STATISTICS API: DBMS-INDEPENDENT ACCESS AND MANAGEMENT OF DBMS STATISTICS IN HETEROGENEOUS ENVIRONMENTS

Tobias Kraft and Bernhard Mitschang
Institute of Parallel and Distributed Systems, University of Stuttgart, Universitätsstraße 38, 70569 Stuttgart, Germany

Keywords: Query optimization, optimizer statistics, database meta data, federation.

Abstract: Many of todays applications access not a single but a multitude of databases running on different DBMSs. Federation technology is being used to integrate these databases and to offer a single query-interface to the user where he can run queries accessing tables stored on different remote databases. So, the optimizer of the federated DBMS has to decide what portion of the query should be processed by the federated DBMS itself and what portion should be executed at the remote systems. Thereto, it has to retrieve cost estimates for query fragments from the remote databases. The response of these databases typically contains cost and cardinality estimates but no statistics about the data stored in these databases. However, statistics are optimization-critical information which is the crucial factor for any kind of decision making in the optimizer of the federated DBMS. When this information is not available optimization has to rely on imprecise heuristics mostly based on default selectivities.
To fill this gap, we propose Statistics API, a JAVA interface that provides DBMS-independent access to statistics data stored in databases running on different DBMSs. Statistics API also defines data structures used for the statistics data returned by or passed to the interface. We have implemented this interface for the three prevailing commercial DBMSs IBM DB2, Oracle and Microsoft SQL Server. These implementations are available under the terms of the GNU Lesser General Public License (LGPL). This paper introduces the interface, i.e. the methods and data structures of the Statistics API, and discusses some details of the three interface implementations.

1 INTRODUCTION

Relational DBMS technology has been used for decades in all kinds of application areas and there are multiple relational DBMSs from different vendors on the market. Furthermore, many of todays applications access not a single but a multitude of databases running on different DBMSs. Left side of Figure 1 shows how such an application scenario looked like in the early days. Given a query combining data of different remote databases, the application programmer was responsible to split this query and decide what portion of the query should be executed on which remote database. Furthermore, the application had to map the data structures returned by the remote systems to its own data structures and to integrate and postprocess the data of the different sources. So, the decision which parts of the overall query are processed by which remote system and which parts are processed by the application itself was coded statically into the application code.

With the development of federated DBMS technology, this decision was handed over to the federated DBMS, in particular to its optimizer. I.e., the application sends the overall query to a single federated DBMS server which dynamically splits and distributes the query among the remote databases (see the center of Figure 1). The federated DBMS also covers schema and data mapping issues, thus decreasing the impedance mismatch. The same holds for Data Grid middleware that provides distributed query processing in general. Additionally, Data Grid middleware may also include management of replicas, data caches, and other resources.

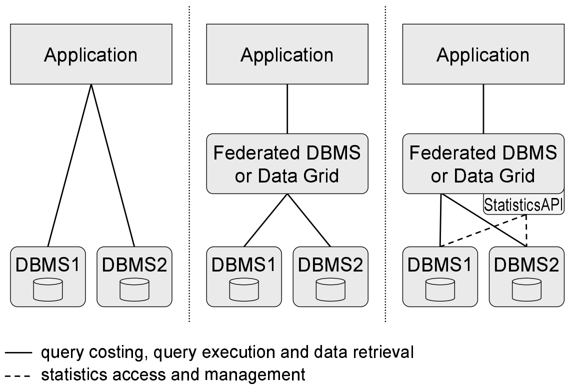
Application
Application
Application
Federated DBMS or Data Grid
Federated DBMS or Data Grid
StatisticsAPI
DBMS1
DBMS2
DBMS1
DBMS2
DBMS1
DBMS2
query costing, query execution and data retrieval
statistics access and management

Figure 1: Different stages of data federation.

The distribution decision is typically based on cost estimates. Thereto, the optimizer of the federated DBMS sends query fragments to the remote database for costing. The typical response contains a cardinality estimate and some cost estimates. However, this information is not sufficient for the task the optimizer of the federated DBMS has to do, i.e., query optimization for the fragment that ought to be processed at the federation tier. Since optimization-critical information like the number of distinct values or the minimum and maximum value of a column is unavailable, the optimizer has to rely again on imprecise heuristics using default selectivities as introduced by Selinger et al. in 1979 (Selinger et al., 1979). To support the optimizer of the federation tier, the access to statistics of the remote databases would be necessary but this is not supported by todays federation technology. This is due to the fact, that the way of storing and accessing the optimizer statistics of a certain DBMS is very proprietary and standardized formats and interfaces for the management of statistics data do not exist. Some DBMSs permit direct access to the respective catalog tables or views whereas others even provide a programming interface for this purpose; in some DBMSs the user is limited to read access whereas others allow to modify the statistics data. Furthermore, DBMSs of different vendors also differ in the amount and type of statistics they offer.

To fill this gap and to enable proper optimization tasks at the federation tier, we propose Statistics API, a JAVA application programming interface (API) that enables DBMS-independent access to statistics data stored in databases running on various DBMSs. Statistics API provides methods to retrieve, modify and delete statistics data. It also defines DBMS-independent data structures used for the statistics data returned by or passed to the methods of the API. We have chosen a very flexible format for histogram data to cover all the different types and classes of histograms (Ioannidis, 2003). Additionally, Statistics API provides methods to retrieve cardinality and cost estimates for arbitrary SQL statements as well as further meta data that may be helpful in conjunction with the offered statistics data. Federated DBMSs and Data Grid middleware can use the interface to retrieve and exploit fine-grained statistics required for query optimization and query planing purposes (see right side of Figure 1). The uniform data structures also allow to compare and exchange statistics data between different DBMSs. However, the application of the Statistics API is not restricted to distributed query processing only. Diverse performance optimization tools that sit on top of a single database system may also benefit from this feature. Modifying statistics and afterwards retrieving the cost estimates based on these new statistics values allows for what-if analysis, i.e. this allows to analyze the behavior of cost estimates for different data distributions without modifying the queried tables. Furthermore, the modification of statistics also enables to set a data distribution on tables temporarily used for intermediate results of query sequences (Kraft et al., 2003) to improve the quality of the execution plans of the queries that access these tables.

We have implemented the Statistics API for the three commercial DBMSs IBM DB2, Oracle and Microsoft SQL Server. The Statistics API and its implementations are freely available under the terms of the GNU Lesser General Public License (LGPL) [1]

The main contributions of this paper can be summarized as follows:

- An API that complements the JDBC-API by providing DBMS-independent access to DBMS statistics, meta data and optimizer estimates.
- DBMS-independent data structures to hold the meta data and a flexible histogram format that abstracts from the proprietary data structures used in the different DBMSs.
- Freely available implementations of the API for the three prevailing commercial DBMSs IBM DB2, Oracle and Microsoft SQL Server.

The paper is organized as follows. Related work is discussed in Section 2. Section 3 describes the Statistics API in more detail, i.e. the methods and data structures provided by this API. Section 4 addresses the implementations for the three mentioned commercial DBMSs. Section 5 concludes this paper and gives an outlook on future work.

[1]StatisticsAPI can be downloaded from the following web page: http://www.ipvs.uni-stuttgart.de/abteilungen/as/forschung/projekte/CEOPS/ceops/StatisticsAPI

2 RELATED WORK

JDBC (Ellis et al., 2001) is a widely-used API for the JAVA programming language that homogenizes the client access to different relational DBMSs. It provides uniform methods for querying and updating data in a local or remote database. Furthermore, JDBC provides the DatabaseMetaData interface which is an interface to let applications know the capabilities of a DBMS in combination with the used JDBC driver. It provides methods to retrieve database meta data and a small set of statistics. The Statistics API also uses JDBC connections to communicate with the databases and it extends the capabilities offered by JDBC's DatabaseMetaData interface. It adds read and write access to fine grained statistics like histograms that cannot be retrieved by the DatabaseMetaData interface, write access to all statistics that can be retrieved by the DatabaseMetaData interface, and read access to optimizer estimates. In practice, DatabaseMetaData interface implementations provide less information than possible due to the fact that it's left to the vendor how much of the meta data can be retrieved via this interface. Furthermore, there is no common behavior for different database systems concerning input parameters of the methods, e.g. search patterns are treated differently and some drivers require that the names of database objects are being passed in upper case letters. The Statistics API implementation also addresses these problems, i.e., it provides as much information as possible and it absolutely shows a common behavior for different DBMSs.

Federation is based on wrapper technology and wrappers typically focus on uniform distribution statistics like cardinality, number of distinct values, lowest and highest value of a column (Lu et al., 1993). Some approaches like Garlic (Roth et al., 1999) (Roth et al., 1996) are able to optionally support more detailed distribution statistics but in practice prototypes are restricted to uniform distribution statistics. In standards such as SQL/MED (Melton et al., 2001) (Melton et al., 2002) and in commercial products like DB2 (IBM, 2004a) wrappers just provide cost and cardinality estimates for query fragments but they do not support the retrieval of statistics. Furthermore, such approaches only support a very limited subset of SQL for costing.

Learning approaches like Leo (Ewen et al., 2005) try to retrieve statistics from remote systems on-the-fly by piggy-backing on query execution or asynchronously by querying the tables of the remote databases. These statistics can only be used reactively for future query planning but not for the first time a query is being executed or only with an appropriate overhead. So, for the first execution such reactive approaches may benefit from an interface like Statistics API.

For the implementations, we made use of the manuals and optimization-related literature offered by the DBMS vendors. However, the management of statistics is often sparsely documented and examples that show how to use these features are hard to find. A short description of the catalog tables of IBM DB2 can be found in the *IBM DB2 Universal Database SQL Reference* (IBM, 2004c). Further information, about the usage of statistics in the optimizer and how to update the statistics tables can be found in the *IBM DB2 Universal Database Administration Guide: Performance* (IBM, 2004b). Similarly, information about the usage of statistics in Oracle and their storage in the catalog tables can be found in the *Oracle Database Reference* (Oracle, 2003b) and in the *Oracle Database Performance Tuning Guide* (Oracle, 2003a). The package DBMS_STAT which enables to modify and delete statistics is documented in the *PL/SQL Packages and Types Reference* (Oracle, 2003c). Additional information can be found in Oracle's web forum *AskTom* (Oracle, 2006). For Microsoft's SQL Server, we made use of the *Transact-SQL Reference* (Microsoft, 2006) which is available online at the Microsoft Developer Network (MSDN).

3 THE STATISTICS API

Statistics API is a JAVA interface that defines methods to uniformly access statistics, meta data and optimizer estimates of different DBMSs. It is a 'low-level' interface that offers a set of basic methods that can be used to compose more powerful methods consisting of multiple basic method calls. Therefore, the Statistics API has an explicit connection management that allows to run multiple methods of the Statistics API using the same connection, i.e., the application has to open a connection to the target database before it can use the access methods and afterwards it has to close the connection again. This may be helpful, e.g., for an application that has to retrieve the names and the statistics data of all columns of a given table. This application can open a connection, retrieve the column names of the table stored as meta data, retrieve the statistics data by calling the appropriate method for each column and close the connection. As Statistics API uses ordinary JDBC connections, they can also be reused by the application for regular data management purposes. However, the Statistics API includes its own *connect* and *disconnect* method. In comparison to JDBC, the connect method doesn't require the user to provide the DBMS-specific connect-url, it only requires the components necessary to build this url like the ip address and the port number of the database server.

3.1 Data Structures

To select the statistics which the Statistics API should support, we analyzed the three commercial DBMSs IBM DB2, Oracle and Microsoft SQL Server. Each of these three DBMSs provides a huge set of statistics and meta data. For the Statistics API, we have selected those statistics and meta data elements that are supported by at least two of the three analyzed DBMSs. Furthermore, we focused on meta data and statistics about relational structures. The biggest overlap we identified in the area of logical statistics, i.e. statistics that describe the data distribution and therefore directly influence selectivity and cardinality estimation. Great differences exist in the semantics of statistics provided for indexes, e.g. DB2 and Oracle both have statistics that reflect the degree of index clustering but their definition and dimension units are totally different. This is due to the fact, that different DBMSs support different index types and that each index type or index implementation has specific statistics. Similarly, there exist DBMS-specific statistics regarding character string data that improve selectivity estimation for prefix patterns or postfix patterns used in LIKE predicates. Statistics that describe physical characteristics of either the data or the hardware of the underlying system are also DBMS-specific and usually not stored in the database catalog.

Table 1 lists all statistics, meta data and optimizer estimates actually supported by the Statistics API. For each item the list contains its name, the associated JAVA data type and a short description. We grouped the items into seven groups according to the type of information they provide and according to the type and granularity of the database object to which they refer: *table statistics, column statistics*, *index statistics, table meta data, column meta data*, *index meta data* and *estimates*. Statistics API provides a JAVA class for each of these groups that contains the associated items as attributes. Statistics, meta data and optimizer estimates represented by long, int and boolean values are stored in attributes of the associated wrapper classes. This allows to distinguish whether a value has been assigned to it or not, i.e., if no value has been assigned to it the associated attribute is set to *null*. For each attribute there exists a *get* and an *isAvailable* method. The *get* method (e.g. *getCardinality*) returns the attribute value. The *isAvailable* method (e.g.

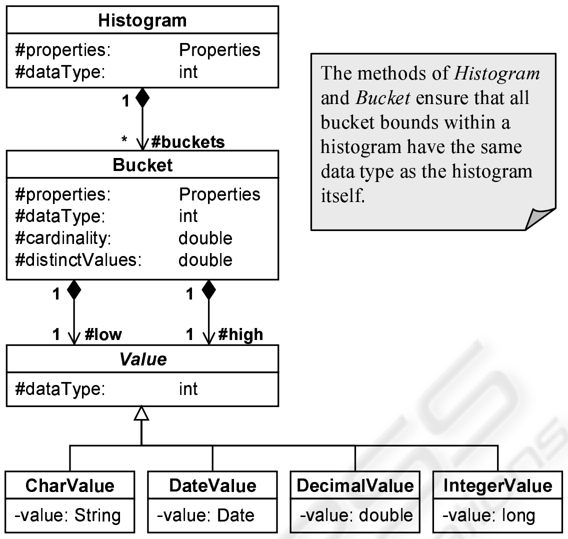

Figure 2: Class diagram of the classes related to histograms (methods are omitted).

isCardinalityAvailable) returns whether a value for the attribute is available or the associated attribute is set to *null*. In addition, statistic attributes have a *clear* and a *set* method assigned. The *clear* method (e.g. *clearCardinality*) resets the statistic attribute to *null*. The *set* method (e.g. *setCardinality*) sets the statistic attribute to the new value passed as parameter. *clear* and *set* methods are not available for meta data and optimizer estimates because the classes associated with meta data and estimates are only used as return values in the methods of the Statistics API but not as input parameters. Therefore, the values stored in these classes needn't be changed by the application.

For histogram data, we provide a flexible data structure that can store different types of histograms. Figure 2 shows a UML class diagram of the histogram-related classes. Independent of their data type all histograms are represented by instances of the class *Histogram*. The *Histogram* class contains a vector that stores the buckets of the histogram. These buckets are instance of the class *Bucket*. Each bucket represents an interval in the domain of the data type associated with the histogram. Therefore, it has a lower bound and an upper bound. Additionally, each bucket stores the number of distinct values and the number of rows that fall into this interval. The lower and upper bounds are stored in data-type specific classes, i.e., for each data type that is supported by the Statistics API an appropriate class exists (*CharValue, DateValue, DecimalValue, IntegerValue*). These classes cover the data-type specific behavior of the value. They inherit from the abstract superclass *Value*. The NULL value that may appear in histograms of any data type is represented

Table 1: Statistics, meta data and optimizer estimates supported by the Statistics API.

TableStatistics		
cardinality	long	number of rows in the table
pagesAllocated	long	number of pages allocated for the table (used + unused)
pagesUsed	long	number of pages containing rows of the table
ColumnStatistics		
avgLength	long	average length of the column (in byte)
histogram	Histogram	a histogram that approximates the data distribution of the column
IndexStatistics		
distinctKeys	long	number of distinct values in the index key columns
distinctKeysFirstNColumns	long[]	number of distinct values in subsets of the index key columns
leafPages	long	number of index leaf pages
levels	long	number of index levels
TableMetaData		
columns	String[]	array containing the column names of the table
indexes	String[][]	array containing name and schema of each index on the table
pageSize	long	size of a page in the associated tablespace (in byte)
ColumnMetaData		
dataTypeName	String	name of the data type of the column in the target DBMS
sqlDataTypeNumber	int	SQL data type of the column from java.sql.Types
histogramDataTypeNumber	int	data type of the histogram (indicates which Value class is used)
size	int	maximum number of characters / precision
digits	int	number of fractional digits
nullable	boolean	true if the column allows NULL values, false otherwise
IndexMetaData		
indexColumns	String[]	array containing the column names of the index key
indexColumnsAscending	boolean[]	array containing the order of the values in the columns of the index key; true for ascending order, false for descending order
includedColumns	String[]	array containing the names of the columns that are included in the index but that are not part of the index key
unique	boolean	true if the index key is unique, false otherwise
clustered	boolean	true if the index is a clustered index, false otherwise
pageSize	long	size of a page in the associated tablespace (in byte)
Estimates		
cardinality	double	estimated cardinality of the result of the given statement
cost	double	estimated total cost of the given statement (DBMS dependent)

by a bucket where the lower and upper bound is set to null. This way of handling NULL values like 'normal values' eases computations on histograms. The *Histogram* class also contains methods to transform the stored histogram into a serial histogram, an equi-width histogram or an equi-height histogram and methods to reduce the number of buckets by merging them. The according algorithms assume uniform distribution within the buckets. These transformation methods are necessary because some DBMSs can only store histograms of a certain type or size. At the moment, we only support unidimensional histograms due to the fact that this kind of histogram is supported by nearly all DBMSs on the market whereas multidimensional histograms are not supported or only rudimentary supported by some research DBMSs.

We treat simple column statistics regarding data distribution also as histograms, i.e., when the lower bound, the upper bound, the cardinality, the number of distinct values and the number of NULL values are available for a given column, we can build a histogram containing a single bucket made up of the given bounds and an additional NULL-bucket when NULL values are present.

Note, that the measuring unit of the cost value depends on the DBMS due to the fact that some DBMS provide cost estimates in milliseconds whereas others have their own abstract cost measure.

3.2 API Methods

For each statistics class (see Table 1) there exists a *delete* method, a *get* method, and a *set* method in the Statistics API. The *delete* method (e.g. *deleteTableStatistics*) deletes existing statistics data in the target database. The *get* method (e.g. *getTableStatistics*) retrieves statistics from the target database and returns an appropriate statistics object. Since not every DBMS supports all statistics covered by the statistics object or some statistics may not have been gathered yet, some of the statistic values in the returned statistics object may be unavailable, i.e., the associated attribute may be set to *null*. The *set* method (e.g. *setTableStatistics*) replaces the statistics values in the database with the new values passed in the appropriate statistics object. For all statistics where the value of the associated attribute in the statistics object is *null*, the value of the associated statistic will not be deleted in the database but the old value will be kept. For meta data, only *get* methods are available because these data contains information about the database structure and physical layout that cannot be changed directly from outside. The same holds for optimizer estimates.

The methods of the Statistics API are not case sensitive regarding identifiers of database objects such as table names or schema names, i.e., identifiers of database objects can be passed in lower case letters, upper case letters or a mixture of both. Since indexes and DDL statements regarding indexes are still not part of the SQL standards, different alternatives to identify an index exist. In some DBMSs an index is a separate object that can be stored in another schema than the associated table. In this case, an index is identified by its name and schema. In other DBMSs an index belongs to the associated table and therefore is identified by its name and the associated table's name and schema. To support both alternatives of index identification the index related methods of the Statistics API require the name and schema of the index as well as the name and schema of the associated table as input. Even when an index can be identified by its name and schema in the target DBMS, an interface implementation should check whether the given name and schema of the associated table is correct. When the index is identified by its name and the associated table an implementation should ignore the value passed as index schema.

The Statistics API provides its own exception classes *StatisticsAPIException*, *NoSuchObjectException* and *UnsupportedMethodException*. The two latter ones are specializations of the first one. A *NoSuchObjectException* is being thrown when the object addressed by the input parameters of a method does not exist. An *UnsupportedMethodException* is being thrown when a method of the Statistics API is being called that is not supported by the chosen implementation. Additionally, the histogram-related classes provide their own exception class *HistogramException*. When such an exception occurs during the execution of an API method, it is being encapsulated into a *StatisticsAPIException* object by this method.

4 IMPLEMENTATIONS

For each DBMS the Statistics API should be used with, there has to exist an appropriate implementation. These implementations behave like wrappers that map the methods of the Statistics API to one or more DBMS-specific SQL statements. These SQL statements directly access catalog tables or execute stored procedures provided by the target database. Only for the retrieval of column meta data the JDBC DatabaseMetaData interface is being used. This is due to the fact, that we exploit the mapping of DBMS-specific data types to the data types of *java.sql.Types* which is offered by the JDBC DatabaseMetaData interface.

In the following sections, we discuss our three Statistics API implementations in more detail.

4.1 IBM DB2 V8.2 Implementation

DB2 offers no special API to access statistics data and meta data but allows for direct access to the appropriate catalog views. So, retrieving this data can simply be realized by querying these catalog views. As most of the columns in these views are updatable, it's also possible to delete or modify existing statistic values. DB2 does some consistency checks when updating these views to avoid serious inconsistencies within the database catalog. So, the order in which the tuples are updated is crucial. Except for histograms, the associated entries for the statistics already exist in the catalog tables and as long as no statistics have been gathered these entries contain a default value that marks them as not available. However, entries in the histogram view *syscat.coldist* only exist when statistics have been gathered. So, updates to this view are not possible as long as RUNSTATS using the WITH DISTRIBUTION option hasn't been called. Finally, we use the EXPLAIN tool to get cost and cardinality estimates. Thereto, we call EXPLAIN with the given SQL statement and query the cost and cardinality information from the appropriate EXPLAIN tables.

The identifiers of database objects such as table names or schema names are stored in upper case letters in the catalog tables of DB2. So, the DB2 implementation of the Statistics API converts the identifiers passed as input attributes to upper case before they are used for comparison within a query. Indexes can be identified by their name and schema. Anyhow, the implementation also asks for the correct name and schema of the associated table.

As DB2 only stores the second-lowest and second-highest value of a column in its catalog tables, we treat the second-lowest as the lowest and the second-highest as the highest value. Thereto, we assume that there is no great difference between the second-lowest and the lowest value and between the second-highest and the highest value. Furthermore, at the moment the *setColumnStatistics* method is restricted to the modification of the quantile data in *syscat.coldist*, frequent values are not being set.

The API implementation for IBM DB2 V8.2 also works with IBM DB2 V9. However, the methods *deleteTableStatistics* and *deleteIndexStatistics* do not reset the values of the statistics columns added in IBM DB9 V9 to the respective catalog tables.

4.2 Oracle 10g Implementation

In an Oracle database system statistics can reside in two different locations: in the database catalog tables or in tables created in the user's schema for this purpose. Due to the fact, that only statistics stored in the catalog have an impact on the cost-based optimizer our API implementation only operates on this location. As the catalog views are not updatable, Oracle provides the DBMS_STATS package that allows to retrieve statistics data as well as to set and delete statistics data. At the moment, we store histograms in Oracle as equi-height histograms. Except for the retrieval of histogram data, our Statistics API implementation solely uses the procedures of the DBMS_STATS package. We do not use DBMS_STATS for histogram retrieval because the associated procedure returns the histogram data in VARRAYs which can only be retrieved when using Oracle's extensions to JDBC. So, we decided to query the appropriate statistics table instead of using Oracle's extended JDBC classes. For the retrieval of meta data we also make use of the appropriate catalog views. Finally, we use the EXPLAIN PLAN command to retrieve the cost and cardinality estimates for a given SQL statement.

The identifiers of database objects are stored in upper case letters in the catalog tables of Oracle. The procedures of the DBMS_STATS package also ask for identifiers in upper case letters. So, the Statistics API implementation converts identifiers passed as input attributes to upper case before they are used for comparison in a query or as parameter in a procedure call. Indexes can be identified by their name and schema. Anyhow, the implementation also asks for the correct name and schema of the associated table.

As the index statistic *distinctKeysFirstNColumns* is not available in Oracle, it is also not available in *IndexStatistics* objects returned by *getIndexStatistics* and it is not considered when setting index statistics. Furthermore, our implementation does not support object tables and nested tables and we do not provide an extended support for statistics of partitioned tables and partitioned indexes. However, *getTableMetaData* also retrieves a cluster index when the target table is part of a cluster which for example is not supported by the JDBC DatabaseMetaData implementation. Furthermore, *getTableMetaData* retrieves page size for 'normal' tables as well as for index organized tables (IOT) and for partitioned tables as well as for non-partitioned tables. This is not trivial because depending on the kind of table and depending on partitioning the information of the page size is stored in different catalog views. Retrieving the columns of an index meets another problem. For each index column that is denoted to be sorted in descending order, Oracle internally adds a hidden column to the associated table which includes the original column as an expression and uses the hidden column in the index key. Unlike the JDBC DatabaseMetaData implementation, we consider this and return the column name included in the expression and not the name of the generated hidden column. Similarly, for indexes defined on expressions we also return the expression instead of the column name in the *IndexMetaData* object.

Please note, to get access to the full functionality of the Statistics API the account used by the application to connect to the database must own the necessary rights to access catalog views and to execute the procedures of the DBMS_STATS package.

4.3 Microsoft SQL Server 2005 Implementation

In SQL Server, the access to catalog views is limited to retrieval. Documentation about how to manipulate statistics data has not been disclosed yet and therefore this functionality is not available in our implementation for SQL Server. So, when a *delete* or *set* method is being called, an *UnsupportedMethodException* will be thrown.

To retrieve statistics and meta data, we make use of the catalog views introduced in SQL Server 2005. Due to the fact, that most of the column and index statistics are not available in these catalog views, we additionally have to call the DBCC-command SHOW_STATISTICS. SHOW_STATISTICS requires an index name or the name of a statistics group as target. Regarding statistics a statistics group in SQL Server is similar to an index. A column can be part of multiple statistics groups and a statistics group can contain multiple columns but detailed information and a histogram is only available for the first column in a statistics group or index. So, we have to query the catalog views to get the names of all statistics groups and indexes where the given column is in the first place. When this query returns multiple occurrences, we choose the latest, i.e., the one that has been updated last. Then we can call SHOW_STATISTICS with the name of this statistics group or index.

To avoid the execution of a query but to get the associated cost and cardinality estimate, we must set SHOWPLAN_ALL ON. Afterwards, when we send a SQL statement to the database, it returns the query plan including some additional information in a tabular format as result set. We read the cardinality estimate and the cost estimate contained in the first return row and set SHOWPLAN_ALL back to OFF.

Indexes can be identified by their name and the name and schema of the associated table. This is due to the fact, that in SQL Server an index is tightly coupled with the associated table. Hence, the same name can be used for multiple indexes as long as they are not associated with the same table.

To get access to the full functionality of the Statistics API, the account used by the application to connect to the database must own the server-role *sysadmin* or the database-role *db_owner*.

5 CONCLUSION AND FUTURE WORK

In this paper we proposed a DBMS-independent JAVA interface that provides read access as well as write access to statistics stored in databases on different relational DBMSs. This interface provides not just a set of methods but also a set of data structures to store the retrieved data in a DBMS-independent format. It can be viewed as an extension and unification of existing interfaces used for statistics retrieval or costing of statements.

In the future, the Statistics API can be enriched by additional statistics or meta data elements as needed. Extension to handle multidimensional histograms are also possible. Further points for improvements are already given in the text. Although relational DBMSs dominate, an extension of our approach to other than relational backend DBMSs and data stores is another valuable next step.

REFERENCES

Ellis, J., Ho, L., and Fisher, M. (2001). *JDBC(TM) 3.0 Specification, Final Release.* Sun Microsystems, Inc.

Ewen, S., Ortega-Binderberger, M., and Markl, V. (2005). A learning optimizer for a federated database management system. In *Proc. BTW, Karlsruhe, Germany*.

IBM (2004a). *IBM DB2 Information Integrator, Wrapper Developer's Guide, Version 8.2.* IBM Corp.

IBM (2004b). *IBM DB2 Universal Database, Administration Guide: Performance, Version 8.2.* IBM Corp.

IBM (2004c). *IBM DB2 Universal Database, SQL Reference Volume 1, Version 8.2.* IBM Corp.

Ioannidis, Y. (2003). The History of Histograms (abridged). In *Proc. VLDB, Berlin, Germany*.

Kraft, T., Schwarz, H., Rantzau, R., and Mitschang, B. (2003). Coarse-Grained Optimization: Techniques for Rewriting SQL Statement Sequences. In *Proc. VLDB, Berlin, Germany*.

Lu, H., Ooi, B. C., and Goh, C. H. (1993). Multidatabase Query Optimization: Issues and Solutions. In *Proc. RIDE-IMS, Vienna, Austria*.

Melton, J., Michels, J.-E., Josifovski, V., Kulkarni, K. G., and Schwarz, P. M. (2002). SQL/MED - A Status Report. *SIGMOD Record*, 31(3):81–89.

Melton, J., Michels, J.-E., Josifovski, V., Kulkarni, K. G., Schwarz, P. M., and Zeidenstein, K. (2001). SQL and Management of External Data. *SIGMOD Record*, 30(1):70–77.

Microsoft (2006). *SQL Server 2005 Books Online - Transact-SQL Reference. http://msdn2.microsoft.com/en-us/library/ms189826.aspx.* Microsoft Corp.

Oracle (2003a). *Oracle Database Performance Tuning Guide, 10g Release 1 (10.1).* Oracle Corp.

Oracle (2003b). *Oracle Database Reference, 10g Release 1 (10.1).* Oracle Corp.

Oracle (2003c). *PL/SQL Packages and Types Reference, 10g Release 1 (10.1).* Oracle Corp.

Oracle (2006). *Ask Tom. http://asktom.oracle.com/.* Oracle Corp.

Roth, M. T., Arya, M., Haas, L. M., Carey, M. J., Cody, W. F., Fagin, R., Schwarz, P. M., Thomas II, J., and Wimmers, E. L. (1996). The Garlic Project. In *Proc. SIGMOD, Montreal, Quebec, Canada*.

Roth, M. T., Ozcan, F., and Haas, L. M. (1999). Cost Models DO Matter: Providing Cost Information for Diverse Data Sources in a Federated System. In *Proc. VLDB, Edinburgh, Scotland, UK*.

Selinger, P., Astrahan, M., Chamberlin, D., Lorie, R., and Price, T. (1979). Access Path Selection in a Relational Database Management System. In *Proc. SIGMOD, Boston, Massachusetts, USA*.