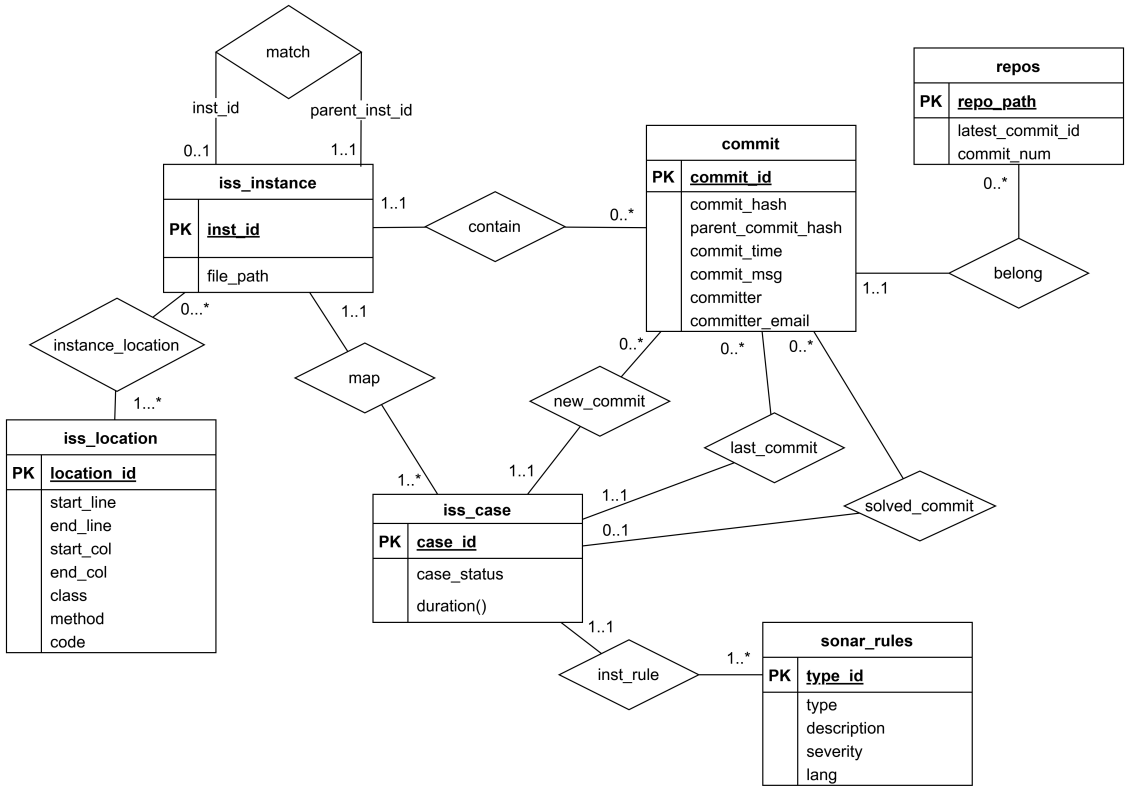
实验报告

王耕宇 17307110209
孙姝然

「 1、数据库ER图」



- 本次代码基本使用 `UUID` 作为主键，相应的 `UUID` 由实例主键增加随机数生成，确保主键的唯一性。
- 这个ER关系模型是根据查询需求和冗余性需求权衡下得出的。外在的查询条件主要是版本、时间和开发人员，因此查询入口在commit实体集中，查询的需求需要根据类型、大类分类，引入状态和解决状态，存续时长等等进行判断，因此主要使用的实体集是 `iss_case` 和 `commit` 。
- 左上角的 `match` 联系集并不对应相应需求，但为了完整的信息存储和需求的可扩展性将其保留。
- 在获取已解决缺陷列表时，主要使用 `last_commit` 联系集的匹配获取详情的缺陷列表。
- 右上角 `belong` 联系集主要用来标识不同版本所属的仓库，来确保各个仓库操作的隔离性。同时 `repos` 实体集中保存了最新提交的comit_id，而不是根据时间获取或者根据 `commit` 的 `parent` 链迭代获取。
- `iss_location` 是 `iss_instance` 的多值属性，这里使用 `iss_location` 联系集和 `iss_location` 实体集进行关联，在存储匹配的过程中可根据 `location` 是否变化，减少 `iss_location` 的入库数量，实现 `location` 和 `instance` 之间的多对多映射。

「 2、数据库表结构」

数据库文件请见db.sql，具体设计框架见ER图，这里对于特殊的约束和其他设计进行说明。

1. 规范化

在数据库规范化层面，我们完善了完整性约束的细节，对于一些用于外键的属性，我们将其设计为 `not null` ，以防止通过值为 `null` 破坏所期望的外键约束。对于 `iss_case` 表的 `case_status` 属性，我们添加约束 `CHECK (case_status in ('NEW', 'UNDONE', 'SOLVED','REOPEN'))` ，将其值规范到我们所需的4个字段之内，防止数据错误。对于一些不能为负值的整型属性，我们将其类型设计为 `unsigned` 。对于 `repos` 表中的 `commit_num` 数据，我们将其设计为某人值0，以方便该表的创建。

2. 去冗余

在数据库去冗余层面，我们大多使用 `UUID` 作为主键来减少表之间关联的数据臃肿问题，这些 `UUID` 主键基本是靠现实主键生成，同时部分使用随机数、当前时间的方法使得主键重复的情况基本不会发生。我们使用mock进行了百万级缺陷实例的导入测试，侧面验证了其唯一性。
使用 `UUID` 的另一个原因，是由于数据库会自动为主键设计索引，如果主键属性过多，相应的索引设计也会变多，导致数据库插入、更新等操作时速度变慢。同时在索引机制下，相较于单值查询，多值查询往往导致更多的IO操作。

3. 高效率

为了查询的方便，我们为一些与查询需求高度相关的属性设计了索引，比如经常用于判断的缺陷状态属性 `case_status` 、经常用于范围比较的缺陷时间属性 `commit_time` 等，我们会在第6部分作出详细介绍和数据分析。

「 3、数据分析需求」

1. 插入、更新需求

在插入、更新等存储需求方面，我们主要关注了数据存储的效率、数据唯一性以及事务三个方面。

效率

在效率方面，由于存储过程中需要获取旧的 `preRawIssue` 信息来与新的 `newRawIssue` 信息进行匹配，如果在每次版本缺陷存储时都要从数据库获取相关信息，则势必会增加较多的IO操作，因此我们将其设计为在多次版本扫描存储情况下，只在第一次操作时取出相关信息生成列表 `List<Matches> matches` ，在之后的连续扫描存储过程中，都会通过该列表在内存中维护相关信息，以加快存储速度。

数据唯一性

在数据唯一性方面，我们主要在多个仓库存储的维护方面做了工作。我们引入仓库表 `repos` ，使得不同仓库的缺陷信息即使在相同的表中，也能做到很好的隔离效果。
同时，我们在仓库表中维护了属性 `latest_commit_id` ，即最近提交的仓库版本，来作为当前仓库的最新版本，以避免多分支等情况对最新 `commit_id` 造成的干扰。

事务

在事务方面，主要是确保数据存储过程中的ACID特性。具体的操作是，我们将每一次版本要插入或更新的信息放入同一事务中执行，如果事务提交成功则继续扫描，若事务提交失败，则程序回滚并中断，必须重新进行扫描。

2. 查询需求

对于查询需求，我们要求必须首先选择要操作的仓库。本次实践中的多表查询方面，我们全部使用 `join` 语法来进行关联，杜绝使用更耗时的笛卡尔积形式。同时对于存续时长，我们在数据库中设计了函数 `duration` 对其进行字符串化处理。

1. 最新版本或指定版本中，静态缺陷数量的分类统计和详细列表： 按类型统计、详情按存续时长排序、按类型统计存续时长的平均值和中位数。

查询函数存在于 `QueryMappingById.java` 文件中。

- 获取最新版本

  函数： `getCommitLatest` ，由于我们实现了 `repos` 表并存储了最新版本信息 `latest_commit_id` ，因此最新版本获取比较方便高效，即使该查询在与存储并发的情况下也能够在更新后立即获取相应情况。
- 引入缺陷、解决缺陷总数获取

函数：`getCountInByCommit_id`、`getCountDoneByCommit_id`，涉及了`iss_case`和`commit`两张表，通过`iss_case`表的`commit_id_new`或者`commit_id_disappear`与`commit`表的`commit_id`进行关联。

- 引入缺陷、解决缺陷按缺陷类型分类统计，存续时长平均数、中位数，存续时长排序

  函数：`getGCountInTypeByCommit_id`、`getCountDoneInTypeByCommit_id`，使用一张`iss_case`表和两张`commit`表，`commit`表的`commit_id`属性分别与`iss_case`表的`commit_id_new`或者`commit_id_disappear`关联，通过类型分类统计。

  对于存续时长平均数，主要通过`timestampdiff`函数，若缺陷已解决，则获取到消除缺陷时间和产生缺陷时间的差值，若缺陷未解决，则另外使用`localtime`函数获取当前时间，计算其与产生时间的差值。这里使用了`case`语句进行判断。

  同时由于`localtime`函数在语句执行开始就得到唯一的值，因此不用担心查询时间过长使得`localtime`取值不同导致结果计算之间的差值过大的问题。

  ```
  duration(avg(TIMESTAMPDIFF(SECOND, c.commit_time, case ic.case_status
  when 'SOLVED' then c1.commit_time else localtime() end))
  ```

  对于中位数，我们先获取到统计数量后，按类型对中位数进行提取，主要是使用了`limit`语句。另外此需求可以设计函数获取，但没有这种方法高效。

  ```
  limit ((intStringTime.getIntValue()+1)/2 - 1), 1
  ```

- 缺陷详细列表，按存续时长排序

  由于一个缺陷可能涉及到多个位置信息，因此我们先对缺陷`id`，类型和路径等信息进行获取，之后再根据`id`信息到关联联系表`instance_location`，进而关联到`iss_location`表进行获取。

  对于解决缺陷列表的获取，主要通过对应`commit_id`与`iss_case`中的`commmit_id_disappear`关联，进而通过该`iss_case`的`commit_id_last`属性关联具体缺陷实例。

- 现存缺陷分类统计

  这个需求主要用在查询最新版本上，用到旧版本不具有意义。主要通过缺陷类型`<> UNSOLVED`进行判断，得到缺陷的缺陷类型和所属`commit_id`，进而找到具体的缺陷。

2. 指定时间段内静态缺陷引入、消除情况的分类统计和详细列表： 按类型统计、详情按存续时长排序、按类型统计存续时长的平均值和中位数。

使用`commit_time`进行比较，允许时间段只有一端约束，允许使用`date`类型进行比较，也运行`date+time`进行比较。引入条件为`commit_id_new`对应时间在允许时间范围内，解决条件为`commit_id_disappear`对应时间在允许时间范围内。其他具体实现方法与需求1类似，不再赘述。

3. 数据分析统计： 指定时间段内引入静态缺陷的数量，解决情况，包括解决率、解决所用的时间，按总量以及分各个缺陷大类和具体类型统计。

- 总量统计、按大类统计、按具体缺陷类型统计

  在这一需求中，三者差别不大，均是通过`if`语句获取相应条件下的查询，每次只需要一次取表即可满足相应的所有需求属性。示例`mysql`代码如下:

```
select count(*) intValue1,
count(if(iss_case.case_status = 'SOLVED',1,null)) intValue2,
concat(round( ( count(if(iss_case.case_status =
'SOLVED',1,null))/count(*))*100, 2),'%') stringValue,
duration(avg(if(case_status='SOLVED',TIMESTAMPDIFF(SECOND,
c1.commit_time, c2.commit_time),null))) time
from iss_case join commit c1 on c1.commit_id = iss_case.commit_id_new
left join commit c2 on c2.commit_id = iss_case.commit_id_disappear
and c1.commit_time >= '2010-12-12'
and c1.commit_time <= '2022-12-12'
and repo = '/repo_path'
```

4. 现存静态缺陷中，已经存续超过指定时长的分类情况统计

通过上述提到的方法获取到存续时长的时间戳，将其与输入格式转换后的时间戳（此方式在java层执行）进行比较即可。
在实际应用过程中，由于我们同时也满足了取出最小存续时长的平均值的需求，因此对每一个缺陷的差值进行了计算并比较。但由于条件指明是未解决静态缺陷，其存续时长由引入时间和 `localtime` 函数获取到的差值决定，因此还可以考虑通过 `localtime` 函数减去最小存续时长来直接获取到指定的最小版本时间，执行一次计算后进行比较即可，不需要每次都真正计算存续时长。

5. 开发人员四种类型分类统计

- 总数统计、分类统计

  对于总数或者分类统计，我们使用了标量子查询的方法来做到一次表获取多次表查询，mysql示例代码如下：

```
select
(select count(if( c1.committer = 'name',1,null))) intValue1,
(select count(if( c2.committer = 'name' and c1.committer <>
'name',1,null))) intValue2,
(select count(if( c1.committer = 'name' and ic.case_status <>
'SOLVED',1,null))) intValue3,
(select count(if( c1.committer = 'name' and c2.committer <>
'name',1,null))) intValue4,
(select avg(if(c1.committer = 'name',timestampdiff(SECOND,c1.commit_time,
case ic.case_status when 'SOLVED' then c2.commit_time else localtime()
end),null))) time1,
(select avg(if( c2.committer = 'name' and c1.committer <>
'name',timestampdiff(SECOND,c1.commit_time, case ic.case_status when
'SOLVED' then c2.commit_time else localtime() end),null))) time2,
(select avg(if(c1.committer = 'name' and ic.case_status <>
'SOLVED',timestampdiff(SECOND,c1.commit_time, case ic.case_status when
'SOLVED' then c2.commit_time else localtime() end),null))) time3,
(select avg(if( c1.committer = 'name' and c2.committer <>
'name',timestampdiff(SECOND,c1.commit_time, case ic.case_status when
'SOLVED' then c2.commit_time else localtime() end),null))) time4,
ic.type_id stringValue
from iss_case ic join commit c1 on ic.commit_id_new = c1.commit_id
left join commit c2 on ic.commit_id_disappear = c2.commit_id
where c1.repo_path = 'repo' group by ic.type_id
```

「4、文件执行」

使用 `jar` 包运行，运行之前您需要在同路径下的 `conf.properties` 中配置您的运行信息。

```
jdbc_url=jdbc:mysql://localhost:3306
jdbc_user=
jdbc_password=
# 您的sonarqube用户名和密码
http_auth_string=admin:1230
```

执行目录下的 `db-jar-with-dependencies.jar` 文件，具体运行方法：

- 如果不带参数会提示并退出

```
PS E:\target> java -jar .\db-jar-with-dependencies.jar
welcome
run     执行查询
save (repo_path) [num] 扫描并存储指定路径仓库，如果指定num，则从距最新版本num位置开始扫描
```

- 执行查询

```
PS E:\target> java -jar .\db-jar-with-dependencies.jar run
welcome
当前存储仓库:
================================================================
E:/repo, 入库版本量: 43
================================================================
请选择要查询的仓库路径#
```

- 执行存储，需要包含仓库路径，第二个整型参数表示从HEAD版本开始追溯，要扫描的版本数，默认扫描 `master` 分支。

```
PS E:\target> java -jar .\db-jar-with-dependencies.jar save E:\repo 3
welcome
cur: 150c0f09b36adf64c851833a37755c32c95e44ad
2:doc: 修改图片, hash: cba0c3b81b46cccccca77aef51bd7ce48d141cbbd
```

「 5、测试数据」

我们准备了相应的测试数据，数据保存在 `resource` 目录下的 `data.sql` 文件中，其在首次运行时会自动入库。
同时，您也可以使用自己电脑上的本地仓库进行扫描，扫描前需要先运行 `sonarqube` ，并确保本地9000端口允许HTTP请求。

「 6、系统性能测试和分析」

数据准备

如果想要重复此测试，您可以使用 `mock` 在另一个仓库啊中生成 `mock` 数据，您可以在 `conf.properties` 中配置相应的数据参数。使用以下命令执行数据存储。

```
PS E:\target> java -jar .\db-jar-with-dependencies.jar mock
```

数据存储完毕后，使用以下命令执行查询，对于本次实验需求，由于数据量过多，大部分操作不会显示具体数据，只会显示相应的统计信息。

```
PS E:\target> java -jar .\db-jar-with-dependencies.jar mocktest
```

在以下测试中，我们模拟的缺陷实例数量为100万，版本数量为2000，仓库数量为4。

index

使用以下命令添加相应索引，索引添加花费时间：6.594s

```
create index commit_time on commit (commit_time);
create index committer on commit (committer);
create index repo_path on commit (repo_path);
create index case_id on iss_instance (case_id);
create index case_status on iss_case (case_status);
```

为了排除缓存池问题对数据产生的影响，我们执行了多次无关、大数据量的查询，同时使用了不同的仓库进行操作，尽可能确保测试结果不受影响。

- 指定版本查询

  使用命令 `latest_defect`

  - 没有引入索引：

    ```
    引入缺陷总数：使用时间: 0ms 引入缺陷数量: 114
    引入缺陷分类统计: 使用时间: 120ms 数据量: 73
    引入缺陷详情: 使用时间: 336ms 数据量: 114
    解决缺陷总数: 使用时间: 26ms 解决缺陷数量: 114
    解决缺陷分类统计: 使用时间: 67ms 数据量: 72
    解决缺陷详情: 使用时间: 328ms 数据量: 114
    现存缺陷类型统计: 使用时间: 297ms
    ```

  - 引入相应索引

    ```
    引入缺陷数量: 使用时间: 0ms 引入缺陷数量: 125
    引入缺陷分类统计: 使用时间: 110ms 数据量: 79
    引入缺陷详情: 使用时间: 235ms 数据量: 125
    解决缺陷数量: 使用时间: 0ms 解决缺陷数量: 125
    解决缺陷分类统计: 使用时间: 79ms 数据量: 80
    解决缺陷详情: 使用时间: 251ms 数据量: 125
    现存缺陷类型统计: 使用时间: 220ms
    ```

  可以看到，整体运行时间都很小，两种情况时间差别不大。

  分析其原因：

  1. commit_id是主键索引情况下，获取指定版本数据较快。
  2. 指定版本数据量较小。
  3. 使用 `join` 语法，同时在 `on` 中直接指定具体版本，表的数据组织量小。

- 指定时间统计

  使用命令 `defect -t 2018-1-1=2022-12-30`

  - 没有引入索引：

```
引入缺陷总数: 使用时间: 0ms 引入缺陷数量: 62649
引入缺陷分类统计: 使用时间: 279793ms 数据量: 124
引入缺陷详情: 使用时间: 114646ms 数据量: 62649
解决缺陷总数: 使用时间: 0ms 解决缺陷数量: 62149
解决缺陷分类统计: 使用时间: 327485ms 数据量: 124
解决缺陷详情: 使用时间: 172060ms 数据量: 62149
```

- 引入相应索引:

```
引入缺陷总数: 使用时间: 8ms 引入缺陷数量: 62931
引入缺陷分类统计: 使用时间: 22168ms 数据量: 124
引入缺陷详情: 使用时间: 91721ms 数据量: 62931
解决缺陷总数: 使用时间: 0ms 解决缺陷数量: 62431
解决缺陷分类统计: 使用时间: 60450ms 数据量: 124
解决缺陷详情: 使用时间: 102993ms 数据量: 62431
```

可以看到，分类统计方面耗时减少了一个数量级，分析主要是由于使用了 `commit_time` 和 `case_status` 索引，不仅使时间范围查找更快，同时也是分类统计更利于执行。

对于缺陷详情，其执行时间改善相比之下不太明显，原因可能是由于错误设计了 `instance_location` 表，增加了表匹配路径长度，且此部分耗时占比较高。

- 指定时间段内数据分析

  使用命令 `analysis 2018-1-1=2022-12-30`

  - 没有引入索引:

```
总数: 使用时间: 18378ms, 数据量62649
按大类分类统计: 使用时间: 4873ms
按类型分类统计: 使用时间: 4835ms
```

  - 引入相应索引:

```
总数: 使用时间: 1913ms 缺陷引入数: 251536
按大类分类统计: 使用时间: 428ms
按类型分类统计: 使用时间: 355ms
```

  可以看到耗时减小了一个数量级，原因同上一部分所述相同。

- 按指定现存缺陷最小存续时长

  使用命令 `duration 1分`

  - 没有引入索引:

```
超过指定存续时长的静态缺陷分类情况统计:
使用时间: 748ms
```

  - 引入相应索引:

```
超过指定存续时长的静态缺陷分类情况统计:
使用时间: 31ms
```

  耗时减少一个数量级。

- 开发人员分类统计

  使用命令 `devs Kubernetes Prow Robot`

  - 没有引入索引:

```
总数:
引入缺陷数量: 29964, 解决他人缺陷数量: 16750, 引入但尚未解决缺陷数量: 276, 引入被他人解决缺陷数量: 16474
使用时间: 4317ms
分类统计: 使用时间: 4225ms 数据量: 496
```

- 引入相应索引:

```
总数:
引入缺陷数量: 27860, 解决他人缺陷数量: 15946, 引入但尚未解决缺陷数量: 121, 引入被他人解决缺陷数量: 16325
使用时间: 301ms
分类统计: 使用时间: 576ms 数据量: 496
```

耗时减少一个数量级。

「7、总结与反思」

1，查询路径长度

对于第6部分提到的查询路径长度，我们认为关联表文件路径长度过长会较大地影响到执行时间，如通过 `inst_id` 寻找相应的位置信息，就需要先通过 `instance_location` 表找到具体的 `location_id` ，再通过 `location_id` 找到相应的位置信息。由于 `iss_instance` 表中没有关联位置信息的缺陷数量占比很少，因此导致 `instance_location` 表中数据量与 `instance` 在一个量级。如果将其作为多值属性存储进 `iss_instance` 表，将其放到一个属性中，以特定分隔符分隔来进行存储。这样可能产生的缺点是，持续存在的缺陷往往 `location` 信息相同，将位置信息作为多值属性存储表中则无法表示多对多关心，会造成大量的数据冗余。

2，只存储有修改的实例

我们考虑过对于相同 `case_id` 的缺陷中，只存储有变动的实例，将 `iss_case` 表中的 `commit_last_id` 存储为最新产生变动的版本id，同时建立版本与实例的映射，这种情况下 `commit` 实体集与 `iss_instance` 实体集之间的联系集 `contain` 变为多对多的关系。这样做的好处是可以大大减少缺陷实例的数据，在仓库较大时很有帮助，坏处是需要在数据库中建立一张缺陷与版本之间的映射表，增加了查询路径长度。
由于时间的关系，且实践前期对需求理解不清楚，导致我们没有实现这一方法。