International Journal of Software and Informatics, ISSN 1673-7288
http://www.ijsi.org, ijsi@iscas.ac.cn, +86-10-62661048
IJSI, 2023, 13(1): 87–115, doi: 10.21655/ijsi.1673-7288.00297
©2023 by Institute of Software, Chinese Academy of Sciences. All rights reserved.

Research Article

BDMasker: Dynamic Data Protection System for Open Big Data Environment

Yaofeng Tu (屠要峰)[1,2], Jiahao Niu (牛家浩)[2], Dezheng Wang (王德政)[1,2], Hong Gao (高洪)[2], Jin Xu (徐进)[2], Ke Hong (洪科)[2], Fang Yang (阳方)[2]

[1] (State Key Laboratory of Mobile Network and Mobile Multimedia Technology, Shenzhen 518057, China)
[2] (ZTE Corporation, Nanjing 210014, China)
Corresponding author: Jiahao Niu, niu.jiahao@zte.com.cn

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 the big data industry and technology requires 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 in regards such as data availability, processing efficiency, and system scalability. This paper proposes the 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 does not change the original business request, which indicates that the whole process of dynamic masking 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.

Keywords big data; data masking; dynamic data masking; SQL rewriting; query dependency

Citation Tu YF, Niu JH, Wang DZ, Gao H, Xu J, Hong K, Yang F. BDMasker: Dynamic data protection system for open big data environment, *International Journal of Software and Informatics*, 2023, 13(1): 87–115. http://www.ijsi.org/1673-7288/297.htm

In the era of big data, big data serves as a national basic strategic resource. Attaching great importance to the development of big data, China has begun to put in place the national big

This is the English version of Chinese article "面向开放大数据环境的动态数据保护系统, 2022, 34(3): 1213–1235. doi: 10.13328/j.cnki.jos.006783"
Funding items: National Key R&D Program of China (2021YFB3101100)
Received 2022-05-14; Revised 2022-07-29, 2022-09-07; Accepted 2022-09-23; IJSI published online 2023-03-30

data strategy in an all-round manner, in which the opening and sharing of data lies at the core of the big data competition strategy. From the perspective of the technological development trend, new technical architectures and big data support platforms are emerging, among which the cloud native and lake-house architecture are reconstructing the big data infrastructure. Stronger capabilities of data security and data sharing are required from access to the data lake and data warehouse to cross-database and cross-domain sharing. Both the open source big data ecosystem and the commercial big data system, however, fall worryingly behind the business development in the security protection capability of big data in an open environment. It is shown by the privacy disclosures in recent years that the release or sharing of unmasked data is highly prone to reveal private data, especially individual sensitive information. In 2018, the data of 87 million users of Facebook, a social media in the US, was illegally used by Cambridge Analytica, a consulting company, and Facebook paid a \$5 billion fine for such an event. Again in 2021, there was a data leak involving another 533 million individual Facebook users. The security problem in the open environment has become a bottleneck in the development and utilization of big data technology. Accordingly, it has become one of the research focuses in big data security as to how to protect the privacy of sensitive data in an open and complex environment while ensuring good data availability and computing efficiency[1, 2].

Data security in the open big data environment differs greatly from traditional data security, with changes seen in the protection method, protection object, and the relationship between management and technology. The application scenario of open big data is committed to the opening and sharing of data, with more diverse roles involved in data processing and the flow of data as normal, which sets higher standards for data security protection. Traditional measures for data security such as data encryption and static masking, thereby, are outmoded. According to relevant research, privacy protection and dynamic data masking technologies represent important means for safe data flow and sharing and credible big data services[3, 4]. By maintaining the availability of data sources without the leak of sensitive information in the data flow, dynamic data masking technology boasts good utility and a broad prospect of application. In an open big data environment, it is a complex problem requiring prompt solutions as to how to dynamically protect sensitive data in an automated, efficient, and scalable manner while softening the impact on normal businesses amid massive multimodal data and highly concurrent access requests[5–7]. The following challenges are mainly involved.

(1) Scalability of heterogeneous environments. To satisfy the timeliness requirement of different data queries and data computing under an open big data scenario, many kinds of big data computing engines are often deployed simultaneously on the same cluster. For instance, Apache Spark[8] is suitable for batch processing of static data with high latency, and Apache Flink[9] is for low-latency or real-time streaming data processing. Faced with complex and diverse business scenarios and multiple computing engines, we should explore how to create, manage, and maintain a uniform data protection strategy for heterogeneous engines and provide standardized access methods for the horizontal expansion of heterogeneous environments. In addition to the capability of dynamic data masking, it is necessary to study how to flexibly support multiple capabilities of dynamic data protection under one framework and support the vertical expansion of the dynamic data protection capability of a single engine.

(2) High efficiency of processing performance. In an open big data environment, data is generated at a faster pace, with its size on the exponential rise. To meet the response time requirements in high-performance real-time protection of massive data, data security protection must be able to operate automatically under the rules and optimize the load of the whole processing process. In this way, it can make full use of the distributed computing capability of the big data execution engine to enhance the processing performance.

(3) Precision of SQL rewriting. SQL is a widely used data query language, and the popular big data computing engines can all provide SQL access. SQL rewriting represents the key technology for dynamic data masking. As SQL requests in the business field are ever-changing, and the SQL rewriting mechanism involves all columns that define masking strategies, the rewriting of complex SQL statements may result in data distortion and reduce data availability, even undermining the accuracy of business logic processing. In the case of complex SQL access requests, it poses a challenge to the design of a dynamic data protection system, especially the SQL rewriting technology, as to how to ensure that the rewritten SQL is completely transparent to the businesses on the premise of not exposing the sensitive information of the underlying physical table so as to make the business logic isolated from the influence of data protection. Next, we take the query statement Query76 of TPC-DS[10] shown in Listing 1 as an example to explain the technical difficulties of SQL rewriting. Query76 covers most of the important syntax rules in SQL statements.

Listing 1 Query76 of TPC-DS

```
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
```

Precise SQL analysis. For an SQL query, the output of its query result set eventually comes from the output field of the "select" statement of the outermost query, and the output field may be from sub-query statements, "join" statements, "union" statements, etc., which may undergo multi-tier conversion through sub-query and nested functions, etc. Therefore, we need to accurately identify and correctly mask the source of sensitive fields in the outermost output parts; otherwise, the sensitive information of the underlying physical table may get exposed. For instance, the fields in the physical table on which the outermost output column `col_ name` in Query76 finally depends include the following: `ss_store_sk` field in the `store_ sales` table, `ws_ship_customer_sk` field in the `Web_sales` table, and `cs_ship_addr_sk` field in the `catalog_sales` table. If masking rules were set for only one of the three fields, the outermost output column `col_name` would cause the leak of sensitive data in the query result set by not obtaining the field information of the underlying physical table on which it depends and applying the corresponding masking rules.

Precise positioning of sensitive fields. The sensitive fields in the SQL query requests may come from different syntactic structures. For example, in Query76, the field `i_category` appears many times in the sub-query output field at different tiers; the field `d_year` is seen in both the sub-query output field and the GROUP BY and ORDER BY statements. Some query output fields come from functions or function nesting, such as SUM (`ext_sales_price`). Therefore, if the complex nesting, tier, alias, function, and other syntaxes and logic relationships in the SQL structure could not be well identified, and we adopted the simple processing method of directly rewriting all the fields defining masking strategies in the SQL statement, the rewritten SQL statement is usually inconsistent with the original request. This can lead to failed or incorrect businesses.

The big data ecosystem SQL-on-Hadoop open source software, such as Apache Spark and Apache Flink, has yet to support dynamic masking and other security capabilities. At the moment, there is yet to be any system that can better solve the above three challenges. Considering this, this paper discusses the design and implementation of the high-performance dynamic data protection system BDMasker under open big data scenarios, with the main efforts and contributions presented as follows.

(1) According to the principle of "zero business awareness", we propose a precise query analysis and query rewriting technology based on the Query Dependency Model (QDM). It can accurately perceive but does not change the original service request, with little impact on the businesses in the whole process of dynamic masking, thus solving the challenge of precise SQL rewriting.

(2) We present a unified security strategy framework for multiple engines, which supports the vertical expansion of multiple dynamic data protection capabilities through plug-in data protection strategy management. In addition, the appropriate data protection strategy can be adopted according to the application scenario, business objectives, and data characteristics, and the horizontal expansion of multiple computing engines can be realized through strategy agents and standardized interfaces.

(3) Given the above SQL analysis rewriting technology and unified security strategy framework, we achieve a dynamic data protection engine in such systems as Apache Hive and Apache Spark. Built into the big data execution process, it makes full use of the distributed computing capability of the big data execution engine to enhance the performance of data protection processing.

(4) We verify the effectiveness, scalability, and performance enhancement effect of the BDMasker system through functional experiments and YCSB and TPC-DS benchmark tests.

Section 1 of this paper introduces the research background and related work. Section 2 presents the system architecture and key technologies. Section 3 validates the functional effectiveness, scalability, and high performance of data protection processing of the BDMasker system through experiments. Finally, Section 4 gives relevant summaries and prospects.

1 Related Work

There are many research programs on dynamic data masking[4, 6, 11–16], and several architectures have been proposed. We divide these programs into those in the agent-based mode and those in the kernel-based mode.

1.1 Agent-based mode

The agent-based mode calls for the use of an external masking agent gateway, which first intercepts the query request and then implements the dynamic masking function by rewriting the request or the query result. This mode involves two technical concepts: rewriting based on the query request and rewriting based on the query result.

The rewriting technology based on the query request first intercepts the query from the client through the agent gateway, then rewrites the query request according to the security strategy, and redirects the rewritten query request to the data source. After receiving the modified query, the data source will output false (mixed) data. According to the query request rewriting method, researchers have proposed the rewriting technology based on rule engines[11] and syntax parsing[12, 13].

Informatica[11] and other manufacturers have adopted dynamic masking technology that matches and rewrites the query SQL by the rule engine in the masking agent gateway. The request is parsed via the connection of the rule engine with the rule tree. If the connection rule is assigned a security rule set, the rule engine parses the SQL request through the security rule tree and rewrites the SQL statement according to the rewriting rule. Despite simple implementation and strong scalability, the rule engine technology also shows some obvious disadvantages: (1) a more complex query request indicates greater difficulty in identification and less accurate matching. As a result, the rule engine cannot accurately match the complex SQL statement and rewrite the fields to be masked, which directly leads to rewriting errors or may change the existing business logic; (2) the rule engine has low processing efficiency, as well as high complexity and poor performance of the modified SQL statement. Moreover, it cannot parse all syntax rules, which is prone to data leaks.

Due to the shortcomings of the rewriting technology based on the rule engine, some researchers have put forward the technology based on syntax parsing. The core idea is to unpack and analyze the SQL statement through lexical analysis or syntax analysis in the masking agent gateway, match the strategy rules, and then rewrite the query request. Highly universal, the syntax parsing mode generally involves parsing the standard SQL syntax, but it can hardly achieve SQL independence, and sensitive data are typically leaked due to the different syntax rules for the dialect assembly logic of each database. In addition, it needs to keep adapting to the changes in the database to connect different database engines. Hence, it is difficult to develop and maintain the system.

In contrast, the rewriting technology based on the query result[12, 13] follows a different core idea. The masking agent gateway intercepts the client query and then forwards it directly to the database engine that produces the data query results. The agent gateway parses the query request statement, replaces the sensitive data in the query result data with the preset information to obtain the masked data, and outputs it to the application program. The main disadvantage of this framework lies in that query requests and result returns need to go through masking middleware, which consumes a lot of network bandwidth and computing and storage resources and leaves this mode unable to serve the response time of real-time masking of big data. Moreover, the result returned by the database needs to be cached in the masking server, which introduces new attack surfaces and data leak risks.

1.2 Kernel-based mode

The kernel-based dynamic masking mode sets the field masking rules through the database's DDL command and implements the dynamic masking function of a single engine by rewriting the query request in the database kernel. This mode is adopted by commercial database manufacturers such as Oracle and IBM[14], open source database OpenGauss[15], and Apache Hive[16], an open source big data engine. The dynamic masking of OpenGauss is to rewrite the statements of queries containing sensitive fields, and the sensitive fields involved in the queries are rewritten via outer-layer nested functions. The main idea of the rewriting mechanism is as follows: Var nodes in the query tree stand for the database resources accessed, and non-Var nodes may contain Var nodes. It is necessary to seek the Var node recursively by reference to its parameters before finally matching the relevant strategies for all the Var nodes identified and replacing the nodes according to the strategy function. Similar to OpenGauss, Apache Hive implements the column-level dynamic masking function through SQL rewriting technology. When the Hive engine parses the SQL, the masking algorithm is used to modify the value of the abstract syntax tree of the field to be masked regardless of the position of this field in the SQL statement.

The SQL rewriting mechanism of OpenGauss and Apache Hive involves all the columns that define masking strategies. The same field may appear in all the syntax structures of a SQL query statement. Thus, this mode of rewriting all matched fields cannot well identify the complex nesting, tier, and other logical relationships in the SQL structure. As a result, the rewritten query statements will bring about the wrong execution results or the incorrect implementation of business logic.

1.3 Scalability

Although the agent-based mode can support the masking of multiple data sources simultaneously, it is at the expense of higher data access overload and costs for system maintenance and hardware, which cannot meet the response time requirement of the real-time masking of big data. In addition, the caching of the result set also introduces a new attack surface. The strategy models proposed by Oracle, OpenGauss, etc., are implemented on the basis of the DDL of their own databases, which are only applicable to their own databases and cannot be expanded to the big data environment. Lacking a unified security technology standard, the open source big data ecosystem cannot be deployed and managed in a unified and effective manner. By lacking support for dynamic masking and other security capabilities at the moment, the open source big data engines such as Apache Spark and Apache Flink are not equipped with a scalable security strategy model for multiple engines.

2 System Architecture and Key Technologies

This section introduces the system architecture and key technologies of BDMasker, a dynamic data protection system for open big data environment. BDMasker adopts a multi-engine-oriented unified security strategy framework and the dynamic data protection technology based on QDM to enable multiple data security protection capabilities such as the dynamic masking of “zero business awareness”. In addition, it supports big data computing engines such as Apache Hive, Apache Spark, and Apache HBase, thus solving the technological challenges faced by dynamic data protection in open big data environments, such as heterogeneous environment expansion, efficient processing performance, and precise SQL rewriting.

2.1 System architecture

The agent mode suffers from low masking performance and poor security in massive data scenarios. Thus, the external masking agent architecture is abandoned in the design of the BDMasker system. Instead, it fulfills data protection functions such as high-performance dynamic data masking by directly building a dynamic data protection engine in the kernel of big data and giving play to the distributed processing capability of the big data engine. Thus, it solves the challenge of efficient processing from the perspective of the architecture process. Through the centralized deployment of strategy engines, it uniformly manages and maintains data protection strategies for heterogeneous engines and provides a standardized access method to solve the challenge of horizontal expansion of heterogeneous environments.

As shown in Figure 1, the system architecture of BDMasker involves three tiers from top to bottom, namely, big data client, big data execution engine, and strategy engine.

Tier of big data client. The client is the application, program, command, or script that accesses the big data service. It sends access requests to the big data execution engine and receives the processing results through the network. It is not necessary to make any change to the clients of various existing open source big data engines.

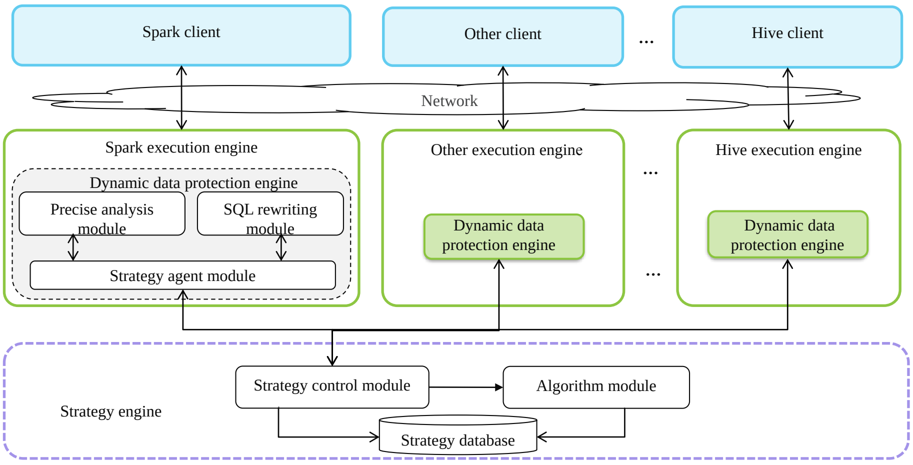


Figure 1 System architecture of BDMasker

Tier of big data execution engine. The big data execution engine is dedicated to the business logic processing of client requests, such as data query analysis. BDMasker enables the dynamic data protection function for the big data system by a built-in dynamic data protection engine in the big data computing engine. The dynamic data protection engine is composed of a strategy agent module, a precise analysis module, and an SQL rewriting module. Specifically, the strategy agent module is designed to obtain the data protection strategies configured by the execution engine from the strategy engine and cache them into the local memory, thus providing the protection strategy query interface for the dynamic protection engine. The precise analysis module generates the SQL QDM by constructing the algorithm. The SQL rewriting module is dedicated to the precise rewriting of SQL statements.

Tier of strategy engine. The strategy engine provides a unified strategy service of data protection for multiple homogeneous/heterogeneous big data execution engines, which is composed of a strategy control module, an algorithm module, and a strategy database. Specifically, the strategy control module designs a uniform data protection strategy model for heterogeneous big data execution engines and provides each execution engine with uniform strategy management service, API interfaces, and operation interfaces. The algorithm module is designed for the centralized management of data protection operators and to provide the framework for operator development, loading, and deployment. The strategy database is dedicated to the centralized storage of configuration data related to the strategy control module and the algorithm module.

The workflow of BDMasker is presented in Figure 2. The administrator can configure different data protection strategies for multiple big data engines through the strategy management interface. First, the administrator selects the instance name, database, and table for the big data engine service, and then sets security rules for the protected sensitive fields, such as the random, replacement, and offset rules. After the setting, the data protection strategy will be saved in the strategy database. The strategy agent module in each execution engine dynamically obtains the corresponding data protection strategy for the engine and loads it into the memory.

As shown on the right side of Figure 2, the big data client sends an SQL access request to the big data execution module, in which the user name will be carried as the user's identity. The SQL parser of the big data execution module carries out lexical analysis on the SQL statement to obtain the general symbolic stream, and then performs syntax analysis on the symbolic stream to construct a syntax tree, and analyzes the syntax tree in-depth to generate the abstract syntax tree

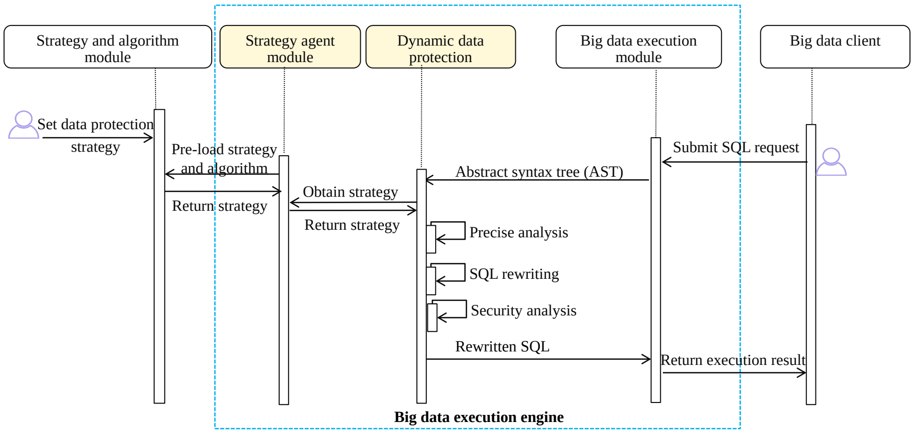


Figure 2 Workflow of BDMasker

(AST). The next step is about the process of dynamic data protection. For dynamic data masking, the precise analysis module of the system first builds the QDM of the SQL request. Then, in combination with the QDM and security strategy, the SQL rewriting module of the system precisely rewrites the SQL. After security analysis, the rewritten SQL statement is submitted to the execution engine. The execution engine re-analyzes the rewritten SQL. After that, the query optimizer optimizes the logic plan, generates and optimizes the physical plan, and converts it into an executable task with the data protection function. Then, the big data execution engine completes the business logic processing in parallel via the distributed computing capability of the cluster. Upon execution, the big data execution module returns the data of the result set to the client.

2.2 Unified security strategy framework for multiple engines

There are multiple big data computing engines in an open big data environment, which generally provide inconsistent security strategy models and external interfaces. Hence, the security strategy is difficult to manage and maintain. Such being the case, BDMasker abstracts and unifies the data protection models of heterogeneous engines and proposes a unified security strategy framework based on metadata for multiple engines to ensure the availability of a unified interface and security strategy model for heterogeneous big data engines.

As shown in Figure 3, BDMasker's unified security strategy model for multiple engines consists of three parts: object, condition, and strategy of rules.

The rule object represents the controlled entity of the big data resource, which adopts the tiered model of service (instance name of the big data execution engine)—database—table—sensitive column. This mode of description can simultaneously manage resource entities at different tiers of heterogeneous engines, with the column as the minimum granularity. Sensitive data domains are used to classify sensitive data, and the functional meaning of columns is described on the basis of column data or column names. For instance, as "ID number" is a type of sensitive data, a unified protection strategy for ID number fields is implemented in all heterogeneous engines. Taking HBase as an example, HBaseSrv1-DB1-Table1-CF1:F1 represents the management of the F1 field of CF1, the column family of Table1 in the DB1 database of the service HBaseSrv1.

As a basic judgment unit that defines trusted access for different types of requests, rule conditions can be defined according to the request context, and different protection methods can be defined for different commands. For instance, the rule condition of identity can restrict users'

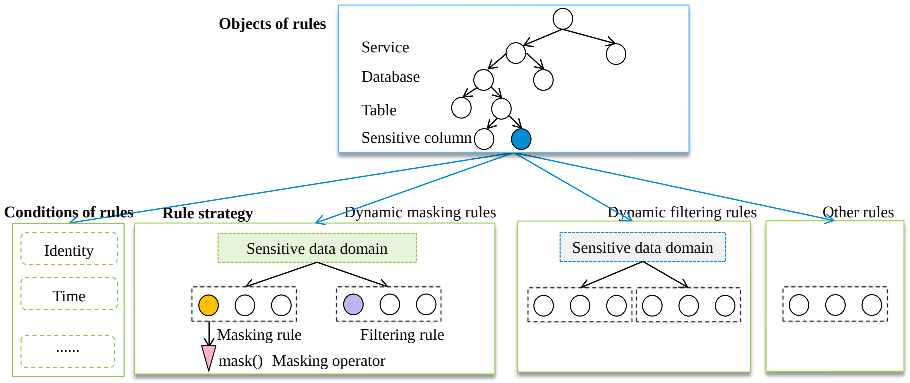


Figure 3 BDMasker's strategy model

access to sensitive data according to user names, user groups, roles, etc.; the rule condition of time can limit the access time; and the rule condition of tier can limit users' access to sensitive data according to data classification and grading conditions.

The rule strategy is described by the data protection rule-execution operator structure. The data protection rules represent specific protection operations for sensitive data, including dynamic data masking rules, dynamic data filtering rules, and other rules. The data masking rules meet the compliance requirements of industry and are used to deform data of specific types or sensitive data domains. The data masking operators, as a kind of execution operator, are used for the underlying conversion execution of masking and are stationed in the big data execution engine in the form of an algorithm library. Operators represent the implementation of the algorithm idea. In algorithm design, we should factor in performance[17] and design different masking algorithms according to various loads and data characteristics[18]. Common algorithms in industry include the modulus algorithm[19] and the format-preserving encryption algorithm[20]. According to such sensitive data domains as the account domain, mailbox domain, name domain, and address domain, BDMasker has designed over 40 masking operators, including the Caesar encryption operator, shift operator, out-of-order operator, and truncation operator. Different labels, such as stability and uniqueness, are set for operators by the algorithm characteristics. The results of the stability operator are consistent no matter how many times it is executed, and the results of the uniqueness operator are globally unique. Different operators are applied in various scenarios. The same masking operator can create multiple different masking rules, and the same masking rule can be distributed in different sensitive data domains. A sensitive data domain matches multiple masking rules. Such flexible design can meet the compliance requirements of various industries.

On the basis of the multi-engine-oriented unified security strategy model, BDMasker has designed the strategy engine and provided a centralized strategy management interface and API access interface. To apply this unified model to heterogeneous big data engines, the system adopts a plug-in design for the strategy agent module, which is built into the big data engine in the form of plug-ins. It loads the data protection strategy configured in this execution engine from the strategy control module and caches it in the local memory. It also provides the strategy interface for the data protection execution engine and dynamically adds, deletes, and checks strategies through the API interface.

The BDMasker system adopts the flexible combination and mutual decoupling of multiple dynamic data protection technologies to solve the vertical expansion problem of the data protection capability of a single computing engine. BDMasker divides the processing of SQL statements into three stages: precise analysis, SQL rewriting, and security evaluation. In each stage, plug-in interfaces are provided to support the flexible expansion of security capabilities. The QDM is constructed by default in the precise analysis stage to obtain the data source on which the output field of the SQL query request depends. This stage enables extended support for the capabilities such as alarms and rule-based access control. The AST undergoes masking-based precise rewriting by default in the stage of SQL rewriting, during which the support for dynamic data filtering is extended. The rewritten SQL statement can support both dynamic masking and dynamic filtering simultaneously, with no impact on the business logic. In the evaluation stage, the security analysis is performed on the reconstructed syntax tree to prevent malicious access or masking bypassing functions, which enables extended support for the functions such as fine-granularity operation audit of sensitive fields, sensitive data alarm, SQL security analysis, and data masking risk assessment. BDMasker's extended security capabilities are deployed in the big data execution engine in the form of plug-ins and loaded into each big data engine by the dynamic data protection engine according to the unified security strategy. They are flexibly combined with the dynamic data masking function during the execution of the SQL statement to jointly achieve security control when different users access the same sensitive data to well intercept illegal access.

With the unified security strategy framework for multiple engines, BDMasker provides a standardized interface for the centralized management of the data protection strategy of each engine and extends it in the form of security plug-ins within a single engine. In this way, it decouples the function of dynamic data protection from the business logic of the big data execution engine, with the impact on the existing big data system reduced. With strategy agent plug-ins and security plug-ins developed in Apache Hive, Apache Spark, and Apache HBase, we have achieved centralized security strategy management, and the business system can adopt appropriate data protection strategies according to the application scenario, business objective, and data characteristics. Meanwhile, the vertical expansion within a single engine via security capability plug-ins can support multiple capabilities of dynamic data protection such as dynamic data masking and filtering.

2.3 Dynamic data protection technology based on QDM

SQL statements are generally composed of the "select" statement, "from" statement, and "body" statement. Among them, the "select" statement represents the query output field. The "from" statement describes the data source on which the query output field depends, which mainly includes tables, nested sub-query statements, and join/union statements. The "body" statement represents the main part of the query statement, including where, group by, order by, and limit. As mentioned before, since SQL statements in the business area are typically complex, the rule engine or shallow parsing cannot ensure the correct application of the security protection strategy due to the lack of precise analysis technology, thus resulting in the leak of sensitive data.

This section introduces the precise query analysis and SQL rewriting technology based on QDM. This technology enables the precise perception of the rewriting without changing the original business request, thus ensuring no impact on the business during the whole process of dynamic data protection.

2.3.1 *Definition of QDM*

One of the challenges facing precise SQL rewriting is to identify the data source on which the outermost query output field finally depends. To enhance the uniform representation of the structured query analysis results and correctly obtain the table and field information on which the query output field finally depends, we refine the query analysis process and propose the

QDM and the sub-QDM, as shown in Figure 4.

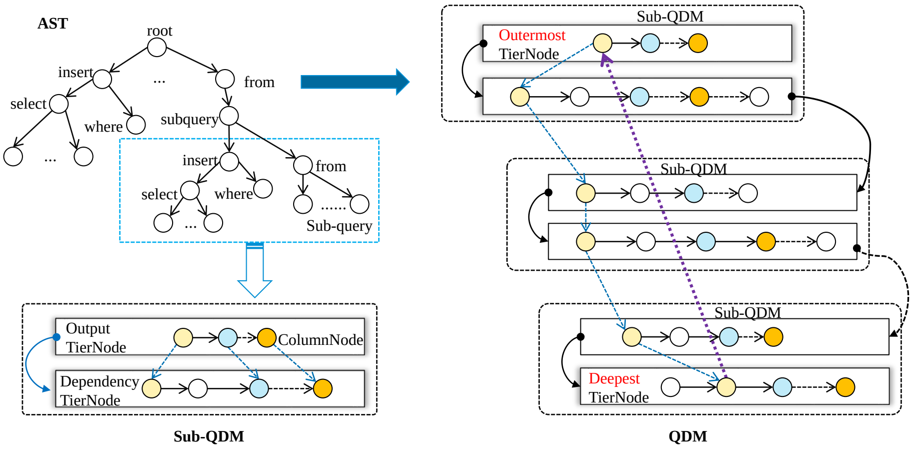


Figure 4 QDM

Definition 1. The model with the following characteristics is known as sub-QDM. This model records the information on the output field dependency of an SQL sub-query, and each SQL sub-query will create a sub-QDM.

(1) Each sub-QDM is composed of two query-tier nodes (TierNode), namely, Output TierNode and Dependency TierNode, which are logically associated through pointers.

(2) TierNode stores field information in a single-linked list. Each node in the single-linked list is known as a field information node (ColumnNode), which is uniformly composed of multi-tuples in the form (TN, ATN, FN, AFN), where TN represents the list name, ATN the sub-query alias, FN the field name, and AFN the field alias.

(3) Each ColumnNode in the Output TierNode stores the information on an output field and each ColumnNode in the Dependency TierNode stores the information on an underlying data source on which the query output field depends.

Definition 2. The model with the following characteristics is known as a QDM, which records the dependency relationship between the output fields of the SQL query tier by tier.

(1) The QDM is a tiered structure composed of several TierNodes. The top-tier TierNode, known as the Outermost TierNode, is the final output field of the query request. The innermost TierNode, known as the Deepest TierNode, is the data source that the Outermost TierNode finally depends on.

(2) The QDM is composed of several sub-QDMs. Each sub-query generates its corresponding sub-QDM. The Dependency TierNode of the parent query points to the output TierNode of the sub-query through the pointer to establish the logical association between the parent and subsidiary queries. Finally, a complete QDM is formed.

2.3.2 *Construction of QDM*

An SQL statement will be converted into an AST after parsing, which represents the syntax structure of the programming language in the form of a tree on which each node represents a structure in the source code. The AST is equipped with all the information required for precise analysis. Therefore, the model construction algorithm adopts the AST as the input for model construction.

This section uses the following simple nested query statements as an example to introduce the construction algorithm of the QDM.

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

Generally, different constant names in different SQL engines uniquely mark the types of AST nodes. In the case of Apache Hive, the parsing process of the AST generates a TOK_TABREF node for each table and a TOK_FROM node for the "from" keyword, which is similar to the process for other nodes. Algorithm 1 takes Apache Hive's AST as an example to construct the QDM. Other big data execution engines such as Apache Spark follow a similar implementation concept.

Algorithm 1. Constructing the QDM algorithm.

```
Input: root node of AST, i.e., ASTRoot
Output: root node of QDM, i.e., Outermost TierNode
1.  TierNode walkAST(ASTRoot)
2.  {
3.    OutputTierNode = newTierNode(); // Build a new query OutputTierNode
4.    DependencyTierNode = newTierNode(); // Build a new dependency DependencyTierNode
5.    OutputTierNode -> next = DependencyTierNode; // Establish the point-to relationship
6.    walkSelectAST(ASTRoot, TOK_SELECT, OutputTierNode); // Call Algorithm 2 to traverse
      the "select" sub tree
7.    walkFromAST(ASTRoot, TOK_FROM, DependencyTierNode); // Call Algorithm 3 to traverse
      the "from" sub-tree
8.    updateOutputTier(OutputTierNode); // Call Algorithm 4 to update the dependency relationship
      between the fields in different tiers of the QDM
9.    return OutputTierNode; // Return the root TierNode of QDM
10. }
```

Algorithm 1 describes the process of constructing a complete QDM for an SQL query request. First, it creates an OutputTierNode and a DependencyTierNode and establishes a dependency point-to relationship (Lines 3–5) between them. Then, it traverses and parses the AST subtree matching the "select" clause to obtain the internal structure of the "select" clause, thus obtaining the query output field information in the current tier (Line 6, corresponding to the generation algorithm of the query output field). After that, it traverses the AST subtree matching the "from" clause, continues to further traverse the sub-query and other syntax structures in the syntax tree, and constructs the corresponding sub-QDM tier by tier via recursion (Line 7, corresponding to the generation algorithm for the dependency information of the query output field). Finally, it updates the dependency information of the top node to form a complete QDM (Line 8, corresponding to the final dependency update algorithm).

Figure 5 describes the process of QDM generation by use of Algorithm 1 in the example SQL. The example SQL generates a complete QDM composed of three sub-QDMs through Algorithm 1, each of which corresponds to a "select" sub-query of the example SQL. The sub-QDM in the first tier corresponds to the outermost "select" query; that in the second tier corresponds to the middle "select" sub-query, and that in the third-tier corresponds to the innermost "select" sub-query. Each sub-QDM stores the output field and field source of the query in the corresponding tier, and each query output field of the outermost "select" query can identify its final dependent physical table information through the relationship between sub-QDMs.

The generation algorithm of the query output field (Algorithm 2) obtains the internal structure of the "select" clause by traversing and parsing the AST subtree corresponding to the

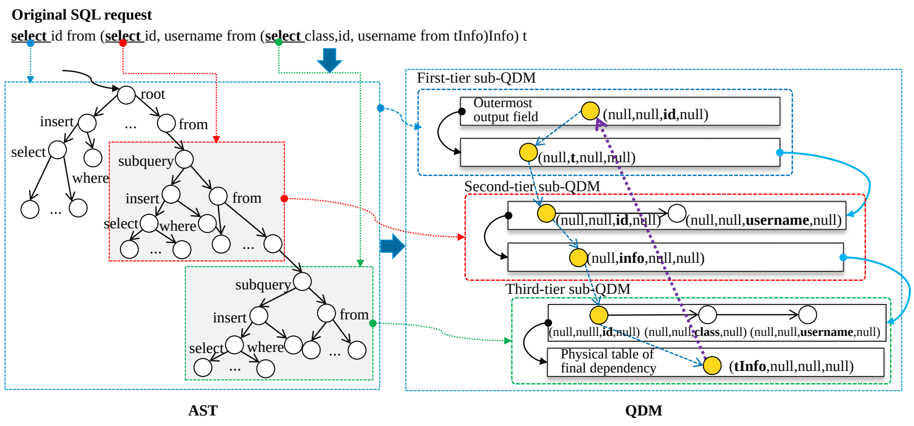


Figure 5 QDM structure

"select" clause and also acquires the query output field information in the current tier. The main steps are as follows: the first step is to perform a depth-first traversal from the AST root node and directly position the root node of the "select" clause, from which the analysis is started (Line 3). Next, the "select" sub-tree is traversed until the process is completed, in which different processing is performed according to the type of the AST node (Lines 4–17). To be specific, if the node type is a field or table information node and also a leaf node, the table name, sub-query alias, field name, and field alias are directly obtained and inserted into the OutputTierNode structure of the sub-QDM. As for function nodes, the function subtree is parsed to obtain all table field information involved in the function and inserted into the OutputTierNode structure of the sub-QDM. As for select*, the name of the table on which the query depends is obtained through parsing, and the corresponding field information of the table is obtained by the calling of the metadata acquisition interface, which is inserted into the OutputTierNode structure of the

Algorithm 2. Generation algorithm of the query output field.

```
Input: AST root node to be analyzed, i.e., ASTRoot, AST node type NodeType, Output tier node
       OutputTierNode
Output: NULL
1.  walkSelectAST(ASTRoot, Nodetype, OutputTierNode)
2.  {
3.  node = walkToAST(ASTRoot, Nodetype); // Traverse AST to position the root node of "select"
      and start traversal from here;
4.  while the sub-tree is not traversed do
5.      if the current node is a leaf node containing field information then
6.          addOneNode(OutputTierNode, node);
7.      end if
8.      if the node type is a function then
9.          // The node type is a function
10.         addAllFunctionColumnNode(OutputTierNode, node);
11.     end if
12.     if the current node type is the * query then
13.         // The status of select*
14.         addAllStarNode(OutputTierNode, node);
15.     end if
16.     Obtain the next node of AST to continue traversal
17. end while
18. }
```

sub-QDM.

Through Algorithm 2, the example SQL generates the query output information of three sub-QDMs from top to bottom. Among them, the top tier represents the final output field of the query request, (null, null, id, null), whose final source is not known at the moment. The query output fields in other tiers are (null, null, id, null) and (null, null, username, null) in the second-tier sub-QDM and (null, null, id, null), (null, null, class, null), and (null, null, username, null) in the third-tier sub-QDM.

The algorithm to generate the dependency information of the query output field (Algorithm 3) obtains the information on the source of the query field in this tier by traversing the AST subtree corresponding to the "from" clause, thus constructing a sub-QDM. Then, it continues the depth-first traversal of the sub-query and other syntax structures in the syntax tree and constructs the corresponding sub-QDM tier by tier through recursion before finally forming a complete QDM.

Algorithm 3. Algorithm to generate the dependency information of the query output field.

```
Input: The AST root node to be analyzed, i.e., ASTRoot, AST node type NodeType, dependency
       tier node DependencyTierNode
Output: NULL
1.  walkFromAST(ASTRoot, Nodetype, DependencyTierNode)
2.  {
3.  node = walkToAST(ASTRoot, Nodetype); // Traverse AST to position the root node of "from" and
      start traversal from here;
4.  while the sub-tree is not traversed do
5.      if the node is a sub-query node, such as SubQuery and CTE then
6.          tTierNode = walkAST(node);
7.          // Sub-QDM Recursively parse the sub-query node, and each SQL sub-query will create a
              sub-QDM
8.          DependencyTierNode -> next = tTierNode; // Establish the point-to relationship
9.      end if
10.     if union/join-type nodes then
11.         WalkJoinSubtree(node, DependencyTierNode); // Obtain the information of the left and
              right tables
12.     end if
13.     if no sub-query then
14.         WalkSingleFrom(node, DependencyTierNode); // No sub-query, and obtain the
              dependency information update
15.     end if
16.     node = take the next node of AST;
17. end while
18. }
```

The main steps are as follows: the first step is to directly position the root node of the "from" clause through traversal from the AST root node, from which the analysis is started (Line 3). Next, the "from" subtree is traversed until the process is completed, in which different processing is performed according to the type of the AST node (Lines 4–17). For the sub-query, this type of SQL statement will generate the corresponding nested sub-query node in the AST and recursively parse the sub-query node by means of depth-first traversal until the table name and field information corresponding to the innermost sub-query are obtained. In the traversal and analysis of the CTE nodes, the data table name and field information that CTE depends on are obtained by the same method. If the nested sub-query contains union, join, or other connection-type nodes, it is required to parse such nodes in the traversal and parsing of the nested sub-query to retrieve all the tables and corresponding field information contained in the node. In the absence of any sub-query, the dependent table name (sub-query alias) and field

Algorithm 4. Final dependency update algorithm.

```
Input: outermost node of QDM, i.e., OutputTierNode.
Output: NULL.
1. updateOutputTier(OutputTierNode)
2. {
3. foreach outputnode in OutputTierNode do
4.     matchColumnNode = outputnode;
5.     matchColumnNode = SearchAllTier(OutputTierNode, matchColumnNode); // Match the
         search dependency tier by tier
6.     UpdateNode(outputnode, matchColumnNode); // Update dependency
7. end foreach
8. }
```

name (alias) are obtained directly via the parsing of the AST node.

Through Algorithm 3, the example SQL generates three sub-QDMs from top to bottom. The source of the query output field obtained in the top tier is (null, t, null, null), which, however, is not the final source of the query output field (null, null, id, null). The source of the query output field in the second-tier sub-QDM is (null, info, null, null), and the source of the query output field in the third-tier sub-QDM is (tInfo, null, null).

The pseudocode corresponding to Algorithm 2 and Algorithm 3 represents a process of depth-first traversal and parsing of AST. In the practical traversal and parsing process, all the SQL syntax structures involved need to be processed; otherwise, syntax compatibility problems will arise. Due to limited space, Algorithms 2 and 3 are not extended.

The final dependency update algorithm (Algorithm 4) is designed to obtain the table field information that all the outermost query output fields finally depend on. This algorithm uses information such as the table name, sub-query alias, field name, and field alias as the matching keywords for all the outermost query output fields for downward matching tier by tier until the deepest tier. By default, the algorithm uses the deepest dependency information as the final source of the outermost output field.

Through Algorithms 2 and 3, the simplified model of QDM generated by the example SQL is as follows:

- the outermost sub-QDM: (null, null, id, null)$\to$(null, t, null, null);
- the second-tier sub-QDM: (null, null, id, null)$\to$(null, info, null, null);
- the innermost sub-QDM: (null, null, id, null)$\to$(tInfo, null, null, null).

In the case of the example SQL by Algorithm 4, the field on which the outermost query output field (null, null, id, null) finally depends is presented as (tInfo, null, id, null), which corresponds to the “id” field of the physical table tInfo. In this way, the complete QDM is generated.

The QDM construction algorithm calls Algorithms 2 and 3 to recursively traverse all the nodes of the AST, with the time complexity as $\mathrm{O}(n)$, where n represents the height of the syntax tree. In the case where there are m tiers of sub-QDMs in the QDM, and there are d ColumnNodes in each tier, the time complexity of Algorithm 4 is $\mathrm{O}(m \times d)$. Therefore, the total time complexity is $\mathrm{O}(n + m \times d)$, where m and d are generally not very large. The new process introduced by the construction algorithm to solve the problem of SQL rewriting is equivalent to an additional parsing of the AST during the execution of the SQL statements. As the SQL statements will eventually be converted into distributed tasks, and a large number of expensive operations such as IO and computation in the execution of tasks determine the performance loss, this algorithm has little impact on the performance of SQL execution.

2.3.3 *SQL rewriting*

The same field may appear in each syntax structure of the SQL query statement. When shallow parsing is adopted as a simple processing approach to directly rewriting all the fields defining the masking strategy in the SQL statements, the rewritten SQL statements are typically inconsistent with the original request. This results in the incorrect returned result set. In contrast, the rule engine parses the SQL request through the security rule tree and rewrites the SQL statements according to the relevant rules. Hence, the matching accuracy is higher when the query request is more complex, and its identification is more difficult.

For syntax such as “create table as select”, “insert (overwrite) into select”, and “view”, sensitive information can also be obtained through the “select” query. If the masking strategy cannot be identified and automatically inherited, sensitive data will also be leaked. There are sensitive fields in the “select” clause of “create table as select” and “insert (overwrite) into select”, for which masking rules are set. BDMasker provides two processing modes for automatic masking conversion of such SQL statements to protect sensitive data in derived tables.

Derivative mode. The masking rule corresponding to the table field name (alias), on which the query output field name (alias) of the “select” clause of “create table as select” and “insert (overwrite) into select” depends, is used as the masking rule of the query output field, and it is inserted into the protection strategy library. In the case of the query of the derived table, we perform the masking conversion according to the inserted masking rule. This mode, not modifying the original data of the derived table, is the default processing mode of this algorithm.

Rewriting mode. We rewrite the “select” clause of “create table as select” and “insert (overwrite) into select”. After their execution, the data of the derived table is masked data.

We set the masking rule of mask(f, ‘*’) for the “id” field of the “tInfo” table in the example SQL, where mask stands for the function of the masking operator, f is the name of the field to be masked, and ‘*’ is the masking parameter.

Figure 6 describes the process of Algorithm 5 for query rewriting.

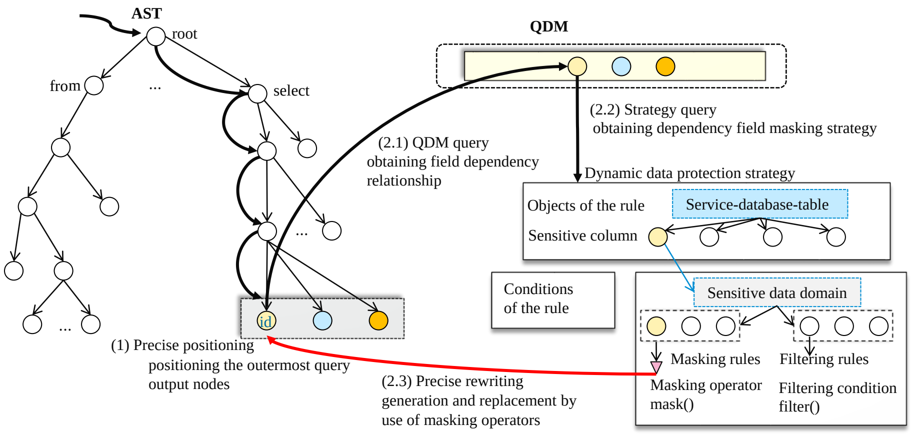


Figure 6 Flow of query rewriting

Unlike traditional methods, Algorithm 5 takes precise positioning and rewriting as its design principle. The algorithm first precisely positions all the output nodes of the outermost query by traversing the AST and only modifies the AST nodes corresponding to the outermost query output fields, without any influence on other nodes in the syntax tree (Line 3, Step 1 in Figure 6). For the “create table as select” and “insert (overwrite) into select” statements, it is required to obtain the name of the derived table and its masking mode (Lines 4–5). Next, for the output

Algorithm 5. SQL rewriting algorithm

```
Input: root node of AST, i.e., ASTRoot, and node in the outermost tier of QDM, i.e., TierNode.
Output: rewritten root node of AST, i.e., ASTRoot.
1.  ASTRoot RewriteSQL(ASTRoot, OutputTierNode)
2.  {
3.  ASTnodes = walkOutSelectAST(ASTRoot); // Obtain the AST node corresponding to the outermost
      output field of the query output
4.  table = walkInsertOrCreateAST(ASTRoot); // The "insert" or "create" statement can obtain the
      corresponding table name
5.  bInherit = getCTASMaskPolicy(table);
6.  foreach astnode in ASTnodes do
7.      qdmnode = getDependencyColumnNode(astnode); // Obtain the corresponding dependency
          table field from QDM
8.      maskPolicy = getMaskPolicy(qdmnode); // Obtain field masking strategy
9.      if maskPolicy != NULL then
10.         if table != NULL && bInherit == TRUE then
11.             addOneMaskPolicy(qdmnode, maskPolicy); // The derived table directly inherits the
                  masking strategy of the field in the parent table
12.         end if
13.         tmpSQLSeg = ConSQLSegment(astnode, maskPolicy); // Use masking operator to
              construct the SQL fragment
14.         ReWriteASTNode(astnode, tmpSQLSeg); // Replace the original AST node value
15.     end if
16. end foreach
17. return ASTRoot;
18. }
```

nodes of the outermost query, we complete the following main processing logic one by one: we first obtain the field name of the final dependency table of the current node from the QDM (Line7, Step 2.1 in Figure 6) and then call the strategy query interface to obtain the masking strategy corresponding to the field name of the final dependency table, including the masking operator function (Line 8, Step 2.2 in Figure 6). If the masking of the derived table is in the derivative mode, we create a new masking rule and use the masking rule of the parent table field as that of the field of the derived table to avoid data leaks of the derived table (Line 11). Finally, we construct an SQL fragment on the basis of the masking operator function and replace the current AST node value with the fragment value (Lines 13–14, Step 2.3 in Figure 6).

In the AST, this algorithm directly and precisely positions the AST node corresponding to the "id" field of the outermost "select" without modifying any other node. In the meantime, the field on which the "id" finally depends (tInfo, null, id, null), i.e., the "id" field of the physical table tInfo, is obtained to acquire the masking operator of the field, namely, mask(f, '*'). The AST value corresponding to the original outermost output field "id" is replaced by the SQL of mask(id, '*').

Through the process of Algorithm 5, the example SQL is rewritten as follows:

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

It can be seen that the example SQL statements have been precisely rewritten, and multiple "id" fields in the sub-query of the original SQL request were not rewritten. This well ensures the correct business logic of the original query request and thus enables good transparency to the business.

The SQL rewriting algorithm recursively traverses the AST according to the syntax structure, without the need to traverse all nodes. The time complexity of the AST is $\mathrm{O}(n)$. In the case of m outermost query output fields, the total time complexity of the algorithm is $\mathrm{O}(n+m)$, where m is usually not very large. The algorithm does not involve the traversal of all nodes and has little impact on SQL execution performance.

3 Experiment Analysis and Evaluation

3.1 Experiment environment

The experiment ran in a big-data distributed environment composed of 15 servers that were interconnected through a 10-gigabit switch. The specific software and hardware configurations of each server are presented in Table 1. The test model and data set selected for the experiment included TPC Benchmark DS (TPC-DS) and YCSB Benchmark (YCSB)[21]. TPC-DS, very close to the real scenario, represents a benchmark test set for the evaluation of decision support systems (or data warehouses) and also a test set of great difficulty. YCSB is a general performance testing tool for NoSQL.

Table 1 Software and hardware configuration in experimental environment

Name	Specification parameter
Operation system	NewStart CGS Linux V5
JDK	1.8
Hadoop	2.7.3
Hive	2.1.0
Spark	2.2.1
HBase	1.2.0
CPU	48core Intel® Xeon®CPU E5-2670 v3 @ 2.30 GHz $\times$ 4
Memory	128 GB $\times$ 1
Hard Disk Drive (HDD)	4 TB SATA $\times$ 12
Network Interface Card (NIC)	2-port 10 Gigabit NIC $\times$ 12

The experiment adopted the mode of masking strategy retention, namely that the next testing statement had the masking strategy of all the previous tested statements by default. Among all the fields of each testing statement, only one or two fields were not set with masking rules.

The performance experiment took the fluctuation ratio of the execution time before and after the masking execution as a quantitative indicator for the evaluation of the system operation efficiency. Each case was tested four times, and the Linux cache was cleaned before each execution. The average in the four tests was used for statistical analysis.

The fluctuation ratio is $|(t_1 - t_0)|/t_0$, where t_1 represents the running time for the opening of the masking switch, and t_0 is that for its closing.

3.2 Functional experiment

The experiment selected Apache Hive, the only open source big data community that supports the dynamic masking function, and BDMasker's Hive for the comparison of the dynamic masking function. It evaluated the system function from the aspects of the correctness of business logic, syntax compatibility, security, and capability scalability.

(1) The correctness of business logic. This indicator is used as the strictest function evaluation, that is, if and only if the modified query request is identical with the result set of the real query request sent by the user, the dynamic masking function can be considered completely correct. There are three indicators to measure the correctness of business logic: the correctness of the entries in the result set, the correctness of the sources of fields to be masked, and the correctness of the masking algorithm.

(2) Syntax compatibility. TPC-DS follows the SQL-99 and SQL-2003 syntax standards. As SQL cases were complex, 99 SQL test cases of TPC-DS were used to set masking rules for

the fields arising in each syntax part of the SQL statement to validate the business logic and the support capability of syntax rules after masking.

(3) Security. This indicator mainly tested the security of derived tables, inserted tables, and views generated on the basis of query requests to prevent the leak of sensitive data from such tables.

(4) Capability expansion. This indicator mainly tested the security function expansion capability of the dynamic data protection engine of the BDMasker system.

3.2.1 *Experiment on correctness of business logic*

Query76 was taken as an example to validate the correctness of the business logic of the modified SQL statement in the BDMasker system. The masking rule `mask_account_Info` was set for the `i_category` field of the "item" table, and the corresponding operator of this rule was `mask_account_info_caesar_string`. The SQL generated after the rewriting of Query76 by the BDMasker system is presented as follows:

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

The other statements remained unchanged.
```

It can be seen from the execution result set in Figure 7 that the entries of the result set are correct before and after the masking of Query76 by the BDMasker system, namely, 10 entries invariably. This can be explained by the fact that the BDMasker system accurately obtains the source of the outermost query output field and the corresponding masking rules via QDM, and the rewritten SQL statement does not affect the original business logic. In this example, the outermost query output field `i_category` is the one from the physical table item, the masking rule of which is `mask_account_info`. The rewritten statement only involves the outermost query output field `i_category`, which does not affect other syntax structures.

				Before masking	**After masking**		
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

Figure 7 Validation results of business logic correctness by Query76

We executed the 99 TPC-DS statements one by one in sequence. The correctness comparison results of business logic are presented in Table 2.

BDMasker's Hive identified the dependency between the tables and fields involved in the SQL statement through precise query analysis based on QDM and eventually merely masked the sensitive fields in the outermost output fields of the "select" query through the precise rewriting algorithm. In this way, it ensured no impact of the masking process on the business

Table 2 Correctness test results of business logic

Functional indicator	Apache Hive	BDMasker Hive
Entries of result set	All incorrect	Correct
Correctness of sources of masking column fields	All incorrect	Correct
Correctness of masking algorithm	Correct	Correct

logic. Apache Hive adopted the shallow parsing mode, and as a result, the original business logic was changed by the rewritten SQL. This lies in the incorrect SQL rewritten algorithm, which is shown in the following aspects: (1) the field to be masked will be masked first when it appears in the WHERE condition, and then the masked field will be applied to the WHERE condition; (2) when the field to be masked appears in any sub-query, it is masked first from the sub-query to the parent query and then applied again; (3) when the field to be masked appears in the `GROUP BY/ORDER BY/SORT BY/DISTRIBUTE BY/JOIN ON` statements, it will be masked first and then be applied to the `GROUP BY/ORDER BY/SORT BY/DISTRIBUTE BY` statements; (4) When the field to be masked appears in any `UDF` function, such as `SUM/AVG/UPPER`, it will be masked first and then applied as an input parameter to the `UDF` function.

3.2.2 *Experiment on syntax compatibility*

The 99 TPC-DS statements were tested in sequence, and the comparison results of syntax compatibility by experiments are presented in Table 3.

Table 3 Test results of syntax compatibility

Syntax	Apache Hive	BDMasker Hive
Simple Select query	Supported	Supported
Nested sub-query	Not supported	Supported
Condition filtering query	Not supported	Supported
Multi-table association	Not supported	Supported
Union and Union all	Not supported	Supported
Sequencing	Not supported	Supported
Groupby function	Not supported	Supported
Create table as select	Not supported	Supported
Insert (overwrite) into	Not supported	Supported
CTE	Not supported	Supported
View	Not supported	Supported
Udf parsing	Not supported	Supported
Distinct duplication removal	Not supported	Supported

It is known from the experiment results that Apache Hive does not support multiple complex SQL syntax structures such as multi-table association and condition filtering. This is because Apache Hive adopts shallow parsing without deeply parsing a large number of syntax structures, and hence, incorrect masking results are produced. The query analysis and rewriting algorithms of the BDMasker system parse and analyze each syntax structure and use 99 TPC-DS statements to perform the strict syntax compatibility test and thus ensure syntax compatibility.

3.2.3 *Security experiment*

The experiment on security protection ability was designed to test the masking capability of the derived table, inserted table, and view generated on the basis of the query request. The "create table as select", "insert (overwrite) into select", and "view" (including nested view) statements were constructed. The comparison results of the security capability by the experiment are shown in Table 4.

The experimental results show that when there are sensitive fields in the "select" clauses of "create table as select" and "insert (overwrite) into select", and masking rules are set for these fields, BDMasker provides two processing modes to protect sensitive data in such SQL

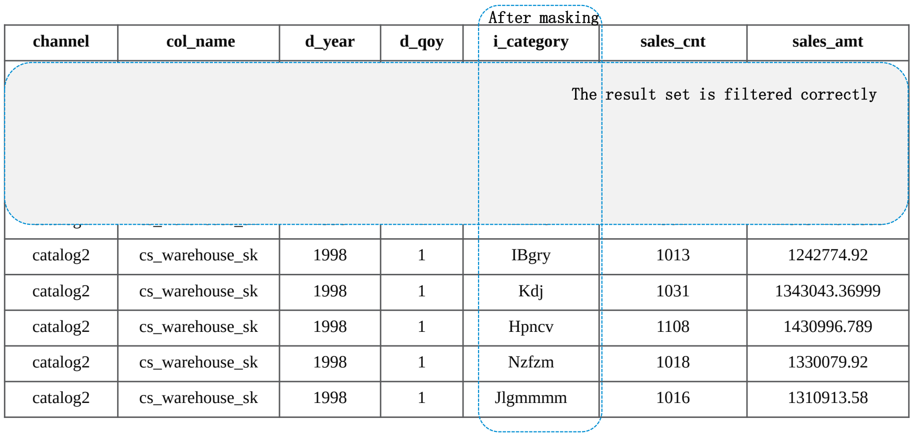
Table 4 Results of security protection capability test

Statement	Apache Hive	BDMasker's Hive
Create table as select	Will leak sensitive data	Provide effective protection
Insert (overwrite) into select	Will leak sensitive data	Provide effective protection
View	Will leak sensitive data	Provide effective protection

statements. This can automatically identify and inherit the masking strategies of the fields in the parent table and thus voluntarily protect the security of the derived tables. Moreover, it also supports view and view inheritance masking to effectively avoid the leak of sensitive data from the derived tables or views. In contrast, as Apache Hive does not automatically apply masking rules to the tables derived from "create table as select", "insert (overwrite) into select", "view", etc., the result set queried in the derived tables is still the original data, which causes the leak of sensitive data.

3.2.4 *Capability expansion experiment*

BDMasker expands the dynamic data filtering function through a plug-in function in the dynamic data protection engine. Taking Query76 as an example, we set the filtering rules for the "`web`" field of the "web_sales" table to filter out the row data whose value was equal to `catalog1` while setting the "`mask_account_info`" masking rules for the `i_category` field of the "item" table. The experimental results are presented in Figure 8.


After masking
The result set is filtered correctly

channel	col_name	d_year	d_qoy	i_category	sales_cnt	sales_amt
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

Figure 8 Experimental results of data protection capability expansion

The execution result set reveals that BDMasker has correctly filtered and masked Query76's result set. BDMakser traverses the AST in precise rewriting and judges that when the current node is a leaf node and physical table, it will call the strategy query interface to obtain the data filtering rules for the physical table and add a sub-query node to the leaf node. The original parent node of the leaf node becomes the parent node of the sub-query node, and the leaf node becomes its subsidiary node. Finally, the SQL statement corresponding to the obtained filtering rule is applied to the sub-query node. The SQL statement rewritten in this way automatically performs dynamic filtering of the data in the physical table according to the predetermined rules in its execution. This processing logic is implemented in the BDMasker system in a plug-in manner, thus protecting the sensitive data via multiple technologies in the same big data engine and well enabling the vertical expansion of the dynamic data security capability of a single engine.

3.3 YCSB performance test

3.3.1 *Single-user benchmark test*

The pressure measurement parameters such as the field length, the number of fields, and the total number of records were set through the YCSB tool to construct 2 TB test data. Then, the random write, random read, and random scan commands were executed to measure the query response time of the benchmark in the single-user mode. The results of the single-user benchmark test are presented in Figure 9.

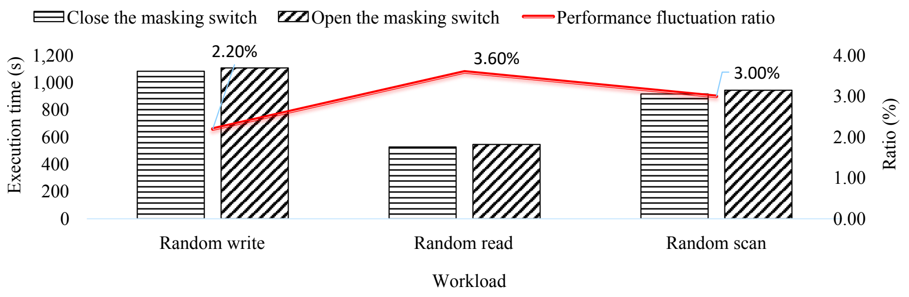


Figure 9 Performance fluctuations of dynamic masking of BDMasker's HBase

The experimental results indicate that BDMasker's HBase always maintains relatively efficient performance in the single-user mode, and the masking process has little impact on business performance. The performance fluctuation of random write, random read, and random scan is 2.2%, 3.6%, and 3.0%, respectively, with the overall average performance loss within 3%. The performance loss of random read and random scan is higher than the average value because the data is subject to masking conversion in the two processes. Under random scan, a large number of data sets are read each time, and the conversion mainly concentrates on one node; the memory use is relatively intense. In contrast, the data under random read is scattered in multiple nodes, and a single record is read each time for masking conversion; the memory use is fragmented, which makes the performance loss of random read relatively higher than that of random scan.

3.3.2 *Dynamic masking performance test of the HBase engine in multi-concurrency scenarios*

The performance of the system under different concurrencies was verified by concurrency size adjustment. The data size of the experiment was 2 TB, and the concurrency sizes went from 10 to 20, 50, and 100. Random write, random read, and random scan commands were executed to test the performance fluctuations of dynamic masking under different concurrency sizes. The experimental results are presented in Figure 10.

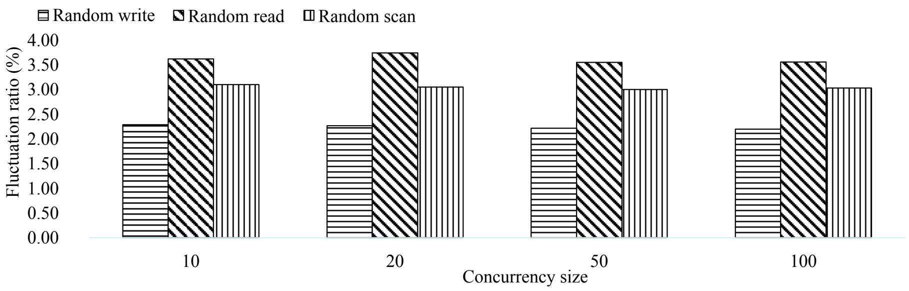


Figure 10 Performance fluctuations of dynamic masking of BDMasker's HBase under different concurrency sizes

According to the experiment results, under different concurrency sizes, BDMasker's HBase reports a random write performance fluctuation of 2.22%, a random read performance fluctuation of 3.55%, and a random scan performance fluctuation of 2.95%. The stable overall performance fluctuation falls within 3%. The increase in the concurrency size does not influence the variation of the dynamic masking performance of HBase. BDMasker's HBase always maintains relatively efficient performance under different concurrency sizes, and the dynamic masking has little impact on business performance, which can effectively ensure data security at a relatively low cost.

3.3.3 *Dynamic masking performance test of the HBase engine under data set scenarios of different sizes*

The performance of the system was verified under the experimental data sizes of 100 GB, 500 GB, 2 TB, 10 TB, 50 TB, and 100 TB. The random write, random read, and random scan commands were executed to test the performance fluctuations of dynamic masking under different data sizes. The experimental results are presented in Figure 11.

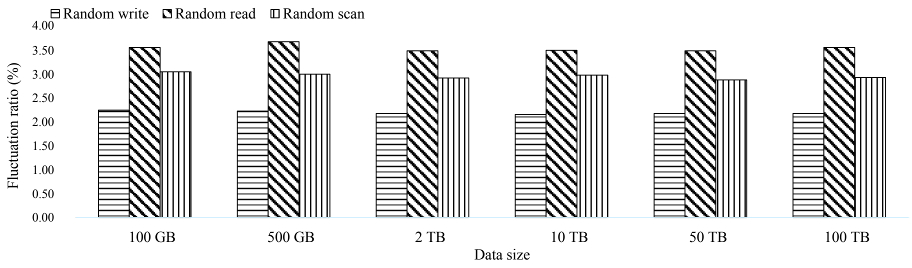


Figure 11 Performance fluctuations of dynamic masking of BDMasker's HBase under data sets of different sizes

Under different data sizes, BDMasker's HBase reports a random write performance fluctuation of 2.21%, a random read performance fluctuation of 3.54%, and a random scan performance fluctuation of 2.96%. The stable overall performance fluctuation falls within 3%. The increase in the data size does not influence the variation of the dynamic masking performance. BDMasker's HBase always maintains relatively efficient performance under different data sizes, and the dynamic masking has little impact on business performance, which can well ensure data security at a relatively low cost.

3.4 Multi-engine scalability and performance experiment

Twenty SQL statements were selected from the 99 TPC-DS statements (see Table 5). They are applied in different business scenarios, and their business logic complexity involves high, medium, and low scenarios; the syntax covers common SQL syntax types such as "sub-query", "join", aggregation, grouping, and condition. The multi-engine scalability and performance of the BDMasker system were verified on the Apache Hive and Apache Spark engines.

3.4.1 *Single-user benchmark performance test*

The single-user benchmark performance test was used to measure the response time of the benchmark query in the single-user scenario and calculate the performance fluctuation ratio, with a data size of 2 TB. The performance fluctuations of the dynamic masking of BDMasker's Hive and Spark are presented in Figures 12 and 13, respectively.

It can be seen that in the single-user mode, BDMasker maintains relatively efficient performance, with little impact on business performance in the masking process, and the average performance loss is 3%. This is because BDMasker's precise SQL rewriting is only targeted at

Table 5 Query statements selected for TPC-DC performance test

Code	Involved syntax
Query3	Select, group by, order by, limit, where, function, etc.
Query7	Select, group by, order by, limit, where, function, etc.
Query12	Select, group by, order by, limit, where, function, in, between, and, etc.
Query15	Select, group by, order by, limit, where, function, in, and, etc.
Query17	Select, group by, order by, limit, where, in, and, etc.
Query19	Select, group by, order by, limit, where, function, etc.
Query20	Select, group by, order by, limit, where, in, and, etc.
Query21	Select, nested select, group by, order by, limit, where, function, between, and, etc.
Query24	Select, nested select, group by, distribute by, order by, having, where, CTE, etc.
Query25	Select, group by, order by, limit, where, function, in, between, and, etc.
Query26	Select, group by, order by, limit, where, function, etc.
Query27	Select, group by, order by, limit, where, function, etc.
Query28	Select, nested select, group by, order by, limit, where, function, in, between, and, etc.
Query31	Select, nested select, group by, order by, limit, where, function, in, between, and, etc.
Query40	Select, nested select, group by, distribute by, order by, having, where, CTE, join, etc.
Query42	Select, group by, order by, limit, where, function, etc.
Query47	Select, nested select, group by, distribute by, order by, having, where, CTE, etc.
Query53	Select, nested select, order by, limit, where, function, and, etc.
Query82	Select, group by, order by, limit, where, function, in, between, and, etc.
Query84	Select, order by, limit, having, where, etc.

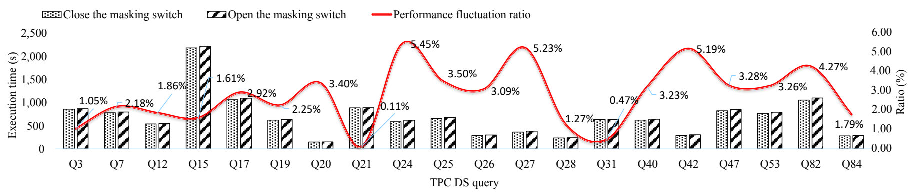


Figure 12 Performance fluctuations of dynamic masking of BDMasker's Hive

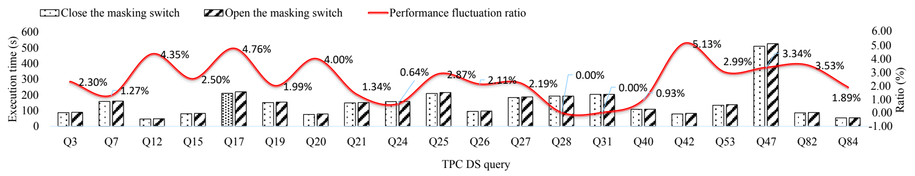


Figure 13 Performance fluctuations of dynamic masking of BDMasker's Spark

the value of the outermost query output node, without spoiling the existing structure of AST. The rewritten SQL statement involves the outermost query output field, and other syntax structures are not influenced, which ensures that the rewriting does not affect the continuous application of various query optimization strategies by the execution engine, and there is no conflict between logic optimization, physical optimization, and masking-based rewriting.

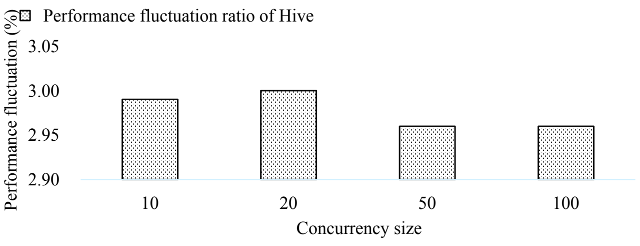
The longitudinal comparison indicates that the dynamic masking performance variations of Hive and Spark in the BDMasker system maintain a similar trend of fluctuations. There is a significant difference between Spark and Hive in the processing efficiency for the same SQL, e.g., Query24. This can be explained by the great differences in the internal optimization and execution mechanism between Hive and Spark, which result in the different performance of the same SQL under the same masking rules.

The performance fluctuation of some query statements exceeds 3% because such SQL statements have a short execution time before masking, but the masking strategy involves a larger number of complex masking algorithms, whose processing has a larger share in the whole task computing. Hence, a large amount of CPU is consumed in the conversion process, which leads to large performance fluctuations. For instance, the Query12 statement of Spark in Figure 13 involves an execution time of 46 seconds before masking and 48 seconds after masking, with a performance fluctuation ratio of 4.35%; the Query42 statement involves an execution time of 78 seconds before masking and 82 seconds after masking, with a performance fluctuation ratio of 5.13%.

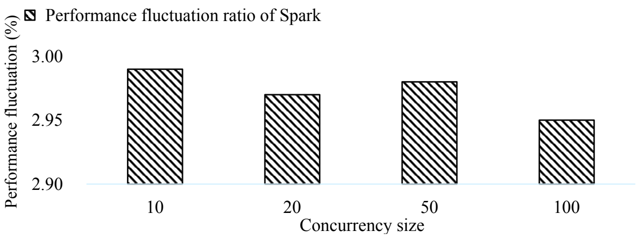
This experiment has also verified the good scalability of BDMasker. BDMasker enables the uniform management of engines such as Hive and Spark with the multi-engine-oriented unified strategy technology. This technology can be applied to most SQL-on-Hadoop engines, showing strong scalability.

3.4.2 *Dynamic masking performance test of SQL engines in multi-concurrency scenarios*

The performance of the system was verified under the concurrency size from 10 to 20, 50, and 100 in experiments. The performance fluctuation ratio was the average fluctuation ratio of the execution time of the 20 SQL statements in Table 5 under this concurrency size. The tests of Hive's and Spark's dynamic masking performance with the increase in concurrencies are presented in Figure 14.



(a) BDMasker's Hive



(b) BDMasker's Spark

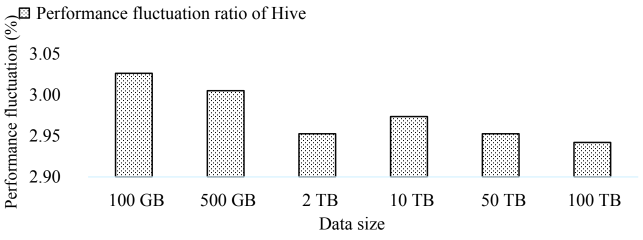
Figure 14 Performance fluctuations of dynamic masking of BDMasker's SQL engines in multi-concurrency scenarios

According to the experiment results, BDMasker always maintains good performance under different concurrency sizes, with an average performance loss of less than 3%, and the Hive and Spark engines report relatively smooth dynamic data masking performance. This shows that the BDMasker system can well ensure data security at a cost not much larger than the non-masking expense. In addition, as the rewritten SQL statements will eventually be converted into big data tasks for distributed execution on the cluster, the performance fluctuations of dynamic masking depend on the performance consumption of data masking operators during execution. Hence, we can ignore the influence of the QDM construction and SQL rewriting algorithms on the operation performance in the whole process.

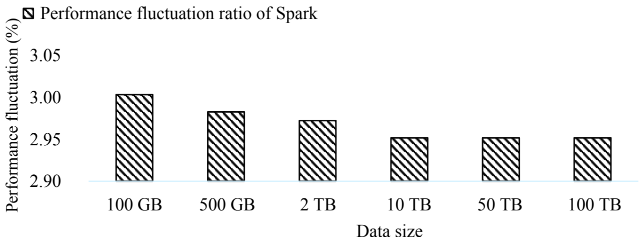
3.4.3 *Dynamic masking performance test of the SQL engine under data set scenarios of different sizes*

The performance of the system was verified under the data size from 100 GB to 500 GB, 2 TB, 10 TB, 50 TB, and 100 TB. The performance fluctuation ratio takes the average fluctuation ratio of the execution time of the 20 SQL statements in Table 5 under this data size. The time overhead of the dynamic masking performance of BDMasker's Hive and Spark SQL with the increase in data size is shown in Figure 15.

According to the experiment results, BDMasker always maintains good performance under the six data sizes, with an average performance loss of less than 3%, and the Hive and Spark engines report relatively smooth dynamic data masking performance. This shows that the BDMasker system can well ensure data security at a cost not much larger than the non-masking expense.



(a) BDMasker's Hive



(b) BDMasker's Spark

Figure 15 Performance fluctuations of dynamic masking of BDMasker's SQL engines under different dataset sizes

With the above experiments, the BDMasker system has passed the strict tests of business logic correctness and SQL syntax compatibility on the 99 TPC-DS statements. It makes the dynamic data masking transparent to the original business requests and overcomes the challenge of precise SQL rewriting. According to the results of TPC-DS and YCSB performance experiments, the BDMasker system can support the masking capability of multiple heterogeneous big data engines simultaneously. The masking process makes full use of the distributed computing power of the big data execution engine for efficient masking, with slight average performance loss and overall performance fluctuations controlled within 3%. It solves the challenges of efficient dynamic data protection processing and scalable heterogeneous environments. The theoretical analysis indicates that the masking gateway mode increases the data access path in comparison with the BDMasker system in the same hardware environment. The masking time linearly relates to the capacity of the result set, which leads to a relatively big loss of overall performance. Accordingly, the BDMasker system reports more efficient direct masking in the kernel of the big data execution engine than the masking gateway mode, and the performance difference depends on the time spent in the result set processing.

4 Conclusion

Data security in the open environment has emerged as the bottleneck restricting the development and use of big data technology. Relevant changes have taken place in the protection mode, protection object, and the relationship between management and technology of data security in the open big data context, which sets higher standards for data security protection. As a result, traditional data security measures are already out of mode. This paper designed and implemented the high-performance dynamic data protection system BDMasker tailored to the open big data environment, which can precisely, efficiently, and dynamically protect sensitive data in a scalable manner. Following the principle of "zero business awareness", we proposed a precise query analysis and SQL rewriting technology on the basis of QDM, which can precisely perceive but does not change the original business request to achieve zero impact of the dynamic masking process on the businesses. We also designed a multi-engine-oriented unified security strategy framework, which manages and supports the vertical expansion of multiple dynamic data protection capabilities through plug-in data protection strategies. Appropriate data protection strategies can be used according to application scenarios, business objectives, and data characteristics. In addition, the framework also achieves the horizontal expansion of multiple computing engines through strategy agents and standardized interfaces, which ensures the use of the distributed computing power of the big data execution engine to improve the data protection processing performance of the system. Finally, relevant experiments have verified the effectiveness, good performance, and scalability of the BDMasker system.

In future work, BDMasker will expand the support for AI-driven dynamic data protection and prevent malicious users from using reasoning or violence technology to bypass masking[22]. We will also study how to work out more general methods via AI so that they can be more easily adapted to big data engines. Closely following the requirements of national laws and regulations such as the *Data Security Law of the People's Republic of China* for data security protection[23], BDMasker, combined with the data security technology system and the application of new technologies, will provide better and more diverse data security services to ensure the security of data use under new standards and requirements in the new era and environments.

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.

[2] Fang BX, Jia Y, Li AP, Jiang R. Privacy preservation in big data: A survey. Big Data Research, 2016, 2(1): 1–18. [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. [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.

[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). [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.

[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. 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. 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.

[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] Software testing help. 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. 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. 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. Proc. of 2017 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. Proc. of the 2nd ACM Symp. on Cloud Computing. New York: ACM, 2011. 1–14. [doi: 10.1145/2038916.2038925]

[22] Yesin V I, Vilihura V V. Some approach to data masking as means to counteract the inference threat. Radiotekhnika, 2019(198): 113–130. [doi: 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.

Yaofeng Tu, Ph.D, researcher. His research interests include big data, distributed systems, and machine learning.

Dezheng Wang, senior engineer. His research interests include big data storage and computing, privacy computing, and blockchain. AI, and privacy computing.

Jiahao Niu, senior engineer. His research interests include big data, as well as data security and privacy protection technology.

Hong Gao, senior engineer. His research interests include big data, AI, data mining, and NLP.

Jin Xu, senior engineer. His research interests include big data,

Fang Yang, big data engineer. His research interests include data warehouse and offline computing.

Ke Hong, senior engineer. His research interests include big data storage and computing and AI.