数据驱动的模式规范化

Thorsten Papenbrock
德国波茨坦 14482 哈索·普拉特纳研究所（HPI）
托尔斯滕·帕彭布罗克@hpi.de

Felix Naumann
德国波茨坦 14482 哈索·普拉特纳研究所（HPI）
费利克斯·瑙曼@hpi.de

摘要

确保符合博伊斯 - 科德范式（BCNF）是消除数据集冗余和异常的最常用方法。将数据集规范化为 BCNF 可以将函数依赖关系（FD）强制纳入主键和外键，从而消除重复值并使数据约束明确化。尽管在理论上已得到充分研究，但将现有数据集的模式转换为 BCNF 仍是一项复杂的手动任务，尤其是因为函数依赖关系的数量庞大，而推导主键和外键属于 NP 难题。

在本文中，我们提出了一种名为 Normalize 的新颖规范化算法，该算法利用发现的函数依赖关系将关系数据集规范化为 BCNF。Normalize 完全由数据驱动运行，这意味着仅在可以观察到冗余的地方才予以消除，并且它是（半）自动化的，这意味着用户可以干预也可以不干预规范化过程。该算法引入了一种计算函数依赖关系集闭包的有效方法以及用于选择适当约束的新颖特性。我们的评估表明，Normalize 能够在几分钟内处理数百万个函数依赖关系，并且约束选择技术支持在规范化过程中构建有意义的关系。

1. 功能依赖关系

函数依赖（FD）是一种形如 X → A 的陈述，其中 X 是关系 R 中的一组属性，A 是关系 R 中的一个单个属性。我们说左侧（Lhs）X 函数决定右侧（Rhs）A。这意味着，每当关系 R 的一个实例 r 中的两个记录在所有 X 值上都相同，那么它们在 A 值上也必须相同[7]。更正式地说，FD X → A 在 r 中成立，当且仅当对于 r 中的所有 t1， t2：t1[X] = t2[X] 则 t1[A] = t2[A]。在接下来的内容中，我们只考虑非平凡的 FD，即 A 不属于 X 的 FD。

表 1 描绘了一个示例地址数据集，其中存在两个函数依赖关系：邮政编码→城市和邮政编码→市长。由于这两个函数依赖关系的左部相同，我们

表 1：示例地址数据集

首先	最后的	邮政编码	城市	市长
托马斯	米勒	14482	波茨坦	雅各布斯
萨拉	米勒	14482	波茨坦	雅各布斯
彼得	史密斯	60329	法兰克福	费尔德曼
茉莉	圆锥体	01069	德累斯顿	奥罗兹
迈克	圆锥体	14482	波茨坦	雅各布斯
托马斯	摩尔	60329	法兰克福	费尔德曼

可以将它们聚合为“邮政编码→城市，市长”这种表示形式。这种函数依赖关系的存在会在数据集中引入异常，因为“波茨坦”“法兰克福”“雅各布斯”和“费尔德曼”这些值被冗余存储，更新这些值可能会导致不一致。因此，例如，如果某位施密特先生当选为波茨坦的新市长，我们必须将所有出现的“雅各布斯”正确地更改为“施密特”。

通过将关系规范化为鲍伊-科德范式（BCNF）可以避免此类异常。关系模式 R 处于 BCNF 状态，当且仅当对于 R 中的所有函数依赖 X→A，其左部 X 为键或超键 [7]。由于在示例数据集中邮政编码既不是键也不是超键，因此此关系不符合 BCNF 条件。要将模式中的所有关系都转换为 BCNF，需要执行六个步骤，稍后将详细解释：（1）发现所有函数依赖，（2）扩展函数依赖，（3）从扩展后的函数依赖中推导出所有必要的键，（4）识别违反 BCNF 的函数依赖，（5）选择一个违反 BCNF 的函数依赖进行分解，（6）根据所选的违反 BCNF 的函数依赖拆分关系。步骤（3）至（5）重复进行，直到步骤（4）不再发现违反 BCNF 的函数依赖，此时生成的模式即符合 BCNF。我们找到了几种用于步骤（1）的函数依赖发现算法，例如 Tane [14] 和 HyFD [19]，但到目前为止，还没有算法能够高效且自动地解决步骤（2）至（6）。

对于示例数据集，FD 发现算法会在步骤（1）中找到 12 个有效的函数依赖关系。在步骤（2）中，这些函数依赖关系必须进行聚合和传递扩展，以便我们找到诸如 First，Last→Postcode,City，Mayor 和 Post-code→City，Mayor 等关系。在步骤（3）中，前一个函数依赖关系使我们能够推导出键 {First， Last}，因为这两个属性功能上确定了关系中的所有其他属性。然后，在步骤（4）中，确定第二个函数依赖关系违反了 BCNF 条件，因为其左部 Postcode 既不是键也不是超键。如果假设步骤（5）能够自动选择第二个函数依赖关系进行分解，那么在步骤（6）中，示例关系将分解为（First, Last, Postcode）和（City, Mayor），其中 {First， Last} 和 {Postcode} 是主键，Postcode 是外键约束。表 2 显示了此结果。再次检查违反函数依赖关系时，我们未发现任何违反情况，因此停止。

表 2：标准化示例地址数据集

首先	最后的	邮政编码
托马斯	米勒	14482
萨拉	米勒	14482
彼得	史密斯	60329
茉莉	圆锥体	01069
迈克	圆锥体	14482
托马斯	摩尔	60329

邮政编码	城市	市长
14482	波茨坦	雅各布斯
60329	法兰克福	费尔德曼
01069	德累斯顿	奥罗兹

规范化过程得到符合 BCNF 的结果。请注意，City（城市）和 Mayor（市长）中的冗余已被消除，数据集的总大小从 36 个值减少到了 27 个值。

由于近年来内存价格大幅下降，出于性能方面的考虑，出现了不进行数据集归一化的趋势。因此，如今有人认为归一化已过时。这种说法是错误的，忽略归一化存在以下危险[8]：

1. 规范化消除了冗余，从而降低了出错的可能性和内存消耗。虽然内存可能相对便宜，但数据错误可能会带来严重且代价高昂的后果，应不惜一切代价避免。

2. 规范化并不一定会降低查询性能；实际上，它甚至可能提高性能。有些查询在规范化之后可能需要额外的连接操作，但其他查询则可以更快地读取更小的关系表。此外，如果必须更改数据，还可以设置更精确的锁，从而提高对数据的并行访问速度。因此，规范化对性能的影响并非由规范化后的数据集决定，而是取决于使用该数据集的应用程序。

3. 规范化提高了对模式以及针对该模式的查询的理解：关系变得更小，更贴近其所描述的实体；其复杂性降低，从而更易于维护和扩展。此外，针对关系的查询也更容易编写，许多错误也更容易避免。例如，对具有冗余值的列进行聚合操作很难正确编写。

总之，规范化应当是默认选择，而反规范化则应是经过深思熟虑后的决定，即“只有在所有其他提高性能的策略都未能满足要求的情况下，我们才应退而求其次采用反规范化设计”，C. J. Date，第 88 页 [8]。

本工作的目标是将给定的关系实例规范化为博伊斯 - 科德范式。请注意，我们既不旨在恢复某个特定的模式，也不旨在使用业务逻辑设计新的模式。为解决规范化任务，我们提出了一种数据驱动的（半）自动规范化算法，该算法在消除所有与函数依赖相关的冗余的同时，仍能提供完整的信息可恢复性。所谓数据驱动，意味着在规范化过程中所使用的所有函数依赖都是直接从数据中提取的，并且所有分解建议都仅基于数据特征。换句话说，我们只考虑在给定关系实例中实际可观察到的冗余。

与最先进的模式驱动方法相比，数据驱动的规范化方法的优势在于，它能够利用数据来展示所有语法上有效的规范化选项，即数据中有证据支持的功能依赖关系，这样算法（或用户）只需决定选择哪条规范化路径，而无需寻找。确实，功能依赖关系的数量可能会变得很大，但我们表明，算法能够有效地提出语义上最合适的选项。此外，了解所有功能依赖关系能够使规范化算法更高效，而不是仅拥有功能依赖关系的子集。

研究挑战。与过去几十年中大量关于规范化的研究不同，我们不假定函数依赖关系（FD）是已知的，因为在实际应用中几乎从未如此。我们也不假定数据专家能够手动识别它们，因为这种搜索本质上就很困难，而且实际的函数依赖关系往往并不明显。例如，我们示例中的函数依赖关系“邮政编码→城市”，虽然通常被认为是正确的，但实际上存在例外情况，即两个城市可能使用相同的邮政编码；而函数依赖关系“大气层→环”，对于人类来说很难发现，但实际上在关于行星的各种数据集中都成立。因此，我们自动发现所有（最小的）函数依赖关系。这带来了新的挑战，因为我们现在要处理的是规模更大、往往存在错误但却是完整的函数依赖关系集。

在规范化过程中使用特定关系实例的所有函数依赖（FD）进一步带来了从这些函数依赖中选择合适的主键和外键的挑战（见步骤（5）），因为大多数函数依赖都是偶然的，即它们在语法上是正确的，但在语义上是错误的。这意味着当数据发生变化时，这些语义上无效的函数依赖可能会被违反，从而不再能作为约束条件发挥作用。因此，我们引入了能够自动识别（并选择）从大量函数依赖中可靠的约束条件的功能，因为这些函数依赖的数量通常多到人工无法逐一检查。

即使所有函数依赖在语义上都是正确的，选择合适的主键和外键仍然很困难。这里所做的决定决定了要执行哪些分解，因为分解选项通常是相互排斥的：例如，如果两个违反函数依赖的规则有重叠，那么一种分解可能会使另一种分解变得不可行。这种情况的发生是因为 BCNF 规范化不具有依赖关系保持性[12]。然而，在所有这些情况下，某些违反函数依赖的规则在语义上比其他规则更优，这就是为什么违反函数依赖的规则不仅要被筛选，还要根据这些质量特征进行排序。

除了引导规范化过程朝着正确的方向发展之外，另一个挑战在于规范化的计算复杂性。Beeri 和 Bernstein 已证明，对于给定的一组函数依赖和一个体现这些依赖的关系模式，判断该模式是否违反 BCNF 的问题在属性数量上是 NP 完全的 [3]。要测试这一点，我们需要检查每个函数依赖的左部是否为键或超键，即每个左部是否能确定所有其他属性。如果所有函数依赖都已完全传递扩展，即它们已传递闭包，则此过程很简单。因此，复杂性在于计算这些闭包（见步骤（2））。由于目前没有算法能够高效地解决闭包计算问题，我们为此子任务的模式规范化提出了新的技术。

总的来说，数据集中的函数依赖关系数量通常远远超出人工专家手动处理的能力[18]。因此，规范化算法必须能够自动处理如此庞大的输入。

贡献。我们提出了一种新颖的基于实例的模式规范化算法，称为 Normalize，它能够自动或在专家监督下对关系数据集进行规范化处理。将人类纳入循环中，使该算法能够将其分析优势与专家的领域知识相结合。通过 Normalize 和本文，我们做出了以下贡献：

a) 模式规范化。我们展示了整个模式规范化过程如何能通过一个算法来实现，这是之前的工作未曾做到的。我们详细讨论了该算法的每个组成部分。我们（半）自动方法的主要贡献在于通过专注于那些最有可能为真的函数依赖，逐步剔除语义上错误的函数依赖。

b) 闭包计算。我们提出了两种高效的闭包算法，一种用于一般的函数依赖结果集，另一种用于完整的结果集。它们的核心创新包括更集中的扩展过程、高效索引结构的使用以及并行化。这些算法不仅在规范化上下文中很有用，而且对于许多其他与函数依赖相关的任务也很有用，例如查询优化、数据清理或模式逆向工程。

c) 违规检测。我们提出了一种紧凑的数据结构，即前缀树，以高效检测违反 BCNF 的函数依赖。这是首次从算法层面改进此步骤的方法。我们还讨论了如何改变此步骤以发现违反其他范式（而非 BCNF）的函数依赖。

d) 约束选择。我们贡献了若干特征来评估候选主键和外键实际成为约束的概率。利用这些结果，在规范化过程中可以对候选进行排序、筛选，并选择作为约束。选择工作可以由专家完成，也可以由算法自动完成。由于之前所有关于模式规范化的工作都假定所有输入的函数依赖都是正确的，因此这是首次针对此前一直被忽视的问题提出解决方案。

e) 评估。我们在多个数据集上对我们的算法进行了评估，展示了在完整的、真实世界的函数依赖结果集上进行闭包计算的效率以及（半）自动模式规范化方案的可行性。

本文其余部分的结构安排如下：首先，在第 2 节中讨论相关工作。然后，在第 3 节中介绍模式规范化算法 Normalize。接下来的几节将更详细地阐述闭包计算（第 4 节）、键派生（第 5 节）以及违反检测（第 6 节）。第 7 节介绍键和外键候选者的评估技术。最后，在第 8 节中对规范化算法进行评估，并在第 9 节中总结。

2. 相关工作

自关系数据模型提出以来，关系数据的范式就得到了广泛的研究[6]。因此，提出了许多范式。在此，我们不对其做全面的介绍，感兴趣的读者可参考[10]。由于 Boyce-Codd 范式（BCNF）[7]能消除关系模式中的大部分冗余，所以它是最受欢迎的范式。这也是本文重点关注这一特定范式的原因。不过，我们提出的大多数技术同样可用于创建其他范式。我们的规范化算法的思路遵循了[12]中提出的 BCNF 分解算法以及许多数据库系统方面的教科书中的算法。该算法消除了与函数依赖相关的所有异常，同时仍能保证通过自然连接实现信息的完全恢复。

模式规范化，尤其是规范化为 BCNF 是研究得较为透彻的问题[3, 5, 16]。伯恩斯坦提出了一种基于函数依赖进行模式综合的完整流程[4]。特别是，他表明计算函数依赖集的闭包是规范化过程中的关键步骤。他还为我们的论文奠定了理论基础。但与大多数其他关于模式规范化的工作一样，伯恩斯坦将函数依赖及其语义有效性视为既定事实——这一假设几乎不适用，因为函数依赖通常隐藏在数据中，必须被发现。因此，现有的模式规范化工作大大低估了非规范化数据集中有效函数依赖的数量，并且忽略了从语法正确的函数依赖中筛选出语义上有意义的函数依赖这一任务。这些原因使得那些规范化方法在实际应用中难以施行。在本文中，我们提出了一种涵盖从函数依赖发现到约束选择再到最终关系分解的整个过程的规范化系统。我们在实际实验中展示了该方法的可行性。

关于模式规范化还有其他一些研究工作，例如迪德里希和米尔顿 [9] 的工作，他们认识到在函数依赖集上计算传递闭包是一项计算复杂度很高的任务，在面对现实世界的函数依赖集时几乎无法实现。作为解决方案，他们建议在计算闭包之前从函数依赖中移除所谓的冗余属性，这能显著降低计算成本。然而，如果所有函数依赖都是最小的（这正是我们规范化过程中的情况），那么就不存在冗余属性，因此所提出的剪枝策略也就毫无用处。

传统规范化方法与我们的算法之间的一个重要区别在于，我们从给定的关系实例中检索出所有最小函数依赖（FD），以利用它们进行闭包计算（语法步骤）和约束选择（语义步骤）。后者在以往的研究中很少受到关注。在[2]中，Andritsos 等人提出根据函数依赖属性集的熵对其进行排序，用于规范化：一个函数依赖消除的重复越多，其效果就越好。这种方法的问题在于，它仅根据有效性对函数依赖进行加权，而未考虑其语义相关性。熵的计算成本也很高，这就是我们使用不同特征的原因。实际上，我们采用了受[20]启发的技术，从包含依赖关系中提取外键。

模式规范化是模式设计和演进中的一个子任务。有许多数据库管理工具，例如 Navicat[1]、Toad[2] 和 MySQL Workbench[3]，支持这些整体任务。它们中的大多数将给定的模式转换为用户可以操作的 ER 图。然后，所有操作都会被转换回模式及其数据。这些工具在一定程度上能够支持规范化过程，但没有一个能够自动根据从数据中检索到的函数依赖关系提出规范化建议。

[1]https://www.navicat.com/

[2]http://www.toadworld.com/

[3]http://www.mysql.com/products/workbench/

在文献[3]中，作者提出了一种解决成员资格问题的高效算法，即测试给定的函数依赖是否在覆盖中的问题。该算法并未解决闭包计算问题，但作者在该算法中提出了一些改进措施，我们的改进闭包算法也采用了这些措施，例如仅测试右部缺失的属性。他们还提出了推导树作为函数依赖推导的模型，即利用阿姆斯特朗推理规则从已知的函数依赖集中推导出更多的函数依赖。由于他们并未给出该模型的算法，因此我们无法将其与我们的解决方案进行比较。

如上所述，从关系数据中发现函数依赖关系是模式规范化的一个先决条件。幸运的是，函数依赖关系的发现是一个研究充分的问题，我们找到了多种解决它的算法。在这项工作中，我们采用了 HyFD 算法，这是当时最高效的函数依赖关系发现算法[19]。该算法——与几乎所有函数依赖关系发现算法一样——在给定的关系数据集中发现所有最小且语法有效的函数依赖关系的完整集合。我们在闭包算法中利用了这些特性，即最小性和完整性。

3. 模式规范化

要将模式规范化为博伊斯 - 科德范式（BCNF），我们采用大多数数据库系统教科书（如[12]）中所展示的直接的 BCNF 分解算法。该算法生成的符合 BCNF 的模式始终是树形雪花模式，即外键结构是分层且无环的。因此，我们的规范化算法并非旨在（重新）构建任意非雪花模式。不过，它会从关系中消除所有与函数依赖相关的冗余。如果需要做出其他导致不同模式拓扑的模式设计决策，用户必须（并且可以！）交互式地选择不同于我们算法所能提出的分解方案。

在接下来的内容中，我们提出了一种规范化过程，该过程以任意关系实例作为输入，并返回其符合 BCNF 的模式。输入数据集可以包含一个或多个关系，除了数据集的模式之外，不需要其他元数据。在规范化过程中逐步变化的此模式为所有算法组件所熟知。我们将数据集的模式定义为其关系集，包括属性、表以及主键/外键约束。例如，表 2 中示例数据集的模式为 {R_1(First 姓 名， Last, Postcode)， R_2(Postcode 编 号， City， 市长)}。下划线表示主键，相同的属性名称表示外键。

图 1 给出了归一化算法的概览，我们将其称为“归一化”。与 [4] 或 [9] 中提出的其他归一化算法不同，归一化算法没有负责最小化函数依赖（FD）或去除多余 FD 的组件。这是因为我们所操作的 FD 集合并非任意的；由于 FD 发现步骤的存在，它仅包含最小的 FD，因此不存在多余的 FD。接下来我们将逐步介绍各个组件，并讨论整个归一化过程。

（1）FD 发现。给定一个关系数据集，第一个组件负责发现所有最小函数依赖关系。可以使用任何已知的 FD 发现算法，例如 Tane [14] 或 Dfd [1]，因为所有这些算法都能够发现完整的最小函数依赖关系集。

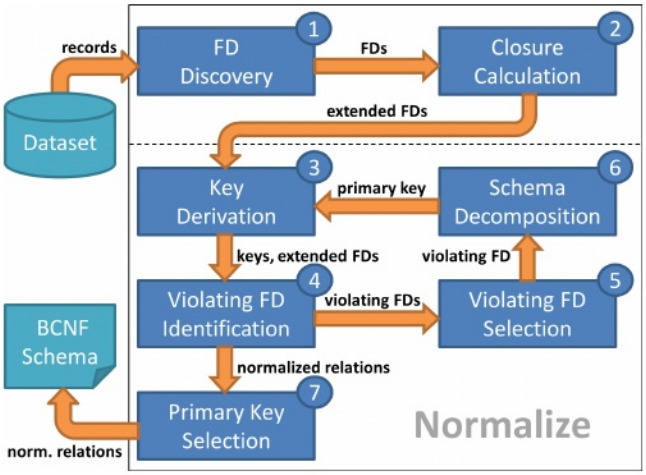

图 1：“标准化”及其组成部分。

关系数据集中的函数依赖。在此我们使用我们的 HyFD [19] 算法，因为它对于此任务是最高效的算法，并且它提供了特殊的剪枝能力，我们可以在后续的规范化过程中加以利用。简而言之，第一个组件读取数据，发现所有函数依赖，并将其发送给第二个组件。有关此发现步骤的更多详细信息，请参阅 [19]。

（2）闭包计算。第二个组件计算给定函数依赖（FDs）的闭包。闭包对于后续组件推断键和范式违规是必需的。形式上，给定函数依赖集 F 的属性集 X 的闭包 X_F^+ 定义为属性集 X 加上所有可以通过 F 和阿姆斯特朗传递公理 [9] 添加到 X 的附加属性 Y。例如，如果 X = {A， B} 且 F = {A →C, C →D}，则 X_F^+ = {A， B， C, D}。我们现在定义给定函数依赖集 F 的闭包 F^+ 为扩展函数依赖集：F 中每个函数依赖 X →Y 的右部 Y 扩展为 $X \cup Y = X_F^+$。换句话说，使用阿姆斯特朗传递公理对 F 中的每个函数依赖进行最大化。由于 Beeri 等人已经证明 [3]，相对于输入关系中的属性数量，这是一个 NP 难问题，我们将提出一种高效的函数依赖扩展算法，通过前缀树查找来发现传递依赖关系。该算法仅迭代函数依赖集一次，并能够并行化其工作。它利用了给定的函数依赖是极小且完备的这一事实（第 4 节）。

（3）密钥推导。密钥推导组件从扩展函数依赖中收集算法进行模式规范化所需的那些密钥。这样的密钥 X 是一组属性，对于这些属性，$X \rightarrow Y \in F^+$ 且 $X \cup Y = R_i$，其中 R_i 是关系 i 的所有属性。换句话说，如果 X 决定了所有其他属性，那么它就是其关系的密钥。一旦发现这些密钥，就会将其传递给下一个组件。我们从扩展函数依赖推导密钥的方法不会揭示模式中所有存在的密钥，但在第 5 节中我们证明，BCNF 规范化仅需要推导出的密钥。

（4）违反函数依赖标识。给定扩展的函数依赖和键集，违反检测组件会检查所有关系是否符合 BCNF 规范。回想一下，关系 R 符合 BCNF 规范，当且仅当对于所有函数依赖 X→A 都成立。

关系的左部 X 要么是主键，要么是超键。因此，规范化会检查每个函数依赖的左部是否包含在键的集合中；如果找不到这样的（子）集，则将该函数依赖报告为 BCNF 违反。请注意，在此组件中可以设置其他规范化标准以实现 3NF 或其他范式。如果发现了函数依赖违反，则将其报告给下一个组件；否则，该模式符合 BCNF 并可发送到主键选择。我们在第 6 节中提出了一种高效的技术来查找所有违反的函数依赖。

（5）违反函数依赖选择。如果某些关系尚未达到 BCNF，则会调用违反函数依赖选择组件，并向其传递一组违反函数依赖的规则。在这种情况下，该组件会对所有违反函数依赖的规则进行评分，以判断其是否适合作为外键约束。利用这些评分，算法会为每个非 BCNF 关系创建违反函数依赖规则的排名。用户从每个排名中选择最合适的违反函数依赖规则用于规范化；如果用户未参与，则算法会自动选择排名最高的函数依赖规则。需要注意的是，如果用户在场，也可以决定不选择任何函数依赖规则，这将结束当前关系的规范化过程。如果所有给出的函数依赖规则在语义上显然不正确，即这些规则只是偶然适用于给定的数据而没有实际意义，那么这种做法是合理的。这些语义上不正确的函数依赖规则在排名末尾会得到相对较低的分数。最终，通过在每一步中仅选择语义可靠的函数依赖规则，迭代过程会自动剔除大部分语义不正确的函数依赖规则。我们在第 7 节中将违反函数依赖选择与键选择一起讨论。

（6）模式分解。在已知违反的函数依赖的情况下，实际的模式分解是一项简单直接的任务：对于给定违反函数依赖 $X \rightarrow Y$ 的每个关系 R，将其拆分为两部分——不含冗余属性的部分 $R_1 = R \setminus Y$ 和包含函数依赖属性的部分 $R_2 = X \cup Y$。此时，X 自动成为 R_2 的新主键，并成为 R_1 的外键。利用这些新关系，算法返回到步骤（3），即键的选择，因为新的键可能出现在 R_2 中，即那些满足 $Z \rightarrow X$ 的键 Z。由于模式分解本身很简单，因此在本文中我们不再对这一部分进行详细说明。

（7）主键选择。主键选择是规范化过程中的最后一个组件。它确保每个符合 BCNF 的关系都有主键约束。由于分解组件在拆分关系时已分配主键和外键，所以大多数关系已经有了主键。只有那些在规范化过程开始时没有主键的关系才会由该组件处理。对于这些关系，算法以（半）自动的方式为其分配主键：对相应关系的所有键进行评分，以确定其作为主键的优劣；然后根据评分对键进行排序，要么由人工从排序结果中选择主键，要么算法自动选择排名最高的键作为关系的主键。第 7 节将更详细地描述键的评分和选择过程。

一旦计算出所有函数依赖的闭包，组件（3）至（6）便形成一个循环：此循环驱动规范化过程，直至组件（4）发现模式符合 BCNF 标准。总体而言，所提出的组件可分为两类：第一类包括组件（1）、（2）、（3）、（4）和（6），在语法层面进行操作；此类的结果定义明确，重点在于性能优化。第二类包括组件（5）和（7），在语义层面进行操作；这里的计算易于执行，但选择困难，结果的质量至关重要。

4. 闭合计算

阿姆斯特朗针对属性集 X、Y 和 Z 上的函数依赖制定了以下三条公理[3]:

1. 自反性：若 $Y \subseteq X$，则 $X \rightarrow Y$。
2. 扩充：若 $X \rightarrow Y$，则 $X \cup Z \rightarrow Y \cup Z$。
3. 传递性：若 $X \rightarrow Y$ 且 $Y \rightarrow Z$，则 $X \rightarrow Z$。

对于模式规范化，我们给定一组函数依赖 F，需要找到一个覆盖 F^+，使 F 中每个函数依赖的右侧达到最大。函数依赖的最大化对于识别键以及正确分解关系非常重要。例如，在我们的示例中，我们可能给定邮政编码→城市和城市→市长。若要正确分解并包含外键邮政编码，则需要邮政编码→城市，市长；否则，由于城市和市长这两个属性会出现在不同的关系中，我们就会丢失城市→市长这一函数依赖。因此，我们应用阿姆斯特朗传递公理对 F 进行计算，以求出其覆盖 F^+。

闭包 F^+ 利用阿姆斯特朗自反性和传递性公理来扩展每个函数依赖。无需使用扩充规则，因为该规则会生成新的非最小函数依赖，而不是扩展现有的函数依赖。分解步骤要求函数依赖的左部必须是最小的，即从 X 中移除任何属性都会使 $X \rightarrow Y$ 无效，因为分解后 X 应该成为最小键。

自反性公理将所有左部属性添加到函数依赖的右部。为减少内存消耗，我们仅隐式地进行此扩展：我们假定左部属性始终也属于函数依赖的右部，而无需在该侧显式存储它们。因此，对于属性集 W、X、Y 和 Z，我们按如下方式应用传递性公理：如果 $W \rightarrow X$，$Y \rightarrow Z$ 且 $Y \subseteq W \cup X$，则 $W \rightarrow Z$。例如，如果给定函数依赖 First，Last→Mayor，我们可以将函数依赖 First，Postcode→Last 扩展为 First，Postcode→Last,Mayor，因为 {First，Last} ⊆ {First, Postcode} ∪{Last}。

接下来，我们将讨论三种从 F 计算 F^+ 的算法：一种是朴素算法，一种是针对任意函数依赖集的改进算法，还有一种是针对最小函数依赖集的优化算法。虽然第二种算法可能在其他上下文中（例如查询优化或数据清理）对闭包计算有用，但我们建议在我们的规范化系统中使用第三种算法。这三种算法都将 F 转换为 F^+ 后存储在变量 fds 中。

4.1 朴素闭包算法

在文献 [9] 中已介绍过的朴素闭包算法如算法 1 所示。对于 fds 中的每个函数依赖（第 3 行），该算法会遍历所有其他函数依赖（第 4 行），并测试这些函数依赖是否扩展了当前的函数依赖（第 5 行）。如果可以扩展，则更新当前的函数依赖（第 6 行）。这些更新可能会使已测试过的函数依赖能够进一步更新。因此，朴素算法会不断遍历函数依赖，直到整个遍历过程不再添加任何新的扩展（第 8 行）。

Algorithm 1: Naive Closure Calculation

```
Data: fds
Result: fds
1 do
2     somethingChanged ← false;
3     foreach fd ∈ fds do
4         foreach otherFd ∈ fds do
5             if otherFd.lhs ⊆ fd.lhs ∪ fd.rhs then
6                 fd.rhs ← fd.rhs ∪ otherFd.rhs;
7                 somethingChanged ← true;
8 while somethingChanged ;
9 return fds;
```

4.2 改进的闭包算法

改进朴素闭包算法的方法有多种，其中一些已在文献[9]和[3]中以类似形式提出。现在我们给出一种改进的闭包算法，它解决了以下三个问题：首先，在扩展一个特定的函数依赖时，算法不应检查所有其他函数依赖，而只需检查那些可能与缺失的右部属性相关联的函数依赖；其次，在寻找缺失的右部属性时，算法不应检查所有可能提供该属性的其他函数依赖，而只需检查与当前函数依赖具有子集关系的那些函数依赖，即那些与扩展相关的函数依赖；第三，变更循环不应遍历整个函数依赖集，因为某些函数依赖必须比其他函数依赖更频繁地进行扩展，从而导致许多扩展测试执行得过于频繁。

算法 2 展示了我们的改进版本。首先，我们移除了对所有其他函数依赖的嵌套循环，并用索引查找来替代。我们提议的索引结构是一组前缀树，也称为字典树。每个字典树存储所有具有相同、特定于该字典树的右部属性的函数依赖左部。为每个右部属性都设置一个索引，使得算法只需检查那些能够提供当前函数依赖实际缺失的右部属性链接的其他函数依赖（第 8 行）。

在算法开始扩展给定的函数依赖（FD）之前，会构建左侧尝试集（第 1 至 4 行）。然后，每次索引查找不必遍历所有引用缺失的右侧属性的 FD；而是会在相应的前缀树中执行子集搜索，因为算法专门寻找其左侧包含在当前 FD 的右侧属性中的 FD（第 9 行）。这种子集搜索比遍历所有可能的扩展候选要高效得多，并且已经在 [11] 中被提议用于 FD 的泛化查找。

作为第三个优化措施，我们建议将变更循环移至 FD 循环内部（第 6 行）。这样一来，在后续迭代中需要多次进行传递扩展的单个 FD 就不会导致对所有 FD 都进行相同次数的迭代，因为大多数 FD 已经完全扩展。

4.3 优化的闭包算法

算法 2 对所有函数依赖集都适用，但在这些函数依赖集包含所有最小函数依赖的假设下，我们可以进一步优化该算法。算法 3 展示了针对包含所有最小函数依赖的完整函数依赖集的更高效版本。

与算法 2 类似，优化后的闭包算法同样利用 Lhs 尝试树来高效地扩展函数依赖，但它无需变更循环，因此对于每个函数依赖所缺失的 Rhs 属性仅迭代一次。该算法还会检查

Algorithm 2: Improved Closure Calculation

```
Data: fds
Result: fds
1  array lhsTries size | schema.attributes | as trie;
2  foreach fd ∈ fds do
3      foreach rhsAttr ∈ fd.rhs do
4          lhsTries[rhsAttr].insert(fd.lhs);
5  foreach fd ∈ fds do
6      do
7          somethingChanged ← false;
8          foreach attr ∉ fd.rhs ∪ fd.lhs do
9              if fd.lhs ∪ fd.rhs ⊇ lhsTries[attr] then
10                 fd.rhs ← fd.rhs ∪ attr;
11                 somethingChanged ← true;
12     while somethingChanged ;
13 return fds;
```

对于子集而言，仅使用当前函数依赖（FD）的左部属性（LHS 属性），而非当前 FD 的所有属性（第 7 行）。这两种优化之所以可行，是因为函数依赖集是完备且最小的，因此对于任何有效的扩展属性，我们总能找到一个子集函数依赖。以下引理对此进行了正式说明：

引理 1. 设 F 是极小函数依赖的完备集。若 $X \to Y \in F$ 且 $X \to A$（其中 $A \in Y$）成立，则必然存在 $X^0 \subset X$ 使得 $X^0 \to A \in F$。

证明。若 $X \to A$ 且 $X \to A \notin F$，则 $X \to A$ 不是最小函数依赖，必然存在一个最小函数依赖 $X^0 \to A$，其中 $X^0 \subset X$。若 X□ → A ∈

如果存在这样的函数依赖 X→Y，那么 F 就不是极小函数依赖集的完整集合，这与前提 F 是完整的相矛盾。□

算法 3 能够正确运行的前提是需要所有最小函数依赖，这一事实的不利之处在于，完整的函数依赖集通常比已经缩减为有意义的函数依赖集要大得多。相反，将函数依赖集缩减为有意义的函数依赖集是一项困难且特定于用例的任务，如果已知函数依赖的闭包，这项任务会更准确。因此，我们在选择函数依赖之前先进行闭包计算，并接受由此带来的处理时间和内存消耗的增加。

处理时间的增加几乎不是问题，因为算法 3 相对于算法 2 在相同规模输入上的性能提升非常显著，所以仍能轻松处理更大规模的函数依赖集。我们在第 8 节中对此进行了展示。

Algorithm 3: Optimized Closure Calculation

```
Data: fds
Result: fds
1 array lhsTries size | schema.attributes | as trie;
2 foreach fd ∈ fds do
3     foreach rhsAttr ∈ fd.rhs do
4         lhsTries[rhsAttr].insert(fd.lhs);
5 foreach fd ∈ fds do
6     foreach attr ∉ fd.rhs ∪ fd.lhs do
7         if fd.lhs ⊇ lhsTries[attr] then
8             fd.rhs ← fd.rhs ∪ attr;
9 return fds;
```

另一方面，如果最小函数依赖集太大，以至于无法全部存入内存，甚至无法全部存入磁盘，那么内存消耗增加就会成为一个问题。此时我们需要对函数依赖进行修剪，但修剪哪些函数依赖才能保证算法 3 对剩余的函数依赖仍能计算出正确的闭包呢？要完全扩展一个函数依赖 $X \to Y$，算法需要所有左侧子集为 X^0 的函数依赖 $X^0 \to Z$（其中 $X^0 \subset X$）都可用。因此，如果我们修剪所有左侧大于 $|X|$ 的超集函数依赖，那么对于 $X \to Y$ 及其所有子集函数依赖 $X^0 \to Z$ 计算出的闭包仍然是正确的。一般而言，我们可以定义一个最大左侧大小，并修剪所有左侧大小超过该值的函数依赖，同时仍能使用算法 3 计算剩余函数依赖的完整且正确的闭包。这种修剪方式非常适合我们的规范化用例，因为正如我们在第 7 节中所论述的，左侧较短的函数依赖在语义上更适合作为主键和外键约束。Normalize 可以免费实现最大左侧大小的修剪，因为它已经在我们用于发现函数依赖的 HyFD 算法中实现了。

这三种闭包算法都可以通过将 FD 循环（分别在第 3 行、第 2 行和第 5 行）分配给不同的工作线程来轻松实现并行化。这是可行的，因为每个工作线程只更改自己的 FD，而对其他 FD 所做的更改可以被该工作线程看到，但并非必须看到。

考虑到这三种算法在输入函数依赖（FD）数量方面的复杂性，朴素算法的时间复杂度为 $O(|fds|^3)$，改进算法为 $O(|fds|^2)$，而优化算法为 $O(|fds|)$。但由于函数依赖的数量可能会随着属性数量呈指数级增长，因此这三种算法在属性数量方面都是 NP 完全问题。我们在第 8 节通过实验对这些算法进行了比较。

5. 密钥派生

在规范化过程中，键非常重要，因为它们由于其唯一性而不包含任何冗余，因此不会导致数据异常。键基本上表明了不需要分解的规范化模式元素，即分解它们不会消除给定关系实例中的任何冗余。在本节中，我们首先讨论如何从扩展函数依赖中推导出键。然后，我们证明推导出的键集足以实现 BCNF 模式规范化。

从扩展函数依赖中推导键。根据定义，键是任何属性或属性组合，其值能唯一确定关系中的所有其他记录[6]。换句话说，键 X 的属性能函数确定关系 R 中的所有其他属性 Y。因此，给定扩展函数依赖，通过检查每个函数依赖 $X \to Y$ 是否满足 $X \cup Y = R$，即可轻松找到键。

然而，我们直接从扩展函数依赖中推导出的键集并不一定包含给定关系的所有最小键。例如，考虑关系 Professor（name, department, salary）、Teaches（name, label） 和 Class（label, room, date），其中 Teaches 是 Professor 和 Class 之间 n:m 关系的连接表。当我们通过计算 R = Professor ./ Teaches ./ Class 对此模式进行反规范化时，我们得到 R（name, label, department, salary, room, date），其主键为 {name， label}。此键不能直接从最小函数依赖中推导出来，因为 name，label→A 对任何 A 而言都不是最小函数依赖 $\in R$; 两个最小函数依赖是 name→department，salary 和 label→room，date。

跳过缺失的键。发现缺失的键是一项代价高昂的任务，尤其是考虑到非规范化数据集中的函数依赖数量可能非常庞大。然而，BCNF 规范化只需要那些我们可以直接从扩展函数依赖中推导出来的键。我们基本上可以忽略缺失的键，因为该算法仅使用函数依赖左部的子集作为键来检查范式违规情况（见第 6 节），而所有这样的键都可以直接推导出来。以下引理对此进行了更正式的说明：

引理 2. 若 X^0 是一个键，且 $X \to Y \in F^+$ 是一个函数依赖，其中 $X^0 \subseteq X$，则 X^0 可直接从 F^+ 中推导得出。

证明。设 X0 是关系 R 的一个键，且 $X \to Y \in F^+$ 是一个函数依赖，其中 $X^0 \subseteq X$。要直接从 F^+ 中推导出键 X^0，我们必须证明存在一个函数依赖 $X^0 \to Z \in F^+$，其中 $Z = R \setminus X^0$。

X 必须是某个函数依赖 $X \to Y_0$ 中的最小左部，其中 $Y^0 \subseteq Y$，因为 $X \to Y \in F^+$ 且 F 是所有最小函数依赖的集合。现在考虑前提 $X^0 \subseteq X$：如果 $X^0 \subset X$，那么 $X \to Y 6\in F^+$，因为 X 是主键，因此它确定的任何属性 A 都不会多于 X^0 所确定的属性。所以，$X = X^0$ 必须成立。此时，我们有 $X \to Y^0 \in F^+$ 且 $X = X^0$。因此，$X^0 \to Y^0 \in F^+$ 也必然成立，这也表明 $Y^0 = Y = Z$，因为 X^0 是主键。 □

在“规范化”中的关键派生组件实际上仅通过检查每个函数依赖 $X \to Y$ 是否满足 $X \cup Y = R$ 来发现与规范化过程相关的那些键。然而，在规范化过程的末尾，主键选择组件必须为那些在任何先前的分解操作中未获得主键的关系发现所有键。为此任务，我们使用了 Heise 等人[13]提出的 DUCC 算法，该算法专门用于键的发现。键的发现是一个 NP 完全问题，但由于此时已规范化的关系比未规范化的初始关系小得多，所以在这个算法阶段它是一个快速操作。

6. 违规检测

给定扩展的函数依赖和键，检测 BCNF 违反情况是直接的：每个左部既不是键也不是超键的函数依赖都必须被归类为违反情况。算法 4 展示了如何再次使用前缀树进行子集搜索来高效地完成此操作。

首先，违反检测算法将所有给定的键插入到字典树中（第 1 至 3 行）。然后，它遍历每个函数依赖（FD），对于每个 FD，检查其左部是否包含空值 $\perp$。这样的 FD 无需考虑分解，因为左部在新的分离关系中会成为主键约束，而 SQL 禁止在键约束中出现空值。请注意，有关可能/确定的键约束的研究允许键中存在 $\perp$ 值[15]，但目前我们仍遵循标准。如果左部不包含空值，则算法在键字典树中查询 FD 左部的子集（第 8 行）。如果找到子集，则该 FD 不违反 BCNF，算法继续处理下一个 FD；否则，该 FD 违反 BCNF。

为保留现有约束，如果违反函数依赖的右侧存在主键，则从该右侧移除所有主键属性（第 11 行）。如果不从函数依赖的右侧移除主键属性，分解步骤可能会将主键拆分。一些关键属性会

Algorithm 4: Violation Detection

```
Data: fds, keys
Result: violatingFds
1  keyTrie ← new trie;
2  foreach key ∈ keys do
3      keyTrie.insert(key);
4  violatingFds ← ∅;
5  foreach fd ∈ fds do
6      if ⊥ ∈ fd.lhs then
7          continue;
8      if fd.lhs ⊇ keyTrie then
9          continue;
10     if currentSchema.primaryKey ≠ null then
11         fd.rhs ← fd.rhs − currentSchema.primaryKey;
12     if ∃ fk ∈ currentSchema.foreignKeys:
13       (fk ∩ fd.rhs ≠ ∅) ∧ (fk ⊈ fd.lhs ∪ fd.rhs) then
14         continue;
15     violatingFds ← violatingFds ∪ fd;
16 return violatingFds;
```

然后将其移入另一个关系中，从而破坏主键约束以及可能引用此主键的外键约束。由于当前模式中也可能包含外键约束，因此我们测试在用于分解时，违反的函数依赖是否保留了所有此类约束：每个外键 fk 必须在两个新关系中的一个中保持完整，否则我们不会使用违反的函数依赖进行规范化（第 12 行）。算法最后将每个保留约束的违反函数依赖添加到违反函数依赖的结果集中（第 15 行）。在第 7 节中，我们提出了一种从中选择一个进行分解的方法。

当使用违反的函数依赖 X → Y 对关系 R 进行分解时，我们得到两个新的关系，即 R_1(R\Y$R_1$ = (X ∪ Y) 和 R_2(XR_1 = (R - X)。由于属性的这种拆分，并非所有先前的函数依赖在 R_1 和 R_1 中都成立。显然，R_1 中的函数依赖恰好是那些满足 V ∪ W ⊆ R_1 且 V → $W^{0\in F+}$（其中 W ⊆ W 0）的函数依赖 V → W，因为对于 V → W 而言，R 的记录在 R_1 中保持不变，只是丢失了一些与所有 V → W 无关的属性。对于 R_1 也有同样的观察结果，尽管记录的数量有所减少：

引理 3. 由函数依赖 X → Y 的分解所产生的关系 R_2(X ∪Y) 保留了所有满足 V ∪W ⊆R_2 且在 R 中有效的函数依赖 V →W 。

证明。（1）关系 R 中任何有效的 V →W 在 R_2:R 中仍然有效。假设 V →W 在 R 中有效但在 R_2 中无效。那么 R_2 中必然至少存在两条违反 V →W 的记录。由于分解仅移除了 V ∪W 中的记录，且 V ∪W ⊆R_2 ⊆R，这些违反的记录在 R 中也必然存在。但这样的记录在 R 中不可能存在，因为 V →W 在 R 中有效；因此，该函数依赖在 R_2 中也必然有效。

（2）关系 R 中有效的 V →W 不可能在 R 中无效：假设 V → W 在 R 中有效但在 R 中无效。那么 R 中必然包含至少两条违反 V →W 的记录。由于这两条记录在 V ∪W 值上不完全相同且 V ∪W ⊆R，分解不会移除它们，它们也存在于 R 中。所以 V →W 在 R 中也必然无效。因此，不存在在 R 中有效但在 R 中无效的函数依赖。□

假设我们不是要保证 BCNF，而是要保证 3NF，3NF 的要求比 BCNF 稍微宽松一些：与 BCNF 不同，3NF 并不会消除所有与函数依赖相关的冗余，但它能保持依赖关系。因此，任何分解都不能拆分除违反 3NF 的函数依赖之外的其他函数依赖 [4]。要计算 3NF 而不是 BCNF，我们可以在算法 4 的结果中，额外移除所有相互排斥的违反 3NF 的函数依赖组，即任何会拆分其他函数依赖左部的函数依赖。要计算比 BCNF 更严格的范式，我们需要检测其他类型的依赖关系。例如，构造 4NF 需要检测所有多值依赖（MVD），因此需要一个能发现 MVD 的算法。那么，规范化算法的工作方式将与上述相同。

7. 约束选择

在模式规范化过程中，我们需要定义主键和外键约束。从语法上看，所有键都是正确的，所有违反函数依赖的键都能形成正确的外键，但从语义上看，主键和违反函数依赖的键的选择是有区别的。从语义角度判断键和函数依赖的相关性对于算法来说是一项艰巨的任务——在很多情况下对于人类来说也是如此——但在接下来的内容中，我们将定义一些质量特征，用于自动评估键和函数依赖是否为“良好”的约束，即不仅在给定实例上有效，而且对于其模式也是正确的约束。

“规范化”中的两个选择组件分别利用这些特征对主键和外键候选进行评分。然后，它们根据得分对候选进行排序。最合理的候选会出现在列表的顶部，而可能是偶然出现的候选则会出现在列表的末尾。默认情况下，“规范化”会使用排名最高的候选并继续进行；如果用户参与其中，她可以选择约束条件或停止该过程。当然，候选列表可能会变得过大，无法进行全面的手动检查，但（1）用户始终只需选择一个元素，即她不需要将列表中的所有元素都分类为正确或错误；（2）在算法的每一步中，候选列表都会变短，因为许多选项会被隐式地排除；（3）在排序的选项列表中找到分割候选的问题比在没有任何排序的情况下找到分割要容易得多，就像没有使用我们的方法时的情况。

7.1 主键选择

如果一个关系没有主键，我们必须从该关系的键集中指定一个。为了找到语义上最佳的键，规范化会使用以下特征对所有键 X 进行评分：

长度得分： $\frac{1}{|X|}$

语义正确的键通常比随机键（在属性数量 |X| 方面）要短，因为模式设计者倾向于使用短键：短键可以更高效地建立索引，并且更容易理解。

（2）价值得分： $\frac{1}{\max(1, \max(X))}$

主键中的值通常较短，因为它们用于标识记录，通常不包含太多业务逻辑。大多数关系型数据库管理系统（RDBMS）也限制主键属性值的最大长度，因为主键默认会被索引，而值过长的索引更难管理。因此，对于值较长的键，我们会将其降级。

使用函数 max(X) 来获取属性（组合）X 中最长的值，若超过 8 个字符；对于多个属性，max(X) 会将它们的值连接起来。

（3）位置得分：$\frac{1}{2}\left(\frac{|X\text{ 左侧}| + 1}{1 + |X\text{ 之间}| + 1}\right)$

在考虑关系中属性的顺序时，关键属性通常位于左侧，且它们之间没有非关键属性。这是合乎直觉的，因为人们倾向于将键放在前面，并将逻辑上相关的属性放在一起。位置得分利用了这一点，根据关键属性 X 左侧的非关键属性数量 left(X) 以及 X 之间的非关键属性数量 between(X) 为其分配递减的得分值。

我们提出的排名公式仅反映我们的直觉。特征列表很可能也不完整，但在我们的实验中，所提出的特征对于关键评分产生了良好的效果。对于最终的关键评分，我们只需计算各个评分的平均值。那么，具有一个属性、最大值长度为 8 个字符且在关系中处于位置 1 的完美键，其键评分为 1；不太完美的键则得分较低。

在评分之后，Normalize 会根据分数对键进行排序，并让用户在排名靠前的键中选择一个主键；如果不需要（或无法）进行用户交互，算法会自动选择排名最高的键。

7.2 违反FD选择

在规范化过程中，我们需要为模式分解选择一些违反的函数依赖（FD）。由于所选的 FD 在分解后会成为外键约束，因此违反 FD 的选择问题类似于外键选择问题[20]，后者对包含依赖（IND）进行评分以确定其是否适合作为外键。然而，两者的观点不同：从 IND 中选择外键旨在识别现有表之间语义正确的链接；而从 FD 中选择外键则是为了形成具有适当键且无冗余的表。

请记住，选择语义上正确的违反函数依赖关系至关重要，因为某些分解是相互排斥的。如果可能的话，用户还应舍弃那些仅在给定关系实例中偶然成立的违反函数依赖关系。否则，规范化可能会过度分解属性集——尤其是那些稀疏填充的属性——将其拆分为单独的关系。

接下来，我们将讨论我们用于将违反外键约束 X → Y 的特征评分定为良好外键约束的特性：

长度得分： $\frac{1}{2}\left(\frac{|X|}{1} + \frac{1}{|Y|}\right)\cdot\frac{1}{|R|-2}$

由于违反函数依赖的左部 X 在分解后会成为左部属性的主键，所以其长度应尽可能短。相反，右部 Y 应尽可能长，这样我们才能创建较大的新关系：较长的右部不仅提高了函数依赖在语义上正确的置信度，还使分解更有效。由于在关系 R 中右部最多只能有 |R| - 2 个属性（一个属性必须是 X，另一个属性不能依赖于 X，这样 X 在 R 中才不是键），所以我们用这个因子来衡量右部的长度。

（2）价值得分： $\frac{1}{\max(1, |\max(X)|-7}$

违反函数依赖的值分数与主键 X 的值分数相同，因为分解后 X 成为了主键。

（3）位置得分：$\frac{1}{2}\left(\frac{1}{1 + |X\text{ 之间的差值}|} + \frac{1}{1 + |Y\text{ 之间的差值}|}\right)$

语义正确的函数依赖的属性由于具有共同的上下文，其位置很可能彼此靠近。我们期望这一规律同时适用于函数依赖的左部和右部。然而，左部属性和右部属性之间的间隔只是一个非常弱的指标，我们不予考虑。基于此，我们根据左部属性之间以及右部属性之间的属性数量反比例地对违反函数依赖的属性进行加权。

（4）重复得分：$\frac{1}{2}\left(2 - \frac{|X\text{ 中的唯一值数量}| + |\text{uniques}(Y)|}{|X\text{ 的值域}| \cdot |Y\text{ 的值域}|}\right)$

如果违反的函数依赖（FD）的左侧 X 和右侧 Y 都可能包含大量重复值，从而存在大量冗余，那么这种分解非常适合进行规范化，因为这样可以消除许多冗余值。对于大多数评分特征而言，左侧值的高重复得分降低了函数依赖偶然成立的概率，因为只有函数依赖左侧的重复值才能使其失效，而左侧 X 中存在大量重复值但没有违反函数依赖的情况是其语义正确的良好指标。在评分时，我们通过 |uniques()| 来估计 X 和 Y 中的唯一值数量；由于精确计算这个数量计算成本很高，所以我们为每个属性创建一个布隆过滤器，并使用其误报概率来高效地估计唯一值的数量。

我们将最终的违反函数依赖（FD）得分计算为各个得分的平均值。通过这种方式，最有希望的违反 FD 是指单个左部属性几乎决定了整个关系，且具有较短且较少不同值的情况。与键评分类似，所提出的特征反映了我们的直觉和观察；它们可能并非最优或完整，但对于一个困难的选择问题，它们能产生合理的结果：在我们的实验中，排名靠前的违反 FD 通常表明了语义上最佳的分解点。

在选择一个违反的函数依赖作为外键约束之后，原则上我们可以决定从该函数依赖的右侧移除个别属性。其中一个原因可能是这些属性也出现在另一个违反的函数依赖的右侧，并且可以在后续的分解中使用。因此，当用户引导规范化过程时，我们会展示所有在其他违反的函数依赖右侧也存在的属性。然后用户可以决定移除这些属性。如果没有用户参与，则不移除任何属性。

8. 评估

在本节中，我们将评估我们的归一化算法 Normalize 的效率和效果。首先，我们将介绍实验设置。然后，我们将评估 Normalize 的性能，特别是其闭包计算组件的性能。最后，我们将评估归一化输出的质量。

8.1 实验装置

已使用 Metanome 数据剖析框架（www.metanome.de）实现了规范化，该框架为不同类型的剖析算法定义了标准接口[17]。特别是，Metanome 提供了 HyFD 函数依赖发现算法的实现。诸如输入解析、结果格式化和性能测量等常见任务由框架进行标准化，并与算法本身解耦。

表 3：数据集、其特征以及处理时间

名字	尺寸	属性	记录	FDs	FD 密钥	FD 光盘。	Closureimpr	Closureopt	密钥派生	小提琴。身份。
马	25.5 千字节	27	368	128,727	40	4157 毫秒	1765 毫秒	486 毫秒	40 毫秒	246 毫秒
Plista	588.8 千字节	63	1000	178,152	1	9847 毫秒	6652 毫秒	857 毫秒	49 毫秒	55 毫秒
汞合金	61.6 千字节	87	50	450,020	2,737	3462 毫秒	745 毫秒	333 毫秒	7 毫秒	25 毫秒
航班	582.2 千字节	109	1000	982,631	25,260	20921 毫秒	132085 毫秒	1662 毫秒	77 毫秒	93 毫秒
MusicBrainz	1.2 千兆字节	106	1,000,000	12,358,548	0	2132 分钟	215.5 分钟	1 分 40 秒	331 毫秒	26 毫秒
TPC-H	6.7 千兆字节	52	6,001,215	13,262,106	347,805	3651 分钟	3 分 48 秒	0.5 分钟	163 毫秒	4093 毫秒

硬件。我们所有的实验均在一台戴尔 PowerEdge R620 服务器上进行，该服务器配备两颗英特尔至强 E5-2650 2.00 GHz 处理器和 128GB DDR3 内存。服务器运行 CentOS 6.7 操作系统，并使用 OpenJDK 64 位 1.8.0 71 版本作为 Java 环境。

数据集。我们主要使用合成的 TPC-H[4] 数据集（规模因子为 1），该数据集模拟了通用业务数据，以及 MusicBrainz[5] 数据集，这是一个由用户维护的关于音乐和艺术家的百科全书。为了评估 Normalize 的有效性，我们将这两个数据集的所有关系通过连接合并为一个通用关系，从而将规范化结果与原始数据集进行比较。对于 MusicBrainz 数据集，由于其表的数量庞大，我们不得不将连接限制在 11 个选定的核心表上。此外，由于关联表在进行完整连接时会产生大量记录，我们还对去规范化的 MusicBrainz 数据集的记录数量进行了限制。

对于效率评估，我们使用了四个额外的数据集，即 Horse、Plista、Amalgam1 和 Flight。我们已在我们的网页上提供了这些数据集以及更详细的描述[6]。在我们的评估中，每个数据集都包含一个关系，其特征如表 3 所示；通常，Normalize 的输入可以包含多个关系。

8.2 效率分析

表 3 列出了六个具有不同属性的数据集。这些数据集中最小函数依赖的数量在 12.8 万到 1300 万之间，因此数量庞大，无法手动挑选出有意义的依赖关系。FD-Keys 列统计了所有可直接从函数依赖中推导出的键。其数量不取决于函数依赖的数量，而是取决于数据的结构：Amalgam1 和 TPC-H 具有雪花型模式，而例如 MusicBrainz 则在其模式中具有更复杂的链接结构。

我们对每个数据集都执行了“规范化”操作，并测量了各个组件的执行时间：（1）函数依赖发现，（2）闭包计算，（3）主键推导，以及（4）违反函数依赖的识别。前两个组件已实现并行化，从而能够充分利用我们评估机器的全部 32 个核心。然而，在两个较大的数据集上，完整函数依赖集的必要发现仍分别需要 36 小时和 61 小时。

首先，我们注意到关键派生和违反 FD 的识别步骤比 FD 发现和闭包计算步骤快得多；它们通常在不到一秒的时间内完成。这一点很重要，因为在规范化过程中这两个组件会被多次执行，而用户可能同时在与系统进行交互。在表 3 中，我们仅展示了这些组件首次调用的执行时间；后续调用的处理速度会更快，因为其输入规模会不断缩小。确定违反 FD 所需的时间主要取决于 FD 键的数量，因为在键的字典树中搜索 Lhs 一般化是开销最大的操作。这解释了 TPC-H 数据集执行时间长达 4 秒的原因。

对于闭包计算，表 3 展示了改进（impr）算法和优化（opt）算法的执行时间。对于 Amalgam1 数据集，原始算法已耗时 13 秒（而改进和优化算法分别耗时不到 1 秒），对于 Horse 数据集耗时 23 分钟（改进和优化算法分别耗时不到 2 秒和不到 1 秒），对于 Plista 数据集耗时 41 分钟（改进和优化算法分别耗时不到 7 秒和不到 1 秒）。这些运行时间比改进和优化算法版本差太多，因此我们停止了对其的测试。优化后的闭包算法在性能上比改进版本高出 2 倍（Amalgam1 数据集）到 159 倍（MusicBrainz 数据集），因为它能够利用给定函数依赖集的完整性。算法必须执行的右部扩展越多，这种优势就越明显。例如，Amalgam1 函数依赖的平均右部大小从 32 增加到 56，而 MusicBrainz 函数依赖的平均右部大小从 3 增加到 40。对于 TPC-H，平均右部大小从 10 增加到 23。总体而言，优化后的闭包计算运行时间与函数依赖发现时间相比是可以接受的。因此，在闭包计算之前没有必要对函数依赖进行过滤。

由于闭包计算不仅对于规范化很重要，对于许多其他用例也很重要，因此图 2 更详细地分析了此步骤的可扩展性。这些图表展示了改进算法和优化算法在输入函数依赖数量不断增加时的执行时间。实验从 1200 万条 MusicBrainz 函数依赖中随机选取这些输入函数依赖；属性数量保持在 106 个不变。我们再次省略了朴素算法，因为它比其他两种方法慢好几个数量级。

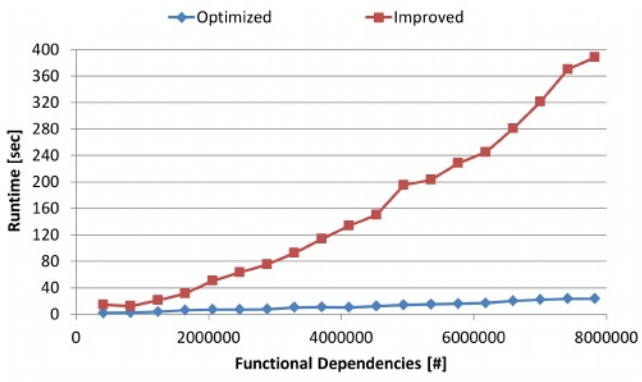

图 2：用于闭包计算的输入文件描述符数量的缩放。

4 http://tpc.org/tpch

5https://musicbrainz.org

6https://hpi.de/naumann/projects/repeatability

图 3：对 TPC-H 进行规范化后的关系。

图 2 中的两种运行时性能似乎都几乎与 FD 的数量呈线性关系，这是因为每个单独 FD 的扩展成本较低，得益于高效的索引查找。然而，随着索引中 FD 数量的增加，索引查找的成本也会增加（如果同时增加属性的数量，查找次数也会增多）。由于改进后的算法比优化版本（即更改后的循环）更频繁地执行索引查找，并且查找的键更大（即 Lhs 和 Rhs），所以优化版本更快，并且在 FD 数量增加时性能扩展性更好：在本次实验中，其速度提高了 4 到 16 倍。

8.3 效果分析

为了进行公平的效果分析，我们自动执行规范化操作，即无需人工干预。在人工监督下，可以生成比下面展示的更好的（但也可能更差的）模式。对于以下实验，我们重点关注 TPC-H 和 MusicBrainz，因为我们之前对这些数据集进行了反规范化处理，因此可以将它们的原始模式用作规范化结果的黄金标准。

图 3 展示了经过 BCNF 归约的 TPC-H 数据集。颜色编码表示不同属性的原始关系。首先我们注意到，归约几乎完美地恢复了原始模式：在归约结果中可以识别出所有原始关系。自动选择的约束条件，即主键和外键，相对于原始模式都是正确的，这之所以可能是因为原始模式是雪花型的。

然而，我们也注意到自动规范化模式存在两个有趣的缺陷：首先，规范化将 LINEITEM 关系分解得有些过头了；从语法上看，结果是正确的，完全符合 BCNF 标准，但从语义上看，那些只有一项依赖且包含三个以上外键属性的拆分并不合理。其次，原本属于 ORDERS 关系的 shippriority 属性被放入了 REGION 关系中。从语法上看，这是一个不错的决定，因为地区也决定了发货优先级，将该属性放入此关系中比放入 ORDERS 关系中能消除更多的冗余值。

图 4 展示了经过 BCNF 规范化的 MusicBrainz 数据集。尽管 MusicBrainz 原本没有雪花模式，但 Normalize 仍能几乎重构出所有原始关系。只有 ARTIST CREDIT NAME 未被重构，其属性现在位于语义上相关的表中。

图 4：对 MusicBrainz 进行规范化处理后的关系。

相关艺术家关系。由于 MusicBrainz 原本并非雪花型结构，规范化生成了一个新的顶级关系，用于表示艺术家、地点、发行厂牌和曲目之间的所有多对多关系。这个顶级关系可以比作事实表。

大多数错误出现在“艺术家信用”关系上，这是最先提出的拆分。这种拆分从其他关系中拿走了一些属性，因为这些属性包含的值不多，将其分配给“艺术家信用”关系在语法上是合理的。如果有人工专家参与，可能会避免这种情况，因为规范化工具会向用户报告这些属性还依赖于其他违反函数依赖关系的左侧属性。不过，总体而言，考虑到没有人工参与创建，这种规范化结果还是相当令人满意的。

我们还在其他各种数据集上对 Normalize 进行了测试，结果类似：如果数据集之前经过了反规范化处理，我们可以在提议的模式中找到原始表；如果存在稀疏列，这些列通常会被移到较小的关系中；如果无人工干预，某些分解会变得详细。所有结果都符合 BCNF 规范且语义清晰易懂。

9. 结论

我们提出了 Normalize 算法，这是一种实例驱动的（半）自动模式规范化算法。该算法表明，任何规模的功能依赖关系分析结果都能高效地用于模式规范化的特定任务。我们还介绍了引导 BCNF 分解算法的技术，以生成语义良好且能适应数据变化的规范化结果。

我们的实现版本可在 http://hpi.de/naumann/projects/repeatability 公开获取。目前它基于控制台，仅提供基本的用户交互。未来的工作应着重于强调用户在循环中的作用，例如通过使用规范化关系及其连接的图形预览。我们还建议研究其他用于主键和外键选择的功能，这可能会带来更好的结果。另一个开放的研究问题是规范化过程应如何处理动态数据和数据中的错误。

10.参考文献

[1] Z. Abedjan, P. Schulze, and F. Naumann. DFD: Efficient functional dependency discovery. In Proceedings of the International Conference on Information and Knowledge Management (CIKM), pages 949−958, 2014.

[2] P. Andritsos, R. J. Miller, and P. Tsaparas. Information-theoretic tools for mining database structure from large data sets. In Proceedings of the International Conference on Management of Data (SIGMOD), pages 731−742, 2004.

[3] C. Beeri and P. A. Bernstein. Computational problems related to the design of normal form relational schemas. ACM Transactions on Database Systems (TODS), 4(1): 30−59, 1979.

[4] P. A. Bernstein. Synthesizing third normal form relations from functional dependencies. ACM Transactions on Database Systems (TODS), 1(4):277−298, 1976.

[5] S. Ceri and G. Gottlob. Normalization of relations and prolog. Communications of the ACM, 29(6):524−544, 1986.

[6] E. F. Codd. Derivability, redundancy and consistency of relations stored in large data banks. IBM Research Report, San Jose, California, RJ599, 1969.

[7] E. F. Codd. Further normalization of the data base relational model. IBM Research Report, San Jose, California, RJ909, 1971.

[8] C. J. Date. Database Design & Relational Theory. O' Reilly Media, 2012.

[9] J. Diederich and J. Milton. New methods and fast algorithms for database normalization. ACM Transactions on Database Systems (TODS), 13(3):339−365, 1988.

[10] R. Fagin. Normal forms and relational database operators. In Proceedings of the International Conference on Management of Data (SIGMOD), pages 153−160, 1979.

[11] P. A. Flach and I. Savnik. Database dependency discovery: a machine learning approach. AI Communications, 12(3):139−160, 1999.

[12] H. Garcia-Molina, J. D. Ullman, and J. Widom. Database Systems: The Complete Book. Prentice Hall Press, Upper Saddle River, NJ, USA, 2 edition, 2008.

[13] A. Heise, J.-A. Quian´e-Ruiz, Z. Abedjan, A. Jentzsch, and F. Naumann. Scalable discovery of unique column combinations. Proceedings of the VLDB Endowment, 7 (4):301−312, 2013.

[14] Y. Huhtala, J. K¨ arkk¨ ainen, P. Porkka, and H. Toivonen. TANE: An efficient algorithm for discovering functional and approximate dependencies. The Computer Journal, 42(2):100−111, 1999.

[15] H. K¨ ohler, S. Link, and X. Zhou. Possible and certain SQL key. Proceedings of the VLDB Endowment, 8(11): 1118−1129, 2015.

[16] H. Mannila and K.-J. R¨ aih¨ a. Dependency inference. In Proceedings of the International Conference on Very Large Databases (VLDB), pages 155−158, 1987.

[17] T. Papenbrock, T. Bergmann, M. Finke, J. Zwiener, and F. Naumann. Data profiling with Metanome. Proceedings of the VLDB Endowment, 8(12):1860−1871, 2015.

[18] T. Papenbrock, J. Ehrlich, J. Marten, T. Neubert, J.-P. Rudolph, M. Sch¨ onberg, J. Zwiener, and F. Naumann. Functional dependency discovery: An experimental evaluation of seven algorithms. Proceedings of the VLDB Endowment, 8(10):1082−1093, 2015.

[19] T. Papenbrock and F. Naumann. A hybrid approach to functional dependency discovery. In Proceedings of the International Conference on Management of Data (SIGMOD), 2016.

[20] A. Rostin, O. Albrecht, J. Bauckmann, F. Naumann, and U. Leser. A machine learning approach to foreign key discovery. In Proceedings of the ACM Workshop on the Web and Databases (WebDB), 2009.