数据库负载样本生成与特征化方案文档

# 一、负载生成器的开发价值

## 1.1 开发背景以及对于参数调优工作的意义

SQL是结构化查询语言，广泛用于关系型数据库管理系统中，用于执行各种数据库操作，如查询、更新、插入等。在现实生产应用中，许多数据库优化问题，如：**数据库参数调优**、慢SQL诊断、数据库性能测试、基数估计、查询优化等诸多任务，都需要大量的SQL查询。

然而，由于数据隐私问题和现实场景中的约束（比如没有任何使用历史的还未上线的新数据库），我们很难获得大量有效的真实SQL查询。同时，有时我们需要SQL语句具有一定的非语义的、在线的、查询特征，如满足特定的基数条件，进一步缩小了有效SQL的数量。**然而，目前的负载生成方法存在很多问题，比如，传统的数据库评测负载生成工具不够灵活、数据库Schema固定，且过于依赖专家模板，迁移性较弱；生成式算法算法生成的SQL质量不高（可能生成空语句）、预训练数据需求量大、生成负载大多同质化，等等。**

因此，负载生成是数据库优化中非常重要的任务。

**负载映射（Workload Mapping）**是数据库参数调优领域的一个概念，它是指在进行数据库性能调优时，考虑到实际运行环境中的负载特征（例如查询类型、访问频率、数据量等），将这些特征与数据库系统的参数设置关联起来，以便为不同的负载配置适当的参数。负载映射方法的目标是为了在保证数据库性能的同时，最大限度地提高资源利用率和系统稳定性。

在负载映射方法中，调优专家或自动化工具首先需要收集和分析数据库系统的负载特征。

基于这些负载特征，调优专家或自动化工具可以确定哪些参数对于当前的负载具有重要意义，并通过调整这些参数来提高系统性能。

**负载映射效果优秀的前提和基础是负载的知识库足够丰富，这就需要大量多样化的负载来做离线训练。而很多时候，由于数据隐私问题和现实场景的约束，我们并没有这样大量的预训练数据，而且数据之间可能存在同质化的问题。如果能提供多场景的、多样化的负载，作为知识库的基础，那么负载映射工作的效果才能得到有效保障。**

简而言之，调优模型需要通过分析实际运行环境中的各种负载特征，帮助调优专家或自动化工具为不同的负载配置适当的参数，以提高数据库系统的性能和稳定性，因此，能否拥有多样化的、有区分度的、语法正确的、符合实际要求且有意义的查询负载，以及能否根据已有负载肖像进行特定负载的生成，对下游的负载映射方法具有重要作用。

## 1.2 当前主流负载生成算法分析

当前的SQL查询语句生成方法可以根据是由查询驱动还是数据驱动分为两大类：基于历史负载的模型，和不依赖于历史负载的模型。

### 1.2.1 查询驱动：基于历史负载的模型

对于基于历史负载的模型来说，区别于NLP领域中文本生成模型（例如LSTM模型）的端到端生成的思想，**查询驱动的负载生成工具不需要按顺序生成SQL语句的每个单词，而只需要将已有的历史负载进行切割分段，在语法合理的情况下重新进行排列组合，进而“生成”新的负载。文献[1]中提出的TreeGAN模型就是此种方法的代表，通过将SQL解析成一颗语法树，然后随机合法地交换子树，以便从已有的负载中生成新的负载，这种思想和数据增强算法有一定的相似之处。**

又或者，连“拆分SQL，再对子句进行合法排列组合生成新负载”这一步都可以省去，**直接在历史负载中选取其中若干条具有代表性的SQL语句，打乱组合，形成新的负载。**例如，使用基于GAN的模型等，使用大量负载数据训练，生成与训练数据相似的查询语句。GAN的基本思想是让生成器和判别器进行对抗性训练。生成器利用随机噪声向量作为输入，试图生成逼真的数据。判别器则接收真实数据和生成器生成的数据作为输入，试图区分哪些数据是真实的，哪些是生成的。生成器通过不断尝试欺骗判别器来提高生成数据的逼真程度，而判别器则通过不断学习区分真伪来提高辨别能力。**文献[2]提出的Warper模型就是这种方法的代表，Warper模型的本意是加快数据库基数估计模型的训练，在数据集足够大的情况下，更新模型时，Warper模型“pick”出最具有代表性的负载，然后“update”当前基数估计训练数据集，来更新基数估计模型。**

此种方法的**不足之处**在于，如此庞大的预训练SQL数据集在现实场景中很难获取，而且数据集必须足够充足、包括足够多样化的负载、不能过于同质化；另外就是也没法满足用户对负载执行特征的需要，比如cardinality（基数大小），即SQL语句返回结果集的大小，再比如cost，即执行该query带来的时间开销。

### 1.2.2 数据驱动：不依赖于历史负载的模型

不依赖于历史负载的模型通常会从**数据及其分布**的角度进行可控的负载生成，此类方法不需要使用大量历史负载进行预训练，但在生成时为了满足负载的质量会根据数据库的数据分布调整比较谓词连接的token。比如，只有解析出某一表的某一列的数据类型和值域分布，才能使得生成的负载更符合实际场景。

对于此类不依赖于历史负载的模型，可以根据是否使用生成模板细分为两类：一是**随机生成方法**，通过随机游走语法解析树的方式进行查询语句生成。**例如，Andreas[3]等提出的SQL Smith是一个开源的SQL查询语句随机生成器，通过随机游走语法解析树生成随机查询语句的抽象语法树，然后通过格式化工具将抽象语法树实例化为随机的SQL查询语句。**

此种方法的**不足之处**在于，像SQL Smith一样采用随机生成方法的模型生成的SQL语句的查询结果可能为空，没有做语义相关的检查，会影响负载整体的质量。

另一类是**基于模板的方法**，通过给定一系列模板作为SQL语句的基础骨架进行查询语句生成，只需要更改模板中的谓词值即可完成生成工作。基于模板的方法广泛用于各种数据库性能测试的基准测试中，**例如，数据库基准测试工具Sysbench[4]，它可以模拟各种负载和工作负荷，包括随机数据生成、查询和更新操作等，并能够以多线程和分布式的方式运行测试，Sysbench中查询负载的生成使用了point\_selects、simple\_ranges等9个SQL语句模版，具体详见附录二；再比如，被广泛使用的TPC-H[5]和TPC-C[6]基准测试负载生成器使用的都是基于模板的方法。TPC-H基准测试负载生成器依赖22条查询语句模板进行查询负载生成，而TPC-C基准测试的查询负载生成器则依赖5个查询事务模板（查询事务模版是由固定顺序的两个或多个查询语句模版组合而成，表示该事务的两个或多个操作组合）进行查询负载生成。**

此种方法的**不足之处**在于，模版化的方法一方面依赖于专家制定的高质量模版，另一方面，当生成模版确定下来，由于只能更改模版中的谓词值，生成的SQL查询语句形式就较为单一，与此同时该负载生成器对应的数据库Schema也被固定下来。

此外，无论是随机生成方法或是基于模版的方法，现有的大多数方法都无法根据用户限定的约束条件进行生成，在实际的应用场景中，当用户需要测试一个数据库的性能时，他们可能需要生成符合一些特定约束条件的查询负载，或者多样化的负载以便更好、更全面的地模拟实际情况。在负载调优任务中，也需要有区分度的、特征不一的多样化负载，对于这些要求，现有的这些方法难以满足。

### 1.3 改进思路分析及相应的解决方案

针对上述方法的不足之处，我们设计负载生成工具时考虑了以下目标：

**◇目标1：**不依赖于大量历史负载做预训练。由于数据隐私问题和现实场景中的约束（比如没有任何使用历史的还未上线的新数据库），我们很难获得大量有效的真实历史SQL查询。

**解决方案：**必须选择更困难但更具备实用价值的生成式方法，而非对已有负载进行shuffle&combine之类操作的数据增强方法。具体来说，我们可以通过构造一个动态化模板（本质上相当于一个有限状态机），对目前已生成SQL的部分处于哪个阶段进行判定，然后根据配置特征计算概率进行状态转移，**状态转移的过程就是往现有的、已生成的、但可能尚不完整的SQL部分里添加一个新的单词，对于单词的解释详见2.1名词解释部分**。

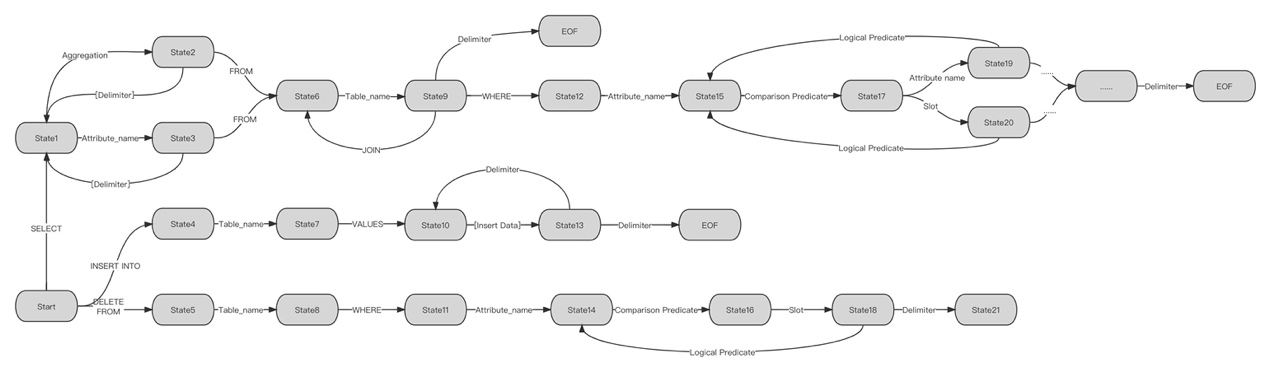


图2.1 生成状态机转移图示

**具体来说，生成过程中，一条SQL语句由多个单词组成，每次生成一个单词的过程也即一次状态转移，每次生成一个单词，必然从一个生成状态转移到另一个状态。有限状态机方法的优势在于，只要自动机设计得当，合理安排转移分支与循环，就可以在满足语法要求的情况下生成尽可能多样的SQL语句。**

**此种方法的优势在于，不同于基于历史负载的查询驱动的方法，不依赖于大量的历史数据，同时保证了生成的可控性。**

**◇目标2：**支持动态、灵活地生成。已有的模板化生成方法大多基于固定的数据库schema，不适用于通用的SQL生成场景。新设计的方法应该能适应各种常规的schema，能根据各种场景生成对应的负载。

**解决方案：**设计一套可以根据Schema改变而改变的“动态模板”，也即图2.1中的自动机流程，先通过配置解析模块将Schema信息提取出来，再通过采样算法利用这些解析出的信息生成SQL，来达到动态、灵活生成的效果。

**此种方法的优势在于，可以生成适用更多场景的负载，不像TPC-H、TPC-C等传统负载生成工具拘泥于固定的数据库Schema。**

**◇目标3：**实现更加可控的、多样化的负载生成。负载映射效果优秀的前提和基础是负载的知识库足够丰富，这就需要大量多样化的负载来做离线训练。为此，需要确保生成的负载不会过于同质化。

**解决方法：**从语义、数据访问模式、执行三个层次刻画负载特征，依据这些特征进行负载生成。对负载特征进行刻画后，负载之间的区别更加明确，更容易生成多样化的负载。

**加入特征刻画后，生成模型的优势在于，可以更好的区分负载，比如，在web应用、数据仓库、OLTP和OLAP等数据库不同应用场景下的负载样本可以通过这些指标加以区分。**

**1.4 项目贡献**

项目具有以下亮点与贡献：

1、设计了一个灵活的、可根据配置参数进行动态负载生成的模型，区别于传统的Benchmark型负载生成工具，不局限于特定的数据库模式，而是可以适用于任何schema；

2、项目包括并兼容了TPC-H、SEATS、SIBench、TATP、TPC-C、Small Bank六个传统的静态负载生成工具，可以根据场景需要调用这些目前主流的负载生成工具；

3、为了辅助用户对生成工具进行配置，编写了辅助用户配置的负载分析工具，可以根据历史负载分析语义特征和数据访问模式特征，给用户提供一定的配置帮助；

4、负载分类更加明确，提供了明确的负载特征指标来区分负载。这样，在严格满足SQL语法的情况下，生成的负载具有多样性、有区分度，适合应用于各种不同的领域和场景。

5、负载生成的速度快、效率高，支持在线解析生成。

# 二、负载生成器框架及技术路线

## 2.1 涉及的名词解释

**◇（SQL包含的）单词或token：**在NLP领域中，分词操作，也即Tokenization，指的是是将句子、段落、文章这类型的长文本，分解为以单词为单位的数据结构，而这里的单词，也就是token。比如，在句子“我很开心”中，利用中文分词得到的列表是{“我”,“很”,“开心”}，列表中的每一个元素代表一个token。由于生成的SQL也相当于一段结构化查询语言的文本，因此在本项目说明书中，**沿用了这种表述习惯**。例如，在查询“SELECT id FROM tb WHERE age=3”中，实际生成的列表为{“SELECT”，“id”，“FROM”，“tb”，“WHERE”，“age”，“=”，“3”}，这一列表的每个元素都为一个单词，也即一个token。

**◇组成SQL的单词种类：**如上，SQL可以逐单词地拆分为一个列表，列表中的token可以分为以下种类：

**关键字（Keyword）：**SQL语言中的关键字是指被SQL语言用于标识操作对象和操作方式等固定的单词或符号，是具有特定含义的保留字。列举一些常见的关键字如下：

·SELECT：用于选取数据表中的数据列；

·FROM：指定要从中选取数据的数据表；

·WHERE：用于筛选符合指定条件的数据；

·INSERT INTO：用于插入新的数据行到数据表中；

·UPDATE：用于更新数据表中的数据行；

·DELETE FROM：用于删除数据表中的数据行；

·ORDER BY：用于对查询结果按指定的列进行升序或降序排序；

·GROUP BY：用于将查询结果按指定的列进行分组；

·JOIN：用于将两个或多个数据表中的数据行组合起来。

**标识符（Identifier）：**包括表名（Table name）、列名（Column name）、视图名（View name）等，标识符需要符合一定的命名规则，例如不能包含空格、不能以数字开头等等。

**字面量（Literals）：**指SQL语言中表示具体值的常量，例如字符串、数字、日期等。字面量通常用于SELECT语句中的列值或WHERE语句中的比较运算符右侧。

**运算符（Operators）：**用于执行操作的符号，包括算术运算符（+、-、\*、/等）、比较运算符（=、<、>、<=、>=、<>等）、逻辑运算符（AND、OR、NOT等）等。

**分隔符（Delimiters）：**用于分隔SQL语句中各个部分的符号，例如分号（;）用于标识SQL语句的结束，逗号（,）用于分隔SELECT语句中的列名等。

**◇谓词（Predicate）：**在SQL语言中，谓词是指用于筛选数据的条件表达式，通常用在SELECT、WHERE、JOIN等语句中作为筛选条件。SQL谓词主要包括比较谓词和逻辑谓词。比较谓词（Comparison Predicate）用于比较两个值是否相等或不相等，包括等于（=）、不等于（<>或!=）、大于（>）、大于等于（>=）、小于（<）和小于等于（<=）等。逻辑谓词（Logical Predicate）：用于连接和组合多个谓词的逻辑运算符，具体包括AND、OR和NOT。

**◇（SQL语句的）骨架和槽：**在组成一条SQL的单词列表中，所有非字面量和标识符的单词合并称为该SQL语句的骨架，在生成完成后保持相对固定，我们称为SQL语句的骨架（SQL skeleton）；而字面量和标识符可以根据列的取值范围进行调整，进而改变SQL的执行特征，相对比较灵活，我们称这一部分为SQL语句的槽（SQL slot）。

## 2.2 整体架构及工程文件目录结构

项目整体架构图如下：

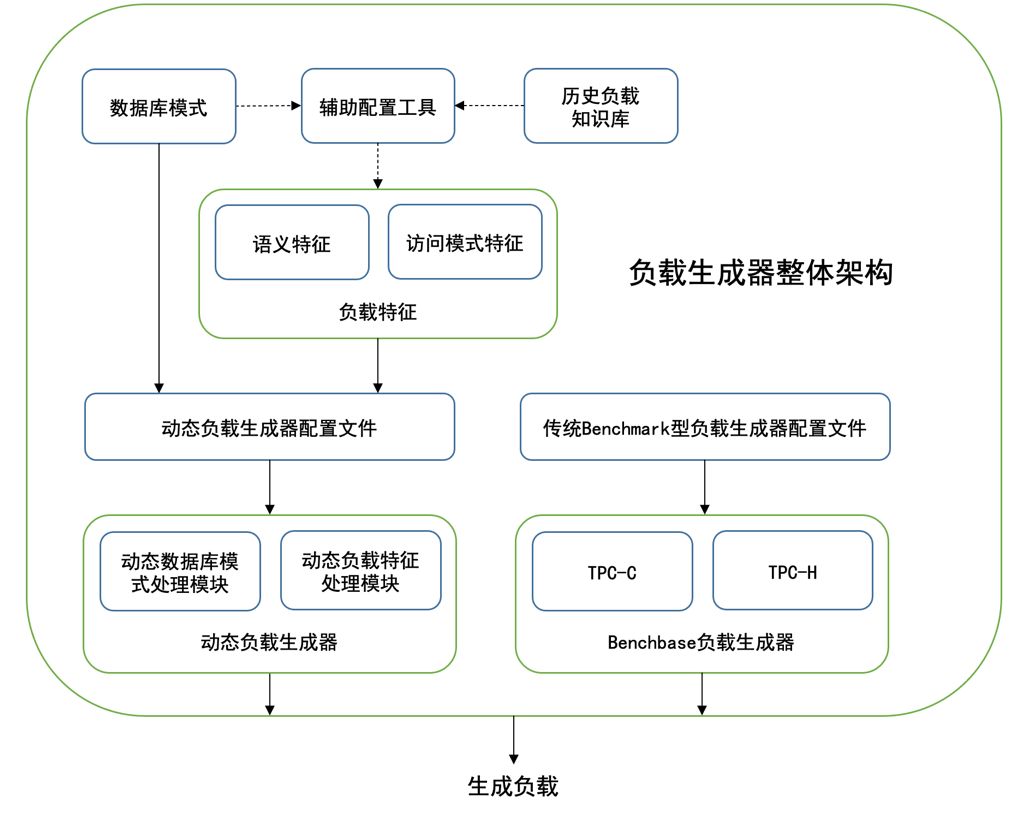


图2.1 负载生成器架构及子模块之间的输入输出关系

**图2.1中，首先根据输入的生成模式选择是使用传统的Benchmark型负载生成工具还是项目自主设计的动态负载生成器；如上图所示，生成的负载可以来自项目自主编写的动态负载生成器，也可以来自项目集成的Benchbase负载生成器；**

**如果选用传统的Benchmark型负载生成工具，则直接使用bin子目录下相应的bash文件即可（前提是PostgreSQL数据库已经在服务器上运行）；**

**如果使用项目自主设计的动态负载生成器，则需要用户正确配置configure文件，也即上图中的“动态负载生成器配置文件”；**

**配置文件本身由两部分组成，一是数据库模式配置，二是负载特征配置。数据库模式配置必须由用户手动完成，负载特征配置较为复杂，可以用户直接手动编写配置，如果有困难，也可以使用用户辅助配置工具分析历史负载，帮助用户对负载进行一定程度的解析，帮助用户编写配置文件。**

**需要注意的是，如果要使用辅助配置工具，必须拥有历史负载数据和对应的数据库模式配置，否则无法分析。这是由语法解析过程决定的。没有历史负载数据自然也无从分析，而必须要有对应的数据库模式配置的原因是，涉及多表连接语句时，所查询的列归属于哪个表可能会产生歧义。因此，辅助配置工具必须拥有完整的历史负载数据和对应的数据库模式配置作为输入。**

**动态负载生成器主要由两部分组成，一个用于处理动态数据库模式的模块，和一个处理动态负载特征的模块。前者保证负载生成工具可以生成不拘泥于数据库Schema的负载，后者保证生成的负载符合给定特征，进一步确保生成负载的多样化。**

工程文件目录和对应的中文解释如下：

SuperWG···························································································根目录

-bin··························································································bash文件

run\_tpch.sh

run\_sibench.sh

run\_seats.sh

run\_smallbank.sh

run\_dwg.sh

run\_parser.sh

run\_tatp.sh

-doc·························································································说明文档

user\_guide.pdf

-res····················································································存放生成结果

dwg.wg

-Benchbase·······················································Benchbase负载工具相关集成

-BenchbaseTools

-TPC-H

-SIBench

-Seats

-SmallBank

-TATP

-TPC-C

-DWG·································································负载生成器和解析器主体

-configure·························································分析负载及其配置文件

sample\_workload\_schema.json

-src··············································································负载生成工具

-WP(WorkloadParser)·······················································辅助配置工具

## 2.3负载生成工具

负载生成工具包含两个部分，一个是集成了现有的Benchbase的工具包，可以随时生成这些传统的benchmark型的负载，具体通过执行bin目录下对应的bash文件实现，但需要提前连接好数据库服务器；第二个部分是项目主要的创新和主体部分，一个多样化的动态负载生成器。

### 2.3.1 BenchbaseTools：传统的benchmark型负载生成工具集成

在项目的这一部分，集成了Benchbase中的六种常用的benchmark型负载：TPC-H、TPC-C、SEATS、Small Bank、SIBench、TATP。

这里以TPC-H和TPC-C为例。TPC-H（Transaction Processing Performance Council Benchmark H）是一个决策支持系统基准测试，旨在衡量各种决策支持系统在执行一组复杂查询时的性能表现。该基准测试由一组广泛适用于各行各业的业务相关自定义查询组成，以便测试系统处理大量数据、执行复杂查询、回答关键业务问题的能力。

TPC-H通过基于模版的方法生成SQL查询语句负载，在源码中主要包含两部分代码逻辑dbgen和qgen，dbgen是一个数据生成器，用来生成用户指定数据规模的元数据；qgen则是负载生成器，根据22条SQL查询语句模版生成可执行查询语句。这些查询语句模版都是查询操作，包含多表关联查询、from子查询、视图等操作，每条查询语句模版中包含若干可替换项，其中可替换项用参数占位符“？”表示，以第一条查询语句模版Q1为例：

1. SELECT
2. l\_returnflag,
3. l\_linestatus,
4. SUM(l\_quantity) AS sum\_qty,
5. SUM(l\_extendedprice) AS sum\_base\_price,
6. SUM(l\_extendedprice \* (1 - l\_discount)) AS sum\_disc\_price,
7. SUM(l\_extendedprice \* (1 - l\_discount) \* (1 + l\_tax)) AS sum\_charge,
8. AVG(l\_quantity) AS avg\_qty,
9. AVG(l\_extendedprice) AS avg\_price,
10. AVG(l\_discount) AS avg\_disc,
11. COUNT(\*) AS count\_order
12. FROM
13. lineitem
14. WHERE
15. l\_shipdate <= DATE '1998-12-01' - INTERVAL ? DAY
16. GROUP BY
17. l\_returnflag,
18. l\_linestatus
19. ORDER BY
20. l\_returnflag,
21. l\_linestatus

代码2-1 查询语句模版Q1

这是一条针对关系数据库的 SQL 查询语句，用于查询 lineitem 表中在指定日期之前发货的产品订单的汇总统计信息。在这条查询语句中，使用了 SUM、AVG 和 COUNT 聚合函数来计算 lineitem 表中符合指定条件的订单的总量、总基本价格、总折扣价格、总价值、平均数量、平均价格、平均折扣和订单数等汇总信息。同时，使用了 GROUP BY 子句对 lineitem 表中的记录按 l\_returnflag 和 l\_linestatus 两个字段进行分组，以便对每个组进行单独统计。最后，使用 ORDER BY 对结果按 l\_returnflag 和 l\_linestatus 进行排序。在 WHERE 子句中，指定了 l\_shipdate 的条件为在指定日期之前 1998-12-01 - INTERVAL ? DAY 发货的订单。其中，? 是一个参数占位符，可以在实际查询时使用具体的数值来替换，以便动态地指定查询的日期范围。每条语句的可替换项都有相对应的默认替换选择，在默认模式下始终生成固定的SQL查询语句，非默认替换选择的情形下，则随机选用适当的替换项。

TPC-H定义了一个固定的schema用于测试，该schema包括八个表，如下表所示：

表格 2‑1 TPC-H基准测试schema中的表

|  |  |
| --- | --- |
| Region（地区） | 地区的信息，包含R\_REGIONKEY（地区标识符）、R\_NAME（地区名字）、R\_COMMENT（描述信息）等字段 |
| Nation（国家） | 国家的信息，包含N\_NATIONKEY（国家标识符）、N\_NAME（国家名字）、N\_REGIONKEY（属于哪个地区）等字段 |
| Supplier（供应商） | 供应商的信息，包含S\_SUPPKEY（供应商标识符）、S\_NAME（供应商名字）、S\_ADDRESS（供应商地址）等字段 |
| Part（零件） | 零件的信息，包含P\_PARTKEY（零件标识符）、P\_NAME（零件名字）、P\_MFGR（生产厂商）等字段 |
| PartSupp（零件-供应商关系） | 零件和供应商之间的关系信息，包含PS\_PARTKEY（零件标识符）、PS\_SUPPKEY（供应商标识符）、PS\_SUPPLYCOST（零件的单价）等字段 |
| Customer（顾客） | 顾客的信息，包含C\_CUSTKEY（顾客标识符）、C\_NAME（顾客名字）、C\_ADDRESS（顾客地址）等字段 |
| Orders（订单） | 订单的信息，包含O\_ORDERKEY（订单标识符）、O\_ORDERDATE（下单时间）、O\_ORDERPRIORITY（订单优先级）等字段 |
| Lineitem（订单详情） | 订单的详细信息，包含L\_ORDERKEY（订单标识符）、L\_PARTKEY（零件标识符）、L\_SUPPKEY（供应商标识符）、L\_QUANTITY（订单数量）等字段 |

在TPC-H测试中，使用指定的schema生成数据集。数据集的大小可以通过参数控制，例如10GB、100GB、1TB等，随后将生成的数据集导入到数据库系统中进行测试。除了数据生成器之外，TPC-H中包括的22个查询（Q1-Q22）语句模版，都是根据这个指定的schema进行设计。

TPC-C（Transaction Processing Performance Council Benchmark C）是一个针对OLTP（Online Transaction Processing）系统的基准测试，用于测试事务处理系统的性能。TPC-C包括五种不同类型和复杂度的并发事务的混合，通过在线执行或排队等待延迟执行，以每分钟事务数（tpmC）为性能的度量单位，可以用来测试数据库服务稳定性、性能以及系统性能等一系列问题。

TPC-C也是通过基于模版的方式进行SQl查询语句生成，与TPC-H不同的是TPC-C模拟了一个商品销售模型，其中包含五类事务，分别是：NewOrder（新订单的形成）、Payment（订单付款）、Orderstatus（最近订单查询）、Delivery（配送）、StockLevel（库存缺货状态）。要了解TPC-C的SQL查询语句生成策略，首先要熟悉TPC-C指定的schema，下表展示了关键的七张表：

表格 2‑2 TPC-C基准测试schema中的表

|  |  |
| --- | --- |
| Warehouse（仓库） | 仓库信息，包含了W\_ID（仓库编号）、W\_NAME（仓库名）、W\_STREET\_1（仓库地址）等字段 |
| District（区域） | 区域信息，包含了D\_ID（区域编号）、D\_NAME（区域名）、D\_STREET\_1（区域地址）、D\_W\_ID（该区域所属仓库编号）等字段 |
| Customer（顾客） | 顾客信息，包含了C\_ID（顾客编号）、C\_FIRST、C\_MIDDLE、C\_LAST（顾客名字）、C\_STREET\_1（顾客地址）、C\_PHONE（顾客电话号码）、C\_D\_ID（顾客所在区域编号）、C\_W\_ID（顾客所属仓库编号）等字段 |
| Order（订单） | 订单信息，包含了O\_ID（订单编号）、O\_ENTRY\_D（订单生成时间）、O\_CARRIER\_ID、O\_OL\_CNT、O\_ALL\_LOCAL（订单状态）等字段 |
| Order Line（订单详情） | 订单详情信息，包含了OL\_O\_ID（订单编号）、OL\_I\_ID（商品编号）、OL\_QUANTITY（订单数量）、OL\_AMOUNT（单价）等字段 |
| Item（商品） | 商品信息，包含了I\_ID（商品编号）、I\_NAME（商品名）、I\_PRICE（商品价格）、I\_DATA（商品详细信息）等字段 |
| Stock（库存） | 库存信息，包含了S\_I\_ID（商品编号）、S\_W\_ID（仓库编号）、S\_QUANTITY（库存数量）、S\_DATA（库存状态）等字段 |

在测试开始之前，TPC-C基准测试规定了数据库的初始状态，也就是数据库中数据的生成规则，可调整的是Warehouse（仓库）的数量W，其余都以规则的形式写定，例如Item（商品）表中固定包含10万种商品、District（区域）表中有W\*10条数据，即每个仓库为10个区域提供服务、Customer（顾客）表中有W\*10\*3000条数据，即每个地区有3000个顾客等等。

上文TPC-H使用了22条独立的SQl查询语句生成模版，与之不同的是，TPC-C依据5类事务设计了5了类SQL语句事务生成模版，每一个事务模版中包含了完成该项事务的一系列固定操作。以Delivery事务为例，这是一个模拟订单交付流程的事务，它包含GetOrderId、DeleteNewOrder、GetCustId、UpdateCarrierId、UpdateDeliveryDateSql、SumOrderAmount、UpdateCustBalDelivCnt七个操作，分别对应七个操作：选择新订单、删除新订单、获取客户信息、更新订单信息、更新订单项信息、计算订单总额、更新客户余额。例如选择新订单操作，事务会选择一个具有给定仓库（warehouse）和区域（district）的最小的新订单，也就是还没有被分配给任何运输服务（carrier）的最早订单；再例如更新订单信息操作，事务会更新之前选择的订单的信息，将其标记为已经交付，并记录交付服务的标识符。选择新订单与更新订单操作的SQL语句生成模版定义如下：

1. public SQLStmt delivGetOrderIdSQL = new SQLStmt(
2. "SELECT NO\_O\_ID FROM " + TPCCConstants.TABLENAME\_NEWORDER +
3. " WHERE NO\_D\_ID = ? " +
4. "   AND NO\_W\_ID = ? " +
5. " ORDER BY NO\_O\_ID ASC " +
6. " LIMIT 1");
7. public SQLStmt delivUpdateCarrierIdSQL = new SQLStmt(

代码 2‑1 delivGetOrderIdSQL查询语句模版

1. "UPDATE " + TPCCConstants.TABLENAME\_OPENORDER +
2. "   SET O\_CARRIER\_ID = ? " +
3. " WHERE O\_ID = ? " +
4. "   AND O\_D\_ID = ?" +
5. "   AND O\_W\_ID = ?");

代码 2‑2 delivUpdateCarrierIdSQL查询语句模版

与一般的模版化生成方式类似，这些模板是根据特定的查询需求和语法规则设计的，每个查询语句模版代表一个特定的操作，通常包含多个占位符，根据特定的条件进行替换以生成具体的查询语句，上述的一系列操作对应的SQl语句生成模版组合起来就是Delivery事务对应的SQL语句事务生成模版。

参数占位符对应数值的确定方式与TPC-H类似，只是没有一组默认的替换项，使用随机数生成器，具体生成方法会因参数类型而异。例如，对于整型参数，可以使用随机数生成器生成一定范围内的整数。对于字符串类型参数，则需要考虑数据长度、字符集等因素。在生成数值时，还需要考虑参数的取值范围和分布情况，以保证生成的数值符合实际场景中的数据分布。

此外，在生成事务时，不同的参数之间可能存在关联关系。例如，生成NewOrder事务时，需要先生成Warehouse ID、District ID和Customer ID，再使用这些值生成Order ID和Order Line。因此，在生成数值时，需要考虑参数之间的关联关系，并按照正确的顺序生成数值，以确保生成的事务符合实际场景。

为了能够高效的整合模版化的方式生成查询负载并快速获取测试结果，实际实现中，本项目在Benchbase数据库测试工具的基础上进行了修改。Benchbase（原名OLTP-benchmark）是一个开源的数据库测试框架，主要用于测试和评估关系型数据库管理系统（RDBMS）的性能。Benchbase本质上是一个多线程负载生成器，可以模拟多个并发用户访问数据库，测试数据库系统在高负载下的性能表现。它通过Java Database Connectivity（JDBC）API连接到数据库系统，可以在多个并发用户之间执行一系列预定义的基准测试，例如TPC-C、TPC-H等。它采用了多线程技术和批处理方式，可以高效地生成负载并快速获取测试结果。

值得注意的是，它的整个测试过程是端到端的，用户无法高效获取到数据库实际执行的查询语句负载，也就无法得到SQL语句模版化生成的结果。因此，本项目对Benchbase的源代码进行了修改，增加了一个功能，即在测试过程中将所有查询语句以文本格式输出，包括TPC-H、TPC-C、Sibench、SEATS、Small Bank、TATP六种基准测试方法，详见附录。

以下本项目将以TPC-H基准测试查询负载生成器中的修改为例详细说明。

Benchbase中的TPC-H基准测试查询负载生成器是基于模版的查询语句生成方法，预先定义了22条SQL模版。代码2-1即benchbase中预先定义的SQL查询语句模版。

### JDBC

在benchbase的负载生成与测试的代码实现中使用了Java Database Connectivity (JDBC)，一个用于在Java编程语言中连接和操作关系型数据库的API。JDBC允许Java程序员使用标准的SQL语句来访问和操作数据库。

JDBC API包含一组Java类和接口，用于连接和操作各种数据库。Java应用程序通过JDBC API与JDBC驱动程序通信，该驱动程序负责将JDBC API调用转换为特定数据库管理系统的原生数据库操作。JDBC驱动程序是由数据库供应商提供的，并且每个驱动程序都是特定于某个数据库管理系统的。因此，benchbase的基准测试方法适用于任何提供JDBC的关系型数据库。

JDBC提供的核心API包括：DriverManag，负责管理JDBC驱动程序的加载和注册；Connection，代表与数据库的连接，并提供对数据库的访问；Statement，允许执行SQL查询和更新操作；PreparedStatement，是Statement的子接口，允许参数化查询；ResultSet，代表查询的结果集。

具体来说，在Benchbase的源代码中对于每个支持的benchmark，都调用了PreparedsStatement类来参数化查询语句模版中的谓词值，并与数据库进行交互。我对源代码的修改根据SQL语句类型可以分为两类，分别是Select语句和Insert/Delete/Update语句，这两种SQL语句将以不同的方式与数据库进行交互。

### Select语句的修改

代码2-2是getOrderId查询语句的代码实现，在该TPC-H基准测试查询负载测试中，所有的select语句生成都以该代码的形式组织。首先通过代码2-2中定义的查询语句模版实例化PreparedStatement类，再通过set函数记录每一个占位符的参数值，随后调用executeQuery方法即可在数据库中执行该条查询语句，并通过ResultSet返回结果集。

1. **private** Integer getOrderId(Connection conn, **int** w\_id, **int** d\_id) **throws** SQLException {
2. **try** (PreparedStatement delivGetOrderId = **this**.getPreparedStatement(conn, delivGetOrderIdSQL)) {
3. delivGetOrderId.setInt(1, d\_id);
4. delivGetOrderId.setInt(2, w\_id);
5. **try** (ResultSet rs = delivGetOrderId.executeQuery()) {
7. **if** (!rs.next()) {
8. // This district has no new orders.  This can happen but should be rare
9. LOG.warn(String.format("District has no new orders [W\_ID=%d, D\_ID=%d]", w\_id, d\_id));
10. **return** **null·**;
11. }
12. **return** rs.getInt("NO\_O\_ID");
13. }
14. }
15. }

代码 2‑4 getOrderId代码实现

通过过ResultSet中实现的getStatment方法可以直接得到执行的查询语句，以字符串的形式输出该条查询语句即可得到查询负载中的所有Select语句。

### Insert/Delete/Update语句的修改

代码2-3是updateCarrierId语句的代码实现，在该TPC-H基准测试查询负载测试中，所有的Insert/Delete/Update语句生成都以该代码的形式组织。与Select语句一致，首先通过预先定义的SQL语句模版实例化PreparedStatement类，并随后通过set函数记录所有占位符的参数值。但executeQuery方法只适用于Select语句的执行，Insert/Delete/Update语句的执行必须调用executeUpdate方法，该函数的返回值为int类型，表示受该条语句影响的行数，因此我们无法拿到结果集ResultSet类的返回值。

1. private void updateCarrierId(Connection conn, int w\_id, int o\_carrier\_id, int d\_id, int no\_o\_id) throws SQLException {
2. try (PreparedStatement delivUpdateCarrierId = this.getPreparedStatement(conn, delivUpdateCarrierIdSQL)) {
3. delivUpdateCarrierId.setInt(1, o\_carrier\_id);
4. delivUpdateCarrierId.setInt(2, no\_o\_id);
5. delivUpdateCarrierId.setInt(3, d\_id);
6. delivUpdateCarrierId.setInt(4, w\_id);
7. int result = delivUpdateCarrierId.executeUpdate();
8. if (result != 1) {
9. String msg = String.format("Failed to update ORDER record [W\_ID=%d, D\_ID=%d, O\_ID=%d]", w\_id, d\_id, no\_o\_id);
10. throw new RuntimeException(msg);
11. }
12. }
13. }

代码 2‑5 updateCarrierId代码实现

为了输出生成的查询负载中所有Insert/Delete/Update语句，我们需要将预先定义的SQL语句模版中所有的占位符替换为指定的参数值，将替换后的结果以字符串形式输出即可。

### Config文件进行参数控制

此外，我们可以通过配置文件设定查询语句生成相关的参数，以TPC-C benchmark为例，可以设定TPC-C的比例因子，即仓库数，可以根据硬件和测试需求进行调整，通常用于控制测试数据的规模和测试的压力；指定测试的负载状况，包括终端数、负载持续时间、提交事务频率、各事务所占权重；指定TPC-C benchmark中各个事务的类型以及各个事务相关的参数，包括名称、预执行等待时间、后执行等待时间等。可以满足生成多样化TPC-C查询负载的要求。

模版化的方法在生成SQL查询负载时，一方面需要依赖专家制定的高质量模版，这些模版包含了常见的SQL查询语句形式、谓词条件、列名以及表名等信息。另一方面，生成SQL查询语句时，只能更改模版中的谓词条件的值，这也导致了生成的SQL查询语句形式相对单一。同时，该负载生成器所使用的数据库schema也被限定为固定的结构。

此外，现有的随机生成方法或基于模版的方法都无法根据用户限定的约束条件进行生成，这意味着在实际的应用场景中，当用户需要测试一个数据库的性能时，他们可能需要生成符合特定约束条件的查询负载，以便更好地模拟实际情况。

### 2.3.2 DWG：动态负载生成

本项目设计了一种基于采样的约束感知SQL查询语句生成方法。进行SQL查询语句生成时，在关注SQL查询语句是否能满足用户指定的约束条件的同时，还要关注如何满足给定的数据库schema。因此本论文采用采样的方法来确定语法树结构与选取参数值，其中数据库表、列名，主外键信息等将从输入的数据库schema文件中获取。当前实现的逻辑有单表查询、多表连接、聚合操作、逻辑判断、插入/删除/更新操作等。接下来将依次介绍约束条件的选择、JSON输入文件的设计和采样算法的实现，其后将按照约束条件的顺序详细介绍SQL查询语句的生成过程。

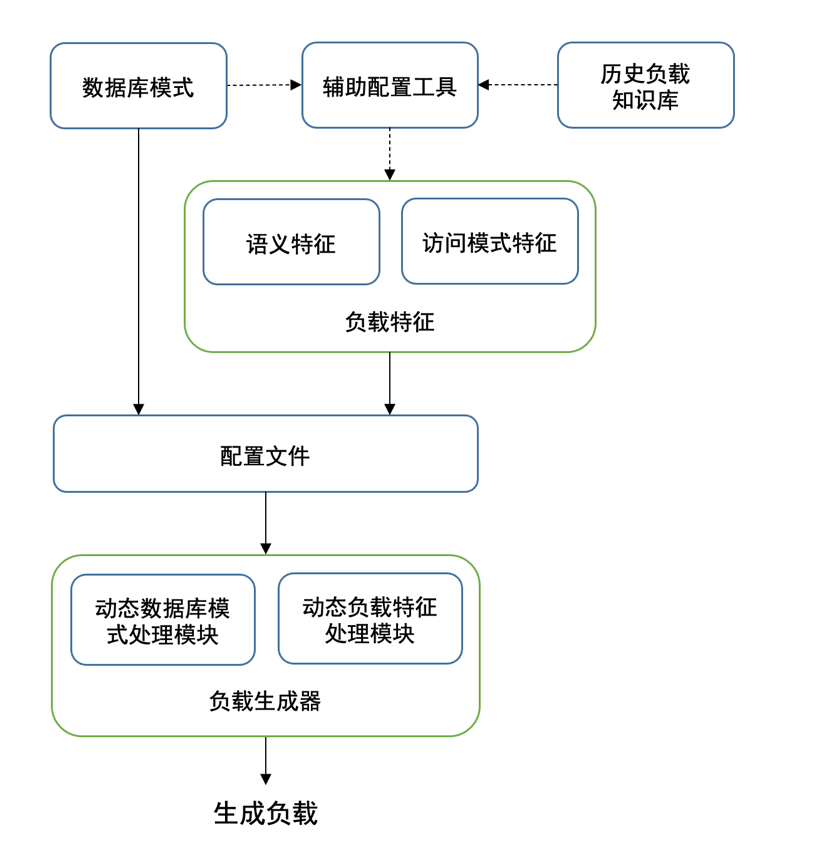


图2.2 SQL生成器内部设计

**项目自主编写的负载生成器在SuperWG/DWG/src目录下，其主要架构如图2.2所示。图2.2中，动态负载生成器主要由两部分组成，一个用于处理动态数据库模式的模块，和一个处理动态负载特征的模块。前者保证负载生成工具可以生成不拘泥于数据库Schema的负载，后者保证生成的负载符合给定特征，进一步确保生成负载的多样化。**在结合两者功能的实现上，是通过抽象类的继承实现的，我们首先设计了一个名为baseGenerator的类，这是一个从未实例化过的抽象类，主要功能为处理动态变化的数据库Schema，确定生成SQL的骨架；然后我们设计了SQLGen类，它继承自baseGenerator类，是实际任务处理的承担者，除了继承来的处理动态数据库Schema的能力，还新增了处理动态负载特征的功能。

**生成首先需要配置好配置文件，代码实现中为方便用户指定约束条件，将数据库schema与用户指定的约束条件通过json文件的方式组织在一起。为了方便用户使用，我们设计了一个用户辅助配置工具，也就是图中的“辅助配置工具”，该工具可以对现有的查询负载进行分析，得到我们想要的约束条件。随后，将相关约束写入配置文件中，我们可以更加方便地对真实查询负载进行仿真，以便于优化数据库的性能和资源利用效率。**

配置文件本身由两部分组成，一是数据库模式配置，二是负载特征配置。数据库模式配置必须由用户手动完成，负载特征配置较为复杂，可以用户直接手动编写配置，如果有困难，也可以使用用户辅助配置工具分析历史负载，帮助用户对负载进行一定程度的解析，帮助用户编写配置文件。

配置文件格式如代码2-6所示，该输入文件中包含数据库名、表名、列名、数据类型、主外键约束，以及用户指定的各类约束条件。

1. {
2. "Table Schema": "",
3. "Tables": [
4. {
5. "Table Name": "",
6. "Table Columns": [
7. {
8. "Column Name": "",
9. "Data Type": "",
10. "Data Distribution": []
11. }
12. ],
13. "Column Distribution" : [],
14. "Primary Key": {
15. "Name": "",
16. "Data Type": ""
17. },
18. "Foreign Key": [
19. {
20. "Foreign Key Name": "",
21. "Foreign Key Type": "",
22. "Referenced Table": "",
23. "Referenced Primary Key": "",
24. "Referenced Primary Key Type": ""
25. }
26. ]
27. }
28. ],
29. "Constraints": {
30. "size of workload" : 0,
31. "read write ratio": 0,
32. "average table num": [],
33. "table-query access distribution": [],
34. "query comparison operator ratio" : [],
35. "table domain distribution" : [],
36. "query logic predicate num" : [],
37. "average aggregation operator num" : [],
38. "average query colomn num":[],
39. "group by ratio if read SQL":[],
40. "order by desc or asc if grouped":[]
41. },
42. "Generation File":"workloadGen.wg"
43. }

代码 0‑6 json文件输入

采样算法的实现参考了朴素的蒙特卡罗方法（Monte Carlo method），这是以摩纳哥的一个赌场命名的，它使用简单的随机事件模拟复杂的概率事件，例如抛掷一对骰子来模拟赌场的整体商业模型。在蒙特卡罗计算中，一个伪随机数生成器被不断调用并返回区间 [0,1] 中的实数，该结果用于生成一个样本分布，其是所研究的目标概率分布的无偏表达。

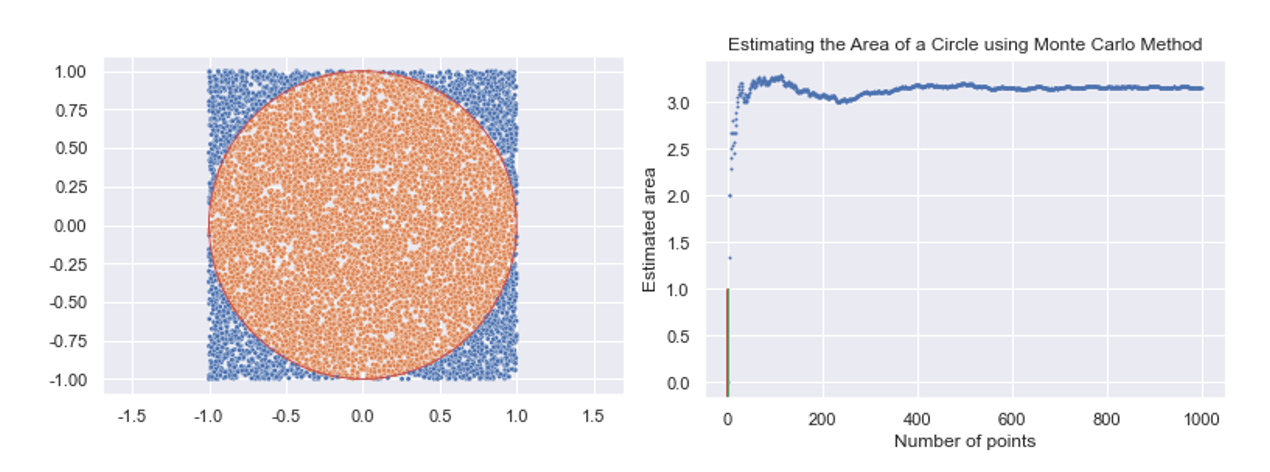
一个典型的例子是计算一个不规则图形的面积，那么图形的不规则程度和分析性计算（比如，积分）的复杂程度是成正比的。蒙特卡罗方法基于这样的想法：假设你有一袋豆子，把豆子均匀地朝这个图形上撒，然后数这个图形之中有多少颗豆子，这个豆子的数目就是图形的面积。当你的豆子越小，撒的越多的时候，结果就越精确。借助计算机程序可以生成大量均匀分布坐标点，然后统计出图形内的点数，通过他们占总点数的比例和坐标点生成范围的面积就可以求出图形面积。如图2.3所示，是用蒙特卡罗方法估计半径为1的圆面积。

图 0.3 蒙特卡罗采样估计圆面积

同样，这个思想还可以用来模拟估算数量，比如计算一个极高维空间 Ω 中计算一个函数的积分：

|  |  |  |
| --- | --- | --- |
|  |  | （2-1） |

这通常通过蒙特卡洛积分来估计。从π(x)中抽取M个样本，

|  |  |  |
| --- | --- | --- |
|  |  |  |

我们可以用样本均值来估算c，

|  |  |  |
| --- | --- | --- |
|  |  | （2-2） |

本项目中采样算法就用到了蒙特卡罗方法的思想，关键点在于代码2-7中rand\_num\_sampling函数的实现，这段代码实现了一个在给定的概率分布图中随机采样的函数，使用这个随机采样函数不断地采样，课逐渐逼近给定的概率分布。输入为左右两侧的index以及给定的概率分布pdg，返回的结果是在区间 [l, r] 内根据给定概率分布随机采样的一个整数。具体来说，它首先检查 pdg 是否符合要求，即长度等于 r-l+1，且概率和为 1。然后将概率分布中的概率值乘以 1000 并四舍五入，得到一个整数数组 pdg\_。接下来，将 pdg\_中所有元素相加得到一个最大值 maxv。最后，随机生成一个 0 到 maxv-1 的整数 num，然后逐个减去 pdg\_ 中的元素值，直到 num 小于 0，此时的索引即为随机采样的结果。

1. def rand\_num\_sampling(l,r,pdg):
2. if r-l+1 != len(pdg):
3. print("l is "+str(l)+" r is "+str(r)+" and pdg is "+str(pdg))
4. print("error : pdg not match [l,r]")
5. return l-1
6. one\_=sum(pdg)
7. if abs(one\_-1.0)>1e-5:
8. print("error : probability sum "+str(one\_)+" do not equal 1")
9. return l-1
10. pdg\_=np.array(pdg)
11. pdg\_=[round(i\*1000) for i in pdg\_]
12. maxv=sum(pdg\_)
13. num=random.randint(0,maxv-1)
14. index=0
15. for i in range(len(pdg\_)):
16. num=num-pdg\_[i]
17. if num<0:
18. index=i
19. break
20. return l+index

代码 2‑7 rand\_num\_sampling代码实现

当前用户指定的约束条件有如下几种：

|  |  |
| --- | --- |
| 特征所属层次 | 特征含义 |
| 语义特征 | 负载规模 |
| 读写比 |
| 平均每条语句查询涉及到的表的数目 |
| 平均每条语句查询约束包含的逻辑谓词数目 |
| 平均每条语句查询包括的聚合函数数目 |
| 平均每条语句包含的GROUP BY操作数量 |
| 平均每条语句查询返回ITEM的有效属性数目 |
| 数据访问模式特征 | 查询涉及到各表的访问分布 |
| 查询涉及到表的各列的访问分布 |
| 查询涉及到的列的查询值域区间的访问分布 |
| WHERE子句涉及到的比较约束种类比例访问分布 |
| 包含排序操作时，升序与降序的排序比例访问分布 |
| 执行特征 | 负载执行有效结果数（基数） |

其中，语义特征和数据访问特征为静态特征，即不需要数据库执行，可以静态分析出的查询负载特征。负载特征是指在数据库系统中，被用户或应用程序提交的SQL查询请求的各种参数、模式等方面的特征。负载特征可以分为两类：静态特征和动态特征。静态特征是指与SQL语句本身、数据或用户访问模式等相关的属性，如语义特征、数据分布和访问频度特征等。这些特征通常可以通过预先分析得到，并且在一定程度上是稳定的，不随环境变化而改变。与之相对的动态特征是指与查询请求在运行时所表现出的性能、执行特征等相关的属性。这些特征通常需要在实际运行中动态观察得到，并且受到运行环境的影响，可能会发生变化，如执行特征、事务逻辑特征等等。

语义特征仅和SQL语句自身有关，体现了工作负载中SQL语句的特点。常见的语义特征包括：读写比（SELECT、INSERT、DELETE、UPDATE的数量比），SELECT子句中的聚集函数（count、sum、avg、max、min）、列，FROM子句中的表，WHERE字句中的过滤范围、Join条件，GROUP BY子句中的列，ORDER BY子句中的列等。比如config文件中的“read write ratio”读写比约束条件，用户可以通过编辑config文件轻松地控制生成SQL查询负载的读写比例。语义特征的选取参考了Benchbase中传统负载生成器使用的SQL专家模板，确保生成语句具有一定的多样性。

数据分布和访问频度特征考虑到了data、schema或用户常用访问模式（某些表、某些列、某些值用户访问比较频繁）对工作负载的影响，如DB schema、查询数据分布区间、查询表的频次期望、查询列的频次期望等。应该注意到，数据分布或访问频度特征和数据特征的区别：前者属于负载（workload）特征，体现了工作负载在不同表、列、值上访问频率的差异；后者属于数据（data）自身的特征，如数据规模、值域、属性等。比如config文件中“Column Distribution”列采样分布约束条件，用户可以轻易地设定访问频繁地列并声称相应的SQL查询负载。数据访问模式的建模特征，参考了数据库垂直分区领域中，传统的统计访问频数矩阵的算法思路，确保生成的语句能从多个角度满足访问模式分布，同样达到了对负载进行分类划分的目标。

生成流程可以被看作是一个自动机状态变化的过程，如图2.4所示。在限制条件"a"为"average table num"的情况下，当用户指定的语句中表的数目为1时，当前状态会转换到下一个状态。此时，可以确定当前的SELECT语句中仅包含一个名为"A"的表（其中，"A"是一个参数占位符，用来帮助表示表的数量，而非具体的表名）。

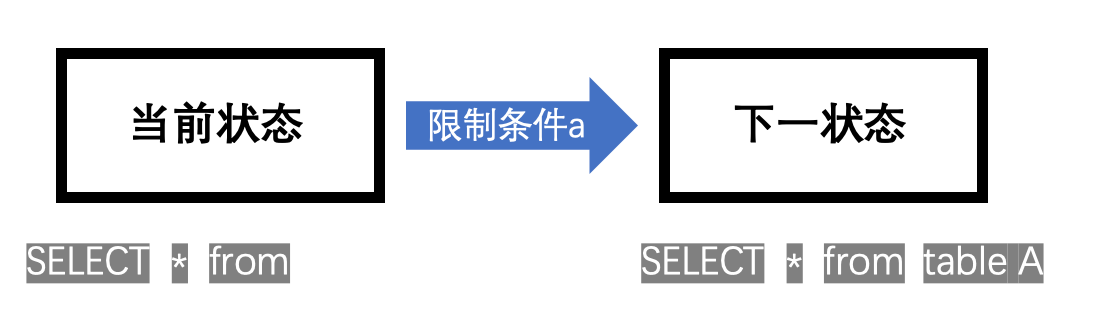
具体来说，首先依据“read write ratio”约束条件确定语句类型，利用rand\_num\_sampling函数进行采样，若随机采样结果落在读语句区间则生成Select查询语句，若随机采样结果落在写语句区间则生成Insert/Delete/Update语句。

图2.4 以average table num为约束条件的生成步骤

采样确定为select语句后，需要依据用户指定的“average table num”约束条件分布确定语句中的表数目，本项目只考虑语句中表数目为1或2的情况。若采样结果为1，则该select语句为单表查询语句；若采样结果为2，则该select语句包含连接操作。

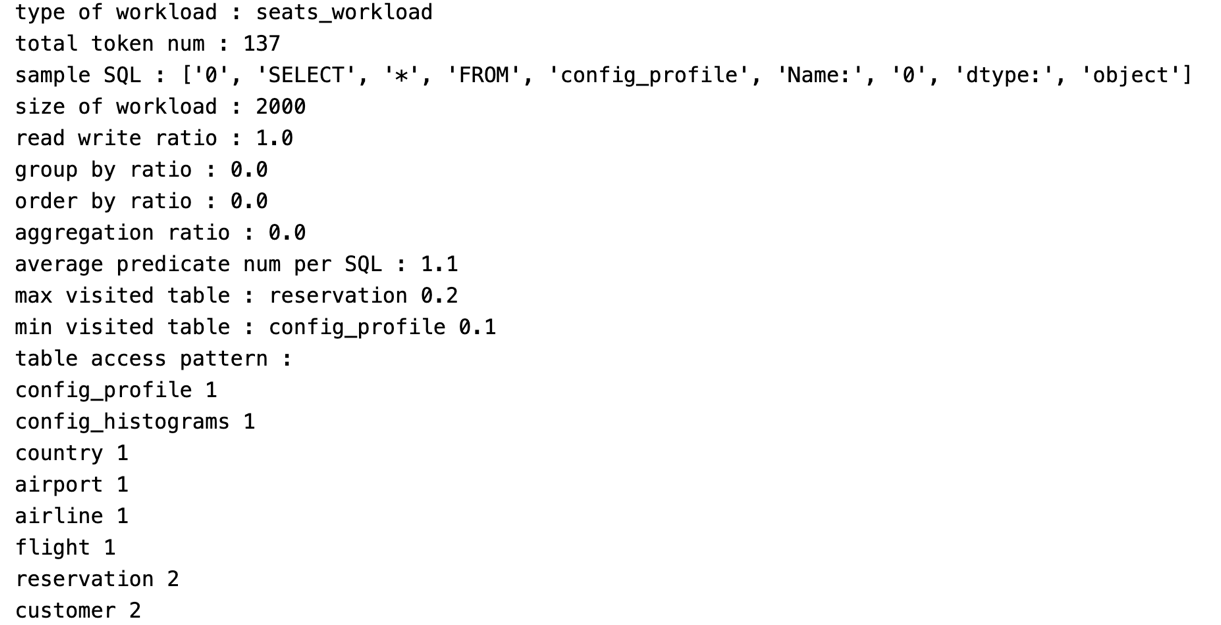
在生成单表单表查询的情况下，依据“table-query access distribution”约束条件采样确定该查询语句中出现的表。这样一来，我们就确定了FROM关键字后出现的表名。接下来，我们依据“average query colomn num”约束条件采样出该查询语句中出现的列的数目（指SELECT关键字后列的数目），并依据“Column Distribution”约束条件采样确定该查询语句中出现的具体的列。接下来通过“average aggregation operator num”约束条件采样确定是否出现聚合操作以及聚合操作的数目，至于具体聚合操作的类型（count、avg、min、max）则是通过均匀分布采样确定，即随机选择。这样，我们确定的字段有如下：select a from tb1 where 。接下来，我们根据“query logic predicate num”约束条件采样确定查询语句中</>/=逻辑判断的出现次数，</>/=左侧的字段则依次通过表分布和列分布采样确定，右侧的字段通过列中数据采样分布（即“table domain distribution”限制条件）采样决定，多个逻辑判断间and/or字段的选择通过均匀分布采样确定。这样一来我们可以得到一条形如select a from tb1 where c>5的Select查询语句。

包含连接操作的情况下，与单表查询类似，但需额外处理连接操作字段，本项目目前只考虑2张表的多表连接语句生成。首先依据“table-query access distribution”约束条件采样确定该查询语句中出现的2张表，并依据“average query colomn num”约束条件采样确定查询语句中出现的列的数目，同样该处列的数目是指SELECT字段后列的数目，并依据给定列采样分布从采样出的两张表中采样对应数目的列。接下来通过“average aggregation operator num”约束条件采样确定是否出现聚合操作以及聚合操作的数目，至于具体聚合操作的类型（count、avg、min、max）则是通过均匀分布采样确定，即随机选择。这样，我们确定的字段有如下：select \* from tb1,b2 where 。接下来，我们根据“query logic predicate num”约束条件采样确定查询语句中</>/=逻辑判断的出现次数，</>/=两侧的字段则依次通过表分布和列分布采样确定，多个逻辑判断间and/or字段的选择通过均匀分布采样确定。最后我们可以得到形如select \* from tb1,b2 where tb1.a>tb2.a or tb1.b<tb2.b的Select查询语句

所有语句中谓词值可以通过参数“table domain distributon”人工指定，目前所有语句中谓词值均可以通过几种固定的采样分布确定，如均匀分布、高斯分布。

### 2.3.3 用户辅助配置工具

本项目中，我们将约束条件视为静态特征，这些约束条件是可以通过分析数据库schema和真实查询负载得到的，但是人工设置这些约束条件往往十分繁琐。为了方便用户使用，我们设计了一个用户辅助配置工具，该工具可以对现有的查询负载进行分析，得到我们想要的约束条件。随后，将相关约束写入配置文件中，我们可以更加方便地对真实查询负载进行仿真，以便于优化数据库的性能和资源利用效率。

图2.5 负载分析部分结果

总的来说，这一用户辅助配置工具会对给定的负载（包括负载本身和相关数据库配置）进行分析，总结出负载的特征作为其肖像。具体来说，这一部分的工具主要解析了负载的**语义特征和数据访问模式特征**。用户指定的约束条件都可以通过静态分析进行确定，这意味着我们可以直接对查询负载进行分析，并根据负载特征推导出必要的约束条件。例如，“read write ratio”约束条件可以通过分析查询负载中select语句和其他语句的占比来确定，因为这个比例可以反映查询负载的读写特征。图2.5为SEATS查询负载的部分负载分析结果，包含负载中所有字段的数目、负载中查询语句的条数、读写比、包含group by语句占比、包含聚合操作语句占比、表格访问模式等等。

需要注意的是，如果要使用辅助配置工具，必须拥有历史负载数据和对应的数据库模式配置，否则无法分析。这是由语法解析过程决定的。没有历史负载数据自然也无从分析，而必须要有对应的数据库模式配置的原因是，涉及多表连接语句时，所查询的列归属于哪个表可能会产生歧义。因此，辅助配置工具必须拥有完整的历史负载数据和对应的数据库模式配置作为输入。

在负载解析的实现方面，首先需要从输入的configure.json文件中解析出数据库Schema等重要配置，获取数据库全貌。这一步是必要的，否则，在涉及多表连接的读语句中，如果在对列数据进行操作时省略列所属的表名，将无法确定选定的列归属于哪张表。为此，项目设计了一系列数据结构，对应解析出的数据库配置，具体包括：包含的表、表包含的列、列数据类型、列值域、主键约束、外键约束等。

在对数据库的配置文件解析完毕后，对负载内容进行逐条解析。其中，执行特征需要负载内容自身与基数进行一一对应，因为没有数据库存储的数据全貌，无法实际执行负载统计结果。

有了这一辅助工具，就可以先使用负载解析器对真实的查询负载进行分析，辅助用户将分析的结果写入config文件中，然后批量生成用户需要的SQL查询负载。

# 三、使用与测试

在SuperWG/DWG目录下配置好configure.json（负载配置），执行SuperWG/bin目录下的run\_dwg.sh即可在终端显示生成结果。在SuperWG/DWG/workloads目录下配置好configure.json（负载配置）和workload.content（负载内容），执行SuperWG/bin目录下的run\_parser.sh即可在终端显示分析结果。

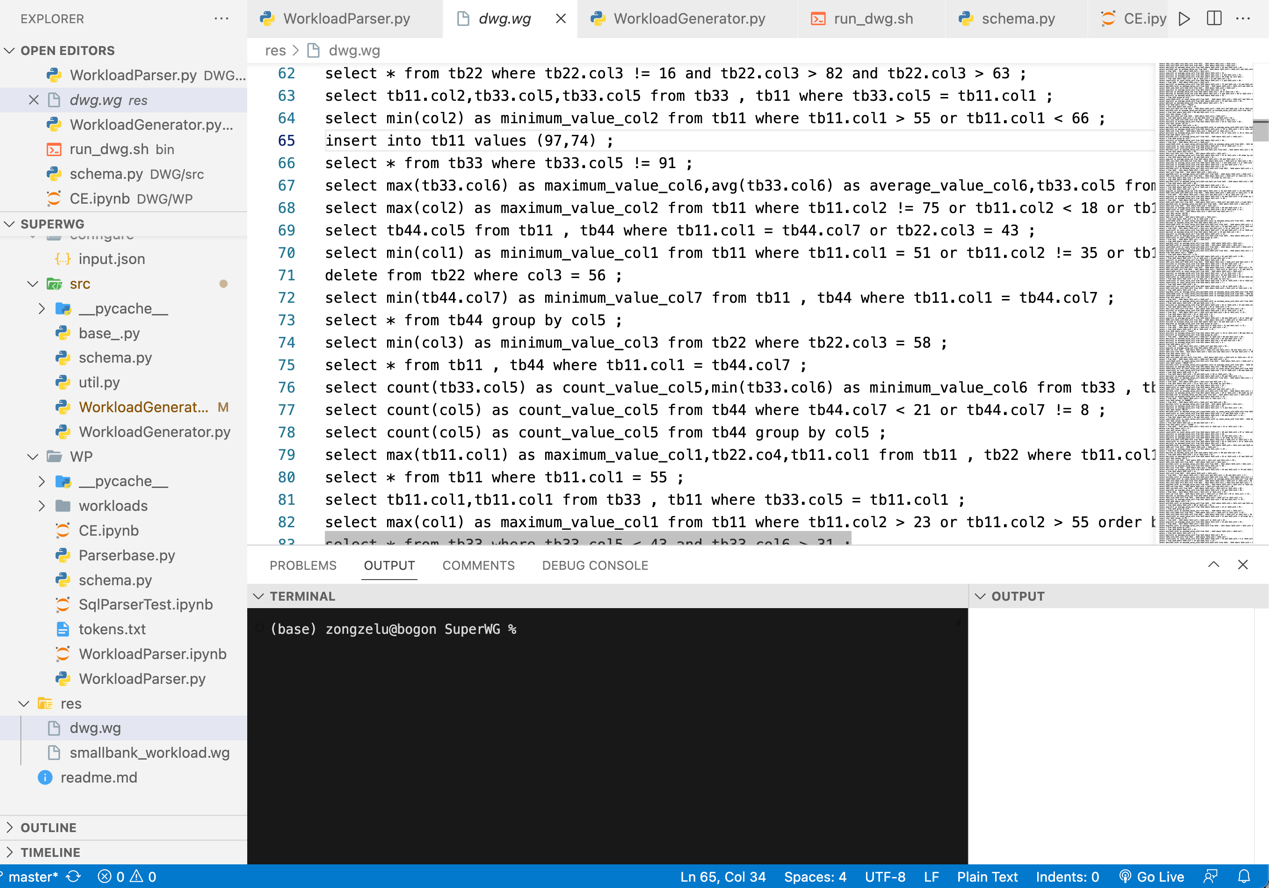


图3.1 示例配置下的查询生成结果

如图3-1所示，dwg.wg文件中是DWG依据附录四中的配置文件所生成的查询负载，其中包含select单表查询语句、insert插入语句、多表连接操作、聚合操作等等。生成语句的多样化程度可以得到保障，同时具有相当高的生成速率（平均每秒生成的查询语句数目在6300到6600之间），可以在短时间内生成大规模的查询负载，具备良好的可扩展性。

图3-2展示了生成的SQL语句中不同语句类型占比与表采样占比。如附录四所示，约束条件“read write ratio”设定为0.6，即读写占比为6:4，如3-2左图所示SELECT语句占比为60.6%，其余三种写语句占比为39.4%，写语句的类型由均匀分布采样结果确定，故三者占比基本一致；同样的，如附录四所ss示，表采样分布约束条件“table-query access distribution”设定为[0.1, 0.4, 0.3, 0.2]，即tb1、tb2、tb3、tb4被采样到的概率分别为[0.1, 0.4, 0.3, 0.2]，如3-2右图tb1、tb2、tb3、tb4的表采样分布严格遵循我们在input文件中指定的约束条件，SuperWG生成语句的准确性可以保证。

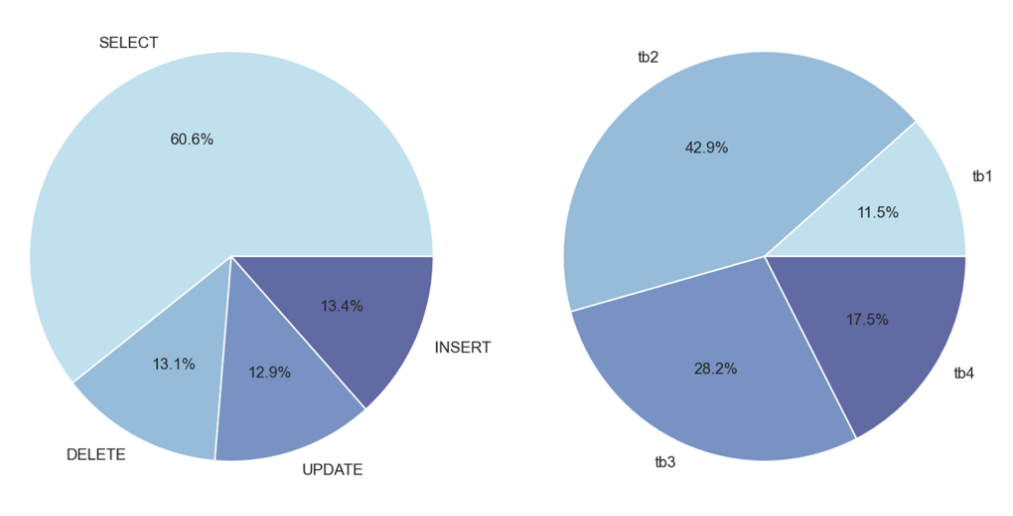


图3.2 生成语句结果与表采样统计分析

代码3-1展示了部分由DWG生成的SQL查询语句，可以看到生成的SQL语句严格遵附录四中用户指定的数据库schema信息，并且包含多样化且语法正确的select语句，如多表连接、聚合操作等等。SuperWG在保证生成的SQL语句符合指定数据库schema的同时也保证了生成语句的语法正确性。

1. select min(co4) as minimum\_value\_co4 from tb22 where tb22.col3 != 23 and tb22.co4 = 88;
2. select tb22.col3,tb11.col2,tb11.col2 from tb11 join tb22 where tb11.col1 = tb22.col3 or tb11.col1 > 19;
3. insert into tb33 value (0rELS,4);
4. select \* from tb33 join tb22 where tb33.col5 = tb22.col3 or tb22.col3 != 88;
5. select avg(tb11.col1) as average\_value\_col1,max(tb33.col6) as maximum\_value\_col6,tb11.col1 from tb11 join tb33 where tb11.col1 = tb33.col5;
6. select tb22.col3,tb44.col7 from tb44 join tb22 where tb44.col7 = tb22.col3
7. select tb44.col7,tb44.col5,tb22.co4 from tb44 join tb22 where tb44.col7 = tb22.col3 and tb11.col1 > 43 or tb22.co4 = 67;

代码 3‑1 DWG生成的部分SQL查询语句

代码3-2、代码3-3分别展示了由修改后的benchbase以模版化生成方式生成的Smallbank和TPC-C的SQL语句，这两类基准测试的生成方法都是预先指定数据库schema，通过实际应用场景设计具体的查询事务模版，并依赖模版进行SQL语句生成。

1. SELECT \* FROM accounts WHERE custid = 510857;
2. SELECT \* FROM accounts WHERE custid = 454019;
3. SELECT bal FROM checking WHERE custid = 510857;
4. UPDATE checking   SET bal = bal + -5.0  WHERE custid = 510857;
5. UPDATE checking   SET bal = bal + 5.0  WHERE custid = 454019;

代码3‑2 模版化方法生成的部分Smallbank SQL查询语句

1. SELECT C\_FIRST, C\_MIDDLE, C\_LAST, C\_STREET\_1, C\_STREET\_2,        C\_CITY, C\_STATE, C\_ZIP, C\_PHONE, C\_CREDIT, C\_CREDIT\_LIM,        C\_DISCOUNT, C\_BALANCE, C\_YTD\_PAYMENT, C\_PAYMENT\_CNT, C\_SINCE   FROM customer WHERE C\_W\_ID = 1    AND C\_D\_ID = 4    AND C\_ID = 1141;
2. SELECT O\_ID, O\_CARRIER\_ID, O\_ENTRY\_D   FROM oorder WHERE O\_W\_ID = 1    AND O\_D\_ID = 4    AND O\_C\_ID = 1141  ORDER BY O\_ID DESC LIMIT 1;
3. SELECT OL\_I\_ID, OL\_SUPPLY\_W\_ID, OL\_QUANTITY, OL\_AMOUNT, OL\_DELIVERY\_D   FROM order\_line WHERE OL\_O\_ID = 1860   AND OL\_D\_ID = 4   AND OL\_W\_ID = 1;
4. UPDATE warehouse   SET W\_YTD = W\_YTD + 1115.489990234375  WHERE W\_ID = 1 ;
5. SELECT W\_STREET\_1, W\_STREET\_2, W\_CITY, W\_STATE, W\_ZIP, W\_NAME  FROM warehouse WHERE W\_ID = 1;

代码3‑3 模版化方法生成的部分TPC-C SQL查询语句

# 附录一

## 参考文献

[1] Liu, Xinyue, et al. "TreeGAN: syntax-aware sequence generation with generative adversarial networks." *2018 IEEE International Conference on Data Mining (ICDM)*. IEEE, 2018.

[2] Li, Beibin, Yao Lu, and Srikanth Kandula. "Warper: Efficiently Adapting Learned Cardinality Estimators to Data and Workload Drifts." Proceedings of the 2022 International Conference on Management of Data. 2022.

[3] Andreas Seltenreich. SQLSmith, 2020. https://github.com/anse1/sqlsmith.

[4] Percona. Sysbench, GitHub, 2021, <https://github.com/akopytov/sysbench>.

[5] TPC Benchmark™ H Standard Specification Revision 2.17.1. Transaction Processing Performance Council, n.d., http://www.tpc.org/tpc\_documents\_current\_versions/pdf/tpc-h\_v2.17.1.pdf.

[6] TPC Benchmark™ C Standard Specification Revision 5.11. Transaction Processing Performance Council, n.d., http://www.tpc.org/tpc\_documents\_current\_versions/pdf/tpc-c\_v5.11.0.pdf.

# 附录二

## Sysbench介绍

Sysbench是一个基于LuaJIT的可脚本化的多线程基准测试工具，旨在测试和分析各种系统的性能，包括数据库、文件系统、CPU、内存和磁盘I/O等。它的编写语言是Lua，以其灵活性、简单性和易用性而闻名。Sysbench特别适用于数据库基准测试，因为它内置了对多个流行数据库的支持，包括MySQL、PostgreSQL、Oracle、MariaDB等。它可以模拟各种负载和工作负荷，包括随机数据生成、查询和更新操作等，并能够以多线程和分布式的方式运行测试。Sysbench所使用的SQL语句生成模版如下：

1. local stmt\_defs = {
2. point\_selects = {
3. "SELECT c FROM sbtest%u WHERE id=?",
4. t.INT},
5. simple\_ranges = {
6. "SELECT c FROM sbtest%u WHERE id BETWEEN ? AND ?",
7. t.INT, t.INT},
8. sum\_ranges = {
9. "SELECT SUM(k) FROM sbtest%u WHERE id BETWEEN ? AND ?",
10. t.INT, t.INT},
11. order\_ranges = {
12. "SELECT c FROM sbtest%u WHERE id BETWEEN ? AND ? ORDER BY c",
13. t.INT, t.INT},
14. distinct\_ranges = {
15. "SELECT DISTINCT c FROM sbtest%u WHERE id BETWEEN ? AND ? ORDER BY c",
16. t.INT, t.INT},
17. index\_updates = {
18. "UPDATE sbtest%u SET k=k+1 WHERE id=?",
19. t.INT},
20. non\_index\_updates = {
21. "UPDATE sbtest%u SET c=? WHERE id=?",
22. {t.CHAR, 120}, t.INT},
23. deletes = {
24. "DELETE FROM sbtest%u WHERE id=?",
25. t.INT},
26. inserts = {
27. "INSERT INTO sbtest%u (id, k, c, pad) VALUES (?, ?, ?, ?)",
28. t.INT, t.INT, {t.CHAR, 120}, {t.CHAR, 60}},
29. }

## SIbench介绍

SIBench是一个用来探索数据库管理系统中快照隔离（snapshot isolation）的微型基准。它包含一个单一的key-value表格和两个事务（分别获取列的最小值或增加条目的单个值）。这个工作负载会创建数据库管理系统必须解决读写冲突的情况，同时通过扫描表格查找最小值来对CPU进行压力测试。

1. // 获取列的最小值
2. public final SQLStmt minStmt = new SQLStmt(
3. "SELECT id FROM " + TABLE\_NAME + " ORDER BY value ASC LIMIT 1"
4. );
5. // 增加条目的单个值
6. public final SQLStmt updateStmt = new SQLStmt(
7. "UPDATE " + TABLE\_NAME + " SET value = value + 1 WHERE id = ?"
8. );

## SEATS介绍

SEATS benchmark模拟了一个航空公司的机票销售系统，客户可以搜索航班并在线预订。它由八个表和六种交易类型组成。大约60％的交易是只读的（例如，客户搜索空余座位），而其他40％涉及创建、更新和删除预订记录。

该benchmark模拟了一个后端系统，处理来自多个应用程序的请求，每个应用程序提供不同的输入。因此，许多交易使用次要索引或外键连接来查找客户预订记录的主键。例如，客户可以使用不同的凭据访问系统，包括他们的常旅客号。

SEATS benchmarl的schema包含8个表：COUNTRY、AIRPORT、AIRPORT\_DISTANCE、AIRLINE、CUSTOMER、FREQUENT\_FLYER、FLIGHT、RESERVATION，其中前四张都为只读表。

CUSTOMER：有N\*10\*\*6个客户，每个客户都有一个整数的客户标识符（CID）和40个属性数据字段，例如姓名、地址、常旅客号码等。20个是长度为8字节的字符串，20个是4字节的整数。因此，总记录长度为244字节。对于N=1，这就是244M字节。CIDs可以被认为是1和N\*10\*\*6之间的正整数。

FLIGHT：有N\*10\*\*5个航班。每个航班都有一个航班标识符（F\_ID），一个航空公司（AL\_ID），出发机场（DEPART\_AP\_ID），出发时间，到达机场（ARRIVE\_AP\_ID），到达时间，以及30个杂项数据属性。每个F\_ID是一个64位的复合ID，包含出发/到达机场ID、飞行日期和一个唯一的标识符。这使得benchmark driver能够执行各种交易。

RESERVATION：每个预订对应一个客户和一个航班；一个预订有10个数据字段，每个字段都是整数（例如，航班的费用和选择的座位）。因此每个航班预订量都在60-100%之间，平均为80%。每架飞机被假定为有150个座位，预计总共有1200万个预订。N=1时，预订表的总大小为624Mbytes。随着航班数量随着N的增加而增加，预订量也会增加。

一共包含六种事务类型：ChangeSeat、FindFlightByAirport、FindFlightByNearbyAirport、FindOpenSeats、NewReservation、UpdateFrequentFlyer、UpdateReservation。大约60%的事务是只读的（例如，客户搜索开放的座位），而其他40%的事务涉及创建、更新和删除预订记录。

例如NewReservation操作OLTP-Bench中事务模版如下：

1. *//查询航班信息*
2. public final SQLStmt GetFlight = new SQLStmt(
3. "SELECT F\_AL\_ID, F\_SEATS\_LEFT, " +
4. SEATSConstants.TABLENAME\_AIRLINE + ".\* " +
5. "  FROM " + SEATSConstants.TABLENAME\_FLIGHT + ", " +
6. SEATSConstants.TABLENAME\_AIRLINE +
7. " WHERE F\_ID = ? AND F\_AL\_ID = AL\_ID");
8. *//查询用户信息*
9. public final SQLStmt GetCustomer = new SQLStmt(
10. "SELECT C\_BASE\_AP\_ID, C\_BALANCE, C\_SATTR00 " +
11. "  FROM " + SEATSConstants.TABLENAME\_CUSTOMER +
12. " WHERE C\_ID = ? ");
13. *//查询座位预定信息*
14. public final SQLStmt CheckSeat = new SQLStmt(
15. "SELECT R\_ID " +
16. "  FROM " + SEATSConstants.TABLENAME\_RESERVATION +
17. " WHERE R\_F\_ID = ? and R\_SEAT = ?");
18. *//查询用户预定信息*
19. public final SQLStmt CheckCustomer = new SQLStmt(
20. "SELECT R\_ID " +
21. "  FROM " + SEATSConstants.TABLENAME\_RESERVATION +
22. " WHERE R\_F\_ID = ? AND R\_C\_ID = ?");
23. *//更新航班剩余座位数目*
24. public final SQLStmt UpdateFlight = new SQLStmt(
25. "UPDATE " + SEATSConstants.TABLENAME\_FLIGHT +
26. "   SET F\_SEATS\_LEFT = F\_SEATS\_LEFT - 1 " +
27. " WHERE F\_ID = ? ");
28. *//更新用户属性表*
29. public final SQLStmt UpdateCustomer = new SQLStmt(
30. "UPDATE " + SEATSConstants.TABLENAME\_CUSTOMER +
31. "   SET C\_IATTR10 = C\_IATTR10 + 1, " +
32. "       C\_IATTR11 = C\_IATTR11 + 1, " +
33. "       C\_IATTR12 = ?, " +
34. "       C\_IATTR13 = ?, " +
35. "       C\_IATTR14 = ?, " +
36. "       C\_IATTR15 = ? " +
37. " WHERE C\_ID = ? ");
38. *//更新FrequentFlyer表*
39. public final SQLStmt UpdateFrequentFlyer = new SQLStmt(
40. "UPDATE " + SEATSConstants.TABLENAME\_FREQUENT\_FLYER +
41. "   SET FF\_IATTR10 = FF\_IATTR10 + 1, " +
42. "       FF\_IATTR11 = ?, " +
43. "       FF\_IATTR12 = ?, " +
44. "       FF\_IATTR13 = ?, " +
45. "       FF\_IATTR14 = ? " +
46. " WHERE FF\_C\_ID = ? " +
47. "   AND FF\_AL\_ID = ?");
48. *//插入新预定*
49. public final SQLStmt InsertReservation = new SQLStmt(
50. "INSERT INTO " + SEATSConstants.TABLENAME\_RESERVATION + " (" +
51. "   R\_ID, " +
52. "   R\_C\_ID, " +
53. "   R\_F\_ID, " +
54. "   R\_SEAT, " +
55. "   R\_PRICE, " +
56. "   R\_IATTR00, " +
57. "   R\_IATTR01, " +
58. "   R\_IATTR02, " +
59. "   R\_IATTR03, " +
60. "   R\_IATTR04, " +
61. "   R\_IATTR05, " +
62. "   R\_IATTR06, " +
63. "   R\_IATTR07, " +
64. "   R\_IATTR08 " +
65. ") VALUES (" +
66. "   ?, " +  *// R\_ID*
67. "   ?, " +  *// R\_C\_ID*
68. "   ?, " +  *// R\_F\_ID*
69. "   ?, " +  *// R\_SEAT*
70. "   ?, " +  *// R\_PRICE*
71. "   ?, " +  *// R\_ATTR00*
72. "   ?, " +  *// R\_ATTR01*
73. "   ?, " +  *// R\_ATTR02*
74. "   ?, " +  *// R\_ATTR03*
75. "   ?, " +  *// R\_ATTR04*
76. "   ?, " +  *// R\_ATTR05*
77. "   ?, " +  *// R\_ATTR06*
78. "   ?, " +  *// R\_ATTR07*
79. "   ? " +   *// R\_ATTR08*
80. ")");

## Smallbank介绍

Smallbank这个工作负载模拟了银行应用程序，其中交易对客户账户执行简单的读取和更新操作。所有交易都涉及少量元组。交易的访问模式呈倾斜状态，以至于只有少量的账户接收到大部分请求。我们还扩展了原始的SmallBank实现，以包括在账户之间转移资金的额外交易。

schema包含三个表：Account(Name, CustomerID), Saving(CustomerID, Balance), Checking(CustomerID, Balance)。Account表代表客户，Name是主键，并将Name映射为唯一的CustomerID。CustomerID是saving表和checking表的主键；Balance列代表一个客户在相应账户中的余额。

一共包含五种事务类型：Balance、deposit-checking、withdraw-from-checking、transfer-to-savings和amalgamate 。每种交易类型都涉及少量简单的读取和更新操作。

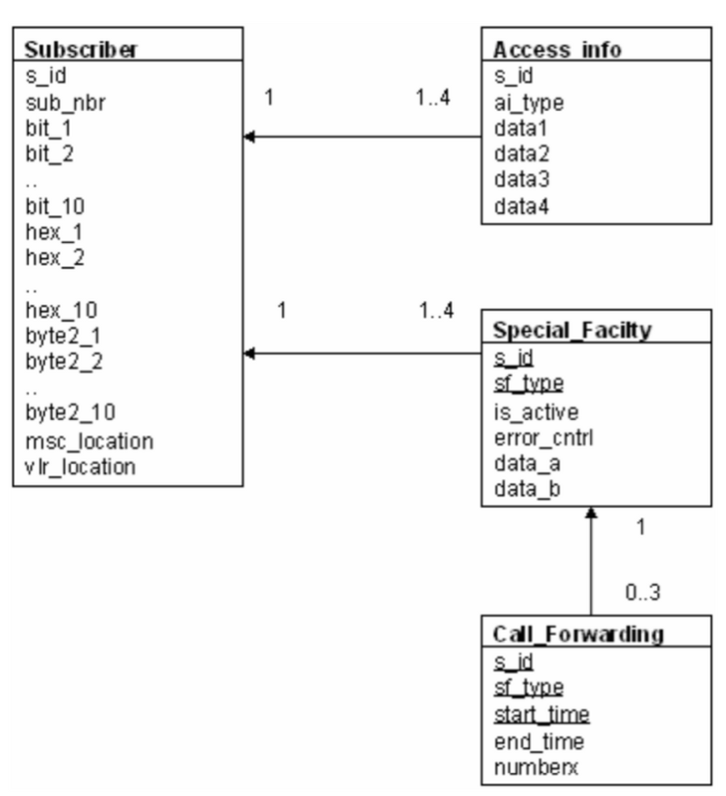
例如查询余额事务Balance在OLTP-Bench中事务模版如下：

1. public final SQLStmt GetAccount = new SQLStmt(
2. "SELECT \* FROM " + SmallBankConstants.TABLENAME\_ACCOUNTS +
3. " WHERE name = ?"
4. );
5. public final SQLStmt GetSavingsBalance = new SQLStmt(
6. "SELECT bal FROM " + SmallBankConstants.TABLENAME\_SAVINGS +
7. " WHERE custid = ?"
8. );
9. public final SQLStmt GetCheckingBalance = new SQLStmt(
10. "SELECT bal FROM " + SmallBankConstants.TABLENAME\_CHECKING +
11. " WHERE custid = ?"
12. );

OLTP-Bench中扩展了最初的SmallBank实现，包括一个额外的事务SendPayment，即在账户之间转移资金。

## TATP介绍

TATP基准测试模拟了移动运营商使用的典型本地位置寄存器（HLR）数据库。HLR是移动网络运营商用于存储所有有关有效订阅者的相关信息的应用程序，包括移动电话号码、已订阅的服务、访问权限以及订户手机的当前位置。每次与移动电话有关的呼叫都需要对双方的HLR进行查找，因此它是一个高负载、高吞吐量的环境的完美示例，所有需要极速运行的应用程序，如电信、金融服务、游戏、事件处理和警报、预订系统等都会涉及到该数据库。

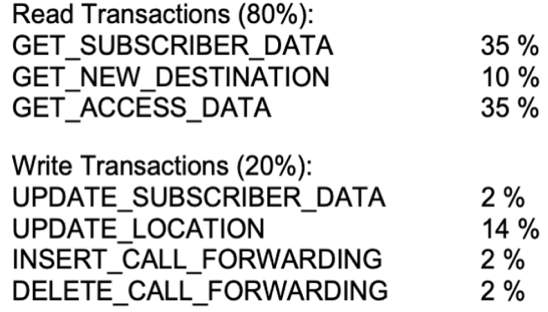


schema包含4张表：Subscriber、Access\_Info、Special\_Facility、Call\_Forwarding

一共包含7种事务类型：GET\_SUBSCRIBER\_DATA、GET\_NEW\_DESTINATION、GET\_ACCESS\_DATA、UPDATE\_SUBSCRIBER\_DATA、UPDATE\_LOCATION、INSERT\_CALL\_FORWARDING、DELETE\_CALL\_FORWARDING。其中除UPDATE\_SUBSCRIBER\_DATA、INSERT\_CALL\_FORWARDING、DELETE\_CALL\_FORWARDING事务外，其余均为单条语句的SELECT或UPDATE操作

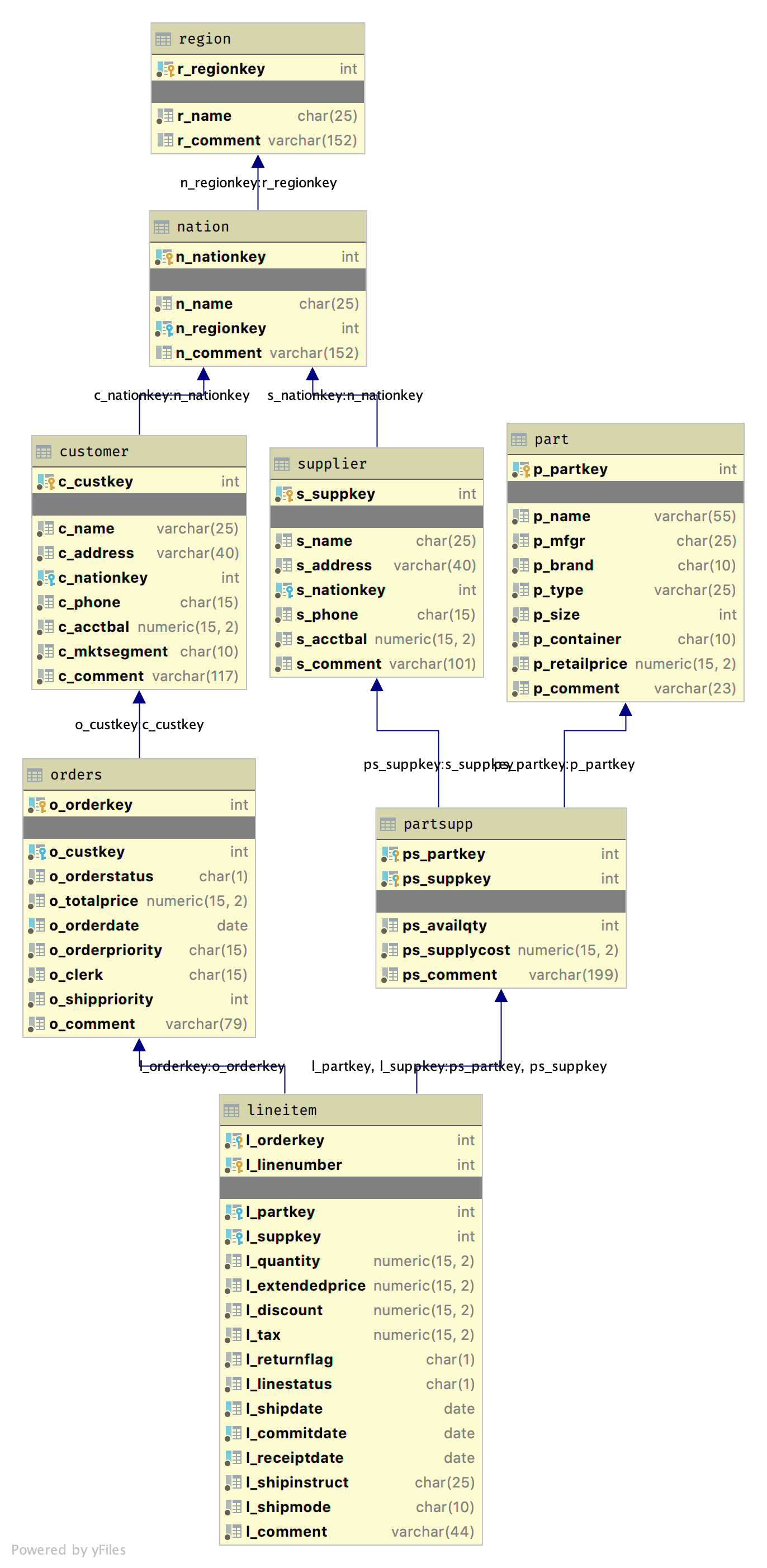
例如UPDATE\_SUBSCRIBER\_DATA、INSERT\_CALL\_FORWARDING、DELETE\_CALL\_FORWARDING在官方Description中事务模版如下

1. *# UPDATE\_SUBSCRIBER\_DATA*
2. UPDATE Subscriber
3. SET bit\_1 = <bit\_rnd>
4. WHERE s\_id = <s\_id rnd subid>;
5. UPDATE Special\_Facility
6. SET data\_a = <data\_a rnd>
7. WHERE s\_id = <s\_id value subid>
8. AND sf\_type = <sf\_type rnd>;
9. *# INSERT\_CALL\_FORWARDING*
10. SELECT <s\_id bind subid s\_id>
11. FROM Subscriber
12. WHERE sub\_nbr = <sub\_nbr rndstr>;
13. SELECT <sf\_type bind sfid sf\_type>
14. FROM Special\_Facility
15. WHERE s\_id = <s\_id value subid>:
16. INSERT INTO Call\_Forwarding
17. VALUES (<s\_id value subid>, <sf\_type rnd sf\_type>,
18. <start\_time rnd>, <end\_time rnd>, <numberx rndstr>);
19. *# DELETE\_CALL\_FORWARDING*
20. SELECT <s\_id bind subid s\_id>
21. FROM Subscriber
22. WHERE sub\_nbr = <sub\_nbr rndstr>;
23. DELETE FROM Call\_Forwarding
24. WHERE s\_id = <s\_id value subid>
25. AND sf\_type = <sf\_type rnd>
26. AND start\_time = <start\_time rnd>;

TATP benchmark中各事务占比以及读写比如下：

# 

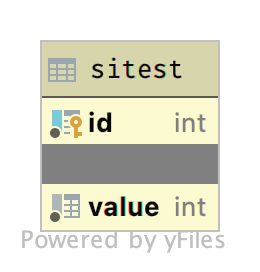
# 附录三

TPC-H指定的schema

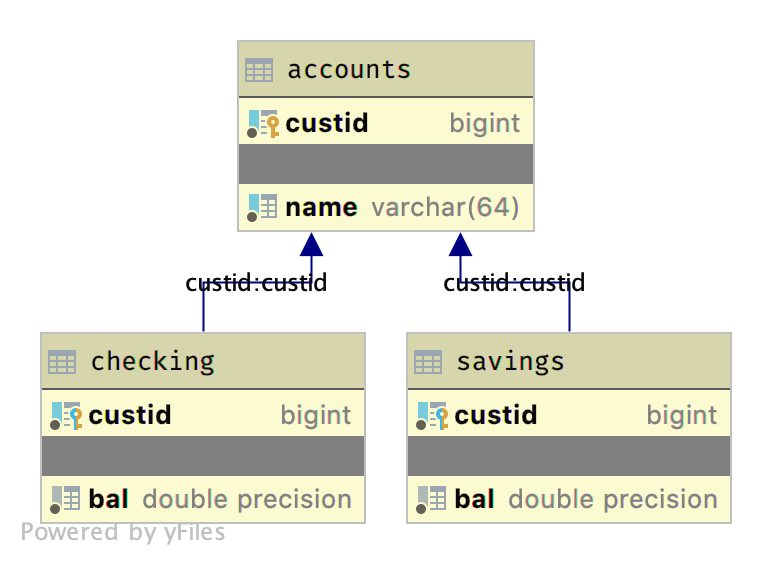
## TPC-C指定的schema

## TATP和SEATS指定的schema

## SIBench指定的schema



## Smallbank指定的schema



# 附录四

## 一份完整的Input输入文件示例

1. {
2. "Table Schema": "schema",
3. "Tables": [ {
4. "Table Name": "tb1",
5. "Table Columns": [ {
6. "Column Name": "col1",
7. "Data Type": "float",
8. "Data Distribution": [0,100]
9. },
10. {
11. "Column Name": "col2",
12. "Data Type": "int",
13. "Data Distribution": [0,100]
14. }
15. ],
16. "Column Distribution" : [0.5, 0.5],
17. "Primary Key": {
18. "Name": "",
19. "Data Type": ""
20. },
21. "Foreign Key": [
22. {
23. "Foreign Key Name": "",
24. "Foreign Key Type": "",
25. "Referenced Table": "",
26. "Referenced Primary Key": "",
27. "Referenced Primary Key Type": ""
28. }
29. ]
30. },
31. {
32. "Table Name": "tb2",
33. "Table Columns": [
34. {
35. "Column Name": "col3",
36. "Data Type": "int",
37. "Data Distribution": [0,100]
38. },
39. {
40. "Column Name": "co4",
41. "Data Type": "int",
42. "Data Distribution": [0,100]
43. }
44. ],
45. "Column Distribution" : [0.5, 0.5],
46. "Primary Key": {
47. "Name": "",
48. "Data Type": ""
49. },
50. "Foreign Key": [
51. {
52. "Foreign Key Name": "",
53. "Foreign Key Type": "",
54. "Referenced Table": "",
55. "Referenced Primary Key": "",
56. "Referenced Primary Key Type": ""
57. }
58. ]
59. },
60. {
61. "Table Name": "tb3",
62. "Table Columns": [
63. {
64. "Column Name": "col5",
65. "Data Type": "varchar",
66. "Data Distribution": ["minv","maxv"]
67. },
68. {
69. "Column Name": "col6",
70. "Data Type": "int",
71. "Data Distribution": [0,100]
72. }
73. ],
74. "Column Distribution" : [0.5, 0.5],
75. "Primary Key": {
76. "Name": "",
77. "Data Type": ""
78. },
79. "Foreign Key": [
80. {
81. "Foreign Key Name": "",
82. "Foreign Key Type": "",
83. "Referenced Table": "",
84. "Referenced Primary Key": "",
85. "Referenced Primary Key Type": ""
86. }
87. ]
88. },
89. {
90. "Table Name": "tb4",
91. "Table Columns": [
92. {
93. "Column Name": "col7",
94. "Data Type": "int",
95. "Data Distribution": [0,100]
96. },
97. {
98. "Column Name": "col5",
99. "Data Type": "varchar",
100. "Data Distribution": ["minv","maxv"]
101. }
102. ],
103. "Column Distribution" : [0.5, 0.5],
104. "Primary Key": {
105. "Name": "",
106. "Data Type": ""
107. },
108. "Foreign Key": [
109. {
110. "Foreign Key Name": "",
111. "Foreign Key Type": "",
112. "Referenced Table": "",
113. "Referenced Primary Key": "",
114. "Referenced Primary Key Type": ""
115. } ] }
116. ],
117. "Constraints": {
118. "size of workload" : 10000,
119. "read write ratio": 0.6,
120. "average table num": [0.5, 0.5],
121. "table-query access distribution": [0.1, 0.4, 0.3, 0.2],
122. "query comparison operator ratio" : [0.25, 0.25, 0.25, 0.25],
123. "table domain distribution" : [0.5, 0.5],
124. "query logic predicate num" : [0.4,0.4,0.2],
125. "average aggregation operator num" : [0.5,0.3,0.2],
126. "average query colomn num":[0.3,0.3,0.2,0.2],
127. "group by ratio if read SQL":[0.8,0.2],
128. "order by desc or asc if grouped":[0.85,0.15]
129. },
130. "Generation File":"workloadGen.wg"
131. }

# 附录五

## Benchbase中postgres测试TPC-C的配置文件

1. <?xml version="1.0"?>
2. <parameters>
3. <!-- Connection details -->
4. <type>POSTGRES</type>
5. <driver>org.postgresql.Driver</driver>
6. <url>jdbc:postgresql://localhost:5432/benchbase?sslmode=disable&amp;ApplicationName=tpcc&amp;reWriteBatchedInserts=true</url>
7. <username>postgres</username>
8. <password>1024</password>
9. <isolation>TRANSACTION\_SERIALIZABLE</isolation>
10. <batchsize>128</batchsize>
11. <!-- Scale factor is the number of warehouses in TPCC -->
12. <scalefactor>1</scalefactor>
13. <!-- The workload -->
14. <terminals>1</terminals>
15. <works>
16. <work>
17. <time>60</time>
18. <rate>10000</rate>
19. <weights>45,43,4,4,4</weights>
20. </work>
21. </works>
22. <!-- TPCC specific -->
23. <transactiontypes>
24. <transactiontype>
25. <name>NewOrder</name>
26. <!--<preExecutionWait>18000</preExecutionWait>-->
27. <!--<postExecutionWait>12000</postExecutionWait>-->
28. </transactiontype>
29. <transactiontype>
30. <name>Payment</name>
31. <!--<preExecutionWait>3000</preExecutionWait>-->
32. <!--<postExecutionWait>12000</postExecutionWait>-->
33. </transactiontype>
34. <transactiontype>
35. <name>OrderStatus</name>
36. <!--<preExecutionWait>2000</preExecutionWait>-->
37. <!--<postExecutionWait>10000</postExecutionWait>-->
38. </transactiontype>
39. <transactiontype>
40. <name>Delivery</name>
41. <!--<preExecutionWait>2000</preExecutionWait>-->
42. <!--<postExecutionWait>5000</postExecutionWait>-->
43. </transactiontype>
44. <transactiontype>
45. <name>StockLevel</name>
46. <!--<preExecutionWait>2000</preExecutionWait>-->
47. <!--<postExecutionWait>5000</postExecutionWait>-->
48. </transactiontype>
49. </transactiontypes>
50. </parameters>