硕士学位论文

（学术学位论文）

Spark 平台上的数据时效性管理的关键技术研究

RESEARCH ON KEY TECHNOLOGIES OF DATA CURRENCY MANAGEMENT ON DISTRIBUTED PLATFORM

黄光辉

哈尔滨工业大学

2022 年 6 月

国内图书分类号：TP392　　学校代码：10213
国际图书分类号：004　　密级：公开

硕士学位论文

Spark 平台上的数据时效性管理的关键技术研究

硕 士 研 究 生：黄光辉

导　　师：李建中教授

申 请 学 位：工学硕士

学　　科：计算机科学与技术

所 在 单 位：计算机科学与技术学院

答 辩 日 期：2022 年 6 月

授予学位单位：哈尔滨工业大学

Classified Index: TP392
U.D.C: 004

Dissertation for the Master Degree in Engineering

RESEARCH ON KEY TECHNOLOGIES OF DATA CURRENCY MANAGEMENT ON SPARK PLATFORM

Candidate： Huang Guanghui
Supervisor： Prof. Li Jianzhong
Academic Degree Applied for： Master of Engineering
Speciality： Computer Science and Technology
Affiliation： School of Computer Sci
Date of Defence： June, 2022
Degree-Conferring-Institution： Harbin Institute of Technology

摘 要

当前人类社会正处于大数据时代，数据的数量正以前所未有的速度急剧增长，然而数据的质量也同样在快速下降，数据质量（数据可用性）问题已经成为数据管理领域不可忽视的热点问题。作为数据可用性问题 5 个维度的一员，数据时效性问题已经成为不可忽视的关键问题。由于现实生活中的事物会随着时间的流逝而产生变化，因此作为现实事物客观反映的数据在数据库中会过时失效。在实际应用场景中，往往缺少完整可靠的时间戳，当前数据时效性领域的研究主要包括时效性判定和时效性修复两个重要的问题。本文根据已有的相关研究成果，改进并提出了一系列大数据时效性管理的关键技术。

在时效性判定问题上，目前已有的时效性判定算法根据实体属性上的时效规则以及实体的冗余记录构造时效图，并基于时效图判定数据的时效性。本文针对时效图构造算法时间复杂度过高，在海量数据的应用场景中表现不佳的问题，基于时效规则提出了状态图的概念，状态图相比时效图具有更好的空间复杂度，非常适合在分布式场景中传输，用状态图替代时效图的构造算法可以大大优化时效性判定算法在分布式场景中的执行效率。另外，本文还针对用户对数据库不同维度的查询请求，给出了不同维度的时效性度量方法。

在时效性修复问题上，本文深入研究了一种特殊的数据时效性关联错误。给出了实体冗余记录是否具有数据时效性关联错误的判定原理，并给出整个数据库是否有数据时效性关联错误的判定算法。然而仅仅判定数据库是否有数据时效性关联错误是不够的，为了数据时效性判定算法的正确计算，还需要对实体的冗余记录进行时效性关联错误修复。本文将给出一种时效性关联错误的检测方法，并基于检测方法给出一种高效的时效性关联错误修复算法。

关键词：大数据；数据质量；数据时效性；时效性判定；时效性修复

Abstract

This is an era of big data., and the amount of data is increasing rapidly at an unprecedented rate. However, the quality of data is also rapidly declining. The problem of data quality (data availability) has become a hot issue in the field of data management that cannot be ignored. As one of the five dimensions of data availability, data currency has become a key issue that cannot be ignored. Since things in real life change over time, data that is an objective reflection of real things will become outdated in the database. In practical application scenarios, complete and reliable timestamps are often lacking. The current research in the field of data currency mainly includes two important issues: currency determination and currency repair. Based on the existing research results, this paper improves and proposes a series of key technologies for the currency management of big data.

On the problem of currency determination, the existing currency determination algorithm determines the currency according to the currency rules of entity attributes and the redundant records of the entity, and calculates the currency of entity according to the currency graph. Aiming at the problem that the currency graph construction algorithm has large time complexity and poor performance in the application scenario of massive data, the concept of state graph based on currency rules is proposed, and the construction algorithm of currency graph with state graph is proposed. Compared with the currency graph, the state graph has better space complexity and is very suitable for transmission in distributed scenarios, thus greatly optimizing the execution efficiency of the currency determination algorithm in distributed scenarios. In addition, this paper also provides different dimensions of currency measurement methods for users' query requests to different dimensions of the database.

On the problem of currency repair, this paper delves into a special kind of data currency error. The principle of judging whether the entity redundant record has data currency error is given, and the judgment algorithm of whether the entire database has data currency error is given. However, it is not enough to only determine whether the database has data currency errors. In order to correctly calculate the data currency determination algorithm, it is also necessary to perform currency error repair on

redundant records of entities. In this paper, a detection method of currency errors will be given, and an efficient currency error repair algorithm will be given based on the detection method.

Keywords: big data, data quality, data currency, currency determination, currency repair

目 录

第 1 章　绪　论

1.1　课题来源

课题来源自国家自然科学基金重大项目《基于超算的大数据分析处理基础算法与编程支撑环境》(U1811461)。

1.2　课题背景及研究的目的和意义

当前人类社会正处于大数据时代，据统计调查，每过一分钟谷歌公司就有 200 万次的来自用户的搜索请求，每过一分钟脸谱 App 就有 180 万次用户点赞，每过一秒 Instagram 就有 3600 张用户照片被上传。数据的数量正以前所未有的速度急剧增长，然而数据的质量也同样在快速下降，劣质数据正无时无刻地通过网络在不同数据源之间传输。目前已经有许多相关的统计调查表明，每年在全球范围内，数据质量（数据可用性）问题都会产生无法估量的严重后果。因此数据质量（数据可用性）问题已经成为数据质量研究工作中不可忽视的热点问题。

一般说来，数据可用性问题可以从 5 个维度讨论：

（1） 时效性：数据集合中的记录保证在当前时刻是有效的，而不是陈旧过时的。例如，某用户在 2022 年结婚，而该用户的婚姻状态在数据库中存储的记录仍为“单身”；

（2） 一致性：数据集合中的记录保证其语义正确且不存在矛盾冲突。例如，记录（邮编=“410000”，城市=“哈尔滨”）就存在不一致错误，因为 410000 是长沙的邮编而非哈尔滨的邮编；

（3） 完整性：数据集合中的某一实体保证其所有记录是完整无缺失的。例如，图书馆数据库存储的某读者的借书记录被丢失后就会导致数据不完整，造成图书管理的漏洞；

（4） 精确性：数据集合中描述某一实体的记录能够保证数据是准确的。例如，数据库中记录的长沙全市总面积为 1 万平方公里，但实际约为 1.18195 万平方公里；

（5） 实体同一性：在各个数据源中记录同一个实体的数据值必须是相同的。例如，某些会员制商家要求会员卡的申请人和会员卡的使用人必须是

同一个人，从而获取更多的利益。

作为数据可用性问题 5 个维度的一员，数据时效性问题已经成为不可忽视的关键问题。由于现实生活中的事物会随着时间的流逝而产生变化，因此作为现实事物客观反映的数据在数据库中会过时失效。随着时间的推移，真实数据库中的数据质量会迅速下降。研究表明，客户档案中 2%的记录在一个月内就会过时。也就是说，在一个包含 500,000 个客户记录的数据库中，有 10,000 条记录每个月可能会过时，每年 12 万份记录，两年内大约 50%所有的记录可能都过时了。有鉴于此，我们经常会发现多重价值同一实体的一部分驻留在一个数据库中，这曾经是正确的，也就是说，它们是正确的某一时刻实体的真实值。然而，它们中的大多数已经是过时的和不准确的。举个日常生活中的例子，当一个人搬到一个新地址时，a 银行可能会保留她原来的地址，更糟糕的是在相当长的一段时间内，信用卡账单可能仍然会被以这个旧地址发送。过期数据是影响数据质量的核心问题之一。众所周知，脏数据每年给美国企业造成 6000 亿美元的损失，而且不新鲜数据损失占损失的很大一部分。

同时，数据可用性的其他维度如数据不一致、不精确、不完整等，都有可能是由于数据的过时失效导致的。例如，如果学校没有更新在校学生的家庭住址信息，就会造成学生的家庭住址和邮编的不一致；随着新生的入学和毕业生的离开，学校学生学籍管理数据库中在校生的人数会因为毕业生数据的过时而变得不再精确；期末考试后学生的成绩信息没有录入学生信息管理系统将会导致数据库不完整。上述例子说明，数据的时效性对于数据可用性至关重要。

在数据质量（数据可用性）领域，目前还没有成体系地针对数据时效性进行研究，数据时效性问题仍然存在许多挑战性问题。首先，现实应用场景中的许多数据库由于成本及数据迁移等各种原因，都缺少完整可用的时间戳。这导致用户很难判定大多数数据库中的数据记录在给定时刻的时效性。其次，针对不同应用场景或不同的用户查询请求，时效性有不同的度量标准。在许多场景下可能无法判定数据库在给定时刻的绝对时效性，那么针对用户查询请求的相对时效性就至关重要了。第三，需要给出数据时效性修复方法修复数据库中的那些过时数据，然而，尽管当前数据可用性领域有许多针对数据修复的工作，但还缺乏针对数据时效性的数据修复方法。

1.3 国内外研究现状

随着数据质量下降的问题越来越受到关注，数据质量（数据可用性）问题已

经成为数据管理领域不可忽视的热点问题。文献[1][2][3][4][5][6][7]综合概述了当前数据可用性管理领域中主要的一些研究工作成果。下面针对本文研究的数据时效性，详细介绍数据时效性管理相关的国内外研究现状。

1.3.1 时效性判定研究现状

在数据时效性管理领域，针对数据时效性的判定问题是确保数据质量中数据时效性的关键。无论是数据时效性问题的发现还是数据时效性问题的修复，都首先需要对数据时效性加以判定。在用户使用数据时，往往也首先要求判定数据的时效性。所谓数据的时效性，通常用“过时”和“非过时”这两种布尔类型表达，但也可以用[0,1]的小数来表达数据记录过时的可能性。当前主要有两种判定数据时效性的方法：基于时间戳的时效性判定和基于规则的时效性判定。

当数据库中存在完整可用的时间戳来表示数据的生成时间时，基于时间戳的时效性判定能够根据数据库中的时间戳来判定当前数据记录值是否过时失效。该方法能够判定数据在给定的时间点的时效性，然而需要的辅助信息过多，真实应用中往往没有完整的时间戳，因此该判定方法的适用范围很有限。

文献[8][9][10][11][12][13]基于数据年龄 age()的概念从一些新的角度定义了时效性，它表示数据本次使用的时间点减去上一次更新的时间点。其中文献[8]和文献[11]为数据给定了一个确定的保质期 ShelfLife()，当数据年龄 age()大于保质期 ShelfLife()时，则视为数据过时失效。对于给定的记录值 value，文献[8]用数据过时失效的概率 $P(ShelfLife(value) - age(value) > 0)$作为记录值 value 的时效性，而文献[11]要求数据满足约束 $ShelfLife(value) - age(value) > 0$，并将数据时效性直接定义为 age(value)。文献[9][10]定义了时效性衰减函数 decline(value)，用来刻画记录值 value 的时效性随时间流逝的减弱程度，并用 $\exp(-decline(value) \times age(value))$定义 value 的时效性。文献[12]与文献[11]类似，直接将数据年龄 age()作为数据时效性的度量。基于模糊逻辑，文献[13]通过提出一种推断时效性衰减函数 decline(value)来判定数据的时效性。

数据库有完整可用的时间戳来表示数据的存储时间是基于时间戳的时效性判定方法的必要条件，当时间戳不完整或是完全不存在时，将无法计算文献[8]和文献[11]中提出的数据时效性度量数据年龄 age()，同样也无法使用基于年龄提出的衰减函数 decline(value)、模糊度量推导方法。而且在真实的应用场景中，仅通过数据年龄 age()作为判定数据时效性的唯一度量是不完全可靠的，数据年龄大的数据记录时效性不一定差。因此，基于时间戳的时效性判定有其明显的缺

陷，一旦数据库中时间戳不完整或是错误，则基于时间戳的判定方法都没有意义，也就是说，其必要条件过于严苛。尽管基于时间戳的时效性判定有着诸多缺点，但在数据库中有足够完整可用的时间戳的前提条件下，基于时间戳的时效性判定有着判定相对简单、判定正确性有保障、判定算法通俗易懂等优势，并且判定算法计算复杂性较低，执行效率和可扩展性较高。

由于实际应用场景中存在大量没有完整可用时间戳的情况，需要新的判定方法，因此基于规则的时效性判定被提出。文献[14][15]首次研究了基于规则判定数据时效性的相关理论，给出了不依赖于时间戳的数据时效性度量模型：时序关系（Currency Order）、时效约束（Currency Constraint）、不同数据源间的拷贝函数（Copy Function）。两个元组 t_i, t_j的在属性 A 上时序关系 $t_i <_A t_j$ 表示元组t_i在属性 A 上的值比元组t_j在属性 A 上的值旧。时效约束定义为一阶逻辑语句 $\forall t_i, \ldots, t_j: R(\bigwedge_{j\in[1,k]} t_1[EID] = t_j[EID] \wedge \phi \rightarrow t_u <_A t_v)$，$\phi$表示一个连接谓词，通过$\phi$推导属性值的时效顺序，它能够描述数据的语义信息。该约束表示如果一组元组$t_i, \ldots, t_j$描述的实体相同，且满足特定条件φ，那么就有结论$t_u <_A t_v$。当数据来自于多个数据源时，拷贝函数$R_1[\vec{A}] \leftarrow R_2[\vec{B}]$表示数据源$R_1$的$\vec{A}$向量表示的属性拷贝自数据源$R_2$的$\vec{B}$，描述了不同数据源间的依赖关系。上述工作仍有许多问题没有解决：(1) 仅仅推测实体的属性值是否是最新的，没有给出数据时效性的方法；(2) 没有给出任何针对数据库的时效性判定算法；(3) 模型依赖于时效规则，但没有讨论时效规则的挖掘方法。

文献[16][17]沿用了文献[14]中定义的时序关系和时效约束，提出了一系列算法。针对最新值查询和历史序列查询，文献[17]提出了查询相关时效性和用户相关时效性，其依据时效约束为数据集合中描述同一实体的每一条记录的同一属性上的属性值构建一个时效图，并通过对时效图进行拓扑排序，从而确定同一实体的所有记录在某一属性上的历史序列和最新值。文献[16]在同一实体的记录集合能够推出时效性目标元组集合的情况下，将时效约束和主数据集匹配的命题逻辑统称为规则，给出了推导时效准确属性值的一般算法和 top-k 启发式算法。文献[18]在文献[17]的基础上定义了时效性修复规则，通过统计方法对数据时效性进行了判定，文献[19]在文献[17]的基础上定义了不确定规则，给出了不确定规则的学习算法，从而定量的判定了数据的时效性。

文献[15][20][21]结合了数据时效性来讨论数据可用性的其他维度，文献[15][20]通过结合数据时效性的相关技术来解决数据不一致问题，在时序关系与时效约束规则的基础上，引入条件函数依赖（CFDs）解决数据不一致问题。文献[22][23][24][25]给出了生成条件函数依赖的算法。文献[15][20][10,11]还提出了一种由用户参与解决数据

冲突的数据冲突解决系统，用户补充数据记录间缺少的时序关系，必要时反馈给用户一些合理的建议。文献[21]利用条件函数依赖 CFDs 和匹配规则[26][27][28]将实体同一性、一致性及完整性相结合提高数据可用性。

基于规则的时效性判定克服了基于时间戳的时效性判定对时间戳的依赖，利用时效规则、拷贝函数、条件函数依赖 CFDs 和匹配规则解决了数据时效性判定问题。但是其算法复杂度相比基于时间戳的判定方法要复杂的多，面对当前应用场景中普遍的大数据量，其判定时间长。现有的研究工作仍然未能解决时效性判定算法的效率问题。

1.3.2 时效性修复研究现状

当前数据管理领域针对数据修复问题的研究工作主要集中在如何消除数据中的不一致和冗余情况，对于时效性相关的数据修复问题则考虑得较少。在文献[29][20]的研究工作中基于文献[14]中提出的模型，给出了某一实体某个属性上最新属性值的推断算法，并用新的属性值代替旧的属性值，从而提高了数据时效性。在时效规则足够多的情况下，该算法能够有效地找到实体的最新值，但存在如下不足之处：首先，算法本质上将时效性的判定作为辅助，解决的是数据不一致的修复问题。其次，其假定数据在相应属性上的最新值一定存在于当前数据集合中，但实际场景中，该假定并不总能满足。

虽然目前针对时效性错误的修复研究工作较少，但是一些工作提出了通用的数据修复框架，一定程度上能够解决数据时效性错误的修复问题。主要分为基于规则的数据修复和基于统计的数据修复。

在基于规则的数据修复工作中，许多工作借助了数据库中的依赖作为修复规则，并基于此修复数据。函数依赖（Fanction Dependency）和包含依赖（Include Dependency）是最常见的两种数据库依赖，FD 形式为 $X \rightarrow Y$，其表示数据库中两条记录在属性 X 上的值相同时，其在属性 Y 上的值也必须相同。IND 形式为 $R_1[X] \subseteq R_2[Y]$，其表示关系R_1中属性 X 的值域必须是关系R_2中属性 Y 值域的子集。基于函数依赖和包含依赖的修复技术，通常的思路是：筛选出数据库中满足规则左部的所有记录，遍历这些记录判断其是否满足规则的右部。需要修复所有不满足规则的记录。文献[30]使用函数依赖 FD 对错误数据进行修复，对于给定的函数依赖 FD 集合和数据库，给出了对数据库修改最少的修复算法。文献[31]为了合理的估计数据修复的代价，提出了一个代价模型，并基于这个代价模型使修复算法的代价最小化。文献[32]通过相关信度（Relative Trust）限制数据集合中候选

修复数据的数量，因为相关信度可以用来判断是修复数据还是约束。文献[33]通过抽样获得一个基数最小化空间（Cardinality-set-minimal），它是一个包含数据和依赖的修复空间，该空间不仅考虑了集合最小化（Set-minimal)，还能综合了基数最小化（Cardinality-minimal）。同时允许绝对正确的依赖集合，并定义为强约束（Hard Constraint），强约束在修复过程中不会被修改。文献[34]对基于函数依赖 FD 修复方法的 Repair-checking 问题进行了研究，Repair-checking 问题判断给定的两个修复哪一个更优。对于给定的函数依赖 FD 和数据库，该工作指出可以在多项式时间解决 Globally-optimal Repair-checking 问题的情况有哪些。对于上述基于规则的修复技术，用于数据修复的规则挖掘也是众多工作关注的重点：文献[35][36][23][37][38][39][40][41]针对函数依赖 FD 规则挖掘问题，提出了一系列挖掘算法，如 FASTFDS 等。文献[42]对比了各类函数依赖 FD 规则挖掘算法的性能。文献[43][44][45]提出了一系列类似的数据修复技术，在细节方面他们各有不同，但主要思想却不尽相同。主要思想都是：为了提高数据质量，使数据库记录最大化的满足所有条件依赖，采用通过对整条元组或元组属性值进行修改来达到目的。其中，文献[43]研究了两个被证明是难解的问题：寻找满足规则约束的修复算法修复给定的数据库、对数据库进行增量地修复。该工作给出针对上述问题的启发式算法，并进一步通过统计方法，证明其算法的输出能够满足预设的精确度。基于主数据，文献[44]为了对确定正确的修复操作进行描述，给出了确定域（Certain Region）的定义。对于确定域和主数据已知的情况，正确的修复策略可以由编辑规则（Editing Rules）确定。上述文献研究了有关编辑规则的基础问题的计算复杂度，并对最小确定域的判定给出了算法。文献[21]研究了 CFD 和 MD 之间相互作用的问题，并讨论了数据一致性修复和实体识别要如何同时进行，给出了一个基于完整性约束、主数据和匹配规则的通用框架，来对数据进行修复和实体识别。进一步的，对于相关的基础问题，文献证明了其难解性。规则一致性在 CFD 和 MD 同时使用时被证明是 NP-完全，蕴含问题在 CFD 和 MD 同时使用时被证明是 coNP-完全的；数据修复问题和数据修复的可终止性被证明是 NP-完全的和 PSPACE-完全的。文献还提出了相应的数据修复算法。基于否定约束（Denial Constraint），文献[46]在一个框架下将各种类型的约束统一起来，现有的所有约束种类均可以否定约束的形式表达，且数据中的不一致可以建模为冲突超图（Conflflict Hypergraph），并对多种异构约束综合起来修复数据。文献对于数据修复给出了基于该框架的算法，并给出了新的数据修复语义 Repair Context——用表达式对不同属性值关系和相应修复要求进行描述的集合。基于文献[48]的 Cleaning Egd（Equality Generating Dependencies）规则，文献[47]研究了如何在同

一个框架下执行数据集成中的模式匹配（SchemaMapping）操作和数据修复操作。其对现有的模式匹配和数据修复框架进行了扩展，通过设计相应的匹配语义和数据修复，给出了基于 Chase 的修复匹配算法。针对 CFD 和 CIND 不能表达属性取值范围的缺点，文献[49]通过扩展 CFD 和 CIND 得到 CFDp 和 CINDp。CFDp 和 CINDp 的蕴含问题和可满足性问题，上述文献也给出了根据扩展后的规则可以自动地产生一组 SQL 查询的算法，将产生的 SQL 查询在数据库 D 上执行，可以返回所有违反对应的 CFDp 和 CINDp 的元组。文献[50]将一类基础修复规则的扩展规则定义为 Fixing 规则。前述规则不能指明具体是哪个值错了，只能用于发现错误存在于哪条或者哪几条元组，而 Fixing 规则不但能检测出错误，还能够识别错误位置和采取的修复策略。上述文献研究了 Fixing 规则的基础理论问题，并给出了基于 Fixing 规则的修复算法。文献[51]更进一步对基础的修复规则扩展得到 Sherlock 规则。通过 Sherlock 规则对比当前数据表和一个修复好的数据表（Reference Table），可以同时发现当前数据表的错误、识别错误位置以及找出无需修复的那些值。

尽管目前基于规则有一系列通用的数据修复算法，但是针对数据时效性修复的研究工作并不多，现有的能够用来修复数据时效性的数据依赖或规则，只有由文献[14]提出的确定的时效约束。但确定的时效约束要求时效约束均为确定成立的，这对于许多场景是不足以修复完全的，因为确定的时效约束使得其表达能力受到影响。因此，针对数据时效性错误的修复研究仍然充满挑战。另外，目前基于规则的一系列通用数据修复算法对错误数据进行修复主要是通过数据依赖或规则引入相关的领域知识，再利用这些领域知识对数据记录进行错误检测及修复。然而，真实场景的数据记录中存在某些基于规则的修复方法难以检测并修复的错误。比方说，同一家公司有多个办公地址，同一个人可能在多个公司机构存在职位，对于这些较为复杂的情况，简单的基于关联规则的修复算法难以有效修复错误数据。针对上述较为复杂的修复场景，需要新的修复策略，从而基于统计的数据修复方法被提出[40][52]。数据修复工具 ERACER 和 SCAREd 是上述工作的主要成果，它们首先针对训练数据集，采用统计学习方法训练出属性值的概率分布，然后利用这些概率分布对错误数据进行的时效性修复。通过条件概率描述的数据间关联关系巧妙解决了基于规则无法解决的修复难题，。然而，基于统计的修复方法也存在着缺陷：首先，因为需要从样本数据集中训练得到条件概率 P(Y|X)，当属性 X 和 Y 的值域较大时，条件概率的训练代价很高且非常耗时，而且容易过拟合。其次，基于统计的修复方法忽视了领域知识的重要性，没有发挥出领域知识对指导数据修复的重要作用。文献[53][54]提出了针对实体的识别方法，对来自不同时

刻的数据区分实体。在此基础上，文献[55][56]研究了基于统计的实体识别方法，来对带有时间戳的数据记录较好地实现了实体识别。文献[55]提出了“突变”地概念（Mutation），“突变”用来表达某个属性上的属性值突然变化成不属于该属性值域的值。该工作基于“突变”给出了通过统计信息推断实体各个属性上属性值变化规律的算法。文献[56]提出了 MAROON 框架，该框架不仅考虑了更为复杂的数据变化模式，并且在实体识别的同时将数据来源纳入了考虑范围，从而将数据源的更新频率以及时间戳变化二者相结合，共同作为影响数据变化模式的重要因素，推断数据变化。为了检测数据错误的位置以及出现错误的原因，文献[57]通过贝叶斯分析建立了一个用于评估检测结果的代价模型，并提出 DATAXRAY 框架，给出了一个分布式地面向海量数据的数据检测算法。DATAXRAY 将数据元素和一系列特征对应起来，提出了元素-特征模型。而对于元素-特征模型中的每一个特征，将一个可能的错误原因与之对应，通过贝叶斯分析得到这种对应的好坏。对于数据最终的检测结果，可以用一个线性时间迭代算法来计算，该算法依据特征的层次化关系，自顶向下地对数据进行最终检测。最后，正如前面提到的，上述算法是一个分布式地面向海量数据的数据检测算法，因此需要部署到并行的 MapReduce 分布式计算平台中。对于时效性错误修复，上述提到的诸多工作都具有参考价值，当存在大量的训练数据集时，可以采用统计学习的算法从训练集中训练出数据变化的规律，然后通过训练得到的变化规律修复目标数据。然而，目前该方向的修复方法没有考虑到领域知识的重要作用，通过领域知识来指导变化规律的训练，一定能取得更好的修复效果。

1.4 本文主要研究内容

本文将根据数据时效性管理领域已有的相关研究成果，改进并提出一系列数据时效性管理的关键算法。本文主要的研究内容如下：

（1） 针对时效图构造算法时间复杂度和空间复杂度过高的问题，对时效图的数据结构进行优化，以适应分布式场景中传输成本，进而优化时效性判定算法在分布式场景中的执行效率；

（2） 针对用户对数据库不同维度的查询请求，给出不同维度的时效性度量方法；

（3） 针对一种由于时效规则推断的时序关系与真实时序关系相矛盾导致的错误，给出定义以及判定原理。并根据判定原理给出该错误是否存在的判定算法；

（4） 针对上述错误，研究该错误的检测方法，并基于检测方法给出一种高效的

错误修复算法。

1.5 本文内容安排

第一章：绪论。介绍了课题来源、研究背景、国内外研究现状以及本文主要研究内容。

第二章：基于分布式的数据时效性判定技术。本章针对时效图构造算法时间复杂度和空间复杂度过高，在海量数据的应用场景中表现不佳的问题，基于时效规则提出了状态图的概念及其构造算法。状态图相比时效图具有更好的空间复杂度，从而大大优化了时效性判定算法在分布式场景中的执行效率。另外，本文还针对用户对数据库不同维度的查询请求，给出了不同维度的时效性度量方法。

第三章：基于时效规则的时效性关联错误修复技术。本章针对一种特殊的数据时效性关联错误，给出了实体冗余记录是否具有数据时效性关联错误的判定原理。为了数据时效性判定算法的正确计算，需要对实体的冗余记录进行时效性关联错误修复。本文给出了一种时效性关联错误的检测方法，并基于检测方法给出一种高效的时效性关联错误修复算法。

第四章：结论。本章总结了全文研究内容。

第 2 章　基于分布式的数据时效性判定算法

2.1　引言

数据库时效性管理领域中，实体记录时效性的判定问题是一个研究的重点问题。无论是检测数据记录是否过时，还是对过时的数据进行时效性修复，亦或是用户需要查询使用数据，都需要先判定数据的时效性。所谓的判定既可以指代“过时”和“非过时”这样的二进制状态，也可以指代数据记录或数据集合过时的可能性。

本章针对数据库中缺少完整、精确、可用的时间戳的场景，利用实体的冗余记录和领域专家给出的时效规则，设计实现基于规则的数据时效性判定算法。冗余记录是指数据库中一个实体存在着多条数据记录。例如，从多个数据源进行数据采集时，可能存在多条来自不同数据源的数据记录描述同一实体；或者在大型企业的数据库中，由于定期的备份数据，同一实体也可能存在多条历史数据。时效规则是由领域专家提供的该领域数据记录的约束规则，可以判定相同实体在不同属性值上的时序关系，从而判定实体记录的时效性，进而得到整个数据库的时效性。

对于多个数据源采集而来的数据库，其中的实体记录可能存在过时的数据。因此，针对用户对数据库 D 中的查询请求，需要先调用时效性判定算法，返回结果的时效性，获取查询请求的结果是否过时以及过时的严重程度。

例 2.1 考虑图 2-1 中的数据表 employee，其描述了某公司职员 Tom 在过去几年的基本情况。其数据模式为 $R = (eId, vId, Name, Age, Salary, Status)$，分别表示实体 id、冗余记录 id、姓名、年龄、薪资、婚姻状况。

表 2-1 数据表 employee

eId	vId	Name	Age	Salary	Status
1	1	Tom	18	5k	Single
1	2	Tom	19	6k	Single
1	3	Tom	19	6k	Married
1	4	Tom	20	7k	Divorced

显然 employee 表中没有时间戳属性来表示数据记录生成的时间，因此需要通过其他的方式判断 Tom 的四条记录之间的生成先后顺序。容易知道该数据模

式有如下条时效规则：

（1） 一个人的薪资不会随时间的增加而下降；

（2） 一个人的婚姻状况只能由 Single 变成 Married，再变成 Divorced。

根据上述时效规则可知，Tom 实体记录在属性 Salary 上生成的顺序为 5k −> 6k −> 7k，在属性 Status 上生成的顺序为 Single −> Married −> Divorced。

从上面的例子可以看出，尽管数据集合没有完整、精确、可用的时间戳，但存在冗余记录，且有足够的时效规则时，仍然可以判断实体的冗余记录之间的生成顺序。然后根据这些冗余记录的生成顺序，可以分析出实体记录的时效性以及整个数据表的时效性。文献[17]研究了基于时效规则对包含冗余记录的数据集合进行时效性判定的相关理论，该工作定义了两类数据相对时效性：查询相关时效性、用户相关时效性，并给出了基于时效图的时效性判定算法。然而该算法仍存在一些不足：其包含的时效图构造算法时间复杂度过高，预处理时间太长；时效图的空间复杂度过高，与数据量呈正相关，在分布式场景中需要频繁地在节点间进行网络传输，严重影响判定算法的执行效率，从而造成判定算法在大数据量分布式场景下性能表现不佳。

本章在文献[17]的基础上，研究了分布式场景下的时效性判定算法。针对时效图构造算法时间复杂度和空间复杂度过高，在海量数据的应用场景中表现不佳的问题，基于时效规则提出了状态图的概念，用状态图替代时效图的构造算法。状态图相比时效图具有更好的空间复杂度，非常适合在分布式场景中进行网络传输，从而大大优化了时效性判定算法在分布式场景中的执行效率。另外，本文还针对用户对数据库不同维度的查询请求，给出了不同维度的时效性度量方法，基于这些度量能够返回单个实体单个属性上的时效性、单个实体多个属性上的时效性、多个实体多个属性上的时效性以及整个数据库的时效性。

本章各节安排如下：2.2 节介绍了数据时效性的表达原理并介绍了改进的时效图——状态图；2.3 节介绍了基于状态图的时效图构造算法；2.4 节给出了优化后的时效性判定算法；2.5 节介绍了基于分布式平台 Spark 实现的不同维度的数据时效性判定算法；2.6 节针对时效性判定算法进行了实验设计与分析。

2.2 数据时效性表达原理

2.2.1 预备知识

假设数据库中的实体存在多条冗余记录，且冗余记录中存在实体 id 和冗余记

录 id。令数据模式$R=(eId,vId,Attr_1,\ldots,Attr_n)$，其中 eId 表示实体 id，vId 表示冗余记录 id，$t_i[eId]=t_j[eId]$表示t_i和t_j是同一个实体的两条记录，$Attr_1,\ldots,Attr_n$是实体的 n 个属性，$t_i[Attr]$表示记录t_i在$Attr$上的属性值。

时序关系描述了同一个实体两个数据记录在属性$Attr$上的值之间的新旧关系，它是时效性判定的基本特性，其形式化定义如下：

定义 2.1（时序关系）设数据集合 D 的数据模式为$R=(eId,vId,Attr_1,\ldots,Attr_n)$。当数据集合 D 中记录t_i和t_j满足下述两个条件时：

（1） $t_i[eId]=t_j[eId]$；

（2） $t_i[Attr]$先于$t_j[Attr]$出现在 D 中。

则记录t_i和t_j在属性$Attr$有时序关系$t_i<_A t_j$。例如，在例 2.1 的$employee$表中，实体 Tom 的记录t_1和t_3之间在属性$Status$上有时序关系$t_1<_{Status}t_3$。

针对数据库中的一个实体，基于实体某一属性上的属性值之间的时序关系，可以将实体在某一属性上的所有属性值用一个有向图表示出来，其中，图的顶点表示实体的属性值，边表示属性值之间的时序关系。通过时序关系的传递性可知，该有向图是一个DAG有向无环图。其形式化定义如下：

定义 2.2（时效图）设数据集合D的数据模式为$R=(eId,vId,Attr_1,\ldots,Attr_n)$，对于数据集合$D$中的任一实体$e$，$Attr$为$R$中任一属性，$S_e=\{t_1,\ldots,t_n\}$是数据集合$D$中描述实体$e$的所有数据记录，若存在有向图$G_e(V,E)$满足下述条件：

（1） $|V|=|S_e|$，S_e中的数据记录t_i对应V中的节点v_i；

（2） 边集 E 中存在有向边(v_i,v_j)当且仅当实体e的数据记录t_i和t_j在属性$Attr$上有时序关系$t_i<_{Attr}t_j$。

称有向图$G_e(V,E)$为实体e的时效图，显然时效图是DAG有向无环图。对于例 2.1 的数据集合$employee$，其属性$Salary$和$Status$上的时效图如下：

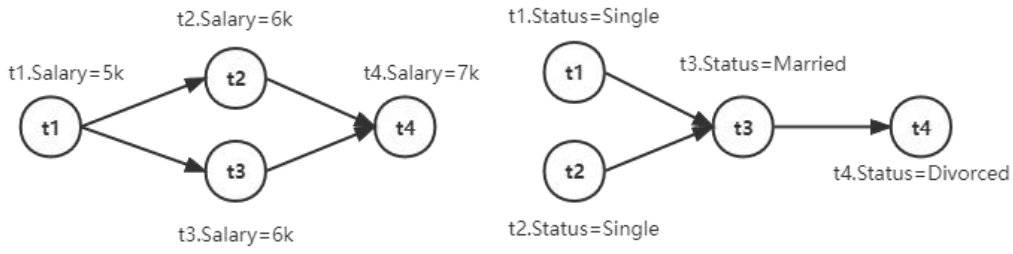


图 2-1 数据表 employee 在属性 Salary 和 Status 上的时效图

根据数据库中实体记录所属的特定领域，领域专家可根据该领域的通用知识对记录的属性值进行约束，这些领域知识可以形式化地转化为计算机能够识别的规则，对于那些能够判定属性值之间时序关系的规则，称之为时效规则，其形式化定义如下：

定义 2.3（时效规则）设$Attr$是数据集合D中模式$R=(eId,vId,Attr_1,\ldots,Attr_n)$的属性。$R$在$Attr$上的时效规则可以用一阶逻辑语句表达，形式如下：

$$\forall t_1,\ldots,t_k:R(\bigwedge_{j\in[1,k]}(t_1[eId]=t_j[eId])\wedge\psi\rightarrow t_u<_{Attr}t_v)$$

其中，$t_1,\ldots,t_k$是模式 R 的任意实例的 k 条记录；ψ是谓词；$\bigwedge_{j\in[1,k]}(t_1[eId]=t_j[eId])\wedge\psi$称为时效规则前件；$t_u<_{Attr}t_v$称为时效规则后件。

例 2.2 数据模式$employee(Id,Name,Salary,Status)$上的$Salary$属性上有时效规则：同一个人的工资不会随时间下降，其逻辑语句如下：

$$\forall s,t:Emp(t[eId]=s[eId]\wedge t[Salary]<s[Salary]\rightarrow t<_{Salary}s)$$

数据模式 employee(Id, Name, Salary, Status)上的 Status 属性上有时效规则：同一个人的婚姻状况只能由 Single 变成 Married 再变成 Divorced，其逻辑语句如下：

$$\forall x,y,z:Emp(x[eId]=y[eId]=z[eId]\wedge(x[Status]=Single)\wedge(x[Status]=Married)\wedge(x[Status]=Divorced)\rightarrow x<_{Status}y<_{Status}z)$$

对于上述两种常见的时效规则，我们称例 2.1 中的时效规则为单调时效规则，例 2.2 中的时效规则为状态时效规则。

定义 2.4（单调时效规则）设$Attr$是数据集合D中模式R的属性。R在$Attr$上的单调时效规则定义为如下的一阶逻辑语句：

$$\forall t_1,\ldots,t_k:R(\bigwedge_{j\in[1,k]}(t_1[eId]=t_j[eId])\wedge t_u[Attr]<t_v[Attr]\rightarrow t_u<_{Attr}t_v)$$

或

$$\forall t_1,\ldots,t_k:R(\bigwedge_{j\in[1,k]}(t_1[eId]=t_j[eId])\wedge t_u[Attr]<t_v[Attr]\rightarrow t_u>_{Attr}t_v)$$

其中，$t_1,\ldots,t_k$是模式R的任意实例的 k 条记录。单调时效规则作用于关系模式的单个属性上，属性的值域空间为无穷集合，且属性值随时间单调递增或递减。由于单调时效规则的时序关系判定简单直接，本文不针对该时效规则进行讨论。

定义 2.5（状态时效规则）设$Attr$是数据集合D中模式R的属性。R在$Attr$上的状态时效规则定义为如下的一阶逻辑语句：

$$\forall t_1,\ldots,t_k:R(\bigwedge_{j\in[1,k]}(t_1[eId]=t_j[eId])\wedge(t_u[Attr]="status1")\wedge(t_v[Attr]="status2")\rightarrow t_u<_{Attr}t_v)$$

其中，$t_1,\ldots,t_k$是模式R的任意实例的 k 条记录。状态时效规则作用于关系模式的单个属性上，属性的值域空间为有穷集合，称每一种属性值为一种状态，状态数远远小于记录数。

2.2.2 状态图

通过上一小节的定义可知，实体在属性$Attr$的所有数据记录可以用时效图来表达。考虑实体在属性$Attr$上的所有时效规则，每一个规则通常作用于两个属性值上，那么所有时效规则覆盖的所有属性值也可以构成一个时效图，其顶点集表示所有覆盖的属性值，边集表示属性值之间的时序关系。为了区别于与实体属性一一对应的时效图，称与作用于属性上的所有时效规则一一对应的时效图为状态图。

定义 2.6（状态图）设数据集合 D 的数据模式为$R = (eId, vId, Attr_1, \ldots, Attr_n)$，对于数据集合 D 中的任一实体$e$，$Attr$为$R$中任一属性，$S_e = \{t_1, \ldots, t_n\}$是数据集合$D$中描述实体$e$的所有数据记录，存在单调时效规则或状态时效规则：

$$\forall t_1, \ldots, t_n: R(\bigwedge_{j\in[1,n]} (t_1[eId] = t_j[eId]) \wedge \psi \rightarrow t_u <_{Attr} t_v)$$

$\mathrm{S_{Attr}} = \{\mathrm{s_1}, \ldots, \mathrm{s_k}\}$是时效规则作用在属性 Attr 上的所有属性值，有向图$\mathrm{G_A(V, E)}$称为时效规则的状态图，当且仅当下述条件同时满足：

1）$|V| = |S_{Attr}|$，且V中的任意节点v_i对应S_{Attr}中的一个属性值；

2）$s_i <_{Attr} s_j$当且仅当E中存在有向边(v_i, v_j)。

容易看出单调时效约束的状态图是一条无穷顶点的单向链；而状态时效约束的状态图可以是任意的有向无环图。

对于例 2.1 的数据集合$employee$，其属性$Salary$和$Status$上的状态图如下：

图 2-2 数据表 employee 在属性 Salary 和 Status 上的状态图

数据时效性表达

对于多个数据源采集而来的数据库，其中的实体记录可能存在过时的数据。因此，针对用户对数据库D中的查询请求，需要先调用时效性判定算法，返回结果的时效性，获取查询请求的结果是否过时以及过时的严重程度。

根据查询的对象可以分三个维度定义时效性：针对用户对数据库D中的某一实体e的某一属性$Attr$的查询请求，返回该实体e在属性$Attr$上的时效性$curr_{Attr}$；而针对用户对数据库D中某一实体 e 的多个属性$Attr_1, \ldots, Attr_n$的查询请求，返回该实体 e 在多个属性$Attr_1, \ldots, Attr_n$上的时效性$curr_{Attr_1,\ldots,Attr_n}$；最后，针对用

户对数据库D中多个实体$e_1,\ldots,e_n$的查询请求，返回数据库D的时效性$curr_{e_1,\ldots,e_n}$。

根据查询的结果可以分两种类型定义时效性：针对用户对数据库D中属性的最新值查询请求，返回该最新值的时效性$curr_{CVQ}$；针对用户对数据库D中属性的历史值查询请求，返回该历史值序列的时效性$curr_{HSQ}$。

上述两种定义方式组合起来有6种时效性查询，它们的形式化定义和计算表达式如下：

定义 2.7（时效性查询）将用户对数据库D的时效性查询Q形式化定义为为三元组$Q=(EntitySet, AttrSet, Type)$，其中$EntitySet$表示用户查询的实体集合；$AttrSet$表示用户查询的属性集合；$Type$表示用户的查询类别，包括最新值查询（$CVQ, Current\ Value\ Query$：查询返回结果为实体的最新值）和历史值查询（$HSQ, History\ Sequence\ Query$：查询返回结果为实体的历史值序列）两种。

定义 2.8（针对用户查询的数据时效性）根据用户查询的三个维度和两种类型可将时效性查询分为6种，针对每一种时效性查询可将数据时效性用以下6种度量表达：

（1） 当时效性查询Q的$EntitySet$为单个实体e，$AttrSet$为单个属性$Attr$，$Type=CVQ$时，

$$curr_Q=\frac{\min\limits_{value_i\in S_{e,Attr}} cnt(value_i)}{\sum_{value_i\in S_{e,Attr}} cnt(value_i)} \tag{2-1}$$

其中，$S_{e,Attr}$为实体e的冗余记录在属性$Attr$上的最新值集合，$cnt(value_i)$表示$value_i$在冗余记录中重复出现的个数。

（2） 当时效性查询Q的$EntitySet$为单个实体e，$AttrSet$为多个属性$Attr$，$Type=CVQ$时，

$$curr_Q=\sum_{Attr_i\in AttrSet} w_{Attr_i}\times\frac{\min\limits_{value_j\in S_{e,Attr_i}} cnt(value_j)}{\sum_{value_j\in S_{e,Attr_i}} cnt(value_j)} \tag{2-2}$$

其中，$S_{e,Attr_i}$为实体e的冗余记录在属性$Attr_i$上的最新值集合；$cnt(value_j)$表示$value_j$在冗余记录中重复出现的个数；w_{Attr_i}表示每个属性$Attr_i$的权重。

（3） 当时效性查询Q的$EntitySet$为多个实体e，$AttrSet$为多个属性$Attr$，$Type=CVQ$时，

$$\mathrm{curr_Q} = \frac{1}{\mathrm{cnt(EntitySet)}} \sum_{e_k \in EntitySet} \left(\sum_{Attr_i \in AttrSet} w_{Attr_i} \times \frac{\min\limits_{value_j \in S_{e_k,Attr_i}} cnt(value_j)}{\sum_{value_j \in S_{e_k,Attr_i}} cnt(value_j)} \right) \tag{2-3}$$

其中，$S_{e_k,Attr_i}$为实体e_k的冗余记录在属性$Attr_i$上的最新值集合，$cnt(value_j)$表示$value_j$在冗余记录中重复出现的个数。$cnt(EntitySet)$表示不同实体的数量。

（4） 当时效性查询Q的$EntitySet$为单个实体e，$AttrSet$为单个属性$Attr$，$Type = HSQ$时，

$$curr_Q = cos\frac{\pi}{2 \times cnt(S_{e,Attr})} \tag{2-4}$$

其中，$S_{e,Attr}$为实体e的冗余记录在属性$Attr$上历史值序列的集合，$cnt(S_{e,Attr})$表示不同历史值序列的个数。

（5） 当时效性查询 Q 的$EntitySet$为单个实体e，$AttrSet$为多个属性$Attr$，$Type = HSQ$时，

$$curr_Q = \sum_{Attr_i \in AttrSet} w_{Attr_i} \times cos\frac{\pi}{2 \times cnt(S_{e,Attr_i})} \tag{2-5}$$

其中，$S_{e,Attr_i}$为实体e的冗余记录在属性$Attr_i$上历史值序列的集合，$cnt(S_{e,Attr_i})$表示不同历史值序列的个数；w_{Attr_i}表示每个属性$Attr_i$的权重。

（6） 当时效性查询Q的$EntitySet$为多个实体e，$AttrSet$为多个属性$Attr$，$Type = HSQ$时，

$$curr_Q = \frac{1}{\mathrm{cnt(EntitySet)}} \sum_{e_j \in EntitySet} \left(\sum_{Attr_i \in AttrSet} w_{Attr_i} \times cos\frac{\pi}{2 \times cnt(S_{e_j,Attr_i})} \right) \tag{2-6}$$

其中，$S_{e_j,Attr_i}$为实体e_j的冗余记录在属性$Attr_i$上历史值序列的集合，$cnt(S_{e,Attr_i})$表示不同历史值序列的个数；w_{Attr_i}表示每个属性$Attr_i$的权重。

上述时效性表达式的意义如下：cur_Q的值域为[0,1]，其值越接近 1 则查询结果的时效性越好，越接近 0 则表示查询结果的时效性越差。对于CVQ查询来说，当有多个候选最新值时，应该返回重复次数最少的最新值，因为刚更新的新鲜数据在往往只在某一个数据源中进行了更新；并且当候选最新值的数量越多，时效性趋近于 0，说明CVQ查询的时效性越差。对于HSQ查询来说，当历史值序列的候选结果越多，则说明HSQ查询的时效性越差，且随着候选结果数量的上升，时

效性快速趋近于 0，这表明当结果数量过多时，查询对象的时效性越差，查询结果就没有了参考意义。

2.3 基于状态图的时效图构造算法

2.3.1 状态图的构造算法

对于当前已有的基于时效图的时效性判定算法，其时效图构造算法复杂性较高，通常作为预处理操作。其基本思路是：将所有实体记录作为顶点初始化时效图，然后遍历每一对记录，若满足时效规则，则将对应的有向边添加到时效图中。时效图得构造算法如下：

算法 2-1 时效图构造算法

输入：实体e，属性$Attr$，S_e，$Rule$

输出：$G_{e,Attr}$

1 $G \leftarrow (V,E), V \leftarrow \emptyset, E \leftarrow \emptyset$;
2 ***for*** $t_i \in S_e$ ***do***
3 add v_i into V;
4 ***end for***
5 ***for each*** r $\in Rule$ ***do***
6 ***for each*** 能由规则 r 推导得到的 $t_i < t_j$ ***do***
7 add (v_i,v_j) into ***E***;
8 ***end for***
9 $G_{e,Attr} \leftarrow G$;
10 ***Return*** $G_{e,Attr}$

算法的输入为实体e，属性$Attr$，实体冗余记录集合S_e，作用在属性$Attr$上得规则集合$Rule$；输出为实体 e 在属性$Attr$上的时效图$G_{e,Attr}$。算法第 1 行初始化时效图$G_{e,Attr}$，顶点集 V 和边集 E 初始化为空；第 2 行到第 4 行，将实体冗余记录集合S_e中的每一条记录t_i作为一个顶点v_i添加到顶点集 V 中；第 5 行到第 8 行，对于规则集合$Rule$中的每一条规则 r，遍历所有记录对t_i, t_j，若能根据规则 r 推导出$t_i < t_j$，则将对应的有向边(v_i,v_j)添加到边集 E 中；第 9 行到第 10 行，返回时效图$G_{e,Attr}$。

假设实体记录数为 n，则常规的时效图构造算法时间复杂度为$O(n^2)$。空间复杂度为$O(n)$。

考虑到大数据量的场景，实体存在大数据量的冗余记录，上述时效图构造算

法将存在两个显著的缺陷：1）作为结果返回的时效图将占用大量内存，因为其顶点数等价于数据量。2）基于时效图执行时效性判定算法时，对顶点的遍历造成巨大的时间成本和计算资源。

状态图的构造成功避免了上述两个问题。首先，和时效图顶点与冗余记录一一对应不同，状态图顶点与时效规则作用属性值一一对应，因此和时效图的构造需要遍历所有冗余记录不同，状态图的构造只需遍历时效规则集，而时效规则的数量远远小于冗余记录的数量，状态图构造的时间成本将远远小于时效图的构造。对于第二个问题同理，作为结果返回的状态图占用内存也将远远小于时效图。状态图的构造算法如下：

算法 2-2 状态图构造算法

输入：属性$Attr$，$Rule$

输出：G_{Attr}

1 $G \leftarrow (V, E),\ V \leftarrow \emptyset,\ E \leftarrow \emptyset$;
2 ***for each*** r $\in Rule$ ***do***
3 规则 r 作用的两个属性值为$value_1$、$value_2$;
4 $v_1 \leftarrow G.\text{get}(value_1)$;
5 ***if*** (v_1 == null)
6 为属性值$value_1$新建节点v_1，$v_1.numRep \leftarrow 0$, V $\leftarrow v_1$;
7 ***end if***
8 $v_2 \leftarrow G.\text{get}(value_2)$;
9 ***if*** (v_2 == null)
10 为属性值$value_2$新建节点v_2，$v_2.numRep \leftarrow 0$, V $\leftarrow v_2$;
11 ***end if***
12 规则推出的有向边为(v_1,v_2)，$E \leftarrow (v_1, v_2)$;
13 ***end for***
14 $G_{Attr} \leftarrow G$;
13 ***Return*** G_{Attr}

算法的输入为属性$Attr$，作用在属性$Attr$上的规则集合$Rule$；输出为属性$Attr$对应的状态图G_{Attr}。算法第 1 行初始化状态图G_{Attr}，顶点集 V 和边集 E 初始化为空；第 2 行到第 11 行，为规则集作用的每一个属性值$value$创建一个顶点v添加到顶点集 V 中，每个顶点存放一个参数$numRep$为之后构造时效图记录重复数，且对于规则 r 推导出$value_i < value_j$，将对应的有向边(v_i,v_j)添加到边集 E 中；第 8 行到第 9 行，返回状态图G_{Attr}。

假设实体记录数为n，时效规则数为m，则状态图构造算法时间复杂度为$O(m)$，与实体记录数无关。空间复杂度为$O(m)$。

2.3.2 基于状态图的时效图构造算法

之前提到过，针对大数据场景下基于时效图的时效性判定问题，时效图存在缺陷，因此需要对时效图“瘦身”。考虑到时效性判定问题依靠时效规则对各个冗余记录之间推断时序关系来进行判定，而基于时效图的时效性判定算法用时效图的有向边来表达时序关系，因此可以在保证有向边完整的前提下，对时效图进行删减：首先，可以删去那些不存在边的顶点，即时效规则作用不到的冗余记录对应的顶点；其次，删去不存在边的顶点后，由于时效规则的数量远远小于冗余记录的数量，记录在属性值上存在大量重复，可以将重复属性值对应的顶点合并为一个顶点，并记录用一个计数器记录重复数。

通过上述分析，直观地可以基于时效图进行上述删减操作，但这么做仍然存在需要将完整时效图存入内存进行删减导致的空间占用过大问题。通过状态图的定义可知，上述删减操作后的时效图和状态图基本相似，区别在于状态图包含时效规则作用的所有属性值，而实体的冗余记录未必包含时效规则作用的所有属性值，也就是说，删减后时效图的顶点集是状态图顶点集的子集，边集也是状态图边集的子集。于是，可以通过对状态图进行删减得到“瘦身”后的时效图，这样只需要在内存中存放占用空间很小的状态图，完美解决了内存占用的问题。状态图的删减操作只需要删去那些冗余记录没有的属性值对于的顶点即可。基于状态图的时效图构造算法如下：

算法 2-3 基于状态图的时效图构造算法

输入：实体 e，属性 Attr，S_e，$Rule$

输出：$G_{e,\mathrm{Attr}}$

1 $G_{\mathrm{Attr}} \leftarrow$ 状态图构造算法(属性 Attr, $Rule$)
2 ***for*** $t_i \in S_e$ ***do***
3 获取t_i在属性 Attr 上属性值$value_i$对应的顶点v_i;
4 $v_i.numRep$ ++;
5 ***end for***
6 初始化一个队列 que;
7 遍历G_{Attr}的顶点集 V, 将入度为 0 的顶点入队;
8 ***while***(que.nonEmpty)
9 $v_i \leftarrow$ que. dequeue;
10 将v_i的后继节点中入度为 0 的顶点入队;
11 ***if***($v_i.numRep == 0$)
12 从G_{Attr}中删去v_i;

13　　更新v_i前驱节点和后继节点的入度出度信息;
14　***end if***
15　***end while***
16　$G_{e,\text{Attr}} \leftarrow G_{\text{Attr}}$;
17　***Return***　$G_{e,\text{Attr}}$

算法的输入为实体e，属性$Attr$，实体冗余记录集合S_e，作用在属性$Attr$上得规则集合集合$Rule$；输出为实体e在属性$Attr$上的时效图$G_{e,Attr}$。算法第 1 行调用状态图构造算法获取状态图G_{Attr}；第 2 行到第 5 行，遍历实体冗余记录集合S_e中的每一条记录t_i，找出t_i对应的顶点v_i并更新$v_i.numRep$；第 6 行初始化一个队列 que；第 7 行遍历状态图G_{Attr}的顶点集 V，将入度为 0 的顶点入队；第 8 行到第 15 行，借助队列，遍历状态图中每一个顶点，若其$v_i.numRep$为 0，则从G_{Attr}中删去v_i，并更新v_i前驱节点和后继节点的入度出度信息；第 16 行到第 17 行，返回时效图$G_{e,Attr}$。

假设实体记录数为 n，时效规则数为 m，则状态图构造算法时间复杂度为$O(\text{n}+m)$，空间复杂度为$O(m)$。

2.4　数据时效性判定算法

2.4.1 CVQ 时效性判定算法

本节我们针对时效性查询$Q=(EntitySet, AttrSet, Type)$，其中$EntitySet$为单个实体$e$，$AttrSet$为单个属性$Attr$，$Type=CVQ$，$curr_Q=\frac{\min\limits_{value_i \in S_{e,Attr}} cnt(value_i)}{\sum_{value_i \in S_{e,Attr}} cnt(value_i)}$。其中，$S_{e,Attr}$为实体$e$的冗余记录在属性$Attr$上历史值序列的集合，$cnt(S_{e,Attr})$表示不同历史值序列的个数。

由时效图的定义可知，最新值集合$S_{e,Attr}$可以用出度为 0 顶点组成的集合表示，$cnt(value_i)$等价于出度为 0 顶点的参数$numRep$。因此，CVQ时效性判定算法的基本思想为：遍历时效图中出度为 0 的所有顶点，用一个变量min记录顶点参数$numRep$的最小值，用一个变量 sum 记录顶点参数$numRep$的总和，最后返回min/sum的值。算法如下：

算法 2-4 CVQ时效性判定算法

输入：$G_{e,Attr}$
输出：$curr_Q$
1　$sum \leftarrow 0$; $min \leftarrow$ Integer.MAX;

2 ***for*** $v_i \in G_{e,Attr}$.V ***do***
3 ***if***(v_i.outDegree == 0)
4 sum ← sum + v_i. numRep;
5 ***if***(v_i.numRep < min)
6 min ← v_i. numRep;
7 ***end if***
8 ***end if***
9 ***end for***
10 $curr_Q \leftarrow min/sum$;
11 ***Return*** $curr_Q$

算法的输入为$G_{e,Attr}$；输出为时效性$curr_Q$。算法第 1 行初始化变量sum为 0 和变量min为一个极大数；第 2 行到第 9 行，遍历时效图的所有顶点，每当遇到出度为 0 的顶点，更新变量sum，若顶点的参数 *numRep* 小于变量min，则更新min的值；第 10 行到第 11 行计算并返回时效性$curr_Q$。

假设顶点数为 n，则CVQ时效性判定算法时间复杂度为$O(n)$。

2.4.2 HSQ 时效性判定算法

本节我们针对时效性查询$Q = (EntitySet, AttrSet, Type)$，其中$EntitySet$为单个实体$e$，$AttrSet$为单个属性$Attr$，$Type = HSQ$，$curr_Q = cos\frac{\pi}{2\times cnt(S_{e,Attr})}$。其中，$S_{e,Attr}$为实体$e$的冗余记录在属性$Attr$上历史值序列的集合，$cnt(S_{e,Attr})$表示不同历史值序列的个数。

由时效图的性质可知，历史值序列可以视为时效图的拓扑序列，因此$cnt(S_{e,Attr})$可以等价于时效图不同拓扑序列的个数。因此，HSQ时效性判定算法的基本思想为：用计数器 *count* 记录不同拓扑序列的个数，从入度为 0 的顶点出发，采用深度遍历的方式，遍历时效图的所有顶点；在遍历完所有顶点时，更新计数器，然后回溯到上一个顶点继续遍历。算法如下：

算法 2-5 HSQ时效性判定算法

输入：$G_{e,\mathrm{Attr}}$
输出：$\mathrm{curr_Q}$
1 count ← 0; sum ← $|G_{e,\mathrm{Attr}}.\mathrm{V}|$; result ← 0;
2 Set ← null;
3 ***for*** $v_i \in G_{e,\mathrm{Attr}}$.V ***do***
4 ***if***(v_i.inDegree == 0)

```
5          Set.add(v_i);
6       end if
7    end for
8    for v_i ∈ Set do
9       dfs(G_e,Attr, v_i, Set, count);
10   end for
11   curr_Q ← cos π/(2×result);
12   Return curr_Q
```

func：dfs

输入：$G_{e,Attr}$, v, Set

输出：null

```
1    count ++;
2    if(count == sum)
3       result ++;
4    end if
5    Set.remove(v);
6    for v_i ∈ v.nextVertexes do
7       v_i.inDegree --;
8       if( v_i.inDegree == 0)
9          Set.add(v_i);
10      end if
11   end for
12   for v_j ∈ Set do
13      dfs(G_e,Attr, v_j, Set, count);
14   end for
15   for v_i ∈ v.nextVertexes do
16      if(v_i.inDegree == 0)
17         Set.remove(v_i);
18      end if
19      v_i.inDegree ++;
20   end for
21   Set.add(v);
22   Return null
```

算法的输入为$G_{e,Attr}$；输出为时效性$curr_Q$。算法第 1 行到第 2 行初始化计数器为 0，集合 *Set* 为空；第 3 行到第 7 行，遍历时效图$G_{e,Attr}$的顶点集 V，将入度为 0 的顶点添加到集合 Set 中；第 8 行到第 10 行，对集合 *Set* 中的每一个顶点调用 dfs 方法；第 11 行到第 12 行，根据计数器的值返回时效性$curr_Q$。

dfs 方法的输入为时效图$G_{e,Attr,}$ 顶点v,和集合 *Set*；输出为 null；算法第 1 行到第 4 行，若当前顶点v的出度为 0，则计数器 *count* 加 1，并直接返回 null；第 5 行，将顶点v从集合 *Set* 中移除；第 6 行到第 11 行，遍历顶点v的后继顶点，将它们的入度减 1，并将入度为 0 的后继顶点添加到集合 Set 中；第 12 行到第 14 行，对集合 *Set* 中的每一个顶点调用 dfs 方法；第 15 行到第 20 行，遍历顶点v的后继顶点，将入度为 0 的后继顶点从集合 *Set* 中移除，并将它们的入度加 1；第 21 行到第 22 行，将顶点v添加到集合 *Set* 中，返回 null；

假设顶点数为 n，则HSQ时效性判定算法时间复杂度为$O(n^2)$。

2.5 分布式框架下的时效性判定算法

上一节讨论了针对时效性查询$Q = (EntitySet, AttrSet, Type)$，其中$EntitySet$为单个实体$e$，$AttrSet$为单个属性$Attr$，$Type = CVQ$或$HSQ$的时效性判定算法。容易发现，在数据表中，时效性判定算法的输入就是多行元组在特定属性上的值，输出是一个[0,1]的数值，其本质上和 SQL 中的一个聚合函数（如：avg(), sum()等）无异。

2.5.1 用户自定义聚合函数 UDAF

分布式框架 SparkSQL 提供了 Aggregator 接口来实现用户自定义的聚合函数。在大数据场景下，数据时存储在分布式集群中的，分布在集群中不同机器上的实体冗余数据不需要拉取到一台机器上执行算法的计算任务，而是通过类似 MapReduce 的计算模型，分布式地执行大部分计算后，最后聚合到主节点上完成计算。

实现了 Aggregator 接口的聚合函数需要实现如下几个算子：

（1）zero 是每个分布式节点上中间结果的初始值；

（2）reduce 将当前节点上的每一条数据与当前节点上的中间结果进行自定义的计算操作，返回中间结果；

（3）merge 将两个节点上的中间结果通过自定义的合并操作转换为新的中间结果；

（4）finish 对聚合后的中间结果进行最后的计算操作，返回函数最终的计算结果。

SparkSQL 用户自定义聚合函数的实现原理如下图：

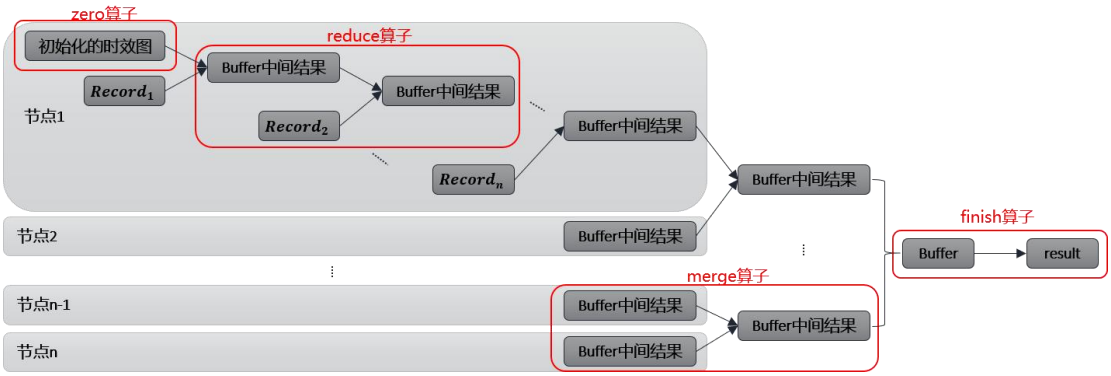


图 2-3 SparkSQL UDAF 分布式框架图

对于时效性判定算法在上述分布式框架下的实现，可以将时效图构造算法进行分解：可以将状态图构造算法封装为 zero 算子，即将状态图作为中间结果的初始值；而时效图构造算法中，遍历冗余记录并更新时效图中对应顶点的参数 numRep 的一系列操作可以封装为 reduce 算子，返回集群单个机器上所有冗余记录对应的时效图（还未完成删减操作）；通过 merge 算子将集群各个节点上的时效图合并为一个时效图；最后通过 finish 算子对聚合后的时效图进行删减操作。至此，基于分布式框架的时效图构造算法设计完成了。而时效性判定算法则直接封装在 finish 算子中，其结果直接作为聚合函数的结果返回。

综上，zero、reduce、merge 算子组合成了分布式时效图构造算法；而 finish 算子则封装了时效性判定算法。

merge 算子中有一个时效图的合并操作需要单独设计，算法如下：

算法 2-6 时效图合并算法

输入：时效图G_1, G_2

输出：时效图G

1 ***for*** $v_i \in G_1.V$ ***do***

2 $v_j \leftarrow G_2.get(v_i)$

3 $v_i.\text{numRep} \leftarrow v_i.\text{numRep} + v_j.\text{numRep}$;

4 ***end for***

5 $G \leftarrow G_1$

22 ***Return*** G

2.5.2 通过 UDF 实现其他维度的时效性判定算法

目前仅讨论了针对时效性查询$Q = (EntitySet, AttrSet, Type)$，其中$EntitySet$为单个实体$e$，$AttrSet$为单个属性$Attr$，$Type = CVQ$或$HSQ$的时效性判定算法。而对于$AttrSet$为多个属性$Attr$，甚至$EntitySet$为多个实体$e$的时效性

查询维度还没有给出实现算法。然而，通过计算表达式可以直观地发现，多个属性 Attr 的时效性判定就是在单个属性的基础上进行了加权累加操作；多个实体 e 的时效性判定则直接将每个实体的时效性求平均即可。

分布式框架 SparkSQL 提供了简单的用户自定义函数 UDF 来实现对每一行元组的一个或多个属性值进行计算，将结果作为新的列输出。因此，可以将 UDAF 聚合函数的输出结果（单实体单属性上的时效性）作为 UDF 的输入，用 UDF 实现加权累加操作以及求平均操作，从而实现多实体多属性上的数据时效性算法。

2.6 实验结果及分析

2.6.1 实验配置

实验采用四台机器组成的分布式集群作为硬件环境，单机的硬件配置如下表所示：

表 2-2 单机的硬件配置

项目	硬件
CPU	Intel Core i7-11700T@3.4GHz
GPU	Intel UHD Graphics 750
RAM	DDR4 3200MHz 64GB
ROM	SAMSUNG PM9A1 256GB

软件环境方面，实验使用了 Docker 开源的应用容器引擎在硬件网络的基础上构建了分布式集群环境，表中所示的机器的软件环境：

表 2-3 单机的软件环境

项目	软件
OS	Ubuntu20.04
Java	Open JDK 1.8.0_261
Scala	2.12.13
Docker	20.10.7

基于 Docker 容器，搭建了 Spark 分布式运行环境。这里简单介绍 Spark 计算环境的搭建过程：首先，Spark 镜像选择的是 bitnami 提供的 docker 镜像，并在该镜像的基础上配置 Dockerfile 文件，包装了 HDFS 作为存储层，新的镜像作为后续集群容器创建的统一镜像；然后，通过 Docker Swarm 建立了 Docker 跨主机的通信，并在此基础上通过 Docker Stack 统一部署了 Spark 分布式计算环境，部

署的方式只需要编写 Docker Compose yml 配置文件即可。Docker 容器中的环境配置如下：

表 2-4 Docker 容器中的软件环境

项目	软件
OS	bitnami 镜像封装的 OS
Java	Open JDK 1.8.0_261
Scala	2.13.8
Spark	3.2.0
Hadoop	3.3.0

Spark 集群包含 1 个 Master 节点，32 个 Worker 节点，其中每个 Spark Worker 容器内存为 7G，core 为 2，Spark 集群参数配置如下：

表 2-5 Spark 集群参数配置

项目	软件
total executor cores	64
executor memeory	7G
executor cores	2

基于上述实验环境，实验代码采用 scala 编写，软件开发环境为 Intellij IDEA，使用 Maven 管理项目并打 jar 包到集群分布式环境中运行测试结果。

2.6.2 实验数据

我们在构造的虚拟数据集上进行了实验，实验数据包含一个表 D，其数据模式为 $R = (eId, vId, name, age, city, grade, status)$。我们分别构造了 1 亿、5 亿、10 亿、50 亿条数据文件，并存储在 HDFS 中。对于 10 亿条数据，还针对属性 grade 和 status 上不同规则数的情况，分别构造了 10、50、100 条规则的数据文件。

2.6.3 CVQ 判定算法执行效率分析

我们从三个角度考量了 CVQ 判定算法的执行效率：

（1） 与为优化的时效性判定算法对比：对比两种算法再不同数据量下的执行效率差距；

（2） 数据量：分别以 1 亿、5 亿、10 亿条记录的数据表作为输入，grade 和 status 上分别 10 条规则，考察数据量对算法执行效率的影响；

（3） 规则数：分别在属性 grade 和 status 上以 10、50、100 条规则的 10 亿条记录的数据表作为输入，从而分别考察参考属性和修复属性上规则数对算法执行效率的影响；

（4） 不同维度的用户查询：针对用户对数据库不同维度的时效性查询请求，给出 CVQ 判定算法的测试用例，它们可以分别用如下 SQL 语句表达：

a) SQL1：select single_cvq(vId, grade) as cvq from table where eId = 1，表示对实体 eId = 1'在属性 grade 上的 CVQ 时效性判定；

b) SQL2：select multi_cvq(hsq1, hsq2) as cvq from (
select single_cvq(vId, grade) as cvq1, single_cvq(vId, status) as cvq2
from table where eId = 1)，表示对实体 eId = 1 在属性 grade 和 status 上的 CVQ 时效性判定；

c) SQL3：select eId, multi_cvq(hsq1, hsq2) as cvq from (
select eId, single_cvq(vId, grade) as cvq1, single_cvq(vId, status) as cvq2
from table where eId >= 1 and eId <= 10 group by eId)，表示对实体 eId = 1,…, 10 在属性 grade 和 status 上的 CVQ 时效性判定；

d) SQL4：select db_cvq(cvq) from (
select eId, multi_cvq(cvq1, cvq2) as cvq from (
select eId, single_cvq(vId, grade) as cvq1, single_cvq(vId, status) as cvq2
from table group by eId)，表示对整个数据库在属性 grade 和 status 上的 CVQ 时效性判定；

基于数据量对 CVQ 判定算法的考量结果如图所示，可以看出随着数据量的增大，CVQ 时效性判定算法的执行时间呈线性增长，即执行时间基本和数据量呈正比。这是由于判定算法的实现完美契合分布式计算的思想，各个 Worker 节点上计算出的中间结果与数据量无关，因此不会对 Master 节点 fetch 中间结果造成内存压力。这说明判定算法在 Spark 分布式平台上的代码实现具有良好的可扩展性，即若不考虑节点资源的限制，算法不会因为数据量的膨胀导致执行时间非线性增长。

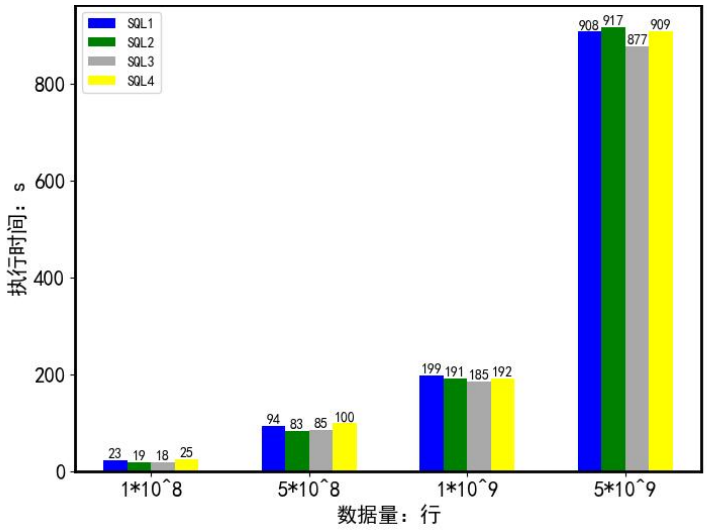


图 2-6 不同数据量对 CVQ 判定算法的测试结果

与未优化的 CVQ 时效性判定算法的实验结果对比图如下，优化后的 CVQ 时效性判定算法在数据量为 1 亿条时，执行时间减少了超过 90%。并且未优化的判定算法在跑 5 亿条及以上的数据时，Spark 集群会出现 OOM 错误，即在大数据量的场景下会耗费过多的内存资源。这是因为未优化的时效性判定算法，其时效图中的顶点与数据量正相关，随着数据量的增长，其内存占用也随之增长且在网络中传输成本极高。

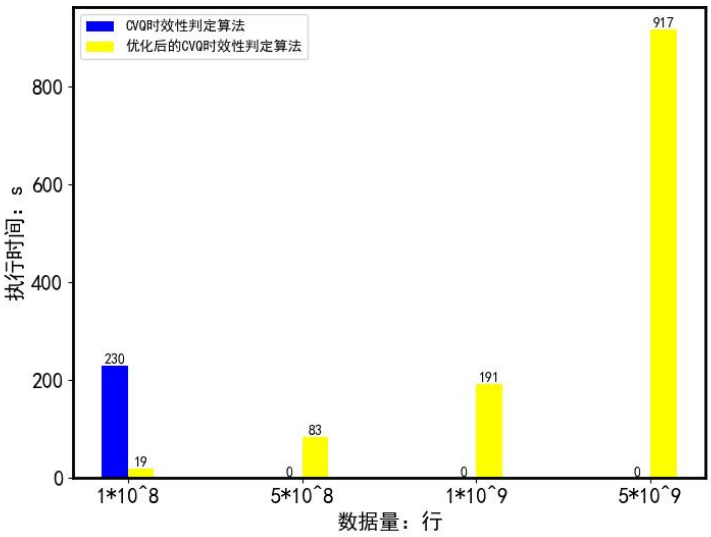


图 2-7 与未优化的 CVQ 判定算法对比结果

基于时效规则数对 CVQ 判定算法的考量结果如下面两个图所示。容易看出，SQL4 的执行时间受到时效规则数的影响最大，SQL1~SQL3 的执行时间几乎不受时效规则数的影响；这是因为时效性判定算法的执行时间基本与时效图的个数呈正相关，而时效图与单个实体的单个属性一一对应，时效图的大小与时效规则的数量线性相关。因此，SQL1~SQL3 针对的是少数实体的时效性查询，而 SQL4

是针对整个数据库所有实体的时效性查询。所以，在时效规则数较少时，SQL4的执行时间还没有明显高于 SQL1~SQL3，而当时效规则数迅速增大后，SQL4的执行时间也开始收到影响。

另外，对比两张图发现，在属性 grade 和 status 上分别增加时效规则数并没有产生较大差异的影响，这是因为实验测试中，对不同属性赋予了相同的权重，因此没有产生较大差异。

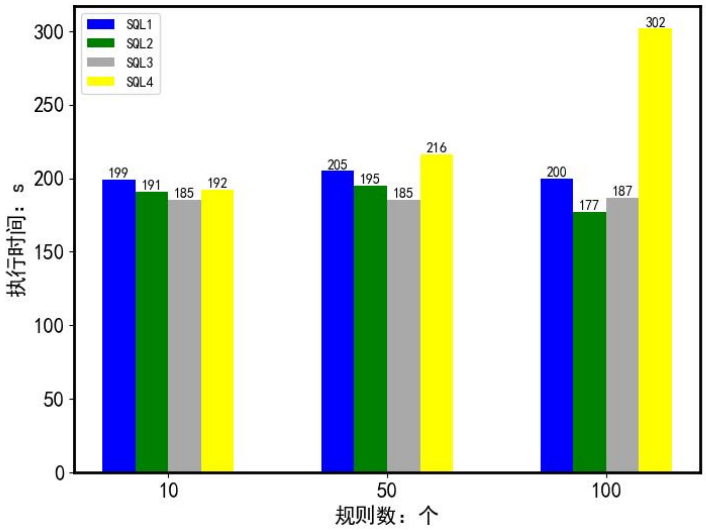


图 2-8 参考属性上不同规则数对 CVQ 判定算法的测试结果

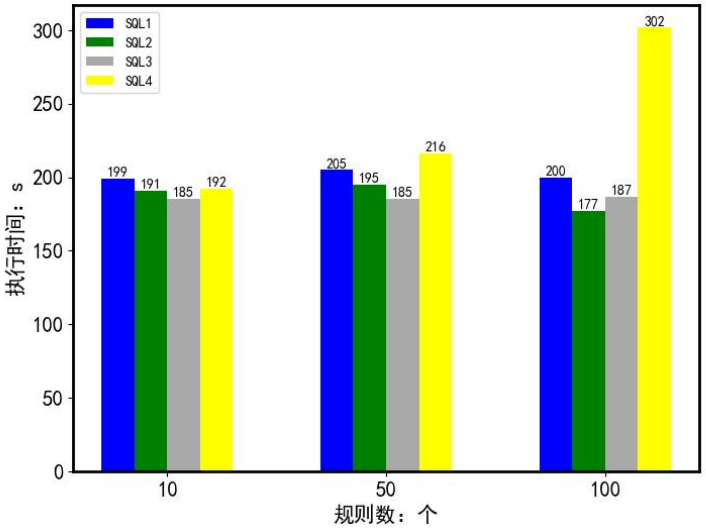


图 2-9 修复属性上不同规则数对 CVQ 判定算法的测试结果

2.6.4 HSQ 判定算法执行效率分析

我们依旧从三个角度考量了 HSQ 判定算法的执行效率：

（1） 与为优化的时效性判定算法对比：对比两种算法再不同数据量下的执行效率差距；

（2） 数据量：分别以 1 亿、5 亿、10 亿条记录的数据表作为输入，grade 和 status 上分别 10 条规则，考察数据量对算法执行效率的影响；

（3） 规则数：分别在属性 grade 和 status 上以 10、50、100 条规则的 10 亿条记录的数据表作为输入，从而分别考察参考属性和修复属性上规则数对算法执行效率的影响；

（4） 不同维度的用户查询：针对用户对数据库不同维度的时效性查询请求，给出 HSQ 判定算法的测试用例，它们可以分别用如下 SQL 语句表达：

a) SQL1：select single_hsq(vId, grade) as hsq from table where eId = 1，表示对实体 eId = 1'在属性 grade 上的 HSQ 时效性判定；

b) SQL2：select multi_hsq(hsq1, hsq2) as hsq from (
select single_hsq(vId, grade) as hsq1, single_hsq(vId, status) as hsq2
from table where eId = 1)，表示对实体 eId = 1 在属性 grade 和 status 上的 HSQ 时效性判定；

c) SQL3：select eId, multi_hsq(hsq1, hsq2) as hsq from (
select eId, single_hsq(vId, grade) as hsq1, single_hsq(vId, status) as hsq2
from table where eId >= 1 and eId <= 10 group by eId)，表示对实体 eId = 1,…, 10 在属性 grade 和 status 上的 HSQ 时效性判定；

d) SQL4：select db_hsq(hsq) from (
select eId, multi_hsq(hsq1, hsq2) as hsq from (
select eId, single_hsq(vId, grade) as hsq1, single_hsq(vId, status) as hsq2
from table group by eId)，表示对整个数据库在属性 grade 和 status 上的 HSQ 时效性判定；

基于数据量对 HSQ 判定算法的考量结果如图所示，可以看出随着数据量的增大，HSQ 时效性判定算法的执行时间呈线性增长。通过对比发现，其与 CVQ 判定算法的实验结果基本相同，这是因为判定算法的大部分时间耗费在各个分布式节点的单机计算以及节点之间网络传输上，而 CVQ 和 HSQ 的唯一区别仅在于 Master 节点收集到合并后时效图上执行的判定算法。而在时效规则数较少的情况下，时效图上执行的判定算法时间复杂度并不高。

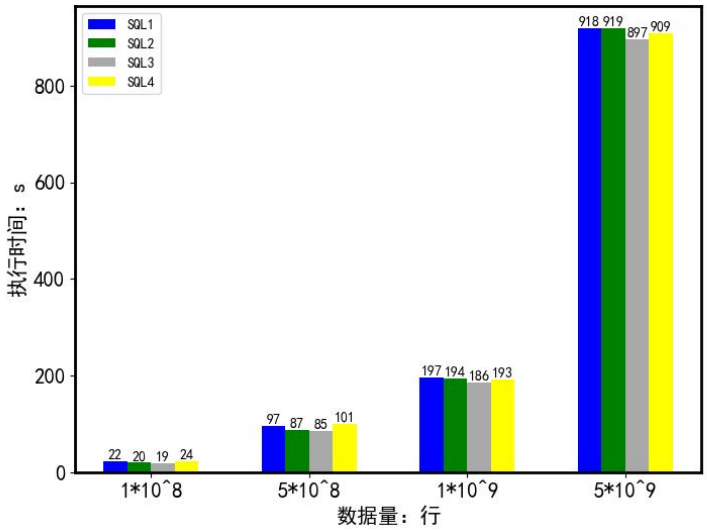


图 2-10 不同数据量对 HSQ 判定算法的测试结果

与未优化的 HSQ 时效性判定算法的实验结果对比图如下，和 CVQ 判定算法一样，未优化的 HSQ 时效性判定算法不适用于大数据量的场景，在数据量过高时会出现 OOM 错误，且执行效率很低。

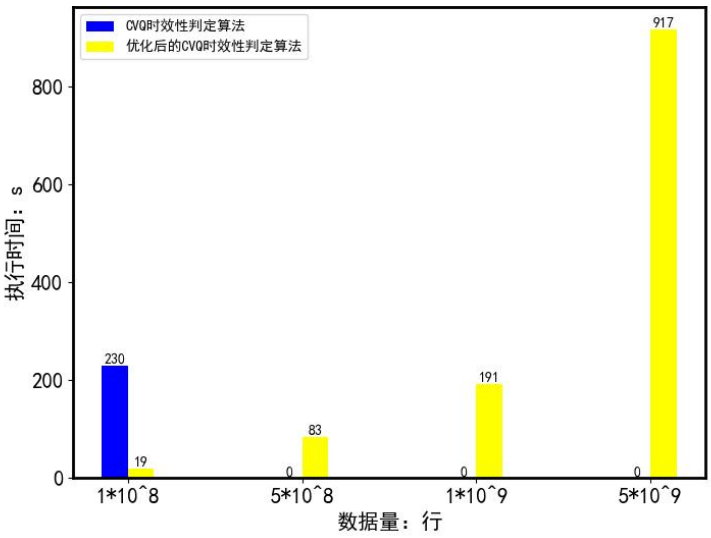


图 2-11 与未优化的 CVQ 判定算法对比结果

基于时效规则数对 HSQ 判定算法的考量结果如下面两个图所示。容易看出，SQL1~SQL4 的执行时间受时效规则数的影响与 CVQ 判定算法基本相同，SQL4 的执行时间受到时效规则数的影响最大，SQL1~SQL3 几乎不受影响；与 CVQ 判定算法不同的地方在于，HSQ 判定算法受规则数的影响比 CVQ 判定算法更大，当规则数增加时，查询 HSQ 时效性所耗费的时间相比查询 CVQ 时效性要更多，这是因为在 Master 节点收集到合并后时效图上执行的判定算法，HSQ 判定算法的时间复杂度要比 CVQ 判定算法高。

另外，对比两张图发现，与 CVQ 判定算法相同，在属性 grade 和 status 上

分别增加时效规则数并没有产生较大差异的影响。

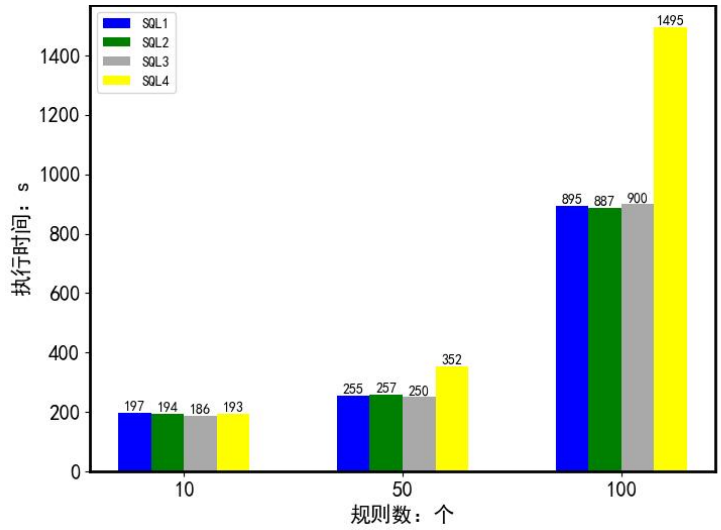


图 2-12 参考属性上不同规则数对 HSQ 判定算法的测试结果

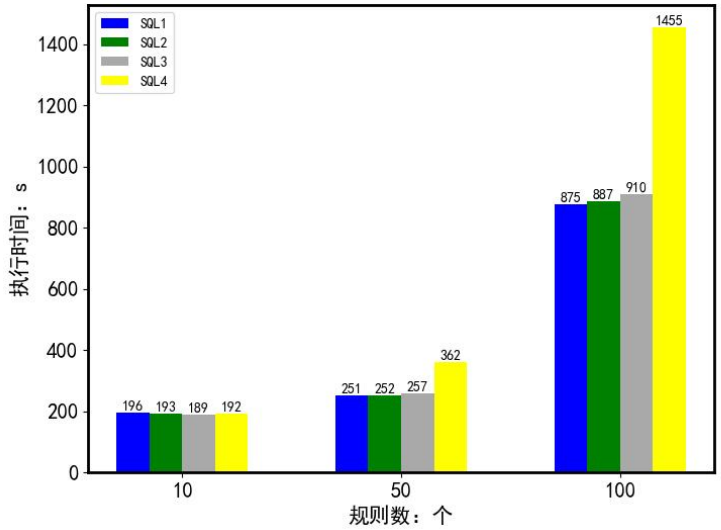


图 2-13 修复属性上不同规则数对 HSQ 判定算法的测试结果

2.7 本章小结

本章针对现有时效性判定算法中的时效图构造算法时间复杂度过高，在海量数据的应用场景中表现不佳的问题，基于时效规则提出了状态图的概念。由于时效图中的顶点与实体的冗余记录一一对应，因此在海量数据的场景中，时效图的大小与数据量呈正相关，这意味着它会非常占用内存空间。从而，时效图的构造不仅时间复杂度高，空间复杂度也很高。而状态图中的顶点与时效规则作用的所有属性值一一对应，时效规则数远远小于实体记录数，因此，状态图占用的内存空间与数据量无关，是常数级别的。状态图相比时效图具有更好的空间复杂度，

非常适合在分布式场景中传输，本章用状态图替代时效图的构造算法，从而大大优化了时效性判定算法在分布式场景中的执行效率。

另外，本文还针对用户对数据库不同维度的查询请求，给出了不同维度的时效性度量方法，基于这些度量能够返回特定实体特定属性上的时效性、特定实体多个属性上的时效性、多个实体多个属性上的时效性以及整个数据库的时效性。基于上述不同维度的时效性度量方法，本章通过分布式框架 SparkSQL 提供的用户自定义聚合函数(UDAF)实现了不同维度的时效性判定算法。通过将时效性判定算法划分成不同部分，封装在 UDAF 的各个算子中，巧妙利用了 SparkSQL 分布式计算模型，使得时效性判定算法具备了分布式框架的诸多优点，包括：对海量数据的扩展性、分布式并行计算的高效、数据存储的容错性等等。

第 3 章　基于时效规则的数据时效性错误修复算法

3.1　引言

上一章针对用户对数据库的时效性查询请求（CVQ、HSQ），设计了不同维度的时效性判定算法，获取查询请求的结果是否过时以及过时的严重程度。在查询请求的结果具有良好时效性的前提下，用户可放心的对数据库进行最新值查询或历史值序列查询。

然而，数据库可能存在着某些时效性关联错误，这些错误可能会影响时效性判定算法的结果，导致为用户提供了错误的判定信息。下面以最新值查询为例，结合图中的数据表，介绍时效性关联错误会带来的严重后果。

例 3.1 考虑表 3-1 中的数据表$student$，其描述了学生Tom在过去几年的基本情况，其数据模式为$R=(eId,vId,Name,Age,Grade,Status)$，分别表示实体 id、冗余记录 id、姓名、年龄、年级、婚姻状况。

表 3-1 数据表 student

eId	vId	Name	Age	Grade	Status
1	1	Tom	18	HighSchool	Single
1	2	Tom	19	Bachelor	Single
1	3	Tom	19	PHD	Single
1	4	Tom	20	Master	Married

有如下时效规则：

（1） 一个人的年级只能由$HighSchool$变成$Bachelor$，再变成$Master$，最后变为 PHD；

（2） 一个人的婚姻状况只能由$Single$变成$Married$，再变成$Divorced$。

考虑时效性查询$Q=(EntitySet,AttrSet,Type)$，$EntitySet=\{1\}$，$AttrSet=\{Grade,Status\}$，$Type=CVQ$，其查询实体 1 在属性$Grade,Status$上的最新值。根据表中数据，容易推断结果为$(Grade,Status)=(PHD,Married)$。然而当查询结果返回给用户后，用户想知道在属性$Grade,Status$上具有最新值的记录，其 Age 属性值是多少时，执行如下SQL语句：

$$select\ age\ from\ student\ where\ grade=PHD\ and\ status=married;$$

根据表中数据容易知道，上述SQL语句返回的结果为空。因为数据表中并没有满足$(Grade,Status)=(PHD,Married)$的元组。造成这种错误的原因是，元组

$t_1(1,3,Tom,19,PHD,Single)$ 和 $t_2(1,4,Tom,20,Master,Married)$ 之间在属性 $Grade$和$Status$上存在时效性关联错误，因为根据属性$Grade$上的时效规则可知，t_1与t_2的时序关系为$t_1 > t_2$，而根据属性 Status 上的时效规则可知，t_1与t_2的时序关系为$t_1 < t_2$，从而产生了矛盾。

从上面的例子可以看出，数据时效性关联错误会严重影响时效性判定算法的判定结果。本章将深入研究数据时效性关联错误，分析实体冗余记录是否具有时效性关联错误的判定原理，并给出整个数据库是否有时效性关联错误的判定算法。然而仅仅判定数据库是否有时效性关联错误是不够的，为了数据时效性判定算法的正确计算，还需要对实体的冗余记录进行时效性关联错误修复。本章将给出一种时效性关联错误的检测方法，并基于检测方法给出一种高效的时效性关联错误修复算法。

本章各节安排如下：3.2 节定义了数据时效性关联错误，并给出了数据时效性关联错误的判定原理及其证明，进一步提出了时效性关联错误的判定算法；3.3 节给出了时效性关联错误的检测算法，并基于检测算法给出了时效性关联错误的修复算法；3.4 节针对时效性关联错误修复算法进行了实验设计与分析。

3.2 数据时效性关联错误的判定算法

对于有完整可用时间戳的数据库，时间戳通常记录了数据的生成时间，根据时间戳很容易判断数据记录间的时序关系。

定义 3.1（真实时效图）设有完整可用时间戳的数据集合D，数据模式为$R = (timestamp, Attr_1, \ldots, Attr_n)$，对于数据集合$D$中的任一实体$e$，$S_e = \{t_1, \ldots, t_n\}$为实体$e$的冗余记录集合。根据时间戳可以为实体$e$构造时效图$G_e$，容易知道，它是一个链式的有向图。区别于通过其他属性上的时效规则构造的时效图，称由时间戳构造的时效图为真实时效图，用$G_{e,real}$表示。

然而对于缺少完整可用时间戳的数据库，无法通过时间戳得到实体记录的真实时效图。但数据记录的生成时间是客观存在的，它们之间的时序关系也是客观存在的，因此实体记录的真实时效图$G_{e,real}$是客观存在的。

定义 3.2（时效性关联错误）设数据集合D缺少可用的时间戳，数据模式为$R = (eId, vId, Attr_1, \ldots, Attr_n)$，$Attr_i$为$R$中任一属性，对于数据集合$D$中的任一实体$e$，$S_e = \{t_1, \ldots, t_n\}$是数据集合$D$中描述实体$e$的所有数据记录，有作用在属性$Attr_i$上的时效规则集$Rule$。若存在实体记录$t_i, t_j \in S_e$，根据时效规则集$Rule$有$t_i <_{Attr_i} t_j$，而真实时效图$G_{e,real}$所表达的记录$t_i, t_j$的时序关系为$t_i > t_j$，称这种矛盾为实体$e$在属性$Attr_i$上的时效性关联错误。

例 3.2 考虑表 3-2 中带有时间戳的数据表$student$，其描述了学生Tom在过去几年的基本情况，其数据模式为$R=(timestemp, eId, vId, Name, Grade)$，分别表示时间戳、实体 id、冗余记录 id、姓名、年级。

表 3-2 带有时间戳的数据表 student

timestemp	eId	vId	Name	Grade
20220424083005	1	1	Tom	HighSchool
20220424083015	1	2	Tom	Bachelor
20220424083025	1	3	Tom	PHD
20220424083035	1	4	Tom	Master

上表中的实体Tom根据时间戳可以得到如下真实时效图：

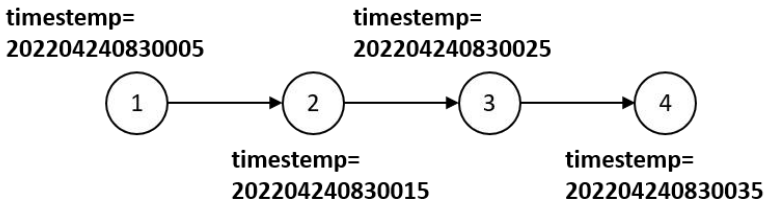


图 3-1 表 student 中实体 Tom 的真实时效图

而上表中的实体Tom根据属性$Grade$上的时效规则可以得到如下时效图：

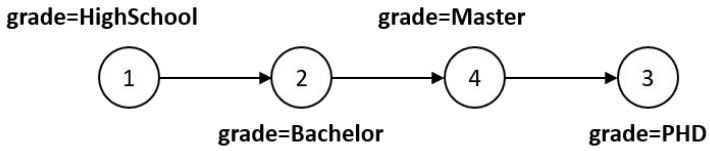


图 3-2 表 student 中实体 Tom 在属性 Grade 上的时效图

从图 3-1 和图 3-2 可以看出，实体Tom在属性$Grade$上存在时效性关联错误。

然而，由于缺少完整可用时间戳的数据库无法构造真实时效图$G_{e,real}$，从而无法根据定义判定实体冗余记录是否存在时效性关联错误。但依然可以寻求近似的数据时效性关联错误判定方法。

3.2.1 数据时效性关联错误判定原理

假定时效规则是绝对可靠的，即在数据记录准确无误的前提下，通过时效规则推导出的数据记录间的时序关系符合客观存在的真实时序关系。以此假定为前提，针对时效性关联错误的定义，推导出数据时效性关联错误的判定原理。

定理 3.1 设数据集合D的数据模式为$R=(eId, vId, Attr)$，对于任一实体e，$S_e=$

$\{t_1,\ldots,t_n\}$是数据集合D中描述实体e的冗余记录集合，有作用在属性$Attr$上的时效规则集$Rule$。设当实体记录在属性$Attr$上没有时效性关联错误时，在属性$Attr$上通过时效规则集$Rule$得到的时效图$G_{e,Attr}$中，任意有向边(v_i,v_j)，在真实时效图$G_{e,real}$中也一定能找到由v_i到v_j的路径。

证明：根据时效性关联错误的定义，由实体记录在属性$Attr$上没有时效性关联错误可得，对于$\forall r\in Rule$，$r\Rightarrow t_i>_{Attr}t_j$，$\nexists$路径$v_j,\ldots,v_i\in G_{e,real}$，其中$t_i$对应顶点$v_i$，$t_j$对应顶点$v_j$；由于真实时效图$G_{e,real}$是一条链式的有向图，有$\forall r\in Rule$，$r\Rightarrow t_i>_{Attr}t_j$，$\exists$路径$v_i,\ldots,v_j\in G_{e,real}$；即$\forall(v_i,v_j)\in G_{e,Attr}.E$，$\exists$路径$v_i,\ldots,v_j\in G_{e,real}$。

引理 3.1 根据时效图是DAG有向无环图的性质，已知$G_{e,Attr}(V,E)$、$G_{e,real}(V,E)$是DAG图，则实体记录在属性$Attr$上没有时效性关联错误的充分必要条件是$G'(V,E)$是DAG图，其中$V=G_{e,Attr}.V=G_{e,real}.V$，$E=G_{e,real}.E\cup G_{e,Attr}.E$。

证明：“⇒”反证法，若$G'(V,E)$有环，即$\exists$路径$v_i,\ldots,v_j$和路径$v_j,\ldots,v_i\in G'(V,E)$；由于$G_{e,Attr}(V,E)$、$G_{e,real}(V,E)$无环，则有向环$v_i,\ldots,v_j,\ldots,v_i$中的有向边分别属于$G_{e,Attr}(V,E)$、$G_{e,real}(V,E)$；不失一般性，对于$\forall(v_k,v_{k'})\in v_i,\ldots,v_j,\ldots,v_i$且$(v_k,v_{k'})\in G_{e,Attr}(V,E)$，由定理 1 可知：$\exists$路径$v_k,\ldots,v_{k'}\in G_{e,real}$；于是将有向环$v_i,\ldots,v_j,\ldots,v_i$中所有有向边$(v_k,v_{k'})\in G_{e,Attr}(V,E)$替换为路径$v_k,\ldots,v_{k'}\in G_{e,real}$，将在$G_{e,real}$中得到一条有向环，矛盾。

“⇐”反证法，若实体记录在属性$Attr$上有时效性关联错误，根据时效性关联错误的定义，$\exists(v_i,v_j)\in G_{e,Attr}(V,E).E$，$G_{e,real}$中有一条由$v_j$到$v_i$的路径；从而$G'(V,E)$有环，矛盾。

接下来考虑真实数据集上可能存在的时效性关联错误的情况，某个属性$Attr$对应时效图$G_{e,Attr}$中的某些顶点间的偏序关系可能违背真实时效图$G_{e,real}$中相应顶点之间的偏序关系，从而导致$G'(V,E)$有环，其中$E=G_{e,real}.E\cup G_{e,Attr_i}.E$。于是根据引理 3.1 的否命题，有如下引理 3.2：

引理 3.2（时效性关联错误判定定理） 已知$G_{e,Attr}(V,E)$、$G_{e,real}(V,E)$是DAG图，对于图$G'(V,E)$，其中$V=G_{e,Attr}.V=G_{e,real}.V$，$E=G_{e,real}.E\cup G_{e,Attr}.E$，$G'(V,E)$有环的充分必要条件是实体记录在属性$Attr$上存在时效性关联错误。

证明：略，引理 3.1 的否命题。

理想情况下，可以通过上述时效性关联错误判定定理，根据$G'(V,E)$是否有环，判定实体记录在属性$Attr$上存在时效性关联错误。但由于实际场景中，真实时效图$G_{e,real}$的缺失，上述时效性关联错误判定定理没有实际应用的价值。于是，尝试将时效性关联错误判定定理扩展到所有属性$Attr_1,\ldots,Attr_n$上，有

引理 3.3 已知$G_{e,Attr_1}(V,E),\ldots,G_{e,Attr_n}(V,E),G_{e,real}(V,E)$是$DAG$图，有图$G'(V,E)$，其中$V=G_{e,Attr_i}.V=G_{e,real}.V$，$E=\bigcup_{i=1,\ldots,n}(G_{e,Attr_i}.E)\cup G_{e,real}.E$，$G'(V,E)$有环的充分必要条件是实体记录在属性$Attr_1,\ldots,Attr_n$中的一个或多个上存在时效性关联错误。

证明：“⇒”反证法，若假设实体记录在属性$Attr_1,\ldots,Attr_n$上没有时效性关联错误；则对于任一$Attr_k$，$k\in\{1,\ldots,n\}$，从$G'(V,E)$的边集E中删去$G_{e,Attr_k}.E$后得到的$G''(V,E)$，$E=\bigcup_{i=1,\ldots,n且i\neq k}(G_{e,Attr_i}.E)\cup G_{e,real}.E$仍然有环；因为根据定理 1，时效图$G_{e,Attr_k}$中任意有向边$(v_i,v_j)$，在真实时效图$G_{e,real}$中一定能找到由$v_i$到$v_j$的路径，所以$G'(V,E)$中的环不会因为$G_{e,Attr_k}$中任意边$(v_i,v_j)$的删去而断开；同理，从$G'(V,E)$的边集$E$中依次删去所有属性$Attr_1,\ldots,Attr_n$对应时效图的边集$\bigcup_{i=1,\ldots,n}(G_{e,Attr_i}.E)$得到的时效图$G'''(V,E)$，$V=G_{e,Attr_i}.V=G_{e,real}.V$，$E=\bigcup_{i=1,\ldots,n}(G_{e,Attr_i}.E)\cup G_{e,real}.E-\bigcup_{i=1,\ldots,n}(G_{e,Attr_i}.E)$仍然有环；容易知道，$\bigcup_{i=1,\ldots,n}(G_{e,Attr_i}.E)\cup G_{e,real}.E-\bigcup_{i=1,\ldots,n}(G_{e,Attr_i}.E)\subseteq G_{e,real}.E$；而$DAG$图去除有限条边后还是$DAG$图，因此时效图$G'''(V,E)$是$DAG$图，矛盾。

“⇐”根据时效性关联错误的定义可知，$G'(V,E)$有环。

上述引理 3.3 依然需要真实时效图$G_{e,real}$的参与。于是，根据DAG图去除有限条边后还是DAG图的性质，将真实时效图$G_{e,real}$去除，有如下引理 3.4：

引理 3.4 已知$G_{e,Attr_1}(V,E),\ldots,G_{e,Attr_n}(V,E)$是$DAG$图，若$G'(V,E)$有环，其中$V=G_{e,Attr_i}.V=G_{e,real}.V$，$E=\bigcup_{i=1,\ldots,n}(G_{e,Attr_i}.E)$，则实体记录在属性$Attr_1,\ldots,Attr_n$中的一个或多个上存在时效性关联错误。

证明：根据引理 3.3 的否命题：实体记录在属性$Attr_1,\ldots,Attr_n$上没有时效性关联错误$\Rightarrow G'(V,E)$是DAG图，其中$V=G_{e,Attr_i}.V=G_{e,real}.V$，$E=\bigcup_{i=1,\ldots,n}(G_{e,Attr_i}.E)\cup G_{e,real}.E$；那么根据$DAG$图去除有限条边后还是$DAG$图的性质，上述推导式变为：实体记录在属性$Attr_1,\ldots,Attr_n$上没有时效性关联错误$\Rightarrow G'(V,E)$是$DAG$图，其中$V=G_{e,Attr_i}.V=G_{e,real}.V$，$E=\bigcup_{i=1,\ldots,n}(G_{e,Attr_i}.E)$；于是，再转换为逆否命题就得到引理 3.4。

上述引理 3.4 将真实时效图$G_{e,real}$排除在外，对时效性关联错误的判定有了实际的应用价值。但是，相比于引理 3.2 的时效性关联错误判定定理，引理 3.4 存在着缺陷。引理 3.2 中$G'(V,E)$有环和存在时效性关联错误是充分必要条件，而引理 3.4 中$G'(V,E)$有环只是存在时效性关联错误的充分条件，不是必要条件。这意味着，存在$G'(V,E)$是DAG图，但仍存在时效性关联错误的情况。

3.2.2 数据时效性关联错误判定算法

考虑到真实时效图仅仅是一个客观存在但无法获取的概念，而各个属性上的时效图是容易构造的，因此通过引理 3.4 判定实体冗余记录是否存在时效性关联错误是实际可行的。根据引理 3.4，下面给出了两个属性上的数据时效性关联错误判定算法。对于n个属性上的时效性关联错误判定，只需两两判定即可。算法的基本思路是对合并后的时效图执行拓扑排序算法，记录遍历到的顶点的数量$count$，最后在结束遍历后，判定$count$是否等于时效图的顶点集大小，若不等于则时效图有环，出现异常。

算法 2-6 数据时效性关联错误判定算法

输入：时效图$G_{e,Attr_1}, G_{e,Attr_2}$

输出：isError

```
1  isError ← false; G_e ← null; count ← 0;   queue ← null;
2  for (v_i, v_j) ∈ G_{e,Attr_2}.E do
3     if((v_i, v_j) ∉ G_{e,Attr_1}.E)
4        G_{e,Attr_1}.E ← G_{e,Attr_1}.E∪{(v_i, v_j)};
5     end if
6  end for
7  G_e ← G_{e,Attr_1};
8  for v ∈ G_e.V do
9     if(v.inDegree == 0)
10       queue.add(v);
11    end if
12 end for
13 while(queue.nonEmpty)
14    v ← queue.first;
15    for v_i ∈ v.nextVertex do
16       if(v_i.inDegree == 0)
17          queue.add(v_i);
18          count ++;
19       end if
20    end for
21 end while
22 if(count ≠ G_e.size)
23    isError ← true;
24 end if
```

25 ***return*** isError;

算法的输入为两个时效图$G_{e,Attr_1}, G_{e,Attr_2}$；输出为时效性关联错误判定结果isError。算法第1行初始化时效性关联错误标记为false、合并的时效图$G_e \leftarrow$ null、计数器为0，队列queue为空；第2行到第7行，两个时效图$G_{e,Attr_1}, G_{e,Attr_2}$合并为$G_e$；第8行到第12行，将$G_e$中入度为0的顶点添加到队列queue中；第13行到第21行，执行拓扑排序算法，每移除一个入度为0的顶点，计数器加1；第22行到第25行，若计数器不等于时效图的顶点数，则返回isError $\leftarrow$ true。

若顶点集为n，边集为m，上述算法的时间复杂度为$O(m+n)$，空间复杂度为$O(n)$。

3.3 数据时效性关联错误的修复算法

通过时效性关联错误判定算法可以判定数据库中实体的冗余记录是否存在时效性关联错误。当存在时效性关联错误时，为了能够继续对数据库中的数据记录进行时效性判定，从而为用户的时效性查询请求提供良好准确的时效性参考，需要对数据库中存在时效性关联错误的数据记录进行修复。

尽管上一小节给出了针对实体的时效性关联错误判定算法，其通过合并各个属性上的时效图，判断合并后的时效图是否有环来判定是否存在时效性关联错误。但该算法仅能判定某一实体的所有冗余记录中是否存在关联错误，无法判断关联错误出现在哪一个实体记录的哪一个属性上，即无法给出错误出现的位置，从而加以修复。为了能够判断某一实体的哪些记录出现了时效性关联错误，需要研究新的检测方法，检测实体冗余记录中时效性关联错误出现的位置，从而基于检测方法给出时效性关联错误的修复算法。

考虑数据集合D，数据模式为$R=(eId, vId, Attr_1, Attr_2)$，对于实体$e$，$S_e=\{t_1,\ldots,t_n\}$是数据集合$D$中描述实体$e$的冗余记录集合。已知实体数据在属性$Attr_1$上不存在时效性关联错误，而通过时效性关联错误判定算法，得知在属性$Attr_1, Attr_2$上存在时效性关联错误，因此，需要修复实体在属性$Attr_2$上的属性值。本节将基于以上问题假设，设计面向海量数据场景的时效性关联错误修复算法。

3.3.1 时效性关联错误检测

根据时效性关联错误的定义，考虑如下时效性关联错误最小修复模型：

定义3.3（时效性关联错误最小修复模型）已知有数据集合D，数据模式为$R=$

$(eId, vId, Attr_1)$，D中仅有 1 个实体e，其冗余记录集合包含 2 条数据记录$S_e = \{t_1, t_2\}$。两条记录的真实生成顺序为$t_1 < t_2$，然而根据属性$Attr_1$上的时效规则推导得到$t_1 >_{Attr_1} t_2$。因此，实体记录在属性$Attr_1$上存在时效性关联错误，需要修复属性$Attr_1$上的属性值。

例 3.3 下表中的数据集合就是一个时效性关联错误的最小修复模型，其中时间戳可知真实生成顺序为$t_1 < t_2$，而根据属性$Grade$上的时效规则可推出$t_1 >_{Grade} t_2$，从而需要修复属性$Grade$上的时效性关联错误。

表 3-3 最小修复模型

timestemp	eId	vId	Name	Grade
20220424083005	1	1	Tom	Bachelor
20220424083015	1	2	Tom	HighSchool

由于没有时间戳，实体记录真实的生成顺序是未知的，需要对上述最小模型增加一个属性$Attr_2$，并已知实体记录在其中一个属性上的属性值没有时效性关联错误。于是，有如下近似的时效性关联错误修复模型：

定义 3.4（近似的时效性关联错误最小修复模型）数据集合D的数据模式为$R = (eId, vId, Attr_1, Attr_2)$，$D$中仅有 1 个实体$e$，其冗余记录集合包含 2 条数据记录$S_e = \{t_1, t_2\}$，已知实体记录在属性$Attr_1$上没有时效性关联错误。两条记录根据属性$\mathrm{Attr}_1$上的时效规则推导有$t_1 <_{Attr_1} t_2$，而根据属性$Attr_2$上的时效规则推导得到$t_1 >_{Attr_2} t_2$，因此，实体记录在属性$Attr_2$上存在时效性关联错误，需要修复属性$Attr_1$上的属性值。

例 3.4 下表中的数据集合是一个近似的时效性关联错误最小修复模型，已知属性$Grade$上没有时效性关联错误，根据属性$Grade$上的时效规则可推出$t_1 <_{Grade} t_2$；然而，属性$Status$上的时效规则可推出$t_1 <_{Status} t_2$。从而需要修复属性$Status$上的时效性关联错误。

表 3-4 近似的最小修复模型

eId	vId	Name	Grade	Status
1	1	Tom	HighSchool	Married
1	2	Tom	Bachelor	Single

上述近似模型本质上是用属性$Attr_1$上时效图所表达的时序关系，来近似实体记录之间生成的真实时序关系。

根据近似的时效性关联错误修复最小模型，用符号$<>_{Attr}$表示属性$Attr$上无法根据时效规则判断时序关系，得到如下**时效性关联错误检测方法：**

对于数据集合 D，数据模式为$R = (eId, vId, Attr_1, Attr_2)$，对于实体$e$，$S_e =$

$\{t_1, t_2\}$。已知记录t_1, t_2在属性$Attr_1$上没有时效性关联错误，根据属性$Attr_1$上的时效规则推导有$t_1 <_{Attr_1} t_2$。对于记录t_1, t_2在属性$Attr_2$上值在时效规则下的时序关系，有如下三种情况：

（1） $t_1 <_{Attr_2} t_2$：则t_1, t_2在属性$Attr_2$上不存在时效性关联错误，无需修复；

（2） $t_1 >_{Attr_2} t_2$：则t_1, t_2在属性$Attr_2$上存在时效性关联错误，需要修复；

（3） $t_1 <>_{Attr_2} t_2$：则t_1, t_2在属性$Attr_2$上不存在时效性关联错误，无需修复。

在第三种情况中，记录t_1, t_2在属性$Attr_2$上的时效图$G_{e,Attr_2}$中对应的顶点v_1, v_2之间没有路径，因此，时效图$G_{e,Attr_2}, G_{e,Attr_2}$合并后不会产生环，从而$t_1, t_2$在属性$Attr_2$上不存在时效性关联错误。

3.3.2 时效性关联错误修复算法

应用上一小节基于最小模型讨论的时效性关联错误检测方法，可以针对大数据量的、分布式存储的、多实体的数据表，设计高效的时效性关联错误修复算法。

考虑数据集合D，数据模式为$R = (eId, vId, Attr_1, Attr_2)$，对于实体$e$，$S_e = \{t_1, \ldots, t_n\}$是数据集合$D$中描述实体$e$的冗余记录集合。已知实体数据在属性$Attr_1$上不存在时效性关联错误，将$S_e$中的记录按$Attr_1$上的时效图$G_{e,Attr_1}$进行拓扑排序，较旧的记录在前，较新的记录在后。这么做的好处是：对于排在任一记录t_i后面的记录t_j，一定不存在$t_i >_{Attr_1} t_j$的情况，即仅有如下两种情况：

（1）$t_i <_{Attr_1} t_j$，即时效图$G_{e,Attr_1}$上存在由v_i到v_j的路径；

（2）$t_i <>_{Attr_1} t_j$，即时效图$G_{e,Attr_1}$上不存在由v_i到v_j的路径。

称上述排序操作为第一趟排序。考虑到可能存在大量的数据记录在属性$Attr_1$上的值相等，将数据记录按$Attr_1$上的属性值分组，并继续对每个分组内的数据记录按$Attr_2$上的时效图$G_{e,Attr_2}$进行拓扑排序，较旧的记录在前，较新的记录在后。这么做的好处是：对于分组内的任意两条记录t_1, t_2，在属性$Attr_1$上有$t_1.Attr_1 = t_2.Attr_1$，在属性$Attr_2$上一定不存在$t_1 >_{Attr_2} t_2$的情况，这就保证了分组内的任意两条记录不会触发时效性关联错误判定生效。称上述组内排序为第二趟排序。

两趟排序算法的主要思路为：首先初始化$Attr_1$、$Attr_2$对应的状态图，在状态图的每个顶点中封装一个数组用来存放相应属性值的所有实体记录；然后，遍历$S_e = \{t_1, \ldots, t_n\}$中的记录并存放到对应$Attr_1$值的顶点数组中；对$Attr_1$上的状态图做拓扑排序，每遍历到一个顶点就输出顶点中存放的记录；接着对$Attr_1$属性值分组，每个分组中的记录单独再存放到$Attr_2$上的状态图做拓扑排序，分组内输出拓扑排序的记录；从而整个S_e中的记录在$Attr_1$实现了拓扑有序，相同$Attr_1$

值的记录在$Attr_2$上实现了拓扑有序。两趟排序算法如下：

算法 2-6 两趟排序算法

输入：$S_e=\{t_1,\ldots,t_n\}$、$Attr_1$、$Attr_2$

输出：S_e

```
1  初始化Attr1、Attr2对应的状态图G_{e,Attr1}、G_{e,Attr2};
2  初始化 tmpArray ← null; resultArray ← null;
3  for t_i ∈ S_e do
4     v ← G_{e,Attr1}.get(t_i.Attr_1);
5     v.tupleArray.add(t_i);
6  end for
7  while(G_{e,Attr1}.nonEmpty)
8     v ← G_{e,Attr1}中任一入度为 0 的顶点;
9     tmpArray.add(v.tupleArray);
10 end while
11 tmpArray.groupBy(t => t.Attr_1)
12 foreach group G_i do
13    初始化Array_i ← null
14    for t_i ∈ G_i do
15       v ← G_{e,Attr2}.get(t_i.Attr_2);
16       v.tupleArray.add(t_i);
17    end for
18    while(G_{e,Attr1}.nonEmpty)
19       v ← G_{e,Attr1}中任一入度为 0 的顶点;
20       Array_i.add(v.tupleArray);
21    end while
22    resultArray.add(Array_i);
25 end foreach
26 S_e ← resultArray
27 return S_e;
```

在完成两趟排序操作后，顺序遍历记录$t_1,\ldots,t_n$，判断相邻的两条记录t_i,t_{i+1}在属性$Attr_2$上的值在时效规则下的时序关系，当$t_i<_{Attr_2}t_{i+1}$或$t_i<>_{Attr_2}t_{i+1}$时，t_i,t_{i+1}在属性$Attr_2$上不存在时效性关联错误，无需修复；当$t_i>_{Attr_2}t_{i+1}$时，t_i,t_{i+1}在属性$Attr_2$上存在时效性关联错误，需要修复。

当需要进行时效性关联错误修复时，已知$t_i<_{Attr_1}t_{i+1}$或$t_i<>_{Attr_1}t_j$，而$t_i>_{Attr_2}t_{i+1}$，有三种修复方式：

（1） 将$t_i.Attr_2$用$t_{i+1}.Attr_2$替换。这种修复方式存在的问题是$t_{i+1}.Attr_2$

的值过于陈旧，导致产生副作用，即t_{i+1}可能$<_{Attr_2}t_i$之前的记录；

（2） 将$t_{i+1}.Attr_2$用$t_i.Attr_2$替换。这种修复方式存在的问题是$t_i.Attr_2$的值太新，导致产生副作用，即t_i可能$>_{Attr_2}t_{i+1}$之后的记录；

（3） 将$t_i.Attr_2$、$t_{i+1}.Attr_2$用$t_{i-1}.Attr_2$替换。这么做的好处是规避了上述两种方式的副作用，但弊端在于其修复了两个记录值，然而在海量数据中错误数据毕竟仅占极少数。

本着尽量少的修复数据的原则，上述修复方式的优先级为方式 1>方式 2>方式 3。为了在一定程度上规避方式 1 和方式 2 的副作用，借助以下两个额外参数来辅助判定：

1. 滑动窗口：用一个参数 left 记录窗口的左边界，另一个参数 right 记录窗口的右边界。left 和 right 初始指向第一条记录，窗口的更新规则如下：

(1) $t_i<>_{Attr_2}t_{i+1}$时，left 不变，right + 1；

(2) $t_i<_{Attr_2}t_{i+1}$时，

① 若窗口内所有记录$<_{Attr_2}t_{i+1}$，且“向后多看一眼”没有出现$t_{i+1}>_{Attr_2}t_{i+2}$需要修复的情况，则 left ← i + 1，right ← i + 1；

② 否则，left 不变，right + 1；

滑动窗口的作用在于，当使用修复方式 1 将$t_i.Attr_2$用$t_{i+1}.Attr_2$替换时，t_{i+1}必须确保$t_{i+1}>_{Attr_2}$或$<>_{Attr_2}$窗口内所有记录，从而规避了方式 1 的副作用。

2. 最大距离$maxDist$：两条记录t_i, t_{i+1}之间的距离定义为，记录在$Attr_2$上时效图对应顶点v_i, v_{i+1}之间的路径距离，若不存在路径，则距离为 0。距离的作用在于，当使用修复方式 2 将$t_{i+1}.Attr_2$用$t_i.Attr_2$替换时，必须确保t_i与窗口内所有记录距离的最小值不超过 maxDist，从而一定程度上规避了方式 2 的副作用。当超过最大距离 maxDist 时，则认为t_i极大可能$>_{Attr_2}t_{i+1}$之后的记录，因此需要采用修复方式 3。

基于滑动窗口的修复算法如下：

算法 2-6 基于滑动窗口的修复算法

输入：$S_e = \{t_1, \ldots, t_n\}$、$Attr_1$、$Attr_2$、maxDist

输出：S_e

```
1   left ← 0; right ← 0;
2   for t_i, t_{i+1} ∈ S_e do
3       flag ← false;
4       if(t_i <>_{Attr_2} t_{i+1})
5           right ++;
6       end if
```

```
7      else if(t_i <_Attr2 t_{i+1})
8        for t ∈ S_e.range(left,right) do
9          if(t<>_Attr2 t_{i+1})  flag ← true;
10       end for
11       if(t_{i+1} >_Attr2 t_{i+2})
12         flag ← true;
13       end if
14       if(flag)  right ++;
15       else  left ← i + 1, right ← i + 1;
16     end else if
17     else
18       for t ∈ S_e.range(left,right) do
19         if(t >_Attr2 t_{i+1})  flag ← true;
20       end for
21       if(flag)
22         t_{i+1}.Attr_2 ← t_i.Attr_2;
23         right ++;
24       end if
25       else
26         t_i.Attr_2 ← t_{i+1}.Attr_2;
27         for t ∈ S_e.range(left,right) do
28           if(t<>_Attr2 t_i)  flag ← true;
29         end for
30         if(flag)
31           right ++;
32         end if
33         else
34           left ← i, right ← i;
35         end else
36       end else
37     end else
38   end for
39   return S_e;
```

通过两趟排序算法和基于滑动窗口的修复算法，可以尽可能少的进行修复操作，使实体冗余记录中的时效性关联错误被修复干净。从而为时效性判定算法提供没有时效性关联错误的数据输入，确保了时效性判定算法输出结果的参考价值。

3.4 实验结果及分析

3.4.1 实验配置

本次实验采用四台机器组成的分布式集群作为硬件环境，单机的硬件配置如下表所示：

表 3-5 单机的硬件配置

项目	硬件
CPU	Intel Core i7-11700T@3.4GHz
GPU	Intel UHD Graphics 750
RAM	DDR4 3200MHz 64GB
ROM	SAMSUNG PM9A1 256GB

软件环境方面，实验使用了 Docker 开源的应用容器引擎在硬件网络的基础上构建了分布式集群环境，表中所示的机器的软件环境：

表 3-6 单机的软件环境

项目	软件
OS	Ubuntu20.04
Java	Open JDK 1.8.0_261
Scala	2.12.13
Docker	20.10.7

基于 Docker 容器，搭建了 Spark 分布式运行环境。这里简单介绍 Spark 计算环境的搭建过程：首先，Spark 镜像选择的是 bitnami 提供的 docker 镜像，并在该镜像的基础上配置 Dockerfile 文件，包装了 HDFS 作为存储层，新的镜像作为后续集群容器创建的统一镜像；然后，通过 Docker Swarm 建立了 Docker 跨主机的通信，并在此基础上通过 Docker Stack 统一部署了 Spark 分布式计算环境，部署的方式只需要编写 Docker Compose yml 配置文件即可。Docker 容器中的环境配置如下：

表 3-7 Docker 容器中的环境环境

项目	软件
OS	bitnami 镜像封装的 OS
Java	Open JDK 1.8.0_261
Scala	2.13.8

Spark	3.2.0
Hadoop	3.3.0

Spark 集群包含 1 个 Master 节点，32 个 Worker 节点，其中每个 Spark Worker 容器内存为 7G，core 为 2，Spark 集群参数配置如下：

表 3-8 Spark 集群参数配置

项目	软件
total executor cores	64
executor memeory	7G
executor cores	2

基于上述实验环境，实验代码采用 scala 编写，软件开发环境为 Intellij IDEA，使用 Maven 管理项目并打 jar 包到集群分布式环境中运行测试结果。

3.4.2 实验数据

我们在构造的虚拟数据集上进行了实验，实验数据包含一个表D，其数据模式为$R=(eId, vId, name, age, city, grade, status)$。已知，属性$grade$没有时效性关联错误，属性$status$存在时效性关联错误，需要对属性$status$进行修复，称属性$grade$为参考属性，属性$status$为修复属性。并针对属性$grade$和$status$，手动构造了时效规则。数据表中共有 1000 个实体，为了方便考量，每个实体的冗余记录数量相同。为了有效分析修复算法的执行效率，针对上述数据表，我们分别构造了 1 亿、5 亿、10 亿条数据文件，并存储在 HDFS 中。对于 10 亿条数据，还针对属性$grade$和$status$上不同规则数的情况，分别构造了 10、50、100 条规则的数据文件。

3.4.3 修复算法执行效率分析

我们从三个角度考量了修复算法的执行效率：

（1） 数据量：分别以 1 亿、5 亿、10 亿条记录的数据表作为输入，$grade$和$status$上分别 10 条规则，考察数据量对算法执行效率的影响；

（2） 规则数：分别在属性$grade$和$status$上以 10、50、100 条规则的 10 亿条记录的数据表作为输入，从而分别考察参考属性和修复属性上规则数对算法执行效率的影响；

（3） 修复率：对 10 亿条记录的数据表随机制造时效性关联错误，分别造

存在 1 万、10 万、100 万、1000 万行时效性关联错误记录的数据表。

考察不同比重时效性关联错误记录对修复算法修复效率的影响。

基于数据量对算法的考量结果如图所示，蓝色圆柱代表算法执行的总时长，绿色圆柱代表排序操作的执行时长，灰色圆柱代表修复操作的执行时长。容易看出，排序操作相比于修复操作几乎不耗费时间，且随数据量的增加，没有明显时间的增加。这是因为在代码实现中，排序操作是通过一个简单的 map 算子，为各个数据记录添加了序号，起到了排序的作用。由于 map 算子是窄依赖算子，计算在单机上进行，计算相比宽依赖算子耗费资源少，计算速度快；而修复操作则直接影响着整个修复算法的执行效率，它通过 groupBy 算子对实体进行分组，然后针对每个实体进行修复。由于分组的过程存在 shuffle 落盘，磁盘 IO 的读写操作是非常耗费时间和资源的。显然，随着数据量的增大，磁盘 IO 耗费的时间会越多，因此修复操作的执行时间越长。综上，时效性修复算法的执行效率直接取决于修复操作的执行效率，且随数据量的增大，执行时间越长。

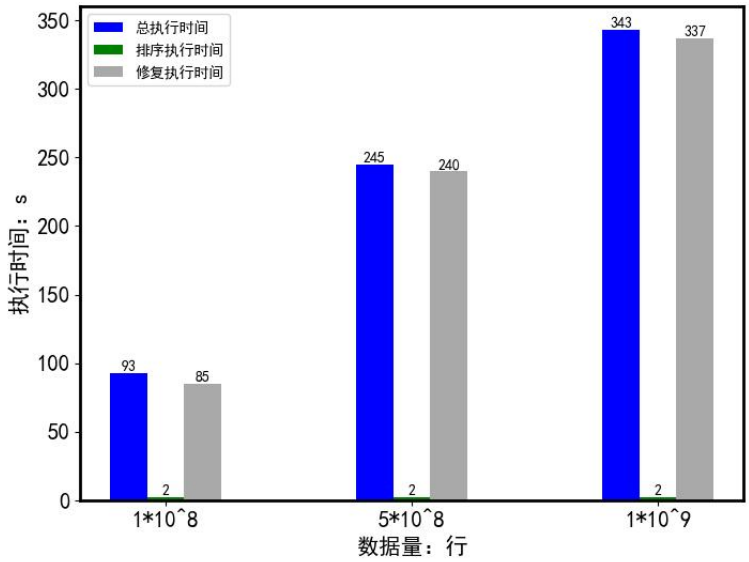


图 3-3 不同数据量对算法的测试结果

基于规则数对算法的考量结果如下两个图所示，蓝色圆柱代表算法执行的总时长，绿色圆柱代表排序操作的执行时长，灰色圆柱代表修复操作的执行时长。数据量选用的是 10 亿条记录的数据表。第一个图是对参考属性上不同规则数（10、50、100）的测试结果，由图中结果可以看出，参考属性的规则数对排序操作执行效率的影响不大，其执行时长相对算法总时长依然微乎其微。但其对修复操作存在一定程度的影响，当规则数上升到 50 以后，执行时长明显大于规则数为 10 的时候，但当规则数继续上升，对修复操作的执行影响有限。

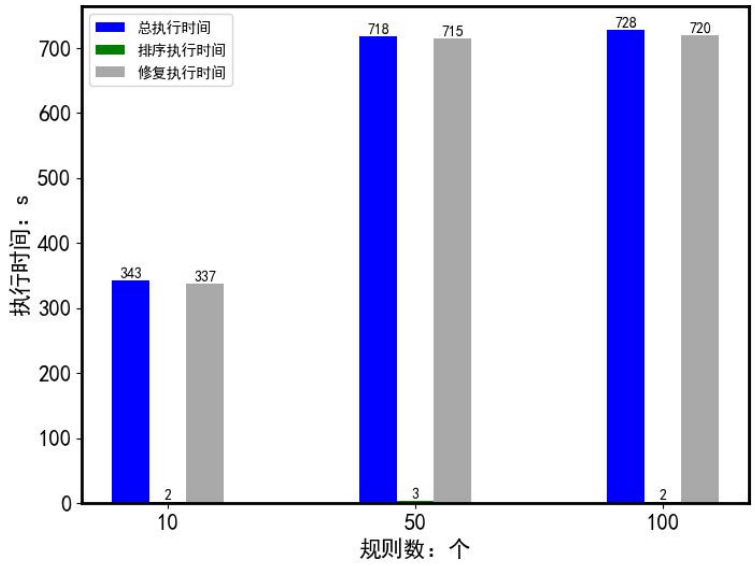


图 3-4 参考属性上不同规则数对算法的测试结果

第二个图是对修复属性上不同规则数（10、50、100）的测试结果，由图中结果可以看出，修复属性的规则数对排序操作执行效率的影响不大，其执行时长相对算法总时长同样微乎其微。但其对修复操作的执行效率起到了决定性的影响，随着修复属性上规则数的增加，修复操作的执行时长也明显上升，从而修复算法的总时长也直线上升。

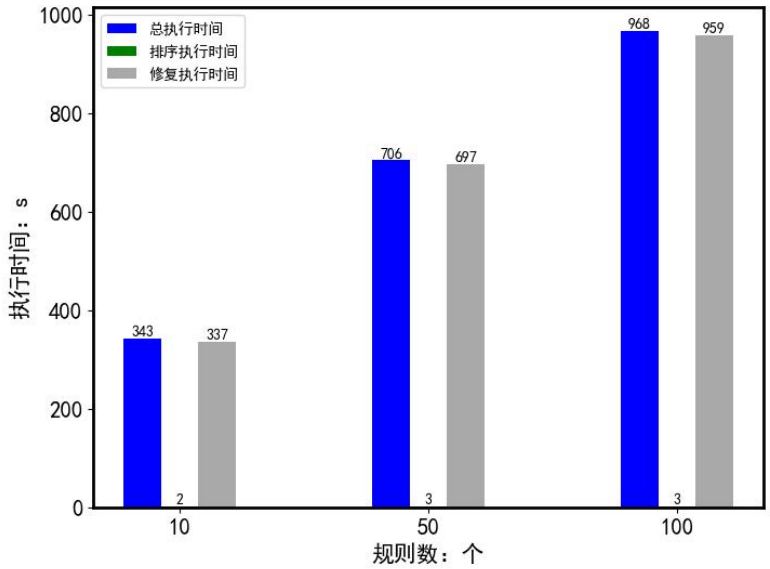


图 3-5 修复属性上不同规则数对算法的测试结果

基于不同时效性关联错误数据比重对算法修复率的考量结果如下图所示，显然，时效性关联错误数据比重越大，修复率越低。且修复率随比重的增大，呈断崖式下降。但在时效性关联错误数据比重低到一定程度时，修复效果还是相当可观的。时效性关联错误与空值错误、零值错误不同，它并不那么常见。所以在大

部分场景下，时效性关联错误数据的比重并不高。因此，时效性关联错误修复算法在大多数错误数据比重偏低的情况下，能取得良好的效果。

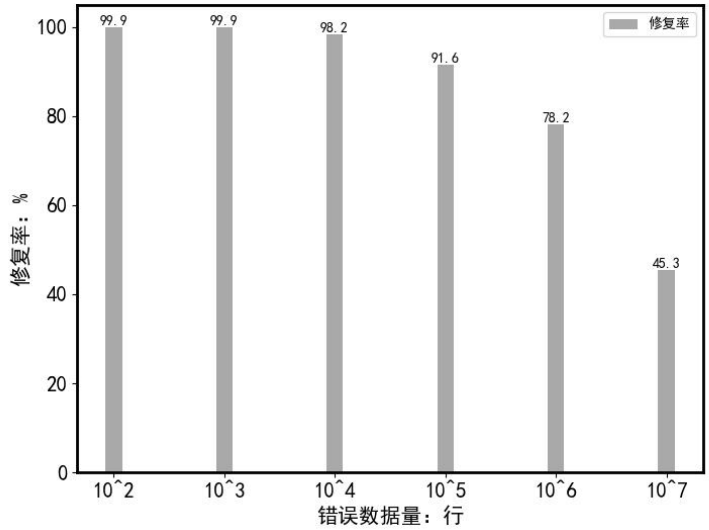


图 3-6 不同时效性关联错误数据比重对算法的测试结果

3.5 本章小结

本章主要针对数据库可能存在的一种时效性关联错误，即实体的冗余记录间根据时效规则推断的时序关系与真实时序关系相矛盾。这种时效性关联错误会严重影响时效性判定问题的结果。因此，本章提出了针对这种时效性关联错误的相关定义、判定理论以及修复方法。

3.2 节首先提出了真实时效图的概念，并基于真实时效图给出了这种时效性关联错误的定义。针对这种时效性关联错误，以时效规则绝对可靠为前提，本节提出了时效性关联错误判定定理，并给出了证明。该定理指出了实体记录在特定属性上存在时效性关联错误的充分必要条件，然而，这一充分必要条件的前提是真实时效图已知。由于真实时效图不可获取，基于时效性关联错误判定定理，继续推导出了引理 3.4，根据该引理可以得到实体记录在特定属性上存在时效性关联错误的充分条件。尽管，充分条件存在缺陷，但依旧是时效性关联错误判定的重要依据。本节的最后，通过引理 3.4 给出了时效性关联错误判定算法。

为了数据时效性判定算法的正确计算，还需要对实体的冗余记录进行时效性关联错误修复。尽管 3.2 节给出了针对实体的时效性关联错误判定算法，但该算法仅能判定某一实体的所有冗余记录中是否存在错误，无法判断错误出现在哪一个实体记录的哪一个属性上。3.3 节提出了近似的时效性关联错误修复最小模型，基于这一模型给出了时效性关联错误检测方法，并基于检测方法给出一种高效的时效性关联错误修复算法。该算法通过先排序后修复的思想，对参考属性和修复

属性进行了拓扑排序预处理，然后对检测到的错误数据进行修复。算法给出了三种修复方法，根据滑动窗口和最大距离两个辅助工具，择优选取合适的修复方法，以避免副作用。

结论

在数据可用性的 5 个维度中，数据时效性扮演的角色至关重要。当前数据时效性领域的研究主要包括时效性判定和时效性修复两个重要的问题。本文根据已有的相关研究成果，改进并提出了一系列大数据时效性管理的关键算法。

本文主要的研究成果如下：

（1） 针对时效图构造算法时间复杂度和空间复杂度过高的问题，基于时效规则提出了状态图的概念，状态图相比时效图具有更好的空间复杂度，非常适合在分布式场景中传输，大大优化了时效性判定算法在分布式场景中的执行效率；

（2） 针对用户对数据库不同维度的查询请求，给出了不同维度的时效性度量方法。判定算法封装为 Spark UDAF 用户自定义函数后，算法的使用具有极高的灵活性，用户可以通过不同的 SQL 从不同维度对数据库进行时效性判定；

（3） 定义了一种特殊的数据时效性关联错误，即实体的冗余记录间根据时效规则推断的时序关系与真实时序关系相矛盾。研究了实体冗余记录是否具有数据时效性关联错误的判定原理以及判定算法；

（4） 研究了一种近似的时效性关联错误修复模型，基于该模型给出了时效性关联错误的检测方法，并基于检测方法提出了一种高效的时效性关联错误修复算法。

从实验结果分析可以看出，本文提出的基于分布式平台的时效性判定算法在分布式场景下具有良好的可扩展性以及和数据量线性相关的时间复杂度。且修复算法在错误数据比重较低的情况下拥有良好的修复率。但是本文仍然有提升的空间：

（1） 状态图仅对单个属性上的时效规则有意义，对于复杂的多属性上的时效规则，还是只能通过暴力法构造时效图；

（2） 修复算法在错误数据比重较高情况下的修复率还有进一步优化的空间。

参考文献

[1] Rahm E, Do H H. Data Cleaning: Problems and Current Approaches.[J]. IEEE Data Eng. Bull., 2000, 23(4): 3–13.

[2] Fan W, Geerts F. Foundations of Data Quality Management[M]. Morgan & Claypool Publishers, 2012.

[3] Elmagarmid A K, Ipeirotis P G, Verykios V S. Duplicate Record Detecti on: A Survey.[J/OL]. IEEE Trans. Knowl. Data Eng., 2007, 19(1): 1–16.

[4] Fan W. Dependencies Revisited for Improving Data Quality. BT - Procee dings of the Twenty-Seventh ACM SIGMOD-SIGACT-SIGART Symposiu m on Principles of Database Systems, PODS 2008, June 9-11, 2008, Va ncouver, BC, Canada[Z](2008).

[5] Johnson T, Dasu T. Data Quality and Data Cleaning: An Overview. BT - Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003[Z/OL] (2003). https://doi.org/10.1145/872757.872875.

[6] 成勇. 大数据的一个重要方面:数据可用性[J]. 山东工业技术, 2014(21): 1.

[7] 郭志懋, 周傲英. 数据质量和数据清洗研究综述[J]. 软件学报, 2002.

[8] Görz Q. An Economics-Driven Decision Model for Data Quality Improve ment - a Contribution to Data Currency[J]. 17th Americas Conference on Information Systems 2011, AMCIS 2011, 2011, 2: 1252–1259.

[9] Heinrich B, Klier M. Assessing Data Currency - a Probabilistic Approac h.[J]. J. Inf. Sci., 2011, 37(1): 86–100.

[10] Heinrich B, Klier M, Kaiser M. A Procedure to Develop Metrics for Cu rrency and Its Application in CRM.[J]. ACM J. Data Inf. Qual., 2009, 1 (1): 5:1-5:28.

[11] Cappiello C, Francalanci C, Pernici B. A Model of Data Currency in M ulti-Channel Financial Architectures. BT - Seventh International Confere nce on Information Quality (ICIQ 2002)[Z](2002).

[12] Cappiello C, Francalanci C, Pernici B. Time Related Factors of Data Ac curacy, Completeness, and Currency in Multi-Channel Information System

s[J]. Journal of Management Information Systems, 2003, 20(3): 71–91.

[13] Heinrich B, Hristova D. A Fuzzy Metric for Currency in the Context of BIG DATA[J]. ECIS 2014 Proceedings - 22nd European Conference on Information Systems, 2014: 1–15.

[14] Fan W, Geerts F, Wijsen J. Determining the Currency of Data[J]. ACM Transactions on Database Systems, 2012, 37(4): 1–46.

[15] Fan W, Geerts F, Tang N, et al. Inferring Data Currency and Consistenc y for Conflict Resolution. BT - 29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013[Z](2 013).

[16] Cao Y, Fan W, Yu W. Determining the Relative Accuracy of Attributes [J]. Proceedings of the ACM SIGMOD International Conference on Mana gement of Data, 2013: 565–576.

[17] 李默涵, 李建中, 高宏. 数据时效性判定问题的求解算法[J]. 计算机学报, 2012, 35(011): 2348–2360.

[18] Li M, Li J. A Minimized-Rule Based Approach for Improving Data Curr ency.[J/OL]. J. Comb. Optim., 2016, 32(3): 812–841.

[19] Li M, Li J, Cheng S, et al. Uncertain Rule Based Method for Determini ng Data Currency.[J/OL]. IEICE Trans. Inf. Syst., 2018, 101-D(10): 2447 –2457.

[20] Fan W, Geerts F, Tang N, et al. Conflict Resolution with Data Currency and Consistency.[J]. ACM J. Data Inf. Qual., 2014, 5(1–2): 6:1-6:37.

[21] Fan W, Ma S, Tang N, et al. Interaction between Record Matching and Data Repairing.[J/OL]. ACM J. Data Inf. Qual., 2014, 4(4): 16:1-16:38.

[22] Chiang F, Miller R J. Discovering Data Quality Rules.[J]. Proc. VLDB Endow., 2008, 1(1): 1166–1177.

[23] Fan W, Geerts F, Li J, et al. Discovering Conditional Functional Depend encies.[J/OL]. IEEE Trans. Knowl. Data Eng., 2011, 23(5): 683–698.

[24] Reingold O. Undirected Connectivity in Log-Space[J]. Journal of the AC M, 2008, 55(4): 1–24.

[25] Golab L, Karloff H J, Korn F, et al. On Generating Near-Optimal Table aux for Conditional Functional Dependencies.[J/OL]. Proc. VLDB Endow., 2008, 1(1): 376–390.

[26] Vries T de, Ke H, Chawla S, et al. Robust Record Linkage Blocking Us ing Suffix Arrays. BT - Proceedings of the 18th ACM Conference on I nformation and Knowledge Management, CIKM 2009, Hong Kong, China, November 2-6, 2009[Z/OL](2009). https://doi.org/10.1145/1645953.164599 4.

[27] Vries T de, Ke H, Chawla S, et al. Robust Record Linkage Blocking Us ing Suffix Arrays and Bloom Filters.[J/OL]. ACM Trans. Knowl. Discov. Data, 2011, 5(2): 9:1-9:27.

[28] Fan W, Jia X, Li J, et al. Reasoning about Record Matching Rules.[J/O L]. Proc. VLDB Endow., 2009, 2(1): 407–418.

[29] Fan W, Geerts F, Tang N, et al. Inferring Data Currency and Consistenc y for Conflict Resolution[J]. Proceedings - International Conference on D ata Engineering, 2013: 470–481.

[30] Kolahi S, Lakshmanan L V S. On Approximating Optimum Repairs for Functional Dependency Violations. BT - Database Theory - ICDT 2009, 12th International Conference, St. Petersburg, Russia, March 23-25, 2009, Proceedings[Z/OL](2009). https://doi.org/10.1145/1514894.1514901.

[31] Chiang F, Miller R J. A Unified Model for Data and Constraint Repair. BT - Proceedings of the 27th International Conference on Data Engine ering, ICDE 2011, April 11-16, 2011, Hannover, Germany[Z/OL](2011). https://doi.org/10.1109/ICDE.2011.5767833.

[32] Beskales G, Ilyas I F, Golab L, et al. On the Relative Trust between In consistent Data and Inaccurate Constraints. BT - 29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013[Z/OL](2013). https://doi.org/10.1109/ICDE.2013.6544854.

[33] Beskales G, Ilyas I F, Golab L. Sampling the Repairs of Functional Dep endency Violations under Hard Constraints.[J/OL]. Proc. VLDB Endow., 2010, 3(1): 197–207.

[34] Fagin R, Kimelfeld B, Kolaitis P G. Dichotomies in the Complexity of Preferred Repairs. BT - Proceedings of the 34th ACM Symposium on Principles of Database Systems, PODS 2015, Melbourne, Victoria, Austra lia, May 31 - June 4, 2015[Z/OL](2015). https://doi.org/10.1145/2745754. 2745762.

[35] Huhtala Y, Kärkkäinen J, Porkka P, et al. TANE: An Efficient Algorith m for Discovering Functional and Approximate Dependencies.[J/OL]. Co mput. J., 1999, 42(2): 100–111.

[36] Novelli N, Cicchetti R. FUN: An Efficient Algorithm for Mining Functio nal and Embedded Dependencies. BT - Database Theory - ICDT 2001, 8th International Conference, London, UK, January 4-6, 2001, Proceeding s.[Z/OL](2001). https://doi.org/10.1007/3-540-44503-X_13.

[37] Abedjan Z, Schulze P, Naumann F. DFD: Efficient Functional Dependenc y Discovery. BT - Proceedings of the 23rd ACM International Conferen ce on Conference on Information and Knowledge Management, CIKM 20 14, Shanghai, China, November 3-7, 2014[Z/OL](2014). https://doi.org/10. 1145/2661829.2661884.

[38] Lopes S, Petit J-M, Lakhal L. Efficient Discovery of Functional Depende ncies and Armstrong Relations. BT - Advances in Database Technology - EDBT 2000, 7th International Conference on Extending Database Tech nology, Konstanz, Germany, March 27-31, 2000, Proceedings[Z/OL](2000). https://doi.org/10.1007/3-540-46439-5_24.

[39] Wyss C M, Giannella C, Robertson E L. FastFDs: A Heuristic-Driven, Depth-First Algorithm for Mining Functional Dependencies from Relation Instances - Extended Abstract. BT - Data Warehousing and Knowledge Discovery, Third International Conference, DaWaK 2001, Munich, Germ any, September 5-7, 2001, Proceedings[Z/OL](2001). https://doi.org/10.100 7/3-540-44801-2_11.

[40] Mayfield C, Neville J, Prabhakar S. ERACER: A Database Approach for Statistical Inference and Data Cleaning[J]. Proceedings of the ACM SIG MOD International Conference on Management of Data, 2010: 75–86.

[41] Flach P A, Savnik I. Database Dependency Discovery: A Machine Learn ing Approach.[J/OL]. AI Commun., 1999, 12(3): 139–160.

[42] Papenbrock T, Ehrlich J, Marten J, et al. Functional Dependency Discove ry: An Experimental Evaluation of Seven Algorithms.[J/OL]. Proc. VLDB Endow., 2015, 8(10): 1082–1093.

[43] Cong G, Fan W, Geerts F, et al. Improving Data Quality: Consistency a nd Accuracy. BT - Proceedings of the 33rd International Conference on

Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007[Z](2007).

[44] Fan W, Li J, Ma S, et al. Towards Certain Fixes with Editing Rules an d Master Data.[J/OL]. VLDB J., 2012, 21(2): 213–238.

[45] Fan W, Li J, Ma S, et al. CerFix: A System for Cleaning Data with Ce rtain Fixes.[J/OL]. Proc. VLDB Endow., 2011, 4(12): 1375–1378.

[46] Chu X, Ilyas I F, Papotti P. Holistic Data Cleaning: Putting Violations i nto Context. BT - 29th IEEE International Conference on Data Engineer ing, ICDE 2013, Brisbane, Australia, April 8-12, 2013[Z/OL](2013). https: //doi.org/10.1109/ICDE.2013.6544847.

[47] Geerts F, Mecca G, Papotti P, et al. The LLUNATIC Data-Cleaning Fra mework.[J]. Proc. VLDB Endow., 2013, 6(9): 625–636.

[48] Geerts F, Mecca G, Papotti P, et al. Mapping and Cleaning. BT - IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014[Z](2014).

[49] Ma S, Duan L, Fan W, et al. Extending Conditional Dependencies with Built-in Predicates.[J]. IEEE Trans. Knowl. Data Eng., 2015, 27(12): 327 4–3288.

[50] Wang J, Tang N. Towards Dependable Data Repairing with Fixing Rules. BT - International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014[Z/OL](2014). https://doi.org/10.11 45/2588555.2610494.

[51] Interlandi M, Tang N. Proof Positive and Negative in Data Cleaning. BT - 31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015[Z/OL](2015). https://doi.org/10.11 09/ICDE.2015.7113269.

[52] Yakout M, Berti-Équille L, Elmagarmid A K. Don’t Be SCAREd: Use S Calable Automatic REpairing with Maximal Likelihood and Bounded Cha nges. BT - Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 2 2-27, 2013[Z/OL](2013). https://doi.org/10.1145/2463676.2463706.

[53] Li P, Dong X L, Maurino A, et al. Linking Temporal Records.[J/OL]. P roc. VLDB Endow., 2011, 4(11): 956–967.

[54] Chiang Y-H, Doan A, Naughton J F. Tracking Entities in the Dynamic World: A Fast Algorithm for Matching Temporal Records.[J/OL]. Proc. VLDB Endow., 2014, 7(6): 469–480.

[55] Chiang Y-H, Doan A, Naughton J F. Modeling Entity Evolution for Te mporal Record Matching. BT - International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014[Z/OL] (2014). https://doi.org/10.1145/2588555.2588560.

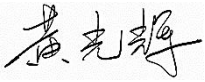
[56] Li F, Lee M-L, Hsu W, et al. Linking Temporal Records for Profiling E ntities. BT - Proceedings of the 2015 ACM SIGMOD International Con ference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015[Z/OL](2015). https://doi.org/10.1145/2723372.2737789.

[57] Wang X, Dong X L, Meliou A. Data X-Ray: A Diagnostic Tool for Dat a Errors. BT - Proceedings of the 2015 ACM SIGMOD International C onference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015[Z/OL](2015). https://doi.org/10.1145/2723372.2750549.

哈尔滨工业大学学位论文原创性声明和使用权限

学位论文原创性声明

本人郑重声明：此处所提交的学位论文《Spark 平台上的数据时效性管理关键技术研究》，是本人在导师指导下，在哈尔滨工业大学攻读学位期间独立进行研究工作所取得的成果，且学位论文中除已标注引用文献的部分外不包含他人完成或已发表的研究成果。对本学位论文的研究工作做出重要贡献的个人和集体，均已在文中以明确方式注明。

作者签名：  日期： 2022 年 6 月 10 日

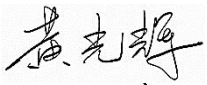
学位论文使用权限

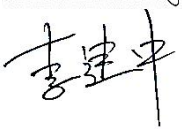
学位论文是研究生在哈尔滨工业大学攻读学位期间完成的成果，知识产权归属哈尔滨工业大学。学位论文的使用权限如下：

（1）学校可以采用影印、缩印或其他复制手段保存研究生上交的学位论文，并向国家图书馆报送学位论文；（2）学校可以将学位论文部分或全部内容编入有关数据库进行检索和提供相应阅览服务；（3）研究生毕业后发表与此学位论文研究成果相关的学术论文和其他成果时，应征得导师同意，且第一署名单位为哈尔滨工业大学。

保密论文在保密期内遵守有关保密规定，解密后适用于此使用权限规定。

本人知悉学位论文的使用权限，并将遵守有关规定。

作者签名：  日期： 2022 年 6 月 10 日

导师签名：  日期： 2022 年 6 月 10 日

致谢

人生的车轮越滚越快，抵达哈工大站的那天仿佛还是在昨天，怀着对梦想学府的敬畏与骄傲踏入校门的那一刻依然历历在目。两年的哈工大研究生之旅转瞬即逝，在这里我收获了国内顶级计算机学府的课程培养、学界最优秀的学者专业的指导、各怀绝技的同窗们热心的帮助以及宿管大爷、校车司机、健身房大叔们热心肠的服务。两年的校园学习以及两次企业的实习经历让我渐渐褪去学生的稚嫩，逐渐成长为一名拥有充足知识储备和专业技能的见习工程师。就在不知不觉中，下一站腾讯深圳已在不远处向我招手。在此，我要感谢这段旅程中指导、支持、陪伴、帮助过我的所有人。

由衷感谢李建中教授和刘显敏副教授对我研究工作的悉心指导。李老师的学者风度和家国情怀深深感染了我，他的一言一行都让我受益匪浅，他曾说过他会随身带一本关于不等式的书方便查阅，在 70 岁这样一个本该早已退休的年纪，李老师依旧身先士卒，站在科研工作的第一线，真正诠释了学者风骨、师者匠心；刘老师工作专注和治学严谨一直是我学习的榜样，他待人随和，与学生没有距离感，他因材施教，考虑到我本科是学数学的，计算机基础相对薄弱，他从论文选题到中期再到最后的结题，帮我克服了许多困难，并为我提供了许多宝贵的建议。

感谢我的父母这两年来对我的支持和鼓励。父亲教育我为人低调谦逊，在我每次取得成就时，总会及时提醒我不要骄傲，要保持谦虚谨慎。正是如此，我才能一直以高标准要求自己，不断进步，实现一个又一个极具挑战的目标。母亲教育我待人心怀善良，做人大气坦荡，懂得换位思考，设身处地为他人着想。正是如此，我才能与身边的每一个人都和睦相处，互相帮助，很少闹矛盾。

感谢我最为重要的好兄弟们：成滋桐、龚利锋、李思睿、李闯、王瀚尉、高猛、刘帅。我们一起经历了研一的课程学习、春招充满挑战的实习面试、暑假紧张忙碌的实习生活、秋招人均大厂的 offer 收割、疫情封校期间的“春风十里”音乐会、毕业设计期间的互帮互助以及每周例行的酒肉时光和篮球切磋。还有我们正在筹划的毕业旅行，相信这会是我们研究生旅程最难忘的终章。

感谢我各怀绝技的同窗们。感谢我的同门李天峰、高宇鹏、成滋桐、任甜在研究生期间对我的热心帮助与指导；感谢梁志宇师兄无数个夜晚和我在乒乓球室挥汗鏖战；感谢王丙昊、崔双双、纪名岳、靳贺霖、穆添愉、邵心玥陪我度过的许多无聊的周末；感谢卢虹远、廖其鑫对我毕设上的建议和生活上的陪伴。

最后，感谢哈尔滨工业大学为我提供了最顶尖的教育平台，我永远不会忘记那句最朴实无华的校训：规格严格，功夫到家！