软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2023,34(3):1213−1235 [doi: 10.13328/j.cnki.jos.006783]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

面向开放大数据环境的动态数据保护系统*

屠要峰[1,2], 牛家浩[2], 王德政[1,2], 高 洪[2], 徐 进[2], 洪 科[2], 阳 方[2]

[1](移动网络和移动多媒体技术国家重点实验室(中兴通讯股份有限公司), 广东 深圳 518057)

[2](中兴通讯股份有限公司, 江苏 南京 210014)

通信作者: 牛家浩, E-mail: niu.jiahao@zte.com.cn

摘 要: 大数据成为国家基础性战略资源, 数据的开放共享是我国大数据战略的核心. 云原生技术和湖仓一体架构正在重构大数据基础设施, 并推动数据共享和价值传播. 大数据产业和技术的发展都需要更强的数据安全和数据共享能力. 然而, 开放环境下数据的安全问题已成为制约大数据技术发展与利用的瓶颈. 无论开源大数据生态还是商业大数据系统, 所引发的数据安全及隐私保护问题都日益凸显. 开放大数据环境下的动态数据保护系统面临着数据可用性、处理高效性和系统可扩展性等方面的挑战. 提出了面向开放大数据环境的动态数据保护系统BDMasker, 通过一种基于查询依赖模型(query dependency model)的精准查询分析及查询改写技术, 能够精准感知但不改变原始业务请求, 实现动态脱敏全过程对业务零影响; 通过面向多引擎的统一安全策略框架, 实现了动态数据保护能力的纵向扩展和在多种计算引擎中的横向扩展; 利用大数据执行引擎的分布式计算能力, 提升系统的数据保护处理性能. 实验结果表明, BDMasker提出的精准SQL分析及改写技术是有效的, 系统具有良好的扩展能力和性能表现, 在TPC-DS和YCSB基准测试中, 整体性能波动在3%之内.

关键词: 大数据; 数据脱敏; 动态数据脱敏; SQL改写; 查询依赖

中图法分类号: TP311

中文引用格式: 屠要峰, 牛家浩, 王德政, 高洪, 徐进, 洪科, 阳方. 面向开放大数据环境的动态数据保护系统. 软件学报, 2023, 34(3): 1213–1235. http://www.jos.org.cn/1000-9825/6783.htm

英文引用格式: Tu YF, Niu JH, Wang DZ, Gao H, Xu J, Hong K, Yang F. Dynamic Data Protection System for Open Big Data Environment. Ruan Jian Xue Bao/Journal of Software, 2023, 34(3): 1213–1235 (in Chinese). http://www.jos.org.cn/1000-9825/6783.htm

Dynamic Data Protection System for Open Big Data Environment

TU Yao-Feng[1,2], NIU Jia-Hao[2], WANG De-Zheng[1,2], GAO Hong[2], XU Jin[2], HONG Ke[2], YANG Fang[2]

[1](State Key Laboratory of Mobile Network and Mobile Multimedia Technology (ZTE Corporation), Shenzhen 518057, China)

[2](ZTE Corporation, Nanjing 210014, China)

Abstract: Big data has become a national basic strategic resource, and the opening and sharing of data is the core of China's big data strategy. Cloud native technology and lake-house architecture are reconstructing the big data infrastructure and promoting data sharing and value dissemination. The development of big data industry and technology require stronger data security and data sharing capabilities. However, data security in an open environment has become a bottleneck, which restricts the development and utilization of big data technology. The issues of data security and privacy protection have become increasingly prominent both in the open source big data ecosystem and the commercial big data system. Dynamic data protection system under the open big data environment is now facing challenges of data availability, processing efficiency and system scalability and etc. This study proposes a dynamic data protection system BDMasker for the open big data environment. Through a precise query analysis and query rewriting technology based on the query dependency model, it can accurately perceive but not change the original business request, which indicates that the whole process of

* 基金项目: 国家重点研发计划(2021YFB3101100)
本文由"大数据治理的理论与技术"专题特约编辑杜小勇教授、杨晓春教授和童咏昕教授推荐.
收稿时间: 2022-05-14; 修改时间: 2022-07-29, 2022-09-07; 采用时间: 2022-09-23; jos在线出版时间: 2022-10-27

dynamic desensitization has zero impact on the business. Furthermore, its multi-engine-oriented unified security strategy framework realizes the vertical expansion of dynamic data protection capabilities and the horizontal expansion among multiple computing engines. The distributed computing capability of the big data execution engine can be used to improve the data protection processing performance of the system. The experimental results show that the precise SQL analysis and rewriting technology proposed by BDMasker is effective, the system has good scalability and performance, and the overall performance fluctuates within 3% in the TPC-DS and YCSB benchmark tests.

Key words: big data; data masking; dynamic data masking; SQL rewriting; query dependency

大数据时代, 大数据成为国家基础性战略资源. 党中央、国务院高度重视大数据发展, 开始全面实施国家大数据战略, 其中, 数据的开放共享是大数据竞争战略核心. 从技术发展趋势看, 新的技术架构和大数据支撑平台不断涌现, 云原生、湖仓一体架构正在重构大数据基础设施. 无论从访问数据湖、数据仓库到跨数据库、跨域共享, 都需要更强的数据安全和数据共享能力. 然而, 无论开源大数据生态还是商业大数据系统, 对于开放环境下大数据的安全保护能力都是严重落后于业务发展的. 近些年发生的隐私泄露事件都表明, 未经过脱敏处理的数据发布或者共享很容易泄露数据隐私, 特别是个体敏感信息. 2018 年, 美国社交媒体 Facebook 的 8 700 万用户数据被咨询公司剑桥分析公司违规使用, 被判赔付 50 亿美元罚金; 2021 年, Facebook 又有 5.33 亿个人用户数据遭泄露. 开放环境下的安全问题, 已成为制约大数据技术发展与利用的瓶颈. 如何在开放的复杂环境下保护敏感数据的隐私性, 同时保证数据的可用性和计算的高效性, 成为大数据安全领域的研究热点之一[1,2].

开放大数据环境下的数据安全与传统数据安全有很大的不同, 保护方式、保护对象、管理和技术的关系均发生了变化. 开放大数据应用场景以数据的开放共享为目标, 参与数据处理的角色更加多元, 数据的流动是常态, 这对数据的安全防护提出了更高的要求, 数据加密、静态脱敏等传统的数据安全措施已不再适用. 研究表明, 采用隐私保护与动态数据脱敏技术, 是促进数据安全流通与共享、确保大数据服务可信的重要手段[3,4].动态数据脱敏技术可以在数据流动过程中不泄露敏感信息, 同时保留数据源的可用性, 具有较好的使用价值和广阔的应用空间. 在开放大数据环境下, 面对海量多模数据和高并发访问请求, 如何以自动化、高效、可扩展方式动态保护敏感数据, 同时减小对正常业务的影响, 是一个十分复杂但亟待解决的问题[5−7]. 面临的主要挑战有:

(1) 异构环境的扩展性. 开放大数据场景下, 为了满足不同的数据查询、数据计算的时效性要求, 在同一套集群上往往同时部署多种大数据计算引擎, 如 Apache Spark[8]适合延时较高的静态数据批量处理场景、Apache Flink[9]适合低延时或实时的流数据处理场景等. 面对复杂多元的业务场景和多种计算引擎, 需要研究如何创建、管理和维护面向异构引擎的统一数据保护策略, 并提供标准化的访问方法, 解决异构环境的横向扩展问题. 同时, 除了动态数据脱敏能力外, 需要研究如何在一套框架下同时灵活支持多种动态数据保护能力, 支持单个引擎动态数据保护能力的纵向扩展.

(2) 处理性能的高效性. 开放大数据环境下, 数据的产生速度越来越快, 数据规模持续指数级增长. 数据安全防护要满足海量数据高性能实时保护的响应时间要求, 就必须能够在规则的引导下自动化进行, 还要能够对全处理流程进行负载优化, 充分利用大数据执行引擎的分布式计算能力, 提升处理性能.

(3) SQL 改写的精准性. SQL 是广泛采用的数据查询语言, 目前, 主流的大数据计算引擎均提供 SQL 访问能力. SQL 改写是实现动态数据脱敏的关键技术. 业务领域的 SQL 请求千变万化, SQL 改写机制会涉及所有定义了脱敏策略的列, 复杂 SQL 语句改写后可能会造成数据失真, 降低数据可用性,甚至影响业务逻辑处理的准确性. 当面对复杂的 SQL 访问请求时, 如何保证改写后的 SQL 在不暴露底层物理表敏感信息的前提下对业务完全透明, 使得业务逻辑不受数据保护的影响, 对动态数据保护系统设计, 尤其是 SQL 改写技术提出了挑战. 下面以图 1 所示 TPC-DS[10]的 Query76 查询语句为例, 对 SQL 改写的技术难点进行说明. Query76 覆盖了 SQL 语句中大部分重要语法规则.

```
SELECT channel, col_name, d_year, d_qoy, i_category, COUNT(*) sales_cnt, SUM(ext_sales_price) sales_amt
FROM (
  SELECT 'store' AS channel, ss_store_sk col_name, d_year, d_qoy, i_category, ss_ext_sales_price ext_sales_price
  FROM store_sales, item, date_dim
  WHERE ss_store_sk IS NULL AND ss_sold_date_sk=d_date_sk AND ss_item_sk=i_item_sk
  UNION ALL
  SELECT 'web' AS channel, ws_ship_customer_sk col_name, d_year, d_qoy, i_category, ws_ext_sales_price ext_sales_price
  FROM web_sales, item, date_dim
  WHERE ws_ship_customer_sk IS NULL AND ws_sold_date_sk=d_date_sk AND ws_item_sk=i_item_sk
  UNION ALL
  SELECT 'catalog' AS channel, cs_ship_addr_sk col_name, d_year, d_qoy, i_category, cs_ext_sales_price ext_sales_price
  FROM catalog_sales, item, date_dim
  WHERE cs_ship_addr_sk IS NULL AND cs_sold_date_sk=d_date_sk AND cs_item_sk=i_item_sk
) foo
GROUP BY channel, col_name, d_year, d_qoy, i_category
ORDER BY channel, col_name, d_year, d_qoy, i_category
LIMIT 100
```

图 1 TPC-DS Query76 查询语句

- SQL 精准分析

对一个 SQL 查询来说, 其查询结果集的输出信息最终来源于最外层查询 select 语句的输出字段, 而最外层的查询输出字段可能来源于子查询语句、join、union 语句等, 可能经过子查询、嵌套函数等层层转换. 因此, 要对最外层输出字段中敏感字段的来源进行精准识别并正确脱敏, 否则可能暴露底层物理表的敏感信息. 例如, Query76 中最外层输出列 *col_name* 所最终依赖的物理表字段包括: *store_sales* 表的 *ss_store_sk* 字段、*Web_sales* 表的 *ws_ship_customer_sk* 字段及 *catalog_sales* 表的 *cs_ship_addr_sk* 字段, 如果这 3 个字段其中之一设置了脱敏规则, 而其他两个字段没有设置脱敏规则, 则最外层输出列 *col_name* 会因没有获取底层依赖的物理表字段信息并应用对应的脱敏规则, 导致查询结果集中的敏感数据泄露.

- 敏感字段精准定位

SQL 查询请求中, 敏感字段可能来自不同的语法结构. 例如, Query76 中的 *i_category* 字段多次出现在不同层次子查询输出字段中; *d_year* 字段既出现在子查询输出字段中, 又出现在 GROUP BY, ORDER BY 语句中. 有的查询输出字段来源于函数或者函数嵌套, 如 *SUM*(*ext_sales_price*). 因此, 如果不能有效地识别 SQL 结构中复杂的嵌套、层次、别名、函数等语法及逻辑关系, 则采用直接改写 SQL 语句中所有定义脱敏策略字段的简单处理方式, 则改写后 SQL 语句与原始请求往往不一致, 造成业务失败或者不正确.

大数据生态 SQL-on-Hadoop 开源软件如 Apache Spark, Apache Flink 等目前尚未支持动态脱敏等安全能力. 目前尚未有系统能够较好地解决上述 3 个挑战. 针对上述问题, 本文探讨了一种面向开放大数据场景下的高性能动态数据保护系统 BDMasker 的设计与实现, 主要工作和贡献如下.

(1) 基于“业务零感知”的原则, 提出一种基于查询依赖模型(query dependency model)的精准查询分析及查询改写技术, 能够精准地感知但又不改变原始业务请求, 实现动态脱敏全过程对业务几乎无影响, 解决了 SQL 改写精准性挑战.

(2) 提出一种面向多引擎的统一安全策略框架, 通过插件式数据保护策略管理支持多种动态数据保护能力的纵向扩展; 可以根据应用场景、业务目标和数据特征选择合适的数据保护策略, 通过策略代理和标准化接口实现了多种计算引擎的横向扩展.

(3) 基于上述 SQL 分析改写技术和统一安全策略框架, 在 Apache Hive, Apache Spark 等系统中实现了动态数据保护引擎, 该动态保护引擎内置于大数据执行进程中, 充分利用大数据执行引擎的分布式计算能力提升数据保护处理的性能.

(4) 通过功能实验和 YCSB, TPC-DS 基准测试对比实验, 验证了 BDMasker 系统的有效性、可扩展性和性能提升效果.

本文第 1 节介绍研究背景和相关工作. 第 2 节介绍系统架构及关键技术. 第 3 节通过实验验证 BDMasker

系统的功能有效性、可扩展性及数据保护处理的高性能. 最后, 在第 4 节对本文进行总结与展望.

1 相关工作

国内外有许多关于动态数据脱敏的研究工作[4,6,11−16], 研究人员提出了若干种体系架构. 我们将这些工作分成基于代理模式和基于内核模式.

(1) 基于代理模式

基于代理模式需要部署外置脱敏代理网关, 代理网关首先拦截查询请求, 再通过改写查询请求或者改写查询结果的方式实施动态脱敏功能. 代理模式有基于查询请求改写及基于查询结果改写两种技术路线.

- 基于查询请求改写技术首先通过代理网关拦截客户端查询; 接着, 根据安全策略对查询请求进行改写, 并将改写后的查询请求重定向到数据源; 收到修改后的查询后, 数据源会输出虚假(混淆)数据. 根据查询请求改写方式, 研究者们提出了基于规则引擎[11]的改写技术及基于语法解析[12,13]的改写技术.
  - Informatica[11]等厂商采用了通过脱敏代理网关中规则引擎匹配并改写查询 SQL 的动态脱敏技术, 通过规则引擎连接规则树解析请求. 如果连接规则分配了安全规则集, 则规则引擎通过安全规则树解析 SQL 请求, 并利用改写规则重写 SQL 语句. 规则引擎技术实现简单, 扩展能力强, 其缺点也比较明显: ① 查询请求越复杂, 识别难度越高, 匹配的准确率就越低, 规则引擎匹配无法精确匹配复杂 SQL 语句, 无法精确改写脱敏字段, 直接导致改写错误或者可能改变已有的业务逻辑; ② 规则引擎处理效率低, 修改后的 SQL 语句复杂度高、性能差, 同时, 无法解析所有语法规则, 容易造成数据泄露.
  - 由于基于规则引擎的改写技术存在不足, 有研究者提出了基于语法解析的技术. 其核心思想是, 在脱敏代理网关中, 通过词法分析或者语法分析对 SQL 语句进行拆包和分析, 匹配策略规则, 然后改写查询请求. 语法解析模式通常对标准 SQL 语法进行解析, 通用性较强, 但很难真正到达 SQL 无关性, 往往会因为每个数据库方言组装逻辑语法规则不同导致敏感数据的泄露. 另外, 为了对接不同的数据库引擎, 需要不断适配数据库的变更, 系统开发及维护难度很大.
- 与基于查询请求改写技术不同, 基于查询结果改写技术[12,13]的核心思想是, 脱敏代理网关拦截客户端查询, 然后直接转发给数据库引擎, 由数据库引擎查询得到数据查询结果, 代理网关解析查询请求语句并利用预设信息替换查询结果数据中的敏感数据, 得到脱敏数据, 并向应用程序输出脱敏数据. 这个技术路线的主要缺点是查询请求和结果返回都需要经过脱敏中间件, 消耗大量网络带宽及计算、存储资源, 该模式无法满足大数据实时脱敏的响应时间要求, 同时, 数据库返回的结果数据需要缓存在脱敏服务器中, 引入新的攻击面, 从而带来了数据泄露的安全隐患.

(2) 基于内核模式

基于内核的动态脱敏模式是通过数据库 DDL 命令设置字段脱敏规则, 通过在数据库内核中改写查询请求, 实现单个引擎的动态脱敏功能. 商用数据库厂商如 Oracle, IBM[14]等、开源数据库 OpenGauss[15]、开源大数据引擎 Apache Hive[16]均采用了此类模式. OpenGauss 动态脱敏的核心思路是: 将包含敏感字段查询的语句改写, 对于查询中涉及的敏感字段, 通过外层嵌套函数的方式改写. 改写机制的主要思想: 查询树中, Var 类型节点代表了访问的数据库资源, 而非 Var 类型节点可能包含 Var 节点; 需要根据其参数递归的寻找 Var 节点, 最后将识别到的所有 Var 节点进行策略匹配, 并根据策略函数进行节点替换. 与 OpenGauss 类似, Apache Hive 通过 SQL 改写技术实现了列级别动态脱敏功能, Hive 引擎在对 SQL 进行解析时, 遇到待脱敏字段就使用脱敏算法修改其抽象语法树的值, 而无论该字段在 SQL 语句中的位置.

OpenGauss、Apache Hive 的 SQL 改写机制会涉及所有定义了脱敏策略的列, 考虑到同一个字段可能出现在 SQL 查询语句的各个语法结构中, 这种对所有匹配到的字段都进行改写的方式不能有效地识别 SQL 结构中复杂的嵌套、层次等逻辑关系, 改写后查询语句会造成业务逻辑执行结果错误或者业务逻辑无法正确实现.

(3) 扩展性方面

基于代理模式虽然可同时支持多个数据源脱敏, 但是这样既增加了数据访问开销, 也带来更大的系统维护成本和硬件成本, 无法满足大数据实时脱敏的响应时间要求, 同时, 结果集缓存也引入了新的攻击面. Oracle, OpenGauss 等提出的策略模型基于自身数据库的 DDL 实现, 只适用于自身数据库, 无法扩展到大数据环境. 大数据开源生态缺乏统一的安全技术标准, 无法进行统一有效部署和管理, 开源大数据引擎 Apache Spark, Apache Flink 目前尚未支持动态脱敏等安全能力, 不具备面向多引擎的可扩展的安全策略模型.

2 系统架构及关键技术

本节介绍面向开放大数据环境的动态数据保护系统 BDMasker 的系统架构及关键技术. BDMasker 采用面向多引擎的统一安全策略框架和基于查询依赖模型的动态数据保护技术, 实现了"业务零感知"的动态脱敏等多种数据安全防护能力, 支持 Apache Hive, Apache Spark, Apache HBase 等多种大数据计算引擎, 解决了开放大数据环境动态数据保护面临的异构环境扩展、处理性能高效、SQL 改写精准等技术挑战.

2.1 系统架构

代理模式在海量数据场景下存在脱敏性能低下及安全性差的问题. BDMasker 系统设计时摒弃了外置脱敏代理架构, 通过在大数据内核中直接内置动态数据保护引擎, 利用大数据引擎自身的分布式处理能力, 实现高性能动态数据脱敏等数据保护功能, 从架构流程上解决处理性能高效性的挑战. 通过集中式部署策略引擎, 统一管理和维护面向异构引擎的数据保护策略, 并提供一种标准化的访问方法, 解决异构环境横向扩展的挑战.

如图 2 所示, BDMasker 系统架构自上而下共分为 3 层, 依次为大数据客户端层、大数据执行引擎层和策略引擎层.

- 大数据客户端层: 客户端是访问大数据服务的应用、程序、命令或者脚本等, 通过网络向大数据执行引擎发起访问请求并接收处理结果. 现有各种开源大数据引擎的客户端不需要做任何改变.
- 大数据执行引擎层: 大数据执行引擎负责完成对客户端请求的业务逻辑处理, 如数据查询分析. BDMasker 通过在大数据计算引擎中置入动态数据保护引擎, 为大数据系统提供动态数据保护功能. 动态数据保护引擎由策略代理模块、精准分析模块和 SQL 改写模块组成: 策略代理模块负责从策略引擎中获取本执行引擎所配置数据保护策略并缓存到本地内存, 为数据动态保护引擎提供保护策略查询接口; 精准分析模块通过构建算法, 生成 SQL 查询依赖模型; SQL 改写模块负责对 SQL 语句进行精准改写.
- 策略引擎层: 策略引擎为多个同构/异构大数据执行引擎提供统一的数据保护策略服务, 由策略控制模块、算法模块和策略数据库组成: 策略控制模块设计了一种面向异构大数据执行引擎的统一数据保护策略模型, 并为各个执行引擎提供统一的策略管理服务、API 接口及操作界面; 算法模块负责统一管理数据保护算子, 并提供算子开发、加载及部署框架; 策略数据库负责统一保存策略控制模块及算法模块相关的配置数据.

BDMasker 系统工作流程如图 3 所示. 管理员通过策略管理界面, 可为多个大数据引擎配置不同的数据保护策略. 首先, 选择大数据引擎服务实例名、数据库和表; 接着, 为受保护的敏感字段设置安全规则, 如随机、替换、偏移等; 设置完成后, 数据保护策略将保存到策略数据库; 各个执行引擎内的策略代理模块动态获取本引擎对应的数据保护策略并加载到内存.

如图 3 右侧所示, 大数据客户端向大数据执行模块发起 SQL 访问请求, 在请求中会携带用户名作为用户的身份标识. 大数据执行模块的 SQL 解析器针对 SQL 语句进行词法分析, 得到词法分析通用符号流, 对符号流再进行语法分析, 构造出语法树并对语法树进行深入分析, 生成取抽象语法树 AST. 之后进入动态数据保护流程. 对于动态数据脱敏操作, 首先由本系统精准分析模块构建出 SQL 请求的查询依赖模型; 接着, 由本系统 SQL 改写模块结合查询依赖模型及安全策略, 对 SQL 进行精准改写; 然后, 经过安全分析流程处理后,

将改写后的 SQL 语句提交给执行引擎. 执行引擎对改写后的 SQL 重新进行分析后, 由查询优化器对逻辑计划进行优化, 生成物理计划并进行优化, 转换为包含数据保护功能的可执行任务, 由大数据执行引擎利用集群的分布式计算能力并行完成业务逻辑处理. 执行完毕后, 由大数据执行模块将结果集数据返回给客户端.

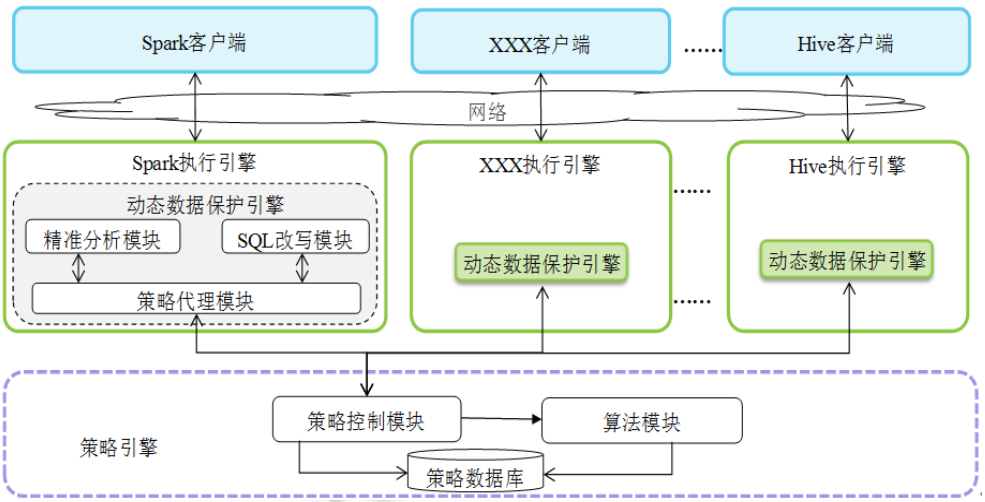


图 2 BDMasker 系统架构

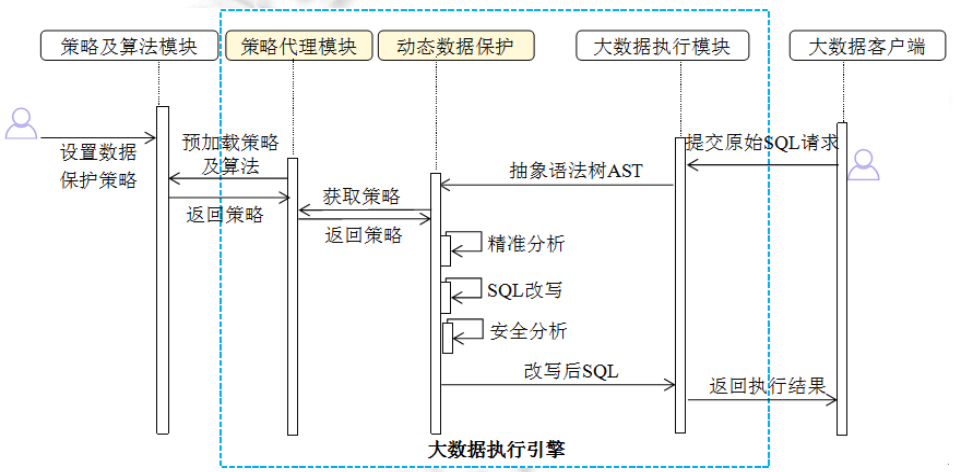


图 3 BDMasker 系统工作流程

2.2 面向多引擎的统一安全策略框架

开放大数据环境下, 存在着多种大数据计算引擎, 这些大数据引擎的安全策略模型往往不一致, 对外接口也不统一, 造成安全策略难以管理和维护. 针对这种情况, BDMasker 对异构引擎的数据保护模型进行抽象和统一, 提出了一种基于元数据面向多引擎的统一安全策略框架, 为异构大数据引擎提供统一的接口和安全策略模型.

如图 4 所示, BDMasker 面向多引擎的统一安全策略模型由 3 部分组成: 规则对象、规则条件、规则策略.

- 规则对象代表了受控的大数据资源实体, 采用服务(大数据执行引擎实例名)-数据库-表-敏感列这种层次模型, 该描述方式可同时管理异构引擎不同层次的资源实体, 最小粒度为列. 采用敏感数据域对敏感数据进行分类, 基于列数据或列名称来描述列的功能含义. 例如, “身份证号”作为一类敏感数据, 对所有异构引擎中的身份证号类字段实施统一的保护策略. 以 HBase 为例, HBaseSrv1-DB1-Table1-

CF1: F1, 表示对 HBasesrv1 这个服务的 DB1 数据库 Table1 表的列族 CF1 的 F1 字段进行管理.

- 规则条件是针对不同的请求类型定义可信访问的基本判断单元, 可以根据请求上下文来定义, 针对不同的命令类型定义不同的保护方法. 例如, 身份规则条件可以基于用户名、用户组、角色等限制用户获取敏感数据, 时间条件可以限制访问时间, 级别规则条件可以根据数据分类分级条件限制用户获取敏感数据.
- 规则策略采用数据保护规则-执行算子结构来描述. 数据保护规则是针对敏感数据采取的具体保护操作, 包括动态数据脱敏规则、动态数据过滤规则等. 数据脱敏规则满足行业的合规要求, 用于变形特定数据类型或者敏感数据域的数据. 数据脱敏算子是执行算子的一类, 用于脱敏底层转换执行, 以算法库的形式驻留在大数据执行引擎中. 算子是算法思想的落地实现, 在算法设计时应该考虑性能[17], 能够根据不同的负载和数据特征设计不同的脱敏算法[18], 业界常用的算法有模数算法[19]、格式保留加密算法[20]等. BDMasker 按敏感数据域如账号域、邮箱域、姓名域、地址域等设计了 40 余种脱敏各种算子, 包括凯撒加密算子、移位算子、乱序算子、截断算子等. 按算法特性为算子设置不同的标签, 包括稳定性、唯一性等, 稳定性算子执行一次与执行多次的结果是一致的, 唯一性算子执行的结果是全局唯一的, 不同的算子有不同的应用场景. 同一个脱敏算子可以创建多个不同的脱敏规则, 同一个脱敏规则可以分布在不同的敏感数据域中, 一个敏感数据域包括多个脱敏规则, 通过这种灵活的设计, 可满足各种行业的合规要求.

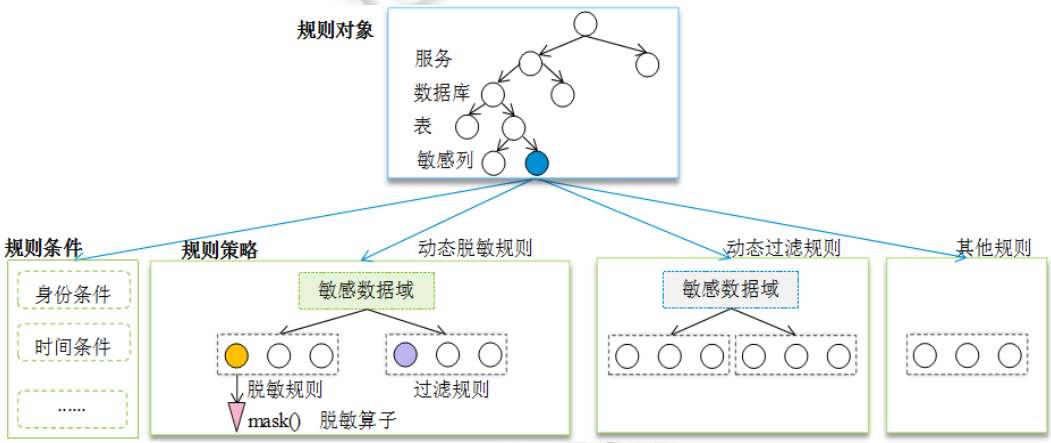


图 4 BDMasker 策略模型图

基于面向多引擎的统一安全策略模型, BDMasker 设计实现了策略引擎, 并提供集中式统一的策略管理界面及 API 访问接口. 为了将统一的安全策略模型应用到异构大数据引擎中, 系统对策略代理模块采用插件式设计, 以插件的方式内置到大数据引擎中, 从策略控制模块中加载本执行引擎已配置数据保护策略并缓存到本地内存, 并为数据保护执行引擎提供策略接口, 通过 API 接口进行策略的动态增删改查.

BDMasker 系统采用多种动态数据保护技术灵活组合又相互解耦的方式, 解决单个计算引擎数据保护能力的纵向扩展问题. BDMasker 将 SQL 语句的处理流程划分为精准分析、SQL 改写及安全评估这 3 个阶段, 在每个阶段均提供插件接口, 以支持安全能力的灵活扩展. 精准分析阶段默认构造查询依赖模型, 获取 SQL 查询请求输出字段所依赖的数据来源, 该阶段可以扩展支持如告警、基于规则的访问控制等能力. SQL 改写阶段默认对抽象语法树进行脱敏精准改写, 该阶段可以扩展支持动态数据过滤能力, 改写后的 SQL 语句可同时支持动态脱敏与动态过滤功能, 对业务逻辑没有影响. 评估阶段对重构后的语法树进行安全分析, 防止恶意访问或者绕过脱敏功能, 可扩展支持如敏感字段细粒度操作审计、敏感数据告警、SQL 安全分析、数据脱敏风险评估等功能. BDMasker 这些扩展的安全能力以插件形式的部署到大数据执行引擎中, 由动态数据保护引擎按统一安全策略加载到各个大数据引擎中, 在 SQL 语句的执行过程中与动态数据脱敏功能灵活组合, 共同实

现不同用户对同一敏感数据访问时的安全控制，从而有效地阻断非法访问.

基于面向多引擎的统一安全策略框架，BDMasker 提供标准化接口对各个引擎的数据保护策略进行统一管理，并在单个引擎内部通过安全插件方式进行扩展，实现了动态数据保护功能与大数据执行引擎的业务逻辑的解耦，减少了对现有大数据系统的影响. 我们分别在 Apache Hive, Apache Spark, Apache HBase 中开发了策略代理插件和安全插件，实现了统一安全策略管理，业务系统能够根据应用场景、业务目标和数据特征选择合适的数据保护策略. 同时，在单个引擎内部，通过安全能力插件进行纵向扩展，可支持动态数据脱敏、动态数据过滤等多种动态数据保护能力.

2.3 基于查询依赖模型的动态数据保护技术

SQL 语句通常由 Select 语句、From 语句和 Body 语句这 3 部分组成: Select 语句属于查询输出字段部分; From 语句描述查询输出字段所依赖数据源，数据源主要包括表、嵌套子查询语句、join/union 语句等; Body 语句是查询语句的主体部分，包括 where, group by, order by, limit 等. 如前所述，业务领域的 SQL 语句往往非常复杂，采用规则引擎或者浅层解析等方式，由于缺少精准分析技术，无法正确应用安全保护策略，从而导致敏感数据泄露.

本节介绍基于查询依赖模型(query dependency model)的精准查询分析及 SQL 改写技术，基于该技术，能够精准感知改写但又不改变原始业务请求，实现动态数据保护全过程对业务零影响.

2.3.1 查询依赖模型定义

SQL 精准改写的难点之一，就是找到最外层查询输出字段最终依赖的数据源. 为增强对结构化查询分析结果的统一表示能力，正确获取查询输出字段最终依赖的表及字段信息，我们对查询分析过程进行提炼，提出了查询依赖模型 QDM 及子查询依赖模型 Sub-QDM，如图 5 所示.

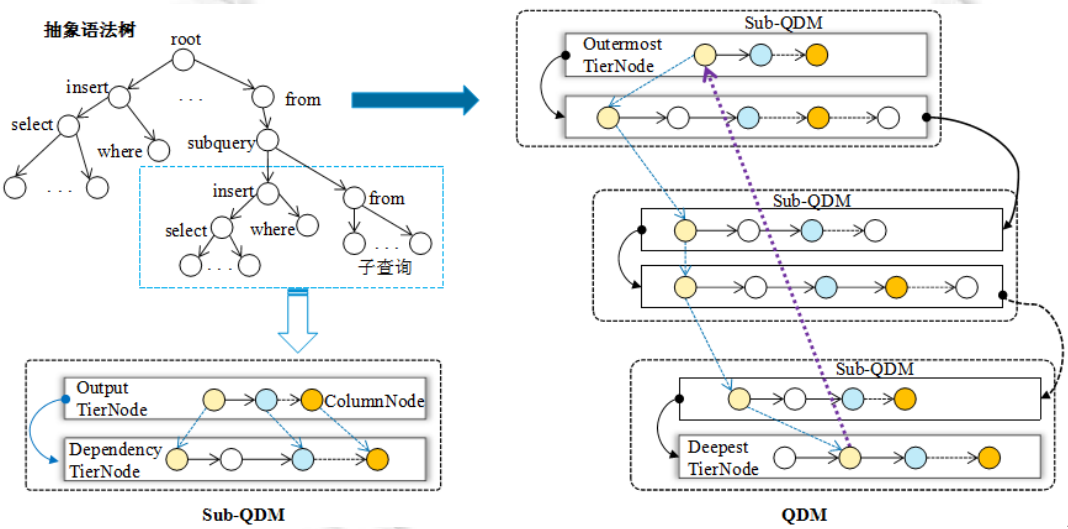


图 5 查询依赖模型 QDM

定义 1. 具备以下特征的模型称为子查询依赖模型(Sub-QDM). 该模型记录了一个 SQL 子查询输出字段依赖信息，每个 SQL 子查询会创建一个 Sub-QDM.

(1) 每一个 Sub-QDM 由两个查询层次节点(TierNode)组成，分别为 Output TierNode 及 Dependency TierNode，二者通过指针建立逻辑关联.

(2) TierNode 内部以单链表方式存储字段信息. 单链表上每个节点称为字段信息节点(ColumnNode)，字段信息节点统一由形如(TN,ATN,FN,AFN)的多元组构成. 其中, TN 表示表名, ATN 表示子查询别名, FN 表示字段名, AFN 表示字段别名.

(3) Output TierNode 中, 每个 ColumnNode 存储一个输出字段信息; Dependency TierNode 中, 每个 ColumnNode 存储查询输出字段所依赖的一个底层数据源信息.

定义 2. 具备以下特征的模型称为查询依赖模型(query dependency model, QDM). 该模型逐层记录了 SQL 查询输出字段间依赖关系.

(1) QDM 是一个层次结构, 由若干 TierNode 组成. 最顶层 TierNode 称为 Outermost TierNode, 为查询请求的最终输出字段. 最内层 TierNode 称为 Deepest TierNode, 它是 Outermost TierNode 所最终依赖的数据源.

(2) QDM 由若干 Sub-QDM 组成, 每一个子查询生成其对应的 Sub-QDM, 由父查询的 Dependency TierNode 通过指针指向子查询 Output TierNode, 建立父子查询间的逻辑关联, 最终构成一个完整的 QDM.

2.3.2 查询依赖模型构建

SQL 语句通过解析后会转换为抽象语法树 AST, 抽象语法树以树状的形式表现编程语言的语法结构, 树上的每个节点都表示源代码中的一种结构, 抽象语法树具备了精准分析所需要的所有信息. 因此, 模型构建算法将抽象语法树作为模型构建的输入.

本节以下面这个简单嵌套查询语句作为示例, 介绍查询依赖模型的构建算法.

```
select id from
  (select id, username from
    (select class, id, username from tInfo) Info
  ) t
```

通常, 在不同的 SQL 引擎内部有不同的常量名称唯一标识抽象语法树节点类型, 以 Apache Hive 为例, 抽象语法树解析过程对每个表生成一个 TOK_TABREF 节点, 对 from 关键字生成一个 TOK_FROM 节点, 其他节点类似. 算法 1 以 Apache Hive 的抽象语法树为例构建 QDM, 其他大数据执行引擎如 Apache Spark 的实现思路类似, 以此类推.

算法 1. 构建 QDM 模型算法.

输入: 抽象语法树根节点 *ASTRoot*.

输出: Outermost *TierNode* 即 QDM 的根节点.

```
1.  TierNode walkAST(ASTRoot)
2.  {
3.      OutputTierNode=newTierNode(·);   //新建一个查询 OutputTierNode
4.      DependencyTierNode=newTierNode(·);   //新建一个依赖 DependencyTierNode
5.      OutputTierNode→next=DependencyTierNode;   //建立指向关系
6.      walkSelectAST(ASTRoot,TOK_SELECT,OutputTierNode);   //调用算法 2 遍历 Select 子树
7.      walkFromAST(ASTRoot,TOK_FROM,DependencyTierNode);   //调用算法 3 遍历 From 子树
8.      updateOutputTier(OutputTierNode);   //调用算法 4 更新 QDM 层间字段依赖关系
9.      return OutputTierNode;   //返回 QDM 根 TierNode
10. }
```

算法 1 描述了为一条 SQL 查询请求构造一个完整查询依赖模型 QDM 的过程: 首先, 创建一个 *OutputTierNode* 节点和 *DependencyTierNode* 节点, 并建立二者的依赖指向关系(第 3–5 行); 接着, 遍历并解析 select 从句对应的 AST 子树, 获得 select 从句的内部结构, 获取出当前层查询输出字段信息(第 6 行, 对应查询输出字段生成算法); 然后, 遍历 from 从句对应的 AST 子树, 对语法树中的子查询等语法结构继续深度遍历, 通过递归逐层构造出对应的 Sub-QDM 模型(第 7 行, 对应查询输出字段依赖信息生成算法); 最后, 更新顶层节点的依赖信息, 构成一个完整的 QDM 模型(第 8 行, 对应最终依赖更新算法).

图 6 描述了示例 SQL 使用算法 1 生成 QDM 的流程.

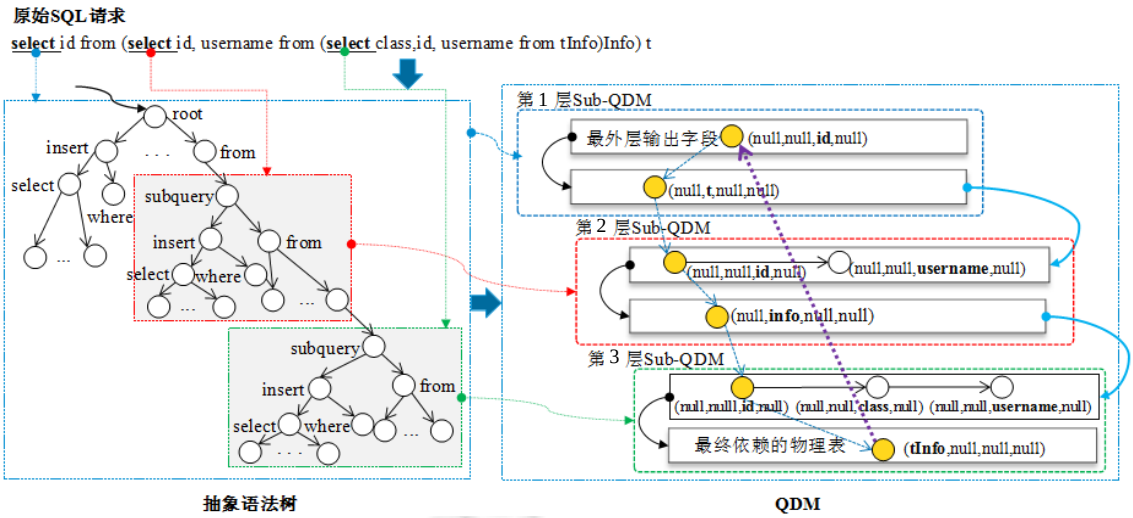


图 6 QDM 构造示意图

示例 SQL 通过算法 1 生成了一个由 3 个 Sub-QDM 组成的完整 QDM 模型. 每个 Sub-QDM 对应该示例 SQL 的一个 select 子查询. 第 1 层 Sub-QDM 对应最外层的 select 查询, 第 2 层 Sub-QDM 对应中间 select 子查询, 第 3 层 Sub-QDM 对应对最内层 select 子查询. 每个 Sub-QDM 存储了本层查询的输出字段和字段来源. 最外层 select 的每个查询输出字段都可以通过 Sub-QDM 间关系找到其所最终依赖的物理表信息.

- 查询输出字段生成算法(算法 2)

算法 2. 查询输出字段生成算法.

输入: 待分析的 AST 根节点 *ASTRoot*, AST 节点类型 *Nodetype*, Output 层次节点 *OutputTierNode*.

输出: *NULL*.

```
1.  walkSelectAST(ASTRoot,Nodetype,OutputTierNode)
2.  {
3.      node=walkToAST(ASTRoot,Nodetype);   //遍历 AST 定位到 select 根节点,从这里开始遍历;
4.      while (子树没有遍历完成)
5.      {
6.          if (当前节点为叶子节点且为包含字段信息)
7.              addOneNode(OutputTierNode,node);
8.          if (节点类型为函数)   //节点类型为函数
9.              addAllFuncitonColumnNode(OutputTierNode,node);
10.         if (当前节点类型为*查询)   //select*的情况
11.             addAllStarNode(OutputTierNode,node);
12.         获取 AST 树下一个节点继续遍历;
13.     }
14. }
```

查询输出字段生成算法通过遍历、解析 select 从句对应的 AST 子树, 获得 select 从句的内部结构, 获取当前层查询输出字段信息. 主要步骤: 首先, 从 AST 根节点开始深度遍历, 直接定位到 select 从句根节点, 从这里开始分析(第 3 行); 接着, 遍历 select 子树, 直到 select 从句遍历完毕, 遍历过程中根据 AST 节点类型进行不同处理(第 4–13 行). 具体地, 如果节点类型为字段或者表信息节点且为叶子节点, 则直接获取表名、子查询

别名、字段名、字段别名信息, 插入 Sub-QDM 的 *OutputTierNode* 结构中; 对于函数节点, 则解析函数子树, 获取该函数所涉及的所有表字段信息, 并插入 Sub-QDM 的 *OutputTierNode* 结构中; 对于 select*, 则通过解析获取查询依赖的表名, 调用元数据获取接口获取得到该表对应的字段信息, 插入 Sub-QDM 的 *OutputTierNode* 结构中.

通过算法 2, 示例 SQL 自顶向下生成了 3 个 Sub-QDM 的查询输出信息: 最顶层为查询请求的最终输出字段(*null*,*null*,*id*,*null*), 此时并不知道该字段最终来源; 其他层的查询输出字段分别为是第 2 层 Sub-QDM 中的(*null*,*null*,*id*,*null*), (*null*,*null*,*username*,*null*)以及第 3 层 Sub-QDM 中的(*null*,*null*,*id*,*null*), (*null*,*null*,*class*,*null*), (*null*,*null*,*username*,*null*).

- 查询输出字段依赖信息生成算法

查询输出字段依赖信息生成算法(算法 3)通过遍历 from 从句对应的 AST 子树, 获得本层查询字段来源信息, 从而构造出一个 Sub-QDM; 对语法树中的子查询等语法结构继续深度遍历, 通过递归逐层构造出对应的 Sub-QDM 模型, 最终构成一个完整的 QDM 模型. 主要步骤: 首先, 从 AST 根节点开始, 通过遍历直接定位到 from 从句根节点从这里开始分析(第 3 行); 接着遍历 from 子树, 直到 from 从句遍历完毕, 遍历过程中, 根据 AST 节点类型进行不同处理(第 4–15 行). 对子查询, 此类 SQL 语句在 AST 树会生成对应的嵌套子查询节点, 采用深度遍历方式, 递归对子查询节点进行解析, 直到获取最内层该子查询所对应的表名及字段信息;对 CTE (common table expression)节点遍历分析时, 采用同样的方法获取 CTE 所依赖的数据表名及字段信息; 如果嵌套子查询中包含 union, join 或其他连接类节点, 在对嵌套子查询遍历解析过程中, 需要对此类节点进行解析, 取出该节点内部包含的所有表及对应字段信息; 对无子查询, 通过解析 AST 节点, 直接得到依赖的表名(子查询别名)、字段名(别名).

算法 3. 查询输出字段依赖信息生成算法.

输入: 待分析的 AST 根节点 *ASTRoot*, AST 节点类型 *Nodetype*, 依赖层次节点 *DependencyTierNode*.

输出: *NULL*.

```
1.  walkFromAST(ASTRoot,Nodetype,DependencyTierNode)
2.  {
3.      node=walkToAST(ASTRoot,Nodetype);   //遍历 AST, 定位到 from 根节点, 从这里开始遍历
4.      while (子树没有遍历完成)
5.      {
6.          if (如果节点为子查询节点, 如 SubQuery/CTE 等情况)
7.              tTierNode=walkAST(node);
8.              //递归对子查询节点解析, 每个 SQL 子查询会创建一个 Sub-QDM
9.              DependencyTierNode→next=tTierNode;   //建立指向关系
10.         if (union/join 类的节点)
11.             WalkJoinSubtree(node,DependencyTierNode);   //获取左右表信息
12.         if (无子查询)
13.             WalkSingleFrom(node,DependencyTierNode);   //无子查询, 获取依赖信息更新
14.         node=取 AST 树下一个节点;
15.     }
16. }
```

通过算法 3, 示例 SQL 自顶向下生成了生成了 3 个 Sub-QDM: 最顶层获取的查询输出字段来源为(*null*,*t*,*null*,*null*), 其实它并不是查询输出字段(*null*,*null*,*id*,*null*)的最终来源; 第 2 层 Sub-QDM 中的查询输出字段来源(*null*,*info*,*null*,*null*), 第 3 层 Sub-QDM 中的查询输出字段来源(*tInfo*,*null*,*null*,*null*).

算法 2 和算法 3 对应的伪代码是对抽象语法树进行深度遍历并解析的过程, 在实际遍历及解析过程中, 需

要处理所有涉及的 SQL 语法结构, 否则会导致语法兼容问题. 限于篇幅, 算法 2 和算法 3 没有展开.

- 最终依赖更新算法

最终依赖更新算法(算法 4)的目的是, 获取最外层所有查询输出字段所最终依赖的表字段信息. 该算法对最外层所有查询输出字段, 逐个以表名、子查询别名、字段名、字段别名信息作为匹配关键字, 逐层向下一层进行匹配, 直到匹配到最深一层, 算法默认将最深层依赖信息作为最外层输出字段最终来源.

算法 4. 最终依赖更新算法.

输入: QDM 查询模型树最外层节点 *OutputTierNode*.

输出: *NULL*.

```
1.  updateOutputTier(OutputTierNode)
2.  {
3.      for each outputnode in OutputTierNode
4.          matchColumnNode=outputnode;
5.          matchColumnNode=SearchAllTier(OutputTierNode,matchColumnNode);   //逐层匹配查找依赖
6.          UpdateNode(outputnode,matchColumnNode);   //更新依赖
7.  }
```

经过算法 2 和算法 3, 示例 SQL 生成 QDM 简化模型为:

- 最外层 Sub-QDM: (*null*,*null*,*id*,*null*)→(*null*,*t*,*null*,*null*);
- 第 2 层 Sub-QDM: (*null*,*null*,*id*,*null*)→(*null*,*info*,*null*,*null*);
- 最内层 Sub-QDM: (*null*,*null*,*id*,*null*)→(*tInfo*,*null*,*null*,*null*).

示例 SQL 通过算法 4, 其查询最外层输出字段(*null*,*null*,*id*,*null*)所最终依赖的字段为(*tInfo*,*null*,*id*,*null*), 对应物理表 *tInfo* 的 *id* 字段. 至此, 完整的 QDM 模型生成.

查询依赖模型构建算法调用算法 2 和算法 3 递归遍历抽象语法树的所有节点, 时间复杂度为 O(*n*), 其中, *n* 为语法树的高度. 在 QDM 模型有 *m* 层 Sub-QDM, 每层有 *d* 个 *ColumnNode* 的情况下, 算法 4 的时间复杂度为 O(*m*×*d*). 因此, 总的时间复杂度为 O(*n*+*m*×*d*), *m* 和 *d* 通常不会很大. 该构建算法为解决 SQL 改写问题而引入的新流程, 相当于在 SQL 语句执行过程中对抽象语法树多做了一次解析, 由于 SQL 语句最终会转换为分布式任务, 任务执行过程中的大量的 IO、计算等消耗决定了性能损失, 该算法对 SQL 执行的性能响微乎其微.

2.3.3 SQL 改写

同一个字段可能出现在 SQL 查询语句的各个语法结构中, 采用浅层解析直接改写 SQL 语句中所有定义脱敏策略字段的简单处理方式, 改写后, SQL 语句与原始请求往往不一致, 造成返回结果集不正确. 而规则引擎通过安全规则树解析 SQL 请求, 并使用规则重写 SQL 语句, 查询请求越复杂, 识别难度越高, 匹配的准确率就越低.

对于 create table as select、insert (overwrite) into select、视图等语法, 也会通过 select 查询获取敏感信息, 如果不能识别并自动继承脱敏策略, 也会造成敏感数据泄露. Create table as select 及 insert (overwrite) into select 中的 select 子句中有敏感字段, 且该字段设置了脱敏规则的情况, BDMasker 提供了两种处理模式对此类 SQL 进行自动脱敏转换, 以保护派生表的敏感数据.

- 派生模式: 将 create table as select 及 insert (overwrite) into select 的 select 从句查询输出字段名(别名)所依赖的表字段名(别名)对应的脱敏规则作为查询输出字段的脱敏规则, 将该脱敏规则插入到保护策略库, 当对派生表进行查询时, 按插入的脱敏规则进行脱敏转换. 这种模式不修改派生表原始数据, 为本算法的默认处理模式.
- 改写模式: 对 create table as select 及 insert (overwrite) into select 的 select 从句进行改写, 在执行 create table as select 及 insert (overwrite) into select 之后, 派生表的数据为脱敏后的数据.

算法 5. SQL 改写算法.

输入: AST 根节点 *ASTRoot*, QDM 最外层层次节点 *Outermost TierNode*.

输出: 改写后的 AST 根节点 *ASTRoot*.

```
1.  ASTRoot RewriteSQL(ASTRoot,OutputTierNode)
2.  {
3.    ASTnodes=walkOutSelectAST(ASTRoot);   //获取查询输出最外层输出节点的对应 AST 节点
4.    table=walkInsertOrCreateAST(ASTRoot);   //insert 或者 create 语句可以获取对应的表名信息
5.    bInherit=getCTASMaskPolicy(table);
6.    for each astnode in ASTnodes
7.      qdmnode=getDependencyColumnNode(astnode);   //从 QDM 中获取对应的依赖表字段
8.      maskPolicy=getMaskPolicy(qdmnode);   //获取字段脱敏策略
9.      if (maskPolicy!=NULL)
10.       if (table!=NULL && bInherit==TRUE)
11.         addOneMaskPolicy(qdmnode,maskPolicy);   //派生表直接继承父表字段脱敏策略
12.       tmpSQLSeg=ConSQLSegment(astnode,maskPolicy);   //使用脱敏算子, 构造 SQL 片段
13.       ReWriteASTNode(astnode,tmpSQLSeg);   //替代原来 AST 节点值
14.   return ASTRoot;
15. }
```

对示例 SQL 中 *tinfo* 表的 *id* 字段设置 *mask*(*f*,'*')的脱敏规则, 其中, *mask* 为脱敏算子函数, *f* 为待脱敏字段名, '*'为脱敏参数.

图 7 描述了查询改写算法 5 流程.

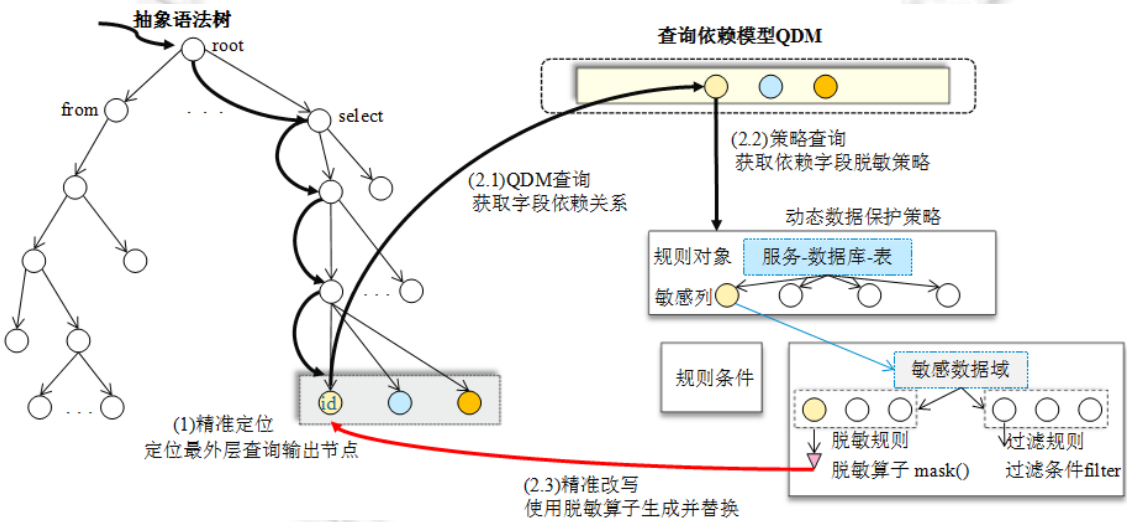


图 7 查询改写流程图

与传统方法不同, 算法 5 以精准定位和精准改写为设计原则. 算法首先通过对抽象语法树遍历, 精准定位到最外层查询所有输出节点, 只修改最外层查询输出字段对应的 AST 节点, 不影响语法树中其他节点(第 3 行, 图 7 步骤 1). 对于 create table as select, insert (overwrite) into select 语句需要获取派生表名及该派生表脱敏模式(第 4、5 行). 接下来, 对于最外层查询输出节点, 逐个完成如下主要处理逻辑: 从查询依赖模型 QDM 中获取当前节点最终依赖表字段名(第 7 行, 图 7 步骤 2.1), 然后调用策略查询接口, 获取当前节点最终依赖表字段

名对应的脱敏策略, 包括脱敏算子函数(第 8 行, 图 7 步骤 2.2), 如果派生表脱敏模式为派生模式, 则新建一条脱敏规则, 将父表字段的脱敏规则作该派生表字段的脱敏规则, 以避免派生表的数据泄露(第 11 行). 最后, 再以脱敏算子函数为基础构造一段 SQL 片段, 以该 SQL 片段值替换当前 AST 节点值(第 12、13 行,图 7 步骤 2.3).

在抽象语法树中, 本算法直接精准定位最外层 select 的 *id* 字段对应的 AST 节点, 而不会修改其他任何节点. 同时, 获取 *id* 最终依赖的字段为(*tInfo*,*null*,*id*,*null*), 也就是物理表 *tInfo* 的 *id* 字段, 获取该字段的脱敏算子 *mask*(*f*,'*'), 以 *mask*(*id*,'*')这一段 SQL 替换原来最外层输出字段 *id* 对应的 AST 值.

经过算法 5 流程后, 示例 SQL 被改写为:

select *mask*(*id*,'*') **from**
  (**select** *id*, *username* **from**
    (**select** *class*, *id*, *username* **from** *tInfo*) *Info*
) *t*

可以看出, 此示例 SQL 语句被精准改写, 原始 SQL 请求中子查询中的多个 *id* 字段并没有被改写, 有效保证了原始查询请求的业务逻辑正确性, 从而实现了对业务透明.

SQL 改写算法是按语法结构递归遍历抽象语法树, 但不需要遍历所有的节点, 抽象语法树的时间复杂度为 O(*n*), 在最外层查询输出字段为 *m* 个的情况下, 算法总的时间复杂度为 O(*n*+*m*), *m* 通常不会很大. 该算法不需要遍历所有节点, 对 SQL 执行性能影响微乎其微.

3 实验分析与评估

3.1 实验环境

实验在 15 台服务器组成的大数据分布式环境上运行, 服务器间通过 1 台万兆交换机互联. 每台服务器具体软硬件配置见表 1. 实验选用的测试模型及数据集包括 TPC Benchmark DS (TPC-DS)和 YCSB Benchmark (YCSB)[21]. TPC-DS 是用于评测决策支持系统(或数据仓库)的基准测试集, 与真实场景非常接近, 也是难度较大的一个测试集. YCSB 是一款通用的 NOSQL 性能测试工具.

表 1 实验环境软硬件配置

名称	规格参数
操作系统	NewStart CGS Linux V5
JDK	1.8
Hadoop	2.7.3
Hive	2.1.0
Spark	2.2.1
HBase	1.2.0
CPU	48 core Intel(R) Xeon(R) CPU E5-2670 v3@2.30 GHz×4
内存	128 GB×1
硬盘	4 TB SATA×12
网卡	双口万兆网卡×1

实验采用脱敏策略保留模式, 后面一条测试语句默认拥有前面所有已测语句的脱敏策略. 每条测试语句仅 1–2 个字段不设置脱敏规则, 其他字段均设置脱敏规则.

性能实验使用脱敏执行前后执行时间波动比例作为系统运行效率的量化考核指标, 每项用例测试 4 次, 每次执行前执行 Linux 清理缓存操作, 取 4 次测试的平均值做统计分析.

波动比例=$|(t_1-t_0)|/t_0$. 其中, t_1 为打开脱敏开关运行时间, t_0 为关闭脱敏开关运行时间.

3.2 功能实验

实验选取目前开源大数据社区唯一支持动态脱敏功能的 Apache Hive 与 BDMasker 系统 Hive 进行动态脱敏功能对比, 从业务逻辑正确性、语法兼容性、安全性和能力可扩展性这 4 个方面评价系统的功能.

(1) 业务逻辑正确性. 该指标作为最严格的功能评价指标, 即当且仅当算法修改后查询请求与用户所发

出的真实查询请求结果集完全相同时, 视为动态脱敏功能完全正确. 衡量业务逻辑正确性的指标有3个: 结果集条目正确性、脱敏字段来源正确性和脱敏算法正确性.

(2) 语法兼容性. TPC-DS 遵循 SQL'99 和 SQL 2003 语法标准, SQL 案例比较复杂, 使用 TPC-DS 99 条 SQL 测试案例, 对 SQL 语句的每个语法部分出现的字段设置脱敏规则, 验证脱敏后业务逻辑及语法规则支持能力.

(3) 安全性. 该指标主要测试对基于查询请求生成的派生表、插入表、视图等安全性进行测试, 防止敏感数据从此类表中泄露.

(4) 能力扩展. 该指标主要测试 BDMasker 系统的动态数据保护引擎的安全功能扩展能力.

3.2.1 业务逻辑正确性实验

以 Query76 为例, 验证 BDMasker 系统修改后的 SQL 语句的业务逻辑正确性. 对 *item* 表 *i_category* 字段设置 *mask_account_info* 脱敏规则, 该规则对应的算子为 *mask_account_info_caesar_string*. BDMasker 系统对 Query76 改写后生成的 SQL 如下:

SELECT *channel*, *col_name*, *d_year*, *d_qoy*, *mask_account_info_caesar_string*(*i_category*),
COUNT(*) *sales_cnt*, *SUM*(*ext_sales_price*) *sales_amt*

其余语句保持不变.

从图 8 的执行结果集可以看出, BDMasker 系统对 Query76 进行脱敏前后结果集条目正确, 均为 10 条. 这是因为 BDMasker 系统通过 QDM 模型精确获取了最外层查询输出字段的来源以及对应的脱敏规则, 改写后的 SQL 语句对原始业务逻辑没有影响. 本例中, 最外层查询输出字段 *i_category* 来源为物理表 *item* 的 *i_category* 字段, 该字段的脱敏规则为 *mask_account_info*. 改写后的语句仅涉及最外层查询输出字段 *i_category*, 对其他语法结构没有破坏.

				脱敏前	脱敏后		
channel	col_name	d_year	d_qoy	i_category	i_category	sales_cnt	sales_amt
catalog1	cs_warehouse_sk	1998	1			23	13777.69000002
catalog1	cs_warehouse_sk	1998	1	Books	Xgnkq	988	1471310.43
catalog1	cs_warehouse_sk	1998	1	Children	Achdakaf	986	1297900.1000
catalog1	cs_warehouse_sk	1998	1	Electronics	Dcwunqg'bbj	1059	1280541.3999
catalog1	cs_warehouse_sk	1998	1	Home	Emhd	961	1280542.079999
catalog2	cs_warehouse_sk	1998	1	Jewelry	IBgry	1013	1242774.92
catalog2	cs_warehouse_sk	1998	1	Men	Kdj	1031	1343043.36999
catalog2	cs_warehouse_sk	1998	1	Music	Hpncv	1108	1430996.789
catalog2	cs_warehouse_sk	1998	1	Shoes	Nzfzm	1018	1330079.92
catalog2	cs_warehouse_sk	1998	1	Sports	Jlgmmmm	1016	1310913.58

图 8 Query76 业务逻辑正确性验证结果

按顺序逐条执行 TPC-DS 99 语句, 业务逻辑正确性对比结果见表 2.

表 2 业务逻辑正确性测试结果

功能指标	Apache Hive	BDMasker Hive
结果集条目	均不正确	正确
脱敏列字段来源正确性	均不正确	正确
脱敏算法正确性	正确	正确

BDMasker Hive 通过了基于 QDM 的精准查询分析找到 SQL 语句所涉及的表及字段间的依赖关系, 通过

精准改写算法, 最终仅对 select 查询最外层输出字段中的敏感字段进行脱敏转换, 从而保证了业务逻辑不受脱敏过程影响. 而 Apache Hive 由于采用了浅层解析模式, 导致改写后的 SQL 改变了原始业务逻辑. 原因是其改写 SQL 算法不正确, 表现在: ① 脱敏字段出现在 WHERE 条件中会先进行脱敏, 再将脱敏后的字段应用到 WHERE 条件中; ② 脱敏字段出现在任何子查询中, 会从子查询就开始脱敏应用到父查询中又再次应用; ③ 脱敏字段出现在 GROUP BY/ORDER BY/SORT BY/DISTRBUTE BY/JOIN ON 会先进行脱敏, 再将脱敏后的字段应用到 GROUP BY/ORDER BY/SORT BY/DISTRBUTE BY 语句中; ④ 脱敏字段出现在任何 UDF 函数中, 比如 SUM/AVG/UPPER 等会先进行脱敏, 再将脱敏后的字段作入参应用到 UDF 函数中.

3.2.2 语法兼容性实验

按顺序对 TPC-DS 99 条语句进行测试, 语法兼容性实验对比结果见表 3.

表 3 语法兼容性测试结果

语法	Apache Hive	BDMasker Hive
简单 Select 查询	支持	支持
嵌套子查询	均不支持	支持
条件过滤查询	均不支持	支持
多表关联	均不支持	支持
distinct 排重	均不支持	支持
Union 和 Union all	均不支持	支持
排序	均不支持	支持
分组聚合函数	均不支持	支持
Create table as select	均不支持	支持
Insert (overwrite) into	均不支持	支持
CTE	均不支持	支持
View	均不支持	支持
udf 解析	均不支持	支持
distinct 排重	均不支持	支持

从实验结果可知, Apache Hive 不支持多表关联、条件过滤等多种复杂 SQL 语法. 这是由于 Apache Hive 采用浅层解析, 对大量的语法结构不进行深度解析, 导致脱敏结果错误. BDMasker 系统的查询分析及改写算法对每一种语法结构进行解析及分析, 并使用 TPC-DS 99 条严格进行语法兼容性测试, 从而保证了语法兼容.

3.2.3 安全性实验

安全保护能力实验测试基于查询请求生成的派生表、插入表、视图的脱敏能力. 通过构造 create table as select, insert (overwrite) into select, View(包括嵌套视图)语句, 安全能力实验对比结果见表 4.

表 4 安全保护能力测试结果

语句	Apache Hive	BDMasker 系统 Hive
create table as select	会泄露敏感数据	有效保护
Insert (overwrite) into select	会泄露敏感数据	有效保护
View	会泄露敏感数据	有效保护

从实验结果可知, create table as select, insert (overwrite) into select 中的 select 子句中有敏感字段, 且该字段设置了脱敏规则的情况, BDMasker 系统提供了两种处理模式保护此类 SQL 中的敏感数据, 能够自动识别并继承父表字段的脱敏策略, 从而自动保护派生表安全. 同时, 也支持视图及视图继承脱敏, 能够有效避免敏感数据从派生表或者视图中泄露. 而 Apache Hive 对 create table as select, insert (overwrite) into select, View 等派生出来的表由于没有自动应用脱敏规则, 导致从派生表中查询的结果集仍然是原始数据, 从而造成了敏感数据的泄露.

3.2.4 能力扩展实验

BDMasker 在动态数据保护引擎中, 通过插件式功能实现动态数据过滤功能的扩展. 以 Query76 为例, 对 *web_sales* 表 *web* 字段设置过滤规则, 过滤该字段值等于 *catalog*1 的行数据, 同时对 *item* 表 *i_category* 字段设置 *mask_account_info* 脱敏规则, 实验结果如图 9 所示.

脱敏后

channel	col_name	d_year	d_qoy	i_category	sales_cnt	sales_amt
结果集被过滤						
catalog2	cs_warehouse_sk	1998	1	IBgry	1013	1242774.92
catalog2	cs_warehouse_sk	1998	1	Kdj	1031	1343043.36999
catalog2	cs_warehouse_sk	1998	1	Hpncv	1108	1430996.789
catalog2	cs_warehouse_sk	1998	1	Nzfzm	1018	1330079.92
catalog2	cs_warehouse_sk	1998	1	Jlgmmmm	1016	1310913.58

图 9 数据保护能力扩展实验结果

从执行结果集可以看出, BDMasker 对 Query76 结果集进行了正确的过滤和脱敏. BDMakser 在精准改写时遍历 AST, 判断当前节点为叶子节点且是一个物理表时, 则调用策略查询接口获取物理表的数据过滤规则, 为该叶子节点新增一个子查询节点, 该叶子节点的原父节点成为子查询节点的父节点, 叶子节点变成子查询节点的子节点, 最后将获取的过滤规则对应的 SQL 语句应用到该子查询节点. 通过这种方式改写后的 SQL 语句执行时自动对该物理表的数据按预定规则进行动态过滤, 这个处理逻辑在 BDMasker 系统中以插件式方式实现, 从而在同一个大数据引擎中以多种技术同时叠加共同保护敏感数据, 有效解决了单个引擎的纵向动态数据安全能力的扩展问题.

3.3 YCSB性能实验

- 单用户基准测试

通过 YCSB 工具设置字段长度、字段数目、总记录数等压测参数, 构造 2 TB 测试数据. 依次执行随机写、随机读、随机扫描命令测量单用户模式下基准的查询响应时间. 单用户基准测试结果如图 10 所示.

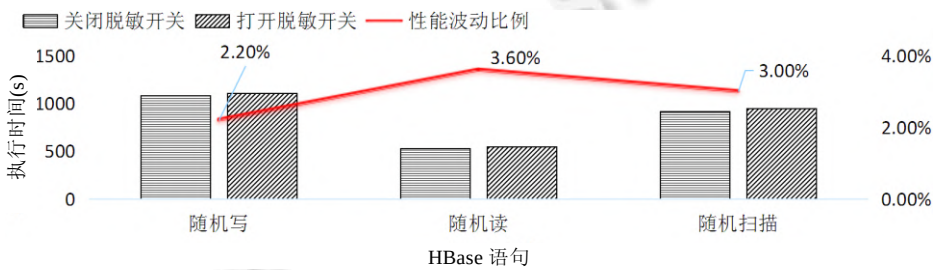


图 10 BDMasker HBase 动态脱敏性能波动图

从实验结果可知, 在单用户模式下, BDMasker HBase 始终保持较为高效的性能表现, 脱敏过程对业务性能影响较小, 随机写性能波动为 2.2%, 随机读性能波动为 3.6%, 随机扫描性能波动为 3.0%, 整体平均性能损失均在 3%以内. 随机读与随机扫描的性能损耗高于平均值, 这是因为随机读与随机扫描的过程中需要对数据进行脱敏转换. 随机扫描每次读取的数据集多且集中在一个节点上进行转换, 内存使用相对集中, 而随机读数据散列在多个节点, 每次读取单条记录脱敏转换, 内存利用碎片化, 从而导致随机读的性能损耗相对高于随机扫描.

- 多并发场景下, HBase 引擎动态脱敏性能测试

通过调整并发规模验证系统在不同并发数的性能表现, 实验数据规模为 2 TB, 并发规模依次为 10, 20, 50, 100, 依次执行随机写、随机读、随机扫描命令, 测试不同并发数下动态脱敏性能波动. 实验结果如图 11 所示.

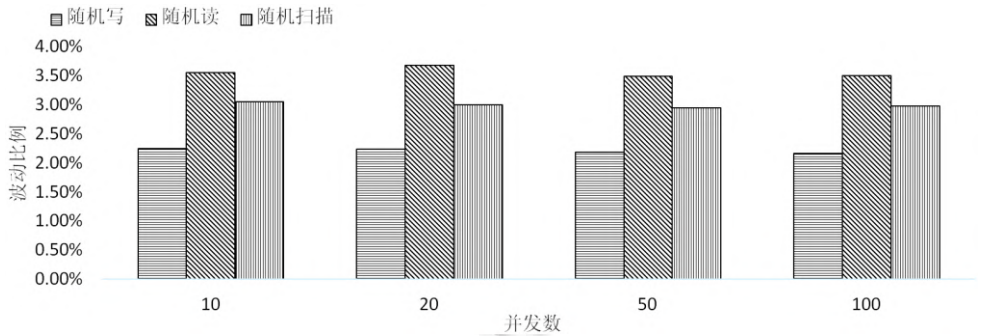


图 11 BDMasker HBase 在不同并发规模下动态脱敏性能波动图

从实验结果可知, 不同并发数下, BDMasker HBase 随机写性能波动平均 2.22%, 随机读性能波动为平均 3.55%, 随机扫描性能波动平均为 2.95%, 整体性能波动稳定在 3%以内. 并发规模的增大, 并不影响 HBase 动态脱敏性能变化值, BDMasker HBase 在不同并发规模下, 始终保持较为高效的性能表现, 动态脱敏对业务性能影响较小, 能够在开销不大的情况下有效保护数据安全.

- 不同规模数据集场景下, HBase 引擎动态脱敏性能测试

通过调整实验数据规模, 验证系统在不同数据规模下的性能表现, 实验数据规模依次为 100 GB, 500 GB, 2 TB, 10 TB, 50 TB, 100 TB, 依次执行随机写、随机读、随机扫描命令, 测试不同数据规模下, 动态脱敏性能波动. 实验结果如图 12 所示.

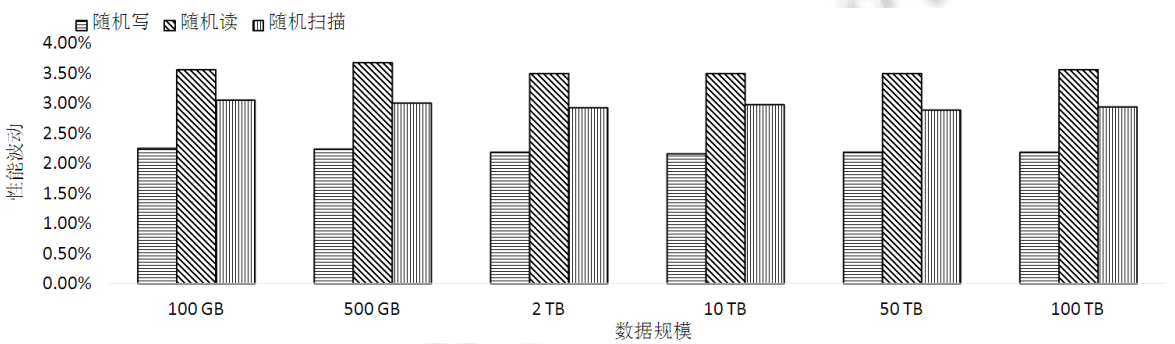


图 12 BDMasker HBase 在不同规模数据集下, 动态脱敏性能波动图

从实验结果可知, 不同数据规模下, BDMasker HBase 随机写性能波动平均 2.21%, 随机读性能波动为平均 3.54%, 随机扫描性能波动平均为 2.96%, 整体性能波动稳定在 3%以内. 数据规模增大, 并不影响动态脱敏性能变化值, BDMasker HBase 在不同数据规模下, 始终保持较为高效的性能表现, 动态脱敏对业务性能影响较小, 能够在开销不大的情况下有效保护数据安全.

3.4 多引擎扩展性及性能实验

从 TPC-DS 99 条语句中挑选 20 条 SQL(见表 5), 这些语句业务场景各异, 业务逻辑复杂度涵盖高中低场景, 语法覆盖子查询、Join、聚合、分组、条件等常用 SQL 语法. 在 Apache Hive, Apache Spark 引擎上验证 BDMasker 系统的多引擎扩展性及性能.

表 5 TPC-DC 性能测试选取的查询语句

编号	涉及语法
query3	Select, group by, order by, limit, where, function 等.
query7	Select, group by, order by, limit, where, function 等.
query12	Select, group by, order by, limit, where, function, in, between, and 等.
query15	Select, group by, order by, limit, where, function, in, and 等.
query17	Select, group by, order by, limit, where, in, and 等.
query19	Select, group by, order by, limit, where, function 等.
query20	Select, group by, order by, limit, where, in, and 等.
query21	Select, 嵌套 select, group by, order by, limit, where, function, between, and 等.
query24	Select, 嵌套 select, group by, distribute by, order by, having, where, CTE 等.
query25	Select, group by, order by, limit, where, function, in, between, and 等.
query26	Select, group by, order by, limit, where, function 等.
query27	Select, group by, order by, limit, where, function 等.
query28	Select, 嵌套 select, group by, order by, limit, where, function, in, between, and 等.
query31	Select, 嵌套 select, group by, order by, limit, where, function, in, between, and 等.
query40	Select, 嵌套 select, group by, distribute by, order by, having, where, CTE, join 等.
query42	Select, group by, order by, limit, where, function 等.
query47	Select, 嵌套 select, group by, distribute by, order by, having, where, CTE 等.
query53	Select, 嵌套 select, order by, limit, where, function, and 等.
query82	Select, group by, order by, limit, where, function, in, between, and 等.
query84	Select, order by, limit, having, where 等.

- 单用户基准性能测试

单用户基准性能测试用于测量单用户场景下基准查询响应时间，并计算性能波动比例，数据规模为 2 TB. BDMasker 系统 Hive, Spark 动态脱敏性能波动分别如图 13、图 14 所示.

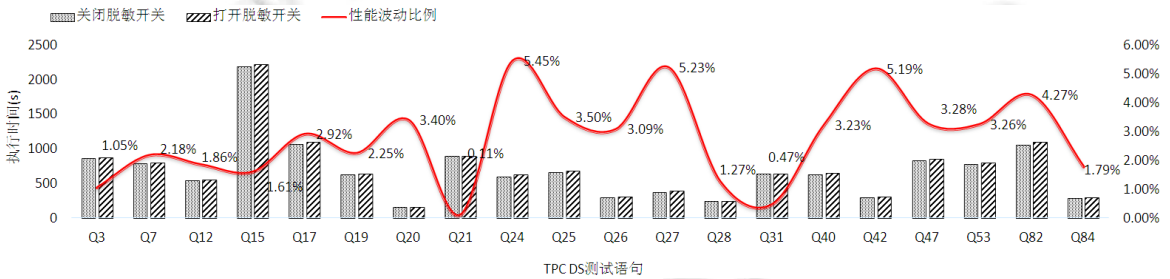


图 13 BDMasker Hive 动态脱敏性能波动图

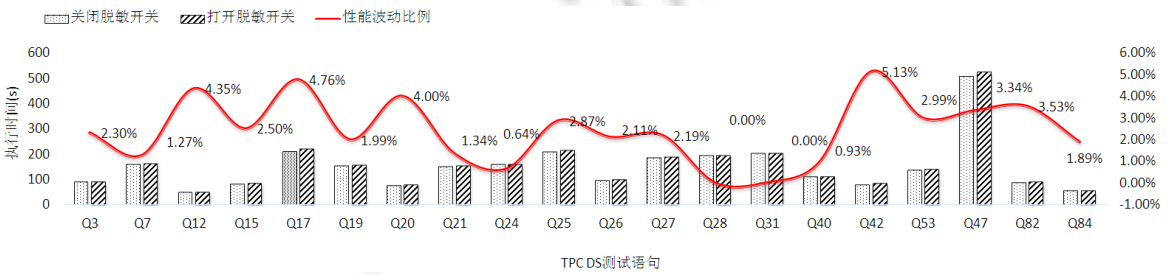


图 14 BDMasker Spark 动态脱敏性能波动图

从图 13、图 14 可以看出，在单用户模式下, BDMasker 保持了较为高效的性能表现，脱敏过程对业务性能影响较小，平均性能损失 3%. 这是由于 BDMasker 的精准 SQL 改写没有破坏现有的抽象语法树的结构，仅对最外层查询输出节点的值改写；改写后的 SQL 语句涉及最外层查询输出字段，其他语法结构不受影响，从而保证改写不会影响执行引擎继续应用各种查询优化策略，逻辑优化、物理优化与脱敏改写没有任何冲突.

通过纵向对比可以看出，BDMasker 系统 Hive 和 Spark 动态脱敏性能变化保持了相似的波动趋势，Spark

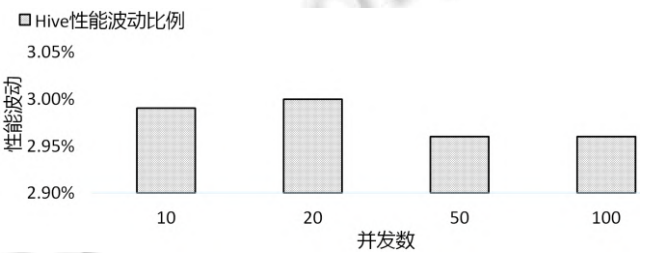
与 Hive 对同一条 SQL 的处理效率差异较大, 如 Query24. 这是由于 Hive 与 Spark 内部优化和执行机制差异较大, 导致同一个 SQL 在相同的脱敏规则下有不同的性能表现.

某些查询语句性能波动超过 3%, 这是因为此类 SQL 语句脱敏前的执行时间较短, 而脱敏策略又设置较多复杂的脱敏算法, 脱敏算法的处理占整个任务计算比例较大, 转换过程消耗了较多的 CPU, 导致性能波动值较大. 例如图 14 中 Spark 的 Quer12 语句, 脱敏前运行时间 46 s, 脱敏后执行时间为 48 s, 性能波动比例为 4.35%; Query42 语句脱敏前运行时间 78 s, 脱敏后执行时间为 82 s, 性能波动比例为 5.13%.

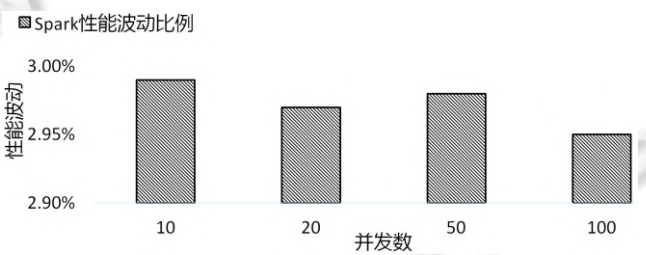
本实验同时验证了 BDMasker 具有良好的扩展能力. BDMasker 基于面向多引擎的统一策略技术实现了对 Hive, Spark 等引擎的统一管理, 该技术能够适用于大部分 SQL-on-Hadoop 引擎, 具有较强的扩展性.

- 多并发场景下, SQL 引擎动态脱敏性能测试

通过调整并发规模验证系统在不同并发数的性能表现, 实验的并发规模依次为 10, 20, 50, 100, 性能波动比例取该并发规模下, 测试表 5 中 20 条 SQL 语句执行时间波动比例的平均值. BDMasker 系统的 Hive 及 Spark 动态脱敏性能随并发数量增加的测试情况如图 15 所示.



(a) BDMasker Hive



(b) BDMasker Spark

图 15 BDMasker 系统多并发场景下, SQL 引擎动态脱敏性能波动图

从实验结果可知, BDMasker 在不同并发数下始终保持了较好的性能表现, 平均性能损失在 3%以内, Hive 与 Spark 引擎的动态数据脱敏性能波动较为平稳. 说明 BDMasker 系统能够在和非脱敏开销相差不大的情况下, 有效保护数据安全. 另外, 由于改写后的 SQL 语句最终会转化为大数据任务在集群上分布式执行, 动态脱敏的性能波动取决于执行过程中数据脱敏算子的性能消耗, QDM 模型构建及 SQL 改写算法在整个作业运行性能过程中的影响可以忽略.

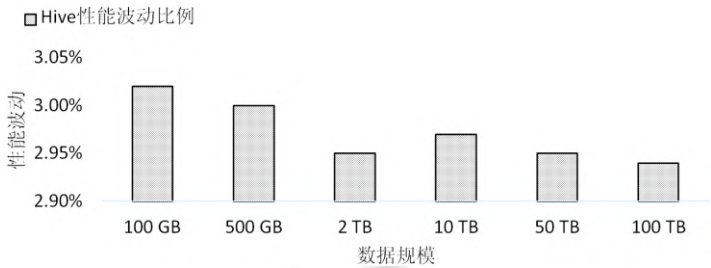
- 不同规模数据集场景下, SQL 引擎动态脱敏性能测试

通过调整数据集规模验证系统在不同数据规模下的性能表现, 数据规模分别为 100 GB, 500 GB, 2 TB, 10 TB, 50 TB, 100 TB, 性能波动比例取该数据规模下, 测试表 5 中 20 条 SQL 语句执行时间波动比例的平均值. BDMasker 系统 Hive 及 Spark SQL 的动态脱敏性能随数据规模增加的时间开销如图 16 所示.

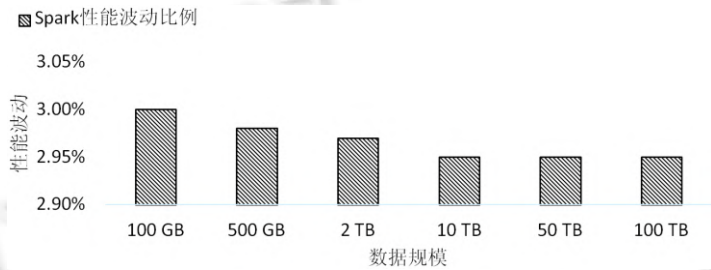
从实验结果可知, BDMasker 在 6 种不同的数据规模实验中始终保持了较好的性能表现, 平均性能损失均在 3%以内. Hive 与 Spark 引擎的动态数据脱敏性能波动较为平稳, 说明 BDMasker 系统能够在和非脱敏开销相差不大的情况下, 有效保护数据安全.

综合上述实验, BDMasker 系统通过了 TPC-DS 99 条语句严格的业务逻辑正确性及 SQL 语法兼容性测试,

实现了动态数据脱敏对原始业务请求的完全透明, 解决了 SQL 改写精准性的挑战. TPC-DS 及 YCSB 的性能实验结果表明, BDMasker 系统可以同时支持多个异构大数据引擎的脱敏能力, 脱敏过程充分利用了大数据执行引擎的分布式计算能力进行高效脱敏, 平均性能损失轻微, 整体性能波动控制在 3%以内, 解决了动态数据保护处理性能高效性和异构环境扩展性的挑战. 理论分析表明, 同样的硬件环境下, 脱敏网关模式相对于 BDMasker 系统增加了数据访问路径, 脱敏耗时与结果集容量线性相关, 整体性能损耗较大, BDMasker 系统在大数据执行引擎内核直接脱敏的效率显著优于脱敏网关模式, 性能差距取决于结果集处理过程的耗时.



(a) BDMasker Hive



(b) BDMasker Spark

图 16 BDMasker 系统在不同规模数据集场景下, SQL 引擎动态脱敏性能波动图

4 总 结

数据的安全问题已成为制约大数据技术发展与利用的瓶颈. 开放大数据环境下, 数据安全的保护方式、保护对象、管理和技术的关系均发生了变化, 对数据的安全防护提出了更高的要求, 传统的数据安全措施已不再适用. 本文设计并实现了一种针对开放大数据环境的高性能动态数据保护系统 BDMasker,能够精准、高效、可扩展地动态保护敏感数据. 基于“业务零感知”的原则, 提出了一种基于查询依赖模型的精准查询分析及 SQL 改写技术, 能够精准感知但不改变原始业务请求, 实现动态脱敏过程对业务零影响; 设计了一种面向多引擎的统一安全策略框架, 通过插件式数据保护策略管理支持多种动态数据保护能力的纵向扩展, 可以根据应用场景、业务目标和数据特征选择合适的数据保护策略, 通过策略代理和标准化接口实现了多种计算引擎的横向扩展, 能够利用大数据执行引擎的分布式计算能力提升系统的数据保护处理性能. 最后,通过实验证明了 BDMasker 系统的有效性、良好的性能及可扩展能力.

在下一步研究中, BDMasker 将扩展支持 AI 驱动的动态数据保护, 防止恶意用户使用推理或暴力技术绕过脱敏[22], 研究如何结合 AI 提炼出更通用的方法, 使其更容易适配大数据引擎. 未来, BDMasker 将紧跟数据安全法等国家法律法规对数据安全保护工作的要求[23], 通过结合数据安全技术体系和新技术应用, 提供更好更丰富的数据安全服务, 保障数据在新时期、新标准、新环境、新要求背景下的使用安全.

References:

[1] Qian WJ, Shen QN, Wu PF, Dong CT, Wu ZH. Research progress on privacy-preserving techniques in big data computing environment. Chinese Journal of Computers, 2022, 45(4): 669−701 (in Chinese with English abstract).

[2] Fang BX, Jia Y, Li AP, Jiang R. Privacy preservation in big data: A survey. Big Data Research, 2016, 2(1):1−18 (in Chinese with English abstract). [doi: 10.11959/j.issn.2096-0271.2016001]

[3] Wu XD, Dong BB, Du XZ, Yang W. Data governance technology. Ruan Jian Xue Bao/Journal of Software, 2019, 30(9): 2830−2856 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5854.htm [doi: 10.13328/j.cnki.jos.005854]

[4] Wang Z, Liu GW, Wang Y, Li Y. Research on the development and trend of data masking technology. Information and Communications Technology and Policy, 2020, 46(4): 18−22 (in Chinese with English abstract).

[5] Chen XY, Gao YZ, Tang HL, Du XH. Research progress on big data security technology. SCIENTIA SINICA Informationis, 2020, 50(1): 25−66 (in Chinese with English abstract). [doi: 10.1360/N112019-00077]

[6] Tong LL, Li PX, Duan DS, Ren BY, Li YX. Data masking model for heterogeneous big data environment. Journal of Beijing University of Aeronautics and Astronautics, 2022, 48(2): 249−257 (in Chinese with English abstract).

[7] Li SY, Ji YD, Shi DY, Liao WD, Zhang LP, Tong YX, Xu K. Data federation system for multi-party security. Ruan Jian Xue Bao/Journal of Software, 2022, 33(3): 1111−1127 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6458.htm [doi: 10.13328/j.cnki.jos.006458]

[8] Zaharia M, Das T, Li H, Hunter T, Shenker S, Stoica I. Discretized streams: Fault-tolerant streaming computation at scale. In: Proc. of the 24th ACM Symp. on Operating Systems Principles. New York: ACM, 2013. 423−438.

[9] Carbone P, Katsifodimos A, Ewen S, Markl V, Haridi S, Tzoumas K. Apache flink: Stream and batch processing in a single engine. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 2015, 36(4): 28−38.

[10] Liu BX. Design and implementation of performance test tool based on TPC-DS [MS. Thesis]. Dalian: Dalian University of Technology, 2018 (in Chinese with English abstract).

[11] Manjunath TN, Hegadi RS, Mohan HS. Automated data validation for data migration security. Int'l Journal of Computer Applications, 2011, 30(6): 41−46.

[12] Gartner. Magic quadrant for data masking technology. 2022. https://www.gartner.com/en/documents/3180344

[13] Softwaretestinghelp. Best data masking tools and software. 2022. https://www.softwaretestinghelp.com/data-masking-tools/

[14] Moffie M, Mor D, Asaf S, Farkash A. Next generation data masking engine. In: Proc. of the Int'l Workshop on Data Privacy Management, Cryptocurrencies and Blockchain Technology. Cham: Springer, 2021. 152−160.

[15] Xu MT. Dynamic data masking of openGauss. 2022. https://blog.opengauss.org/en/post/2022/dynamic-data-masking-of-opengauss/

[16] The Apache Software Foundation. Apache Hive. 2022. https://hive.apache.org/

[17] Baranchikov AI, Gromov AY, Gurov VS, Grinchenko NN, Babaev SI. The technique of dynamic data masking in information systems. In: Proc. of the 5th Mediterranean Conf. on Embedded Computing (MECO). Piscataway: IEEE, 2016. 473−476. [doi: 10.1109/MECO.2016.7525695]

[18] Archana RA, Hegadi RS, Manjunath TN. A study on big data privacy protection models using data masking methods. Int'l Journal of Electrical and Computer Engineering (IJECE), 2018, 8(5): 3976−3983.

[19] Larsonk KS, Boukari S. An improved data masking security solution using modulus based technique (MOBAT) for data warehouse system. Int'l Journal of Science and Engineering Applications, 2020, 9(6): 68−78.

[20] Cui BJ, Zhang BH, Wang KY. A data masking scheme for sensitive big data based on format-preserving encryption. In: Proc. of the IEEE Int'l Conf. on Computational Science and Engineering (CSE) and IEEE Int'l Conf. on Embedded and Ubiquitous Computing (EUC). Piscataway: IEEE, 2017. 518−524. [doi: 10.1109/CSE-EUC.2017.97]

[21] Patil S, Polte M, Ren K, Tantisiriroj W, Xiao L, López J, Gibson G, Fuchs A, Rinaldi B. YCSB++: Benchmarking and performance debugging advanced features in scalable table stores. In: Proc. of the 2nd ACM Symp. on Cloud Computing. New York: ACM, 2011. 1−14. [doi: https://doi.org/10.1145/2038916.2038925]

[22] Yesin VI, Vilihura VV. Some approach to data masking as means to counteract the inference threat. Radiotekhnika, 2019, 3(198): 113−130. [doi: https://doi.org/10.30837/rt.2019.3.198.09]

[23] Shen J, Zhou TQ, Cao ZF. Protection methods for cloud data security. Journal of Computer Research and Development, 2021, 58(10): 2079−2098 (in Chinese with English abstract).

附中文参考文献:

[1] 钱文君, 沈晴霓, 吴鹏飞, 董春涛, 吴中海. 大数据计算环境下的隐私保护技术研究进展. 计算机学报, 2022, 45(4): 669−701.

[2] 方滨兴, 贾焰, 李爱平, 江荣. 大数据隐私保护技术综述. 大数据, 2016, 2(1): 1−18. [doi: 10.11959/j.issn.2096-0271.2016001]

[3] 吴信东, 董丙冰, 堵新政, 杨威. 数据治理技术. 软件学报, 2019, 30(9): 2830−2856. http://www.jos.org.cn/1000-9825/5854.htm [doi: 10.13328/j.cnki.jos.005854]

[4] 王卓, 刘国伟, 王岩, 李媛. 数据脱敏技术发展现状及趋势研究. 信息通信技术与政策, 2020, 46(4): 18−22.

[5] 陈性元, 高元照, 唐慧林, 杜学绘. 大数据安全技术研究进展. 中国科学: 信息科学, 2020, 50(1): 25-66. [doi: 10.1360/N112019-00077]

[6] 佟玲玲, 李鹏霄, 段东圣, 任博雅, 李扬曦. 面向异构大数据环境的数据脱敏模型. 北京航空航天大学学报, 2022, 48(2): 249−257.

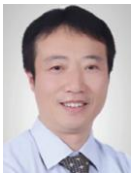
[7] 李书缘, 季与点, 史鼎元, 廖旺冬, 张利鹏, 童咏昕, 许可. 面向多方安全的数据联邦系统. 软件学报, 2022, 33(3): 1111−1127. http://www.jos.org.cn/1000-9825/6458.htm [doi: 10.13328/j.cnki.jos.006458]

[10] 刘宝星. 基于 TPC-DS 的性能测试工具设计与实现 [硕士学位论文]. 大连: 大连理工大学, 2018.

[23] 沈剑, 周天祺, 曹珍富. 云数据安全保护方法综述. 计算机研究与发展, 2021, 58(10): 2079−2098.

屠要峰(1972－), 男, 博士, 研究员, CCF 杰出会员, 主要研究领域为大数据, 分布式系统, 机器学习.

牛家浩(1979－), 男, 高级工程师, CCF 专业会员, 主要研究领域为大数据, 数据安全, 隐私保护技术.



王德政(1974－), 男, 高级工程师, 主要研究领域为大数据存储计算, 隐私计算, 区块链.

高洪(1978－), 男, 高级工程师, CCF 专业会员, 主要研究领域为大数据,人工智能, 数据挖掘, NLP.

徐进(1982－), 男, 高级工程师, 主要研究领域为大数据, 人工智能, 隐私计算.

洪科(1978－), 男, 高级工程师, 主要研究领域为大数据存储计算, 人工智能.

阳方(1984－), 男, 大数据工程师, 主要研究领域为数据仓库, 离线计算.