用户在 GitHub 上的事务数据进行了记录，如 Commit，Fork，Star 等各类事务。GH Archive 按时间顺序对这些数据进行系统的收集，整理并且按小时存档。其将收集到的事务数据信息以 json 文件形式存储，以时间轴排序的方式提供给研究者下载，用户仅通过 http 客户端就可以实现对于数据档案的便捷访问。

我们选取 GH Archive 作为数据集的原因有：

1. 高度规格化数据：GH Archive 提供的规格化 json 格式数据适合数据的存储管理，并为用户，仓库和事务分别提供了唯一编号标识。事务发布者和事务对应代码仓库信息全部包含在事务信息中，方便处理数据之间的逻辑关系，极大地节省了原始数据处理的开销。

2. 丰富的数据多样性：GitHub 中提供了超过 20 种事务类型，从常见的 Fork，Commit 到 Opening New Tickets 和 Commenting。相对应地，GH Archive 中也包含这些多样的事务类型。同时，其中也包括全世界不同地区的 GitHub 用户数据和代码仓库信息，提供了多样化的数据供进一步分析和挖掘。

3. 大量灵活的数据：GH Archive 满足大数据要求，其包含了 GitHub 中近 5 年的全部事务数据，若全部下载整个数据集将达到 PB 级别。此数据集支持按小时下载数据，可以灵活地针对目标数据量，目标分析时间和成本等方面进行进一步的数据筛选和使用。

根据目标数据大小和数据的时效性，我们选取 2020 年 1 月 1 日 0 时至 2020 年 1 月 7 日 23 时的 GitHub 事务数据进行下载。根据 GH Archive 上提供的下载方式进行下载，数据大小总计超过 10GB。

数据集中每一条记录即为一个事务记录，其数据格式如 Fig. 1。


```
{
    "id":"11185281240",
    "type":"ForkEvent",
    "actor":{
        "id":13564443,
        "login":"fagan2888",
        "display_login":"fagan2888",
        "gravatar_id":"",
        "url":"https://api.github.com/users/fagan2888",
        "avatar_url":"https://avatars.githubusercontent.com/u/13564443?"
    },
    "repo":{
        "id":92406528,
        "name":"natashawatkins/linearmodels",
        "url":"https://api.github.com/repos/natashawatkins/linearmodels"
    },
    "payload":{ ... },
    "public":true,
    "created_at":"2020-01-01T00:00:00Z"
}
```


Fig. 1 Fork Event 数据格式示例

[1] 项目网址：https://www.gharchive.org/

GH Archive 中对每条事务以及每个用户和代码仓库设置了唯一编号（id 字段和 actor 下的 id 字段，repo 下的 id 字段）。事务信息包括事务编号，类型，发起者信息（即用户信息），对应仓库信息，详细信息字段，隐私权限信息以及事务时间戳。用户信息包括用户编号，用户昵称，用户头像地址等数据。仓库信息包括仓库编号，仓库名称和仓库地址。

在将上述数据导入数据库前，需要进行数据预处理，考虑到 payload 字段也就是事务的详细信息过于冗长且在不同事务之间具有较大差异，同时其对于用户间交互模式的依赖程度不高，不能提供更多有效的社交网络属性信息，我们删除了该字段。后续测试中也表明，余下更精简的事务数据已经能够有效地支持进一步的数据分析和处理。

# 3 数据存储

为了比较关系型数据库与非关系型数据库的性能差异，本文以 MySQL 作为关系型数据库的代表，Neo4j 与 Redis 作为非关系型数据库的代表进行后续研究。三者都是时下应用相当广泛的数据库。

## 3.1 MySQL

MySQL 是最流行的关系型数据库之一，其由于性能高、成本低、可靠性好，被广泛使用在各种中小型网站中。由于 MySQL 可以利用 SQL 语言进行数据库的查询，同时提供了多种常用编程语言，如 Java，C++，Python 等访问数据库的接口[12]，MySQL 能够为用户提供简单易用同时效率较优的服务。这也是我们选择 MySQL 作为关系型数据库的代表的原因。

根据 GH Archive 数据集中的数据格式和应用需求，我们构建了如 Fig. 2 的 ER 关系模型，并根据该 ER 模型建立 MySQL 数据库。

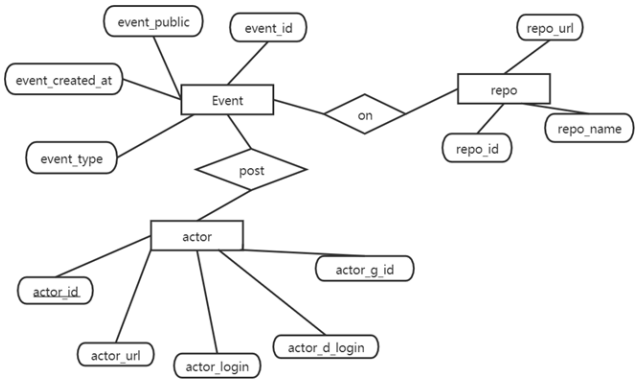



Fig. 2 ER 图

通常情况下，我们对 MySQL 数据库进行插入数据时一般会使用一条 INSERT 语句插入一行数据，但在本次实验选取的数据集中，数据量的行数达到了千万的数量级，普通的插入方式的效率极其低下。这是由于对于每一条 INSERT 语句，数据库的关系引擎都需要解析 INSERT 的语义，在大规模的数据导入中这种重复的操作是大量冗余的[13]，因此需要将多行数据合并在一条 INSERT 语句中进行插入[14]。这里我们将每小时的事务数据合并在一条 INSERT 语句中插入（约 10 万行数据），经过测试发现这种操作可以将插入速度提高数十倍。

利用选取的一周时间内的数据建立 MySQL 数据库总共耗时 3291 秒。以天数作为数据单位，其数据插入速率随着数据量的增加而变化的趋势如 Fig. 3 所示。可以发现，随着数据量的增加，MySQL 的数据插入速率会大幅度下降，降到最高速度的约 40%后趋于稳定。

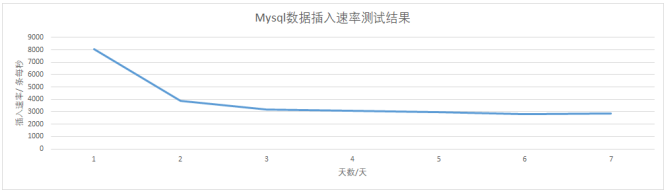



Fig. 3 MySQL 插入数据速率波动

## 3.2 Neo4j

Neo4j 是时下流行的图数据库（Graph Database）。此类数据库有效地利用图模型来储

存，管理和更新数据库，包含节点（Node）和节点之间的关系（Relationship），Fig. 4 提供了 Neo4j 存储的一个例子，其中节点间通过关系相互连接。

为了提高效率，Neo4j 使用了两级缓存机制（文件系统缓存和对象缓存）。在磁盘上，每个节点的大小固定为 9 字节，只包含指向其连接的第一条关系的指针，由此构建一个双向关系链表（与图的存储方式相同），对于节点的属性也采用同样的方法存储。所以根据节点序号进行随即查询的效率为 $\Theta(1)$ 而不是传统的 $\Theta(\log n)$。在磁盘上，关系也采用固定大小（33 字节）存储，包含了其属性以及连接的节点 id，可以经由关系快速访问节点。而在对象缓存中，节点包含其连接的所有关系，可以通过节点访问快速定位所有相关的关系，关系只包含其属性，关系按照类型进行划分，保证了遍历的高效性。[18]

从上述存储结构可以看出，与传统的关系型数据库需要通过遍历表的方式访问关系不同，Neo4j 通过访问节点来完成对关系的访问，很大程度上节省了访问开销。同理其对于复杂连接的查询也具有更好的性能表现。同时，Neo4j 也提供了大量封装的图算法接口供调用，这些算法在涉及图性质操作的场景下表现出显著优势。所以处理大量包含复杂关系的数据时，Neo4j 有着明显的性能优势[19]。考虑到我们所选取的数据集中存在复杂的社交网络关系呈现出一些图的特性，使用 Neo4j 来分析这些复杂关系从而挖掘社交网络属性是很好的选择。[20]

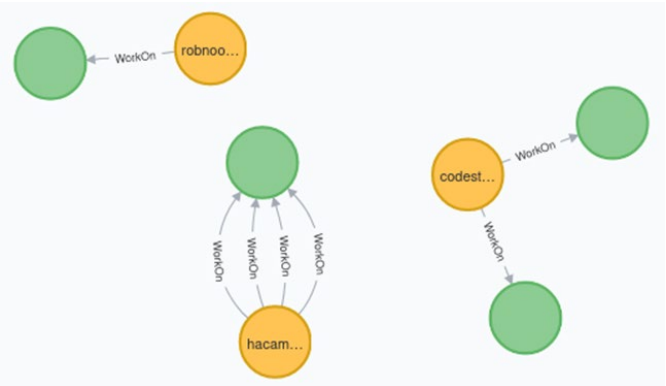



Fig. 4 Neo4j 的存储结构示意图

根据 Neo4j 的以上性质，结合所选数据集的数据结构，在存储时，设置两类节点：用户（Actors）和代码仓库（Repo）。用户和仓库之间通过工作关系（WorkOn）相连接，每条工作关系即代表用户向仓库发起的一次事务。与此同时，为了维护用户和仓库的唯一编号，在用户编号（actor_id）和仓库编号（repo_id）上设置了唯一索引，在保证唯一性的情况下能够加提高查询效率，由于导入数据时需要去重，添加索引也同样能够提高导入数据的速度。

导入数据时通过已有的 json 文件生成 CSV 文件进行导入，考虑到 Neo4j 的导入性能相比其他两种数据库较差，不采取用 CREATE 语句逐条插入的方式来执行所以并未统计单日数据的插入速率，但仍定时对于数据量进行检查统计。实验表明在建立索引的情况下直接导入 CSV 文件的性能是通过逐条导入的 6 倍以上。导入数据时，节点的导入效率较高：花费 55 秒导入超过 161 万个节点；关系的导入效率较差，花费 27255 秒（约 7.5h）导入约 1080 万个关系。显然对于关系的导入是 Neo4j 的瓶颈。并且我们发现，随着导入数据量的增加，关系导入性能急剧下降：初始的 550 万个关系耗时约 2h，接下来第 2~4h 导入了约 200 万条关系，最后 4~7.5h 仅导入约 300 万条关系。导致 Neo4j 插入时效率较低的原因是由于 LOAD CSV 支持实时插入，在新增数据的时候，不仅要统计原有的数据信息，检查并建立新增数据与原有数据的联系（如自增 id，并入已有 label，建立节点之间的关系等），还要动态地维护索引结构，这项开销随着数据数量级的增大而上升明显。并且考虑到 Neo4j 还需要维护数据的图存储结构，包括关系和属性的双向链表，以及节点对关系和属性的指针结构，关系对始末节点的引用，这些额外开销导致了数据导入时的性能较低。

## 3.3 Redis

Redis 是一种开源的，遵守 BSD 协议的[22]，目前常见的一种高性能键值数据库（Key-Value

Database)。相较于其他键值数据库，Redis 的主要优势有二。首先，Redis 是内存数据库，其操作均在内存中运行，运行效率较高，而且同时也提供了可持久化操作，能够通过内存与磁盘的交换来实现对大数据量的访问[23]。其次，Redis 支持内置的多种数据类型。作为键值数据库，Redis 不止支持对字符串的索引，它同时支持 string，list，set，zset，hash 五种数据类型，功能丰富[21]。更具体地阐述，可以把一个 Redis 数据库视为一个字典，其中 key 是字符串，value 是上述五个变量类型中的任意一个，其中，string 类型其实是字符串到整数的字典，list 是无序列表，set 是不重集合，zset 是有序集合，可支持对不重集合的排序，hash 是字符串到字符串的字典。

上面提到的 Redis 特性是我们选择 Redis 作为我们研究的一个数据库的重要原因。由于我们面向用户交互数据的访问和查询，面向的场景往往是在线和及时的，Redis 更有可能满足这些请求的需要。同时，用户交互行为的分析往往又是非常多样的，其查询操作可能出现各种不同的类型，而 Redis 丰富的数据结构便可使得在不同任务采用不同的数据类型来优化实现以及用简单的引用做到复杂的操作提供可能。

根据 GH Archive 的数据特征，以及从 ER 图和图关系吸取相关概念，我们主要选择 Redis 中的 hash 结构来存储主要数据。在 Redis 中，我们同样将所有条目分为三类“实体”，即 actor，repo 和 event，每类实体是一个 hash 字典，以字符串存储了此实体的属性信息。但需要注意的是，不同实体间并无严格界限，仅以键值的前缀名加以区分（如，actor:33403015，repo:230988632)。同时，为了满足之后一系列的查询需求，我们也将在 Redis 中无法体现的关系型数据库的依赖关系，同时也是图数据库中的边集，作为额外信息存储下来，以此协助高频率的查询请求。在之后可以看到，这个操作将极大提高一些查询的计算效率。当然，我们要注意时刻保持数据的一致性问题。

操作上，我们使用 Python 接口访问 Redis 数据库，读取 GH Archive 上用 JSON 存储的事件信息并解析，再调用相关语句将数据存入 Redis。我们统计了插入这批数据的总时间以及分时效率。在 Redis 中，总共插入了约 1300 万条信息以及约 500 万条索引，用时分别为 21833 秒和 5244 秒。值得一提是，随着 Redis 库中数据不断增多，插入数据的效率仍保持在相对稳定的范围之内，这与 Redis 内部的哈希机制有关。Fig. 5 给出了插入不同天数的数据 Redis 耗时的波动情况。

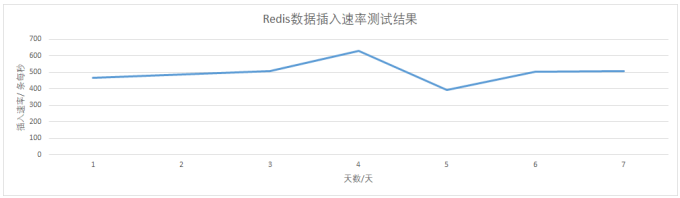



Fig. 5 Redis **插入数据速率波动**

## 3.4 横向对比

综合上述数据导入性能，我们发现关系型数据库 MySQL 在数据导入方面的性能有明显优势，而非关系型数据库 Redis 和 Neo4j 在这里的表现不佳，Fig. 6 展示了三种不同的数据库数据插入耗时对比（其中 Redis 展示的时间未包括插入索引的时间，其索引耗时 5244 秒），Redis 添加索引后的总时间消耗与 Neo4j 几乎相同。这是由于非关系型数据库侧重于查询优化导致其在数据导入时需要维护较多的复杂数据结构（比如 Neo4j 的图结构和 Redis 的哈希），产生了额外的开销。

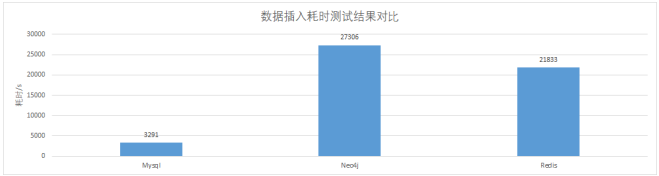
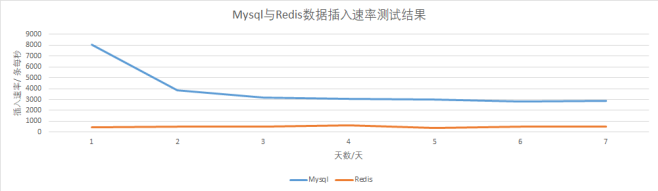



Fig. 6 **三种数据库导入时间**

如 Fig. 7 所示，我们针对 MySQL 和 Redis 提供的单日数据插入速率的数据进行了对比，MySQL

速度全面优于 Redis，这也是由于上述原因导致




的。

Fig. 7 MySQL 和 Redis 插入数据速率波动对比

# 4 数据操作及性能对比

GH Archive 是 GitHub 上用户事件的数据集，我们已将 2020 年第一周的数据以不同形式存入了三类数据库。面向挖掘 GitHub 社交网络属性的需要，我们建立了以事件为关系，用户和仓库为节点的交互图。在此节中，我们将在三个层面研究这个网络的社交属性：单点行为、社群特征以及全局分析。我们对于上面三点针对性地提出了各三种查询操作，并在相同的环境下用三个数据库中存储的数据实现查询，交叉验证了结果的正确性。同时，我们记录每个操作的时间消耗，并详细分析了其内在机理，在此基础上，我们也根据不同数据库的特点分析了不同应用场景下的表现。

实验基于以下配置环境进行：

1. 硬件环境：8GB 内存，inter(R) Core(TM) i7-8750H 6 核 CPU，150G SSD

2. 软件环境：Ubuntu 18.04，Python 3.8.3，MySQL 8.0.2，Neo4j 4.2.1 enterprise，Redis 6.0.9

## 4.1 单点行为

在这一部分我们主要研究个体的行为模式，从 GitHub 上的两种不同的个体即用户和代码仓库的角度，分别观察并统计其交互情况。考虑到 GitHub 上公开的用户交互行为几乎全部通过发布事务来进行，我们通过统计用户和代码仓库的相关事务的时间分布，及事务发布频度与种类来进行研究。

### 4.1.1 指定用户或仓库的活跃时间分布

首先我们希望统计用户和代码仓库的活跃度随时间的分布。在此基础上我们设计了查询来统计七天内指定的用户或者代码仓库相关的事务，并将这些事务按小时归类统计。Fig. 8 和 Fig. 9 展示了三种不同的数据库执行此项操作的性能测试结果对比。

可以看出 Redis 在这一查询操作中有明显优势，这是因为我们在建立 Redis 库时，还加入了一张额外的表，用来存储 Redis 原本无法体现的依赖，同时也相当于 Neo4j 中储存的关系。加之 Redis 内存数据库的特性，这极大提高了查询请求的效率。

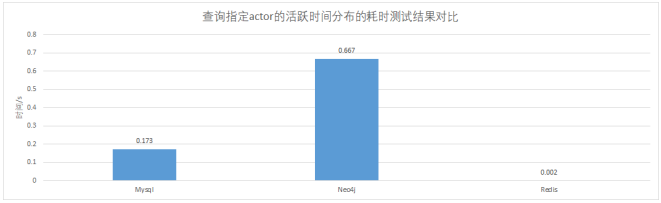



Fig. 8 查询第 108 号用户的活跃时间分布

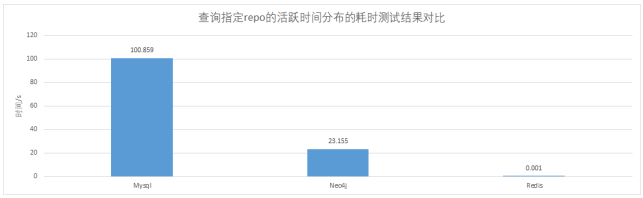



Fig. 9 查询第 230501783 号代码仓库的活跃时间分布

而对于 MySQL 和 Neo4j，由于 MySQL 在基本数据查询方面有很成熟的应用，数据量较小时（比如查询第 108 号用户相关信息）MySQL 的表现较优。但是考虑到 MySQL 每次获取相关事务时都需要查询事务表所以其操作所需时间随数据量呈非线性增长。但 Neo4j 每次查询节点对应的关系只需要常数时间，其耗时随数据量呈线性增长。故当数据量增大时，Neo4j 的相对性能逐渐提高，最

终超过 MySQL。此外考虑到 MySQL 中事务表中用户索引优先级高于代码仓库索引，因此对于指定仓库的查找速度相较于对指定用户的查找也会有所下降。

### 4.1.2 指定用户发布的事务统计

紧接着我们尝试分析某用户发布过的所有事务来获取其在 GitHub 上的行动方式。据此我们设计了对于指定用户查询其相关联的所有事务的操作。Fig. 10 展示了该项查询操作在不同的数据库中的性能对比。

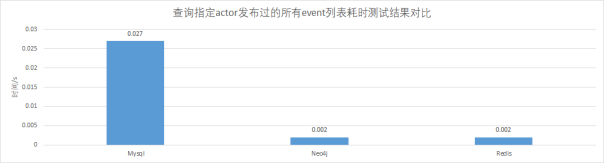



Fig. 10 **查询第** 108 **号用户发布过的所有事务**

从图中不难看出，虽然这是一条简单查询，但与 Redis 和 Neo4j 相比 MySQL 的性能却有着相对明显的差距。

Redis 的高性能与 4.1.1 中的分析结果相同，也进一步验证了前述分析的正确性。而 Neo4j 的高性能取决于其对于数据所采用的图结构储存方式。由于该项查询不需要进行 4.1.1 当中对于事务属性的模糊匹配，只需要返回用户节点对应的所有事务集合，Neo4j 的储存结构优势在该项查询中能够得到极好的体现。故其性能与 Redis 相持平。

### 4.1.3 指定代码仓库的参与者

代码仓库作为 GitHub 上的用户聚集处，是连接不同用户之间的纽带。我们希望查询指定代码仓库的参与者列表来进一步进行分析，为后续的社群特征研究提供支持。

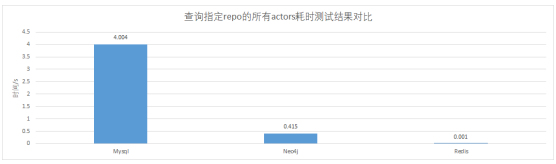



Fig. 11 **查询第** 230501783 **号仓库的所有参与者**

和前两个查询实验结果类似，Redis 在该项查询中延续了其优秀的性能，Neo4j 同样如此，但由于需要查找两组不同的节点之间的对应关系，额外进行了对用户节点中符合要求的用户的查询，产生了额外的开销。结果如 Fig. 11 所示。MySQL 方面与 4.1.1 相似，在数据量较大时查询事务表有较高的开销。

### 4.1.4 小结

通过以上三个查询实验，在简单查询方面，我们不难看出 Redis 有着更好的表现，Neo4j 也展现出了较优的性能。这主要是由于 Redis 针对编号建立了多个额外的索引结构，可以快速根据用户编号或代码仓库编号快速查询对应的事务。Neo4j 对于关系查询的较优性能取决于其图结构的储存模式，在后续复杂查询中的表现更为明显。

## 4.2 社群特征

在第二部分，我们以 4.1 设计的简单查询为基础进行了进一步的分析，着重研究用户在社交网络所处的位置以及其所属群体的结构特点。其中包括某用户在事务发起语境下所建立的交互图的“邻居”，所在连通块的大小及其聚集系数。尝试对 GitHub 上的社交网络属性进行挖掘。

### 4.2.1 指定用户的多跳邻居

邻居是社交网络中的常见概念，通过查询某位用户的邻居，我们可以获取用户在社交网络中的位置以及社交网络中用户的交互关系和信息的传递方向。我们希望实现对于指定用户的 2，4，6 跳邻居的查找。与某点相邻的所有点构成的集合即是该点的单跳邻居（one-hop neighbourhood），多跳邻居（multi-hop neighbourhood）是在此基础上的一个图论概念[15]，是邻居概念的拓展。这里的多跳邻居之所

以选取 2，4，6 是由于不同用户之间通过代码仓库相关联，用户之间的距离均为偶数。而考虑到样本用户群体的大小有限，设计 6 跳邻居足以满足用户关系的分析。Fig. 12，Fig. 13 和 Fig. 14 展示了第 100 号用户（与 2 个代码仓库相关联）的多跳邻居查询。


Fig. 12 **查询第 100 号用户的 2 跳邻居**


Fig. 13 **查询第 100 号用户的 4 跳邻居**


Fig. 14 **查询第 100 号用户的 6 跳邻居**

我们还尝试在与更多代码仓库相关联的用户上寻找其多跳邻居。由于在此基础上的 6 跳邻居并不能在常规时间内得到结论，对于第 108 号节点仅查找了其对应的 2 跳和 4 跳邻居。Fig. 15 与 Fig. 16 展示了相关的性能结果。


Fig. 15 **查询第 108 号用户的 2 跳邻居**


Fig. 16 **查询第 108 号用户的 4 跳邻居**

我们可以看出，Neo4j 在以多跳邻居为代表的图算法查询上有明显优势。其图结构的储存方式，使得其在通过节点访问关系，通过关系访问节点方面相较于关系型数据库都有着更优的性能。故总体上也能够表现出很高的查询效率。

在该项查询中 Redis 的速度相对最慢，其原因内在于中间变量的存储，事实上，此点查询中 MySQL 和 Redis 虽然实现过程几乎一致，但 MySQL 将中间变量，也就是众多邻居信息存到了 Python 数组中，与 MySQL 不同的是，Redis 可以将这些变量存入自己的数据库数据结构中。虽然这样放弃了一定效率，但数据不会因为查询操作的结束而消失，而可持久化存在于数据库中，在下一次查询中，Redis 能够加速其处理过程，而这直接面向批量查询的场景。

### 4.2.2 某用户所处的连通块规模

连通块是图算法中的重要概念，在社交网络中，用户所处的连通块可以视为其所属的社群，这些用户之间可以产生信息的交互。连通块的规模反映了用户所属社群的结构特点。根据上述研究需求我们设计了相对应的算法来实现对于用户所处连通块大小的查询。Fig. 17 展示了查询特定用户所处连通块大小的结果对比。


Fig. 17 **查询第 108 号用户的所在连通块大小**

通过上图可以看出，该项操作中，Neo4j 所消耗的时间极短，相比另两个数据库有明显的优势。Neo4j 实现时直接调用提供的 gds 图算法库，有相当高的性能表现。MySQL 与 Redis 都运用了并查集的算法来完成这条查询，考虑到 Neo4j 提供的 gds 算法进行了很多的底层优化，并查集算法在性能方面不如 Neo4j 所提供的成熟图算法。而 Redis 比较慢的原因与 4.2.1 的分析相同。

### 4.2.3 包含指定用户的闭三点组个数

我们期望在未来对于所选取的数据集进行进一步的深入分析，故设计算法查询包含指定用户的闭三点组个数。首先解释闭三点组的定义：假设图中有一部分点是两两相连的，那么可以找出很多个“三元环”，其对应的三点两两相连，称为闭三点组。闭三点组是图论中聚集系数(clustering coefficient)[16, 17]计算的主要部分，该系数的大小可以衡量社交网络中指定用户与其邻居的紧密程度。在我们的数据框架下，对于闭三点组中“相连”的定义进行了修改：如果两个用户在同一个代码仓库中工作，那我们认为这两个用户是“相连”的。这样更符合我们的分析目标也更贴近实际情况，即用户通过代码仓库相交互。Fig. 18 展示了对于给定用户查询其闭三点组的性能对比。

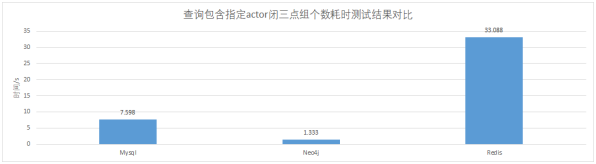



Fig. 18 **查询包含第 108 号用户的闭三点组个数**

尽管在 Neo4j 提供的 gds 库中包含相关的图算法，不过由于前述的改动，无法直接调用。但由于 Neo4j 在算法的每一步中对于邻居的查询都有着更优秀的性能，在该项操作中，Neo4j 也在整体上展现出了更高的效率。同样，Redis 因为将把运算中间变量存入内置数据结构而效率偏慢。

### 4.2.4 小结

在本节中，我们主要基于各种复杂的图算法对 GitHub 上的社交网络属性进行了挖掘。可以看出，Neo4j 表现出了与预期相同的极高的性能。这是由于 Neo4j 的储存结构即是图结构，不需要通过额外的算法进行数据的转化来贴合图结构，在很多情况下，Neo4j 在的社交网络属性挖掘方面还可以调用大量封装好的高性能算法如 gds 库，这也进一步提高了 Neo4j 在这方面的优势。而 Redis 在本节中的性能表现与 4.1 中有较大反差，这是面向为了可持久化、在线以及批量查询等场景做出的 trade-off。

## 4.3 全局分析

在第三部分中，我们针对整个平台为主体进行分析考察，希望观察 GitHub 上的活跃行为是否会出现明显的聚集现象，如统计最受关注的代码仓库，最活跃的用户群体以及不同类型事件的发起频度。

### 4.3.1 查询最受欢迎的仓库群体

首先我们希望实现一个类似“微博热搜”的功能，对于代码仓库，根据其参与者的事务发布频度评价其受关注程度，该评价机制类似于按点击量评价微博热度的机制。基于此想法我们执行查询以统计全部代码仓库中相关联事务数最多的十个仓库。Fig. 19 展示了三种不同的数据库执行此项操作的性能测试结果对比。

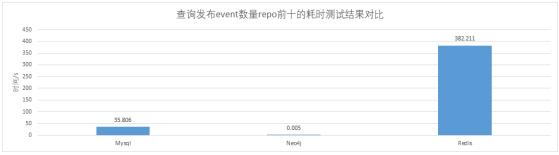



Fig. 19 **查询最受欢迎的仓库群体**

可以看出 Redis 在这项操作中的表现不尽如人意。之所以 Redis 在该项查询中性能不佳是因为该项查询涉及到了排序算法，对于 Neo4j 和 MySQL 来说他们的索引都支持快速排序。但同样地，Redis 能够动态地维护一个平衡树的数据结构，虽然造成了额外的开销，但提供了动态和在线排序的功能，支持在插入数据的过程中随时查询当前的排名，适宜各种线上场景使用，这是另外两种数据库所不支持的。通过该项查询可以看出效率低的

数据库如此处的 Redis 也可能展现出其他方面的优势。而 Neo4j 性能如此之高的原因即是其在对于关系的查询中具有很高的性能，这一点在前文中已经反复提及。

### 4.3.2 查询最活跃的用户群体

在第二部分，与前一部分类似，我们尝试查询 GitHub 中最活跃的用户群体，在进一步的研究中，可以把这些用户看作“微博大 V”，重点研究他们的行为。据此尝试查询发布事务数量前 5% 的用户群体。Fig. 20 展示了三种不同的数据库执行此项操作的性能测试结果对比。


Fig. 20 **查询最活跃的用户群体**

该项查询与 4.3.1 的查询逻辑基本相同，也涉及了排序算法，导致 Redis 性能表现不佳。相较而言，由于所需排序的用户节点数目急剧增大导致 Redis 的性能表现比前一部分更差。

### 4.3.3 不同类型事务的出现频度

除了研究个体行为的集中活跃特征，我们还关注不同事务在 GitHub 上的分布比例，考虑到用户是通过发布事务来进行交互，这有助于我们研究 GitHub 上用户的交互方式。Fig. 21 展示了对于事务频度统计的性能对比。


Fig. 21 **查询每种 event 的类型数量的排序结果**

由上图可知，该项操作中 MySQL 的速度超过 Neo4j 和 Redis。其原因为该项操作需要遍历所有的事务词条，也就是遍历所有的关系，Neo4j 在访问给定节点的关系方面的优势不能得到很好的体现。在这种情况下，由于 MySQL 实现了更多底层的优化，在性能表现上占优。而 Redis 依旧由于其对于排序结构的动态维护产生了较大的额外开销。

### 4.3.4 小结

排序查询中 Redis 由于不支持索引表上的快速排序，导致性能很低，但能实现满足原子性的实时查询。当涉及特定用户或者代码仓库相关的事务查询时，Neo4j 能够体现出明显的图数据库性能优势。而进对于事务表或者关系进行遍历，Neo4j 的优势则表现的并不明显。

## 5 实例展示

根据前述的需求设计了一个简单的服务请求平台 GitHub Search，在用户端提供与其前述需求对应的操作接口，包括四类功能：用户数据分析，用户行为分析，仓库活跃时间分析以及全局访问频度分析。前端采用 Web 架构，考虑美观性和实用性使用 Vue.js + CSS + elementUI + echarts 实现前端搭建，python 程序作为后端执行数据库交互。前后端之间采用 axios 实现通信。根据前述不同数据库在不同数据操作下的性能表现选择最优的数据库执行相应的数据库操作并接受后端返回值，客户端提供对应的数据可视化展示。


Fig. 22 **页面主菜单栏实例展示（按动按钮弹出选项）**

用户数据分析下设用户相关仓库列表查询和用户发布事务频度查询，根据客户端输入的用户 id

返回对应的用户创建的代码仓库(仅展示前 10 个代码仓库）列表或用户对所有事务的发布频度列表，在前端使用 elementUI 绘制表格展示。此项操作调用 Redis 执行。

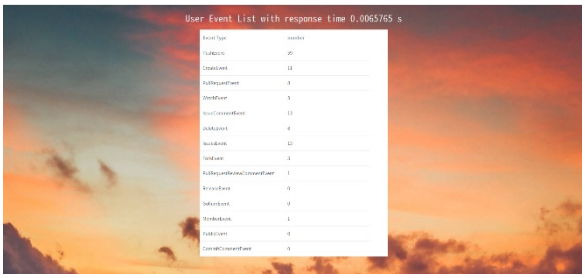



Fig. 23 用户创建的代码仓库列表


Fig. 24 用户发布事务频度表

考虑到用户在 GitHub 上的行为几乎全部通过发布事务（event）的方式表现，用户行为分析即是用户发布事务的时间分布分析。根据前端输入的用户 id 来查询该用户发布的所有事务，对这些事务按小时分类并统计每小时的事件发布总数。在前端调用 echarts 库绘制柱状图展示。此项操作调用 Redis 执行。效果如 Fig. 25。


Fig. 25 第 108 号用户行为的时间分布

对于仓库的活跃时间分析类似于用户行为分析，即统计针对某仓库所发布的事务的时间分析，同样要求前端输入仓库 id，接受返回的事务数据，绘制极坐标系下的堆叠柱状图展示。此项操作调用 Redis 执行。效果如 Fig. 26。

全局访问频度下设三个子选项：最受欢迎的仓库，最活跃用户以及时间发布频度统计。前两项直接统计相关联事务最多的前十位仓库和用户，前端绘制表格展示。此两项操作调用 Neo4j 执行。最后一项统计所有类型的事务发布频度，为方便统计全局的事件类型占比，采用饼图的形式展示。此项操作调用 MySQL 执行。

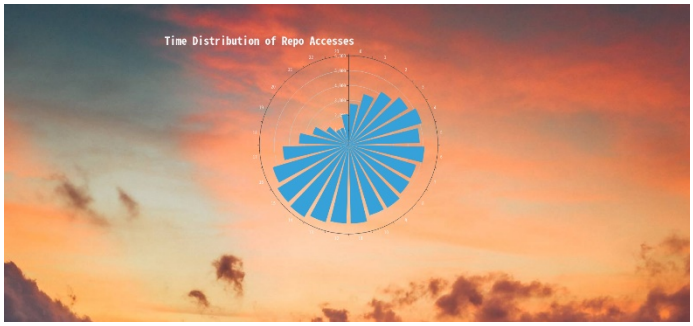



Fig. 26 第 230501783 号代码仓库活跃时间分布

Fig. 27 活跃用户榜 Fig. 28 活跃仓库榜 Fig. 29 事务种类分布如下所示。

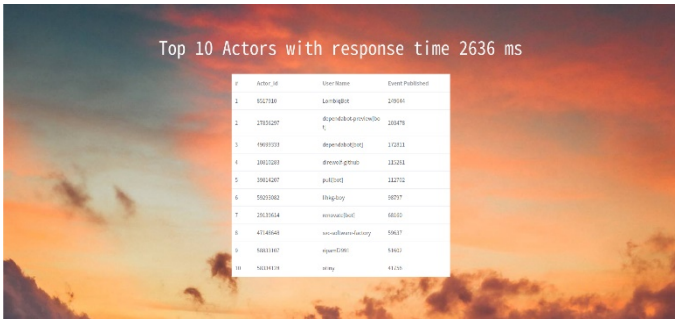



Fig. 27 活跃用户榜

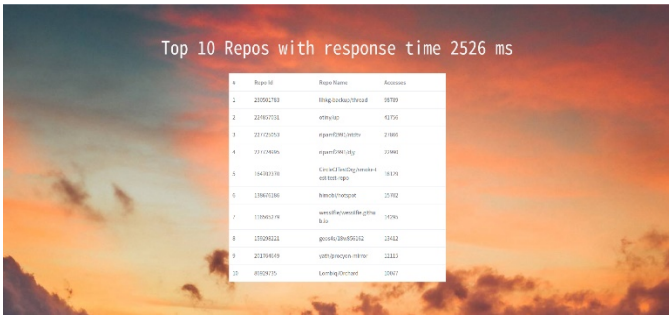



Fig. 28 活跃仓库榜

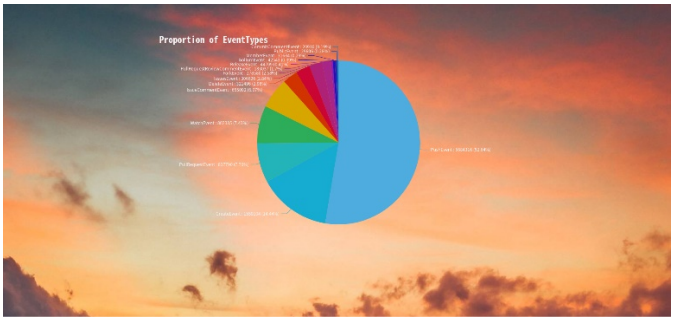



Fig. 29 事务种类分布

# 6 总结

本文中,我们首先从 GH Archive 数据集出发，基于这一用户行为数据集，构建出了 GitHub 上的交互图。为了充分挖掘 GitHub 上的社交网络属性，我们将待发掘的特征分为三点，即单点行为，社群特征及全局分析。在单点行为中，我们关注了个体的交互行为，包括了指定用户或仓库的信息；在社群特征中，我们关注到了单点在整个交互网络所处的环境，观察了用户邻居、所在连通块和聚集系数等信息；在全局分析中，我们关注整个网络的社交特征，例如整个交互网络的热点所在。通过多尺度、多方面的观察和分析，我们尽可能地把 GitHub 上社交属性体现了出来。

在实现上述挖掘的过程中，我们采用了三种数据库来实验，在不同的需求中，三种数据库表现出了不同的性能。Neo4j 在实验条件下大多条件有较为不错的性能，总体上更加适用于我们对 GH Archive 数据处理的需求，这本身是与我们将研究背景设立于社交网络的研究是分不开的，故它在图相关的操作性能相当突出，当查询涉及图结构相关的时候速度会非常之快，远优于其余两个数据库。但是 MySQL 仍能够保持较好的数据依赖，在读入数据速度方面有优势，并且在没有明显适用于其他两个数据库的实验条件下，由于成熟的底层优化可以表现出优越的性能。Redis 在简单查询方面有很高的查询效率，原因内在于其内存数据库的特征，在进行图相关的分析时，它也可以学习图数据库的特点，将图关系同样存储于数据库中，以加速查询的效率，但需要注意的是保持数据的一致性。但是在有排序需求时，性能会大幅下降，但因为能时刻将中间变量存储于内置数据结构中，其中便包含了可排序集合，其能够适应真实的在线服务场景。

我们的工作仍然有提升的空间。首先，基于结果来说，我们希望有时间根据运行出来的数据对 GitHub 上的社交行为进行进一步分析。其次，在请求设计上，可以增加删和改的操作，以面向更真实的场景，并比较其在三个数据库上的性能，此时 Redis 可能便会出现要在程序层面维护一致性的问题。最后，我们可以先从需求出发，寻找更与需求对接的数据库，或是按需求设计自己设计数据存储构架。

# 参考文献


[1] Viswanath, Bimal, et al. "On the evolution of user interaction in facebook." Proceedings of the 2nd ACM workshop on Online social networks. 2009.

[2] Kwak, Haewoon, et al. "What is Twitter, a social network or a news media?." Proceedings of the 19th international conference on World wide web. 2010.

[3] Lin, Tzu-Heng, Chen Gao, and Yong Li. "Cross: Cross-platform recommendation for social e-commerce." Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval. 2019.

[4] Biliris, Alexandros. The performance of three database storage structures for managing large objects. ACM SIGMOD Record 21.2 (1992): 276-285.

[5] Warchał, Łukasz. "Using Neo4j graph database in social network analysis." Studia Informatica 33.2A (2012): 271-279.

[6] 张凤军. 基于 Neo4j 图数据库的社交网络数据的研究与应用. MS thesis. 湖南大学, 2016.

[7] Rautmare S, Bhalerao D M. MySQL and NoSQL database comparison for IoT application[C], 2016 IEEE International Conference on Advances in Computer Applications (ICACA). IEEE, 2016: 235-238.

[8] Tang, Enqing, and Yushun Fan. Performance comparison between five NoSQL databases. 2016 7th International Conference on Cloud Computing and Big Data (CCBD). IEEE, 2016.
[9] Abubakar, Yusuf, Thankgod Sani Adeyi, and Ibrahim Gambo Auta. Performance evaluation of NoSQL systems using YCSB in a resource austere environment. Performance Evaluation 7.8 (2014): 23-27.
[10] Chickerur, Satyadhyan, Anoop Goudar, and Ankita Kinnerkar. Comparison of relational database with document-oriented database (mongodb) for big data applications. 2015 8th International Conference on Advanced Software Engineering & Its Applications (ASEA). IEEE, 2015.
[11] Kamil Kolonko, Performance comparison of the most popular relational and non-relational database management systems, Master of Science in Software Engineering, Feb 2018.
[12] Krogh, Jesper Wisborg, Krogh, and Gennick. MySQL Connector/Python Revealed. Apress, 2018.
[13] Lee, Hwanggyo, and Sang-Won Lee. "Performance improvement plan for MySQL insert buffer." Proceedings of the Sixth International Conference on Emerging Databases: Technologies, Applications, and Theory. 2016.
[14] Schwartz, Baron, Peter Zaitsev, and Vadim Tkachenko. High performance MySQL: optimization, backups, and replication. " O'Reilly Media, Inc.", 2012.
[15] Sun, Zequn, et al. Knowledge graph alignment network with gated multi-hop neighborhood aggregation. Proceedings of the AAAI Conference on Artificial Intelligence. Vol. 34. No. 01. 2020.
[16] Jayant Bisht, Clustering Coefficient in Graph Theory, www.geeksforgeeks.org, Feb 2018.
[17] Zhang, Peng, et al. Clustering coefficient and community structure of bipartite networks. Physica A: Statistical Mechanics and its Applications 387.27 (2008): 6869-6875.
[18] H. Huang and Z. Dong, Research on architecture and query performance based on distributed graph database Neo4j, 2013 3rd International Conference on Consumer Electronics, Communications and Networks, Xianning, 2013, pp. 533-536, doi: 10.1109/CECNet.2013.6703387.
[19] Shalini Batra, Charu Tyagi. Comparative Analysis of Relational And Graph Databases. International Journal of Soft Computing and Engineering (IJSCE), 2(2), May 2012.
[20] Warchał, Łukasz. Using Neo4j graph database in social network analysis. Studia Informatica 33.2A (2012): 271-279.
[21] Tiago Macedo, Fred Oliveira, Redis Cookbook: Practical Techniques for Fast Data Manipulation, published by O'Relly Media, Inc.,2011.
[22] Josiah L. Carlson. 2013. Redis in Action. Manning Publications Co., USA.
[23] Kabakus, Abdullah Talha, and Resul Kara. A performance evaluation of in-memory databases. Journal of King Saud University-Computer and Information Sciences 29.4 (2017): 520-525.

# 项目分工

傅尔正　　签名：

- idea：数据集选取; motivation; 论文结构
- Redis：阅读学习架构实现及操作方法；数据库构建；各操作实现及优化
- 讨论分析：pre 分析部分；论文分析部分
- 论文撰写:英文摘要，引言，Redis 部分结构及分析，总结；校对
- 美工：ppt；海报；Demo；论文排版

田翔宇　　签名：

- 原数据集的获取并处理为 csv 格式
- MySQL 数据库的构建
- 在 MySQL 数据库中进行的查询操作
- 部分 Redis 和 Neo4j 相关的 Python 代码的调试与实现
- Demo 后端框架的搭建
- 论文 MySQL 分析部分；论文摘要
- 性能分析和 pre 的相关讨论

朱　秦　　签名：

- 使用 python 的 json 库数据预处理，删除了所有 payload 字段，并以 json 格式文件储存预处理结果
- 第一次 pre 的文稿
- 制作了第二次 pre 的 PPT，负责了 pre 文稿的撰写。
- 完成了 ER 图的绘制
- 制作了第三次 pre 的 PTT，pre 文稿的撰写
- 第四次 pre 文稿的撰写，海报内容的撰写与汇总
- 论文所有部分初稿的撰写、所有数据的核对和所有图示的制作、论文的修改。

赵予珩　　签名：

- Neo4j 数据库的构建。
- 在 Neo4j 数据库中设计相关的 Cypher 语句进行相应的操作。
- Noe4j 图算法 algo 和 gds 的研究学习。
- 对于 Neo4j 数据库进行性能优化，包括建立索引，优化查询语句，调用 gds 图算法库等。
- Demo 的设计与制作，包括前端框架的搭建，数据的可视化展示（图表的绘制渲染）。
- Neo4j 底层存储方式的研究和分析。
- 论文中的部分数据分析，在文章第 4 部分中负责针对 Neo4j 的性能，相较 MySQL 和 Redis 进行了分析。
- 论文中第 5 部分实例展示的撰写，第 3.2 节 Neo4j 的撰写，及第 2 部分数据集及获取的修改。