RICE UNIVERSITY

Adding Vector and Matrix Support to SimSQL

by

Shangyu Luo

A THESIS SUBMITTED

IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE

Master of Science

APPROVED, THESIS COMMITTEE:

Dr. Chris Jermaine, *Chair*

Associate Professor of Computer Science

Dr. Luay Nakhleh

Associate Professor of Computer Science

Dr. T. S. Eugene Ng

Associate Professor of Computer Science

HOUSTON, TEXAS

JANUARY 2016

Abstract

Adding Vector and Matrix Support to SimSQL

by

Shangyu Luo

In this thesis, I consider the problem of making linear algebra simple to use and efficient to run in a relational database management system. Relational database systems are widely used, and much of the data in the world is stored within them. Having linear algebra integrated into a relational database would provide support for tasks such as in-database analytics and in-database machine learning. Currently, when it is necessary to perform such analyses, one must either extract the data from a database, and use an external tool such as MATLAB, or else use awkward, existing, within-the-database linear algebra facilities. In this thesis, I will focus on my four main contributions: (1) I add vector and matrix types to SQL, the most commonly-used database programming language; (2) I design a few simple SQL language extensions to accommodate vectors and matrices; (3) I consider the problem of making vector and matrix operations efficient via integration with the database query optimizer; and (4) I conduct some experiments to show the efficacy of my language extensions.

Keyword: Vector/Matrix, Linear Algebra, RDBMS, SQL

Acknowledgements

Many thanks go to my advisor, Dr. Chris Jermaine, for your patient guide and instruction. Thank my colleagues for your collaboration for this project. Thank my families for your great support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background and Motivation

Relational database management systems (RDBMS) [1] [2] are widely used and they dominate the current database market.

Generally, a relational database is made up by a collection of *tables*, or *relations*, which describe the relationship between the *attributes* stored in these tables. A table consists of columns and rows. A column represents an attribute of the table, and each attribute is identified by its name and type. A row is a structured combination of attributes, and it represents as a record in the table. The intersection of columns and rows is a cell of the table, and each cell stores the value of an attribute.

SQL (Structured Query Language) [3] is a standard programming language for accessing and managing data in a RDBMS, and currently, it is the most widely used database language [4]. The following code shows how to create a table in a RDBMS by using SQL:

```
CREATE TABLE example_table
(
```

```
        attribute1 integer,
        attribute2 double
);
```

In this example, A table *example_table* is created. It has two attributes: *attribute1*, with type *integer*; and *attributte2*, with type *double*. We can insert data into it in the following format:

```
0 | 0.0 |
1 | 10.0 |
2 | 20.0 |
```

The data contains three records, and one line represents one record. In this example, the attribute *attributte1* has values 0, 1, and 2, while the attribute *attributte2* has values 0.0, 10.0 and 20.0.

However, such a representation does not easily express more complex data structures, such as vectors and matrices. Here, a vector is defined as to a one-dimensional array (an array is an ordered collection of elements, with indexes) [5], and a matrix is defined as a rectangular array with rows and columns [6]. Vectors and matrices are widely used in mathematical computations and scientific applications in many domains, including astronomy, physics, biology and computer science.

It is possible to represent vectors and matrices in a database with the following SQL code, though as we will describe subsequently, there are some problems with this representation:

```
CREATE TABLE vector
(
        pos integer,
```

```
        value double
);
CREATE TABLE matrix
(
        row_index integer,
        col_index integer,
        value double
);
```

In the code above, two tables are created, *vector* and *matrix*, which are used to store a vector and a matrix, respectively. Each element of a vector is depicted by two attributes, *pos* and *value*. *pos* states one element's position in the vector, and *value* indicates this element's value. Each element of a matrix is described by three attributes, *row_index*, *col_index* and *value*. *row_index* and *col_index* tell the position of this element in the matrix. *row_index* indicates the index of the row where this element sits, while *col_index* indicates the index of the column that this element belongs to. *value* stores the value of this element.

There are two obvious problems with this representation, which are illustrated by using the multiplication of two matrices as an example. Two tables *m1* and *m2* are created to hold the two matrices for multiplication. Then, these two matrices can be multiplied with the following code:

```
SELECT m1.row_index, m2.col_index, sum(m1.value * m2.value)
FROM m1, m2
WHERE m1.col_index = m2.row_index
GROUP BY m1.row_index, m2.col_index;
```

The first problem with this code is that it is overly complicated, given that we just wish to perform a simple matrix multiplication operation. Programmers are used to writing this as simply as `m1 * m2` in a language such as Matlab. The problem is even worse when performing more complicated linear algebra.

The second problem is related to performance. When the table *m1* and *m2* become very large, the running time of the code will grow quickly since it requires a join between two large tables. Such a tuple-based vector/matrix representation will typically result in performance that is much slower than what could be obtained by using a typical vector/matrix programming system.

1.2 My Solution and Contributions

From the description in section 1.1, it is evident that there are two main challenges when handling vectors and matrices in a relational database system: How to represent them simply, with a low burden for the programmers? How to run linear algebra and other vector/matrix related operations efficiently?

In this thesis, I introduce *vector* and *matrix* data types as basic attribute types. To accommodate these two new data types, I design a new type system. Also, My colleagues and I have implemented a few powerful operations over vectors and matrices. Such operations are wrapped in aggregate functions, VG functions and built-in functions, which have been designed and tuned specially for vector and matrix manipulations.

Adding vector and matrix attributes has also influenced the query optimizer of the system we implemented. Specifically, the semantic information concerning vector/matrix-related functions is passed to the optimizer. Thus, the optimizer can utilize that information to calculate the output size of each operation, and produce an accurate estimate of the cost of each operation. The optimizer may generate a more

efficient execution plan with the extra cost information, compared to an optimizer without that semantic information.

My colleagues and I have implemented my ideas on top of SimSQL, a scalable, parallel and analytic database system developed by our group [7] [8]. I evaluate my ideas using some popular machine learning algorithms, including Gaussian Mixture Models [9], Bayesian Lasso [10], Latent Dirichlet Allocation [11], and Hidden Markov Models [12]. The experiment results show that the new SimSQL with vector/matrix support has superior performance than the old one without such support.

1.3 Roadmap of This Thesis

This thesis is organized as follows:

Chapter 1 gives some background on relational database systems, and provides the motivation for adding vector and matrix support to an RDBMS. Some examples are given to show how current RDBMS technology can be used to define and store vectors and matrices. Then my proposed alternative solution is described.

Chapter 2 discusses some related work on this topic. More specifically, this chapter describes how the SQL standard regulates the definition and usage of arrays. Then it discusses how commercial relational database systems support vectors and matrices. Next, it considers another kind of database system, array databases, and analyzes how such systems manage vectors and matrices. Lastly, it discusses how user-defined functions can be used to support vectors and matrices.

Chapter 3 describes my ideas for adding vector and matrix data types to a RDBMS in depth. The contents include the definition of vector and matrix data types, some basic functions on vectors and matrices, and how to store/extract vector and matrix data into/from the system. At the end of this chapter, a concrete example, a *Gaussian Mixture Model*, is introduced, and the vector/matrix based implementation of this

model is described.

Chapter 4 considers how the system itself should change to process vectors and matrices. It describes the functions we wrote for users to manipulate vectors and matrices. These functions fall into three categories: aggregate functions, VG functions and built-in functions. The programming interfaces of these functions are given. Moreover, the new type system of accepting vector and matrix type is also described, and how such system influences the query optimizer is explained. The last section of this chapter explicates how to transfer vector and matrix data between disk and memory, as well as transferring them across the network.

Chapter 5 describes the experiments I ran to test our implementation. I implemented each model in two ways: a pure-relational based way and a vector/matrix based way. Then I ran each on a five-machine cluster to obtain and compare their running time. The results show that the vector/matrix based implementations are more succinct, and they greatly outperform the pure-relational ones in most experiments.

Chapter 6 summarizes this thesis, and provides some possible future directions for this work.

Chapter 2

Related Work

This chapter discusses the efforts people have made to facilitate vector and matrix representation and operation in DBMSs. Because a vector can be regarded as a one-dimensional array, and a matrix can be regarded as a two-dimensional array, sometimes I will use *array* as the subject of the discussion, instead of mentioning vectors and matrices explicitly.

2.1 Vector and Matrix in SQL Standard

SQL is the most commonly used database language. It became a standard of the International Organization for Standardization (ISO) in 1987 (ISO/IEC 9075-1:2008 [13]). After that, this SQL standard and its following revisions are widely accepted and adopted by the major database communities, and most database systems conform to those standards. The standard SQL:1999 (ISO/IEC 9075-2:1999 [14]) introduced the *array* data type to SQL standard for the first time. According to this standard, an array was defined as a collection of ordered elements with a declared maximum cardinality [14]. All elements of an array shared the same data type, which could be almost any basic data type defined in the standard. Furthermore, the standard

also defined some basic operations on arrays. Such operations included comparison, reference, and concatenation. According to SQL:1999, a vector can be simply defined as an array. However, SQL:1999 did not support multi-dimensional array or nested array. Thus, the representation of a matrix may not be straightforward, i.e., a matrix can not be defined as a two-dimensional array, or an array of arrays. The next several releases of SQL standard, such as SQL:2003 (ISO/IEC 9075-2:2003 [15]), SQL:2008 (ISO/IEC 9075-2:2008 [16]), and SQL:2011 (ISO/IEC 9075-2:2011 [17]), made minor modifications to the array data type. But the multi-dimensional array is still undefined. However, in 2014, there are reports that ISO intended to include multi-dimensional array into the SQL standard in its next release (Leopold [18] Chirgwin [19]). If this happens, then a matrix could be simply defined as a two-dimensional array.

According to the SQL standard, there are three ways to store a vector in a table in a RDBMS. The first way is to store every element of the vector as an attribute of the table. For example, the following code shows how to store the vector $[1.0, 2.0, 3.0]$ into a table in this way:

```
CREATE TABLE vector1
(
        element1 double,
        element2 double,
        element3 double
);
```

This table *vector1* has three attributes, *element1*, *element2* and *element3*, which hold the vector elements 1.0, 2.0 and 3.0 successively. As the attributes of a table are unordered, some rules should be given to point out which attribute stores which

element. One method is to utilize the name of an attribute to indicate the position of the element. For example, in table *vector1*, a combination of the word "element" and a position index is used to name an attribute. As the example shows, the attribute *element1* stores the first element of the vector. However, it is easy to foresee that such naming process will become tedious when there are many elements in the vector. Also, the number of attributes will easily become large as well. Thus, in practice, such representation is rarely employed. Instead, the following one is used more often:

```
CREATE TABLE vector2
(
        pos integer,
        value double
);
```

Different from the table *vector1*, the table *vector2* has two attributes. The attribute *pos* stores the position of an element, and the attribute *value* stores the value of that element. Compared with the first method, this representation uses a separate attribute to indicate the position of an element. It does not need a method for naming the element attributes. Also, the number of attributes keeps constant when the number of vector elements increases. But this representation may be improved in the case of a dense vector. Hence, the third representation is proposed:

```
CREATE TABLE vector3
(
        v double array[3]
);
```

The table *vector3* only has one attribute *v*. This attribute has *array* data type. The

number in the bracket indicates that the size of this array is 3. Each element of this array has *double* data type. The attribute v stores the whole vector directly. It is clear that compared with the previous two methods, this representation is more succinct and intuitive for dense data.

Similarly, following the SQL standard, there are also three ways to define a matrix. The first way is to list out all elements of the matrix as attributes, similar to the first representation of a vector. Assuming we want to store a matrix $\begin{bmatrix} 1.0 & 2.0 \\ 3.0 & 4.0 \end{bmatrix}$ to a table. The table can be created in the following way:

```
CREATE TABLE matrix1
(
        element11 double,
        element12 double,
        element21 double,
        element22 double
);
```

The attributes *element11*, *element12*, *element21*, and *element22* store the values 1.0, 2.0, 3.0 and 4.0 respectively. Again, the names of the attributes need to be designed carefully, because those names are used to indicate which elements they are referring to. In this example, an attribute's name is given as a combination of the word "element", the row index of the element and column index of the element. For example, the attribute *element11* stores the element in the first row and first column of the matrix. To avoid such inconvenient naming process, the following representation is proposed:

```
CREATE TABLE matrix2
(
```

```
        row_index integer,
        col_index integer,
        value double
);
```

The table *matrix2* uses three attributes to describe each element of a matrix. The attributes *row_index* and *col_index* indicate which row and column the element sits in, and the attribute *value* stores the value of that element. With this method, a fixed number of attributes are used to represent a matrix, even if the number of matrix elements increases. But there is still much repeated information (such as *row_index* and *col_index*) stored, which may be unnecessary in many applications. The array data type can be used to alleviate this:

```
CREATE TABLE matrix3
(
        row_index integer,
        row double array[2]
);
```

Similarly, we can also store a matrix in a column-based way. That is, we use an array attribute to store each column of a matrix, and use another attribute to indicate the index of each column. The example code is given below:

```
CREATE TABLE matrix3
(
        col_index integer,
        column double array[2]
);
```

2.2 Vector and Matrix in RDBMS

Real RDBMSs usually provide their own ways to define and manipulate vectors and matrices. In this section, I introduce several widely-used relational database systems, including MySQL, Microsoft SQL Server, Oracle Database, IBM DB2 and PostgreSQL. I will show how these RDBMSs handle vectors and matrices in practice. In general, these systems comply with the core parts of the SQL standard. But they also provide or support unique operations and functions for vectors and matrices. I will focus on linear algebra operations, as they are important for data analytics, and they are widely used in machine learning and data mining algorithms.

Although array data type has been introduced to SQL standard since SQL:1999, some database systems still do not support array data type. Such systems include MySQL (Widenius and Axmark [20] [21]), Microsoft SQL Sever [22], SQLite [23], etc. If a user of these systems wants to store vector or matrix data, he or she needs to store it as (pos, value) pairs, or (row_index, col_index, value) triples. An alternative way to store arrays in these systems is to use the BLOB (Binary Large OBject) [24] data type. A BLOB attribute can hold a large object by representing it as a binary string. Hence, to store an array, users can firstly concatenate its all elements together to form a string, then convert this string to a binary representation, and finally, store this binary string as a BLOB attribute. However, this method to store an array is not widely adopted. First, not all database systems support BLOB data type. Second, the conversion from arrays to binary strings, or vice versa, is inconvenient.

To facilitate the usage of arrays in the systems without the array data type, some libraries are implemented, and they provide an easy-to-use interface to manipulate arrays in those systems. For example, Dobos, et al. [25] proposed the *SQL Server*

Array Library, a library that added array support to Microsoft SQL Server. The *array* data in this library is stored as binary data underneath. The following code shows how to use this library to define a vector as a one-dimensional array:

```
DECLARE @a varbinary(100) = SqlArray.RealArray.Vector_3(1.0, 2.0, 3.0)
```

Here, a vector a is declared as an array with three elements, 1.0, 2.0 and 3.0, and each element has *real* data type. A matrix can be defined in a similar way:

```
DECLARE @b varbinary(100) = SqlArray.FloatArray.Matrix_2(1.0, 2.0, 3.0, 4.0)
```

This code declares a 2×2 matrix b with float type elements.

Besides the declaration, the SQL Server Array Library also provides some basic operations on vectors and matrices. However, because the array data will be converted and stored as binary data ultimately, the SQL server underneath is not aware of the existence of vectors and matrices. Thus, there is no chance for the system to conduct specific optimizations for those vector/matrix related operations.

While some systems still do not support array data type, more and more database systems are starting to integrate it into their basic data types to comply with the SQL standard. Such systems include Oracle database ([26] Greenwald et al. [27]), IBM DB2 ([28] Chong [29]), PostgreSQL ([30] Stonebraker and Rowe [31]), etc. Usually, these systems have their own definition and usage for array data type, and they also provide some operations and functions on array data.

Oracle database is the #1 DBMS in terms of the market share worldwide as of 2013 [32]. In Oracle database, the array data type is named as *VARRAY* [33]. A VARRAY is defined as a one dimensional array with fixed number of elements. Users can utilize VARRAY to declare a vector data type with the following code:

```
CREATE TYPE Vector IS VARRAY(10) OF DOUBLE;
```

The *Vector* type has 10 elements, and the type of each element is double. With this Vector type, a Matrix data type can be defined as a VARRAY of Vectors:

```
CREATE TYPE Matrix IS VARRAY(100) OF Vector;
```

This *Matrix* type represents a 100×10 matrix. These newly created Vector and Matrix types can be used to define an attribute's type in a table:

```
CREATE TABLE Student_Record
(
   name VARCHAR(30),
   id VARCHAR(10),
   grade Vector
);
```

Additionally, Oracle also provides a package, *UTL_NLA* [34], to support BLAS [35] and LAPACK [36] operations. Such operations include arithmetic operations on vectors and matrices, linear algebra operations, as well as solutions for linear equations. These operations can be used by calling the functions in the UTL_NLA package. However, those functions usually have many parameters, making them difficult for a beginner to learn and use. The following code shows how to conduct matrix multiplication with the function in UTL_NLA package:

```
m1 utl_nla_array_dbl := utl_nla_array_dbl(1, 2, 3, 4, 5, 6);
m2 utl_nla_array_dbl := utl_nla_array_dbl(1.1, 2.2, 3.3, 4.4, 5.5, 6.6);
res utl_nla_array_dbl := utl_nla_array_dbl(0, 0, 0, 0, 0, 0);
utl_nla.blas_gemm(
      transa => 'N, transb => 'N',
```

```
m => 3, n => 3, k => 3,
alpha => 1.0,
a => m1,
lda => 3,
b => m2,
ldb => 2,
beta => 0.0,
c => res,
ldc => 3,
pack => R);
```

In the code above, the variables *m1* and *m2* contain the two matrices for multiplication, and the variable *res* is used to store the result. The function *utl_nla.blas_gemm* does the actual multiplication work. Note that in this example, more than ten parameters are needed to call the function *utl_nla.blas_gemm*, even if it just performs a simple multiplication operation. It is easy to foresee that if users want to achieve more complex operations, the code will become complicated quickly, and the program will be hard to read and understand.

IBM is another important database vendor. The IBM researcher, Edgar Codd, proposed the relational model for the first time in 1970 (Codd [1]). The database product of IBM, *IBM DB2*, includes array as a primary data type. The array data type is usually used to define user-defined data types. Then those new types can be then employed by SQL procedures. The following code is an example:

```
CREATE TYPE dateArray AS DATE ARRAY[100];
CREATE PROCEDURE getWeekends(in myDates dateArray, out weekends dateArray)
```

The last relational database system discussed in this chapter is PostgreSQL. PostgreSQL is a powerful, open source, object-relational database system. It stems from POSTGRES, a project developed by Michael Stonebraker with a research group at UC Berkeley (Stonebraker and Rowe [37] Stonebraker and Kemnitz [38]). Compared with Oracle database and IBM DB2, PostgreSQL provides a more natural way to store and use array data. The following code is an example:

```
CREATE TABLE professor (
        name text,
        course_per_semester integer[3],
        schedule text[][]
);
```

In this example, a table *professor* is created with three attributes, *name*, *course_per_semester* and *schedule*. The attribute *name* has type *text*, which stores a string with any length. The attribute *course_per_semester* indicates how many courses the professor teaches in each semester, assume there are three semesters. So its type is a vector of length 3. The last attribute *schedule* gives the professor's daily arrangement. It is declared as a two-dimensional array, or a matrix, with text type elements that describe the professor's activities. The first dimension of the schedule can be time slots in a day, and the second dimension can be the days in a week.

The representation of vectors and matrices in PostgreSQL is closer to their mathematical forms, making it more intuitive to use. First, the keyword *array* is not needed when defining arrays. Only brackets and element type are required. Second, PostgreSQL supports multi-dimensional arrays. Thus, a matrix type attribute can be directly defined as a two-dimensional array, which is close to the representation that people commonly use.

PostgreSQL also provides some primitive operations on vectors and matrices, including comparison, modification, and concatenation. However, these operations are not powerful enough for complex operations, such as linear algebra. To solve this problem, some external libraries for PostgreSQL are implemented, and they provide a good support for linear algebra and other complicated mathematical computations. MADlib (Hellerstein et al. [39]) is a popular example. It is an open source library, and focuses on scalable in-database analytics and machine learning. The operations supported by MADlib library include arithmetic operations (add, subtract, multiply and divide), statistical operations (mean, square, square root, etc.), linear algebra operations (vector-matrix multiplication, matrix-matrix multiplication, etc.), and other mathematical calculations that are frequently used in machine learning.

The MADlib library is a helpful supplement to PostgreSQL, but there is still space for improvement. First, to use MADlib and PostgreSQL together, users have to install and learn both of them, and make sure they can work together properly. Second, because MADlib is an external library, the PostgreSQL system is not aware of what MADlib is doing. Thus, PostgreSQL may not have a chance to conduct specific optimization for MADlib operations, where many optimization opportunities may exist (Jarke and Koch [40]).

2.3 Vector and Matrix in Array Database

To facilitate the manipulation for array data, a new type of database system, the *array database* (Baumann [41] Baumann [42]), is proposed, which is designed specifically to handle array data. In an array DBMS, a multi-dimensional array is usually treated as the basic processing unit, and the system provides a set of operations to manipulate arrays. Usually, each array DBMS has its unique ways to create, store, modify and retrieve arrays. Because such systems often have special design and optimization

techniques to store and process array data, they tend to have a better performance and scalability when they execute operations on arrays, compared with traditional relational DBMSs. In general, array DBMSs are mainly used to handle domain-specific data, such as raster data in computer graphics, and multi-dimensional image data in geoscience (Gutierrez and Baumann [43]). In this section, I consider three array DBMS, *Rasdaman* ([44] Baumann et al. [45]), *SciDB* (Stonebraker et al. [46] Brown [47]) and *MonetDB* ([48] Idreos et al. [49]).

Rasdaman is regarded as the first fully implemented array database system [44]. It is developed based on Baumann's array algebra (Baumann [42]). In Rasdaman, an array is defined as a combination of *extent* and *cells*. The extent indicates the domain or dimension range of an array, and the cells contains the the array's elements. Users can define their own array type by specifying its dimension boundaries and the cell type. Here is an example:

```
TYPEDEF MARRAY <char, [ 1:200, 1:200 ]> GreyImage
```

The code defines an array type called *GreyImage*. It can be used to create a *collection*, which is similar to the concept of a table in a RDBMS:

```
TYPEDEF SET<GreyImage> GreySet;
CREATE COLLECTION GreySet mr
```

Rasdaman also provides its own query language, *rasql*, to help users to operate on arrays. This language is very similar to standard SQL. The following rasql code shows how to fetch cells from the collection *mr*:

```
SELECT mr[100:150,40:80] / 2
FROM mr
WHERE some_cells( mr[120:160, 55:75] > 250 )
```

The example above shows how to define and use vectors and matrices in Rasdaman. A vector is defined as a one-dimensional array with a dimension range, and a matrix is defined as a two-dimensional array with two dimension ranges. The example also shows some basic operations on arrays. However, although these operations are easy to use, they are not powerful enough to fulfill complex analytic tasks. For example, linear algebra operations are not well supported in Rasdaman.

To fill the need for complicated analytics, another array DBMS, *SciDB*, was developed. SciDB targets on dealing with rich, high dimensional data and in-database analytics on such data. The main features of this system include scalable mathematic operation, data versioning and provenance, uncertain data support, and efficient storage [50]. The linear algebra operations supported by SciDB include matrix inversion, matrix multiplication and matrix transposition. The following code shows how to define and operate matrices in SciDB:

```
CREATE ARRAY M < A1:int32, B1:double > [ I=0:99999,1000,0, J=0:5999,100,0 ]
Multiply (project (M, B1), transpose(project (M, B1)))
```

The code above creates a matrix *M*, and conducts a matrix-wise multiplication on it. Each element of this matrix has two values, *A1* and *B1*, and the dimension range of this matrix is given in the brackets. The function *Multiply* conducts a multiplication between this matrix and its transpose matrix, using *B1* as the element value. The example shows that the declaration and operation of vectors and matrices in SciDB are intuitive, and this system has provided some functions for linear algebra operations such as matrix transpose. However, similar to other systems, the semantics of those functions is not passed to the query optimizer of this system. Without such semantic information, the optimizer cannot have an accurate estimation of the output size of

those functions, so that it may not generate the optimal plan for the query.

Column-oriented database systems (Abadi et al. [51]) stir much interest in recent years, and they have utilized many novel and advanced techniques for data storage (Stonebraker et al. [52]). These techniques can help array databases to store arrays in a more efficient way. A pioneer in this field is MonetDB, which is an open-source column-oriented DBMS. One of its query languages, *SciQL* (Kersten et al. [53]), integrates array support into SQL language, and gives MonetDB the capability of functioning like an array database. However, similar to previous systems, the analytic functions for arrays are missing from MonetDB. The following code shows how to define and use arrays in MonetDB, written in SciQL.

```
CREATE ARRAY matrix (
        x integer DIMENSION[0:4],
        y integer DIMENSION[0:4],
        value float DEFAULT 0
);
SELECT sum(value)
FROM matrix
WHERE (x + y) % 2 = 0
```

2.4 Vector and Matrix in User-Defined Functions

The previous two sections give two ways to handle arrays in database systems. One way is to introduce array as a new data type. The other way is to treat an array as an independent processing unit. However, both solutions exist deficiencies. On one hand, using array simply as a basic data type may unable to support complicated array-related operations. On the other hand, using a new array database system

may require extra efforts from users (set up the system, learn how to use it, etc.), and the data may need moving from a RDBMS to an array DBMS. Based on this inconvenience, a third way to support arrays is proposed: user-defined functions (UDFs) [54] for arrays. More specifically, a list of UDFs are implemented as external libraries, and users can operate on arrays by adding the libraries to their system and calling those functions. Compared with the previous two methods, the user-defined functions usually support more complicated operations, and they are more flexible and extensible. MADlib can be regarded as an example of a UDF library, and more examples will be discussed in the next several paragraphs.

Carlos Ordonez and his colleagues have conducted rigorous work on user-defined functions for DBMS [54] [55] [56]. In [54], they described the idea of using UDFs to provide basic vector and matrix operators for a RDBMS, such as *sum*, *dot product* and *matrix multiplication*. The UDFs were written in C language, and they could be used in SQL queries. By implementing his ideas on a parallel DBMS called *Teradata*, Ordonez claimed that the code written in UDFs could be faster than the code written in naive SQL for specific vector/matrix related operations [54]. Furthermore, Ordonez extended his work to more complex operations [55] [56], including linear correlation, linear regression, naive Bayes, clustering, etc. Nevertheless, Ordonez's ideas need to be verified and evaluated on more database platforms.

Kernert, Köhler and Lehner considered the problem of adding linear algebra to a column-oriented in-memory database system [57] [58]. They treated vectors and matrices as first class citizens in their system, and designed *manipulation primitives* and *linear algebra primitives* for vectors and matrices. These primitives were wrapped up as user-defined functions, and users could borrow them to write their own UDFs. Moreover, the sparse matrices in their system were stored in the CSR (compressed sparse row) format, and a unique *main-delta* architecture was employed. Because of

this, their system showed a good performance for dynamic matrix manipulation [58]. However, their UDFs did not exhibit a good integration with standard SQL language. Also, the focus of their work was the sparse matrix. Thus, some methodologies in their work might not apply to general usages.

2.5 Summary

In this chapter, I have reviewed how arrays, vectors, matrices, as well as their operations, are supported by modern database systems. The SQL standard defines array as an independent data type, and regulates a few very basic operations on arrays. A few RDBMSs conform to the array definition in SQL standard, while some RDBMSs such as PostgreSQL propose more intuitive ways to represent vectors and matrices. In general, traditional RDBMSs have limited support for vector/matrix related operations.

To explore a better way to store and handle array data, array DBMSs are proposed. These systems are designed specifically to manipulate arrays, and they provide a more natural way to represent and access arrays, compared with traditional RDBMSs. The operations provided by these systems are also more adequate. However, in practical applications, people often have their unique needs for operations, and a single system cannot satisfy all requirements. Thus, a bunch of user-defined functions are designed and developed. These functions can be used as external libraries for DBMSs, and endow more choices and flexibilities to manipulate vectors and matrices.

Nevertheless, there is still room for improvement. First, the linear algebra operations are still not well-supported. The functions for those operations are either too complex to use, or not integrated with the underneath system well. Second, almost all systems ignore the semantics of linear algebra operations when they generate the query plan. Thus, the costs of executing those operations may not be estimated cor-

rectly, and the system may lose some optimization opportunities when it optimizes the query plan.

Chapter 3

For SimSQL Users: Vector and Matrix Type

In this chapter, I will provide my definitions for vector and matrix data types in SimSQL. I will also discuss a few basic arithmetic operations and functions for vectors and matrices. After that, I will describe how to construct and deconstruct vectors and matrices in SimSQL. Finally, the Gaussian Mixture Model will be presented as a concrete example to illustrate the usage of vectors and matrices in the real scenario.

3.1 Vector and Matrix Type

In Chapter1, I have described the pure relational way to define and store vectors and matrices in a RDBMS:

```
create table vector_table
(
        pos integer,
        value double
);
create table matrix_table
(
```

```
        row_index integer,
        col_index integer,
        value double
);
```

The two attributes in the table *vector_table* describe each element of a vector. The attribute *pos* indicates an element's position, and the attribute *value* stores this element's value. Now assuming we want to store the vector $[1.0, 2.0, 3.0]$ into this table, then the data should be written in the following format:

```
0 | 1.0 |
1 | 2.0 |
2 | 3.0 |
```

The first line stores the first element, whose position index is 0 and value is 1.0. Then the second line stores the second element, and so on.

Similarly, the three attributes in the table *matrix_table* describe each element of a matrix. The attributes *row_index* and *col_index* indicate an element's row position and column position, and the attribute *value* stores this element's value. Now assuming we want to store the following matrix into the table *matrix_table*:

$$\begin{bmatrix} 1.0 & 2.0 & 3.0 \\ 4.0 & 5.0 & 6.0 \\ 7.0 & 8.0 & 9.0 \end{bmatrix}$$

Then the data should be written in the following format:

```
0 | 0 | 1.0 |
0 | 1 | 2.0 |
```

```
0 | 2 | 3.0 |
1 | 0 | 4.0 |
1 | 1 | 5.0 |
1 | 2 | 6.0 |
2 | 0 | 7.0 |
2 | 1 | 8.0 |
2 | 2 | 9.0 |
```

The first line represents the first element, which has value 1.0 and sits in the first row and first column of the matrix. The second line represents the second element, and so on.

Such relational representation for vectors and matrices may take much space if the vectors and matrices become very large. For example, it needs up to 10^6 (row_index, co_indexl, value) tuples to represent a 1000×1000 matrix. The operations on vectors and matrices represented as so many tuples may be slow as well. Hence, we need more efficient data structures to represent vectors and matrices in a relational database system.

My solution is to introduce two new attribute types into the system: the vector data type and the matrix data type. With these two new data types, the table *vector_table* and *matrix_table* can be redefined in the following way:

```
create table vector_table
(
        v vector[3]
);
create table matrix_table
(
```

```
        m matrix[3][3]
);
```

The new *vector_table* has one attribute v, instead of two attributes (*pos*, *value*). The attribute v has *vector* data type. The number `3` in the bracket indicates the length of this vector. So if the data to be stored in this attribute is not a vector, or the vector does not have three elements, the system will throw out an error. Also, in current SimSQL, each element of a vector has *double* data type.

The vector $[1.0, 2.0, 3.0]$ can be stored into the new *vector_table* with the following format:

```
[1.0, 2.0, 3.0] |
```

Now the data consists of one record instead of three records, and it is more readable.

The new *matrix_table* has one attribute m, instead of three attributes (*row_index*, *col_index*, *value*). The attribute m has *matrix* data type. The number `3` in the first bracket indicates the number of rows, and the number `3` in the second bracket indicates the number of columns. Thus, if the data to be stored in this attribute is not a matrix, or the matrix is not a 3×3 matrix, the system will throw out an error. Besides, each element of a matrix has *double* data type.

The matrix $\begin{bmatrix} 1.0 & 2.0 & 3.0 \\ 4.0 & 5.0 & 6.0 \\ 7.0 & 8.0 & 9.0 \end{bmatrix}$ can be stored into the new *matrix_table* in the following format:

```
[1.0, 2.0, 3.0] [4.0, 5.0, 6.0] [7.0, 8.0, 9.0] |
```

Now the matrix data consists of one record instead of nine records.

A vector or a matrix attribute can also be defined with unknown size. For instance,

a vector attribute can be created with type *vector[]*, which means that there is no specific length requirement for this vector attribute. Thus, a vector of any length can be used as the input data for this attribute. Similarly, a matrix attribute can be created with type *matrix[3][]* or *matrix[][]*. The empty bracket "[]" means there is no specific length requirement for this dimension. Thus, a matrix with any length in the empty bracket dimension can be used as the input data for this attribute.

3.1.1 Linear Regression Model

In this section, a very common model, the linear regression model (Draper et al. [59]), is introduced. This model can further illustrate the difference between the pure relational representation and my vector/matrix-based representation. In linear regression, a $\{response, regressor\}$ data set, represented by $\{y_i, \mathbf{x}_i\}, i = 1, \ldots, n$, is assumed to have this linear relationship:

$$y_i = \mathbf{x}_i^T \boldsymbol{\beta} + \epsilon_i \tag{3.1}$$

Formula 3.1 : *Linear Regression Model*

and the equivalent vector/matrix version is:

$$\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon} \tag{3.2}$$

Formula 3.2 : *Linear Regression Model Represented by Vectors and Matrices*

Here, $\mathbf{y}$ is called *response vector*, and $\mathbf{X}$ is called *regressor matrix*. They are

provided by the user, and are known data. $\boldsymbol{\beta}$ is the parameter vector, and $\boldsymbol{\epsilon}$ is the error vector. They are used to model the relationship between the responses and regressors, and they are usually unknown. Thus, the inference problem of the linear regression model is: given the data set $\{\mathbf{y}, \mathbf{X}\}$, how to estimate the value of the parameter vector $\boldsymbol{\beta}$, so that $\mathbf{X}$ and $\mathbf{y}$ can satisfy the linear relationship described by Formula 3.2. One of the most common estimation methods is the least-squares estimation (Sorenson [60]). According to this method, $\boldsymbol{\beta}$ can be estimated with the following formula:

$$\hat{\boldsymbol{\beta}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \tag{3.3}$$

Formula 3.3 : *Least Square Estimation for Parameter Vector* $\boldsymbol{\beta}$

In the next two parts, I will describe how to write code for linear regression estimation in a traditional relational database system, and how to write such code in SimSQL with vector and matrix support.

Linear Regression In a Traditional Database System

This section considers the implementation of linear regression in a pure relational way. First of all, two tables are created to hold the input data $\mathbf{X}$ and $\mathbf{y}$:

```
create table x
(
        row integer,
        col integer,
```

```
        value double
);
create table y
(
        pos integer,
        value double
);
```

To facilitate the calculation, the Formula 3.3 is re-written to divide the computation of $\hat{\boldsymbol{\beta}}$ into two parts:

$$\begin{aligned} \hat{\boldsymbol{\beta}} &= \mathbf{G}\mathbf{w} \\ \mathbf{G} &= (\mathbf{X}^T\mathbf{X})^{-1} \\ \mathbf{w} &= \mathbf{X}^T\mathbf{y} \end{aligned} \tag{3.4}$$

Formula 3.4 : *Least Square Estimation for Parameter Vector* $\boldsymbol{\beta}$: *Another Way*

Then, the formula to compute $\hat{\boldsymbol{\beta}}$ in a tuple-based way can be written as follows ($\mathbf{X}$ is a $n \times m$ matrix):

$$\begin{aligned}\hat{\boldsymbol{\beta}}_i &= \sum_j^m \mathbf{G}_{j,i}\mathbf{w}_j \\ \mathbf{G}_{i,j} &= (\sum_k^n \mathbf{X}_{k,i}\mathbf{X}_{k,j})^{-1} \\ \mathbf{w}_i &= \sum_j^n \mathbf{X}_{j,i}\mathbf{y}_j\end{aligned} \tag{3.5}$$

Formula 3.5 : *Least Square Estimation for* $\boldsymbol{\beta}$: *Tuple Based*

With Formula 3.5, the SQL code to calculate $\hat{\boldsymbol{\beta}}$ is straightforward. First, the matrix $\mathbf{G}$ can be computed with the following code:

```
CREATE TABLE G(row, col, value) AS
  WITH mi AS MatrixInverse(
                SELECT x1.col, x2.col, SUM(x1.value * x2.value)
                FROM x as x1, x as x2
                WHERE x1.row = x2.row
                GROUP BY x1.col, x2.col
          )
  SELECT mi.rv_out_i, mi.rv_out_j, mi.outValue
  FROM mi;
```

In the code above, the table $\mathbf{X}$ firstly conducts a self-join, conditioning on the row index k. Then, the multiplication $\mathbf{X}_{k,i}\mathbf{X}_{k,j}$ is summed up, grouped by the index pair (i, j). The result of this calculation is $(\mathbf{X}^T\mathbf{X})_{i,j}$. Lastly, a VG function (one of SimSQL's user-defined function interfaces) *MatrixInverse* is applied on $\mathbf{X}^T\mathbf{X}$ to get the final result $(\mathbf{X}^T\mathbf{X})^{-1}$ (i.e., the matrix $\mathbf{G}$). The input of the function *MatrixInverse* is a triple *(row, col, value)*, which represents the elements of a matrix, and the output

is the inverse of the input matrix, with *rv_out_i* as the row index, *rv_out_j* as the column index, and *outValue* as the element value in position *(rv_out_i, rv_out_j)*. Note that VG functions are a set of pre-defined functions provided by SimSQL. Usually they are used to handle computations for stochastic process. More details of VG functions will be discussed later.

The next step is to calculate the vector $\mathbf{w}$. The code is given below:

```
CREATE MATERIALIZED VIEW w(pos, value) as
        SELECT x.col, SUM(x.value * y.value)
        FROM x, y
        WHERE x.row = y.pos
        GROUP BY x.col;
```

In the code above, the table $\mathbf{X}$ and the table $\mathbf{y}$ is joined together, conditioning on the row index j. Then, the multiplication $\mathbf{X}_{j,i}\mathbf{y}_j$ is summed up, grouped by the column index i. The results of this query describe the elements of the vector $\mathbf{w}$. The column index *x.col* stores the position of each element, and the sum result is the value of each element.

With the matrix $\mathbf{G}$ and the vector $\mathbf{w}$, the calculation of $\hat{\boldsymbol{\beta}}$ is simple:

```
SELECT G.col, SUM(G.value * w.value)
      FROM G, w
      WHERE G.row = w.pos
      GROUP BY G.col;
```

Linear Regression In SimSQL with Vector/Matrix Support

This section describes how to estimate the linear regression parameters with vector and matrix support. Similar to the last section, the first step is to create two tables to store the input data $\mathbf{X}$ and $\mathbf{y}$:

```
CREATE TABLE x
(
           pos integer,
           value vector[]
);
CREATE TABLE y
(
           pos integer,
           value double
);
```

The regressor matrix $\mathbf{X}$ is stored in the table x in a row-based way. That is, this first attribute *pos* holds the position of a row, and the second attribute *value*, which is a vector, stores the values of a row. Note that compared with a traditional relational database, the matrix $\mathbf{X}$ in SimSQL is stored with fewer data. The way to store the response vector $\mathbf{y}$ is the same as a traditional database, which uses *(pos, value)* pair to represent each element of the vector $\mathbf{y}$.

With the table x and y at hand, the parameter $\hat{\boldsymbol{\beta}}$ can be computed with the following code, using Formula 3.4 as a guide:

```
SELECT
matrix_vector_multiply(
            matrix_inverse(
```

```
                    sum(outer_product(X.value, X.value))),
            sum(X.value * y.value)
)
FROM x as X, y
WHERE X.pos = y.pos;
```

In the code above, firstly, an outer product is applied on *X.value* and itself. Then, the result of this outer product is summed up to get the matrix $\mathbf{X}^T\mathbf{X}$. Further, the inverse of this matrix is computed by a built-in function *matrix_inverse*. Meanwhile, the multiplication of *X.value* and *y.value* is summed up, with the same value of *pos*, which comprises the vector $\mathbf{X}^T\mathbf{y}$. Lastly, a matrix-vector multiplication is applied on $(\mathbf{X}^T\mathbf{X})^{-1}$ and $\mathbf{X}^T\mathbf{y}$, and the result is just the value of the vector $\hat{\boldsymbol{\beta}}$. Compared with the method introduced in the previous section, the code here is simpler to write and easier to understand.

In addition, the code above has called three functions: *outer_product*, which conducts an outer product between two vectors; *matrix_inverse*, which computes the inverse of the input matrix; and *matrix_vector_multiply*, which does a multiplication between a matrix and a vector. These functions are built-in functions which we have implemented for SimSQL users. Users can also write and use their own built-in functions, following the specific rules. More contents about built-in functions will be discussed in the following sections and chapters.

The linear regression example shows that compared with traditional database systems, the data representation and calculation in a system with vector and matrix support are more succinct. For example, by representing the matrix $\mathbf{X}$ in a vector-based way, we can avoid storing the *(row, col)* pairs to indicate the positions of elements, while such pairs are usually duplicated and useless. Moreover, using vectors

and matrices instead of relational tuples also reduces costly *join* between tables, and avoids moving large numbers of tuples around the system. In practice, these advantages can provide prominent performance improvement for SimSQL, compared with traditional relational database systems.

3.2 Basic Operations and Functions for Vectors and Matrices

This section discusses some basic operations and functions for vector and matrix manipulation.

3.2.1 Basic Operations

SimSQL supports simple arithmetic operations on vectors and matrices. These operations include *add*, *subtract*, *multiply* and *divide*. Their usage is straightforward. Suppose two vector attributes $\mathbf{v}_1$ and $\mathbf{v}_2$ are stored in a table *vectors*:

```
create table vectors
(
        v1 vector[],
        v2 vector[]
);
```

Then, the following code calculates $\mathbf{v}_1 + \mathbf{v}_2$, $\mathbf{v}_1 - \mathbf{v}_2$, $\mathbf{v}_1 * \mathbf{v}_2$, and $\mathbf{v}_1 / \mathbf{v}_2$, respectively:

```
select v1 + v2, v1 - v2, v1 * v2, v1 / v2
from vectors;
```

These operations can be applied on matrices in the similar way.

Besides, the equality check for two vectors or two matrices is also supported. If two vectors/matrices have the same value in every entry, then these two vectors/matrices are *equal*. Otherwise, they are *not equal*.

Note that all arithmetic operations and logical operations must be conducted on two vectors with the same length, or on two matrices with the same dimensions. Otherwise, the system will report an error.

3.2.2 Functions

Sometimes, simple arithmetic operations are not powerful enough to fulfill our tasks. For example, we cannot calculate the Gram matrix $\mathbf{G}$ with the operations described in section 3.2.1. Therefore, more powerful tools are needed. We propose **functions** to support more complex operations. There are three types of functions in SimSQL: aggregate functions, Variable Generation functions (VG functions), and built-in functions. The aggregate functions are used to apply specific calculation on a group of records. The VG functions are used to conduct stochastic analysis on uncertain data. And lastly, the built-in functions are mainly designed for linear algebra operations.

I will use Gram matrix calculation as an example to show how to employ those functions in SimSQL. Given a set of vectors $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$, the Gram matrix of these vectors can be defined as a matrix $\mathbf{G}$, whose element $\mathbf{G}_{ij}$ is calculated as:

$$\mathbf{G}_{ij} = \mathbf{x}_i^T \mathbf{x}_j \tag{3.6}$$

Formula 3.6 : *Gram matrix calculation*

If we define a matrix $\mathbf{X}$ as $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n]$, then the Gram matrix G can be

calculated in a matrix-based version:

$$\mathbf{G} = \mathbf{X}^T\mathbf{X} \tag{3.7}$$

Formula 3.7 : *Gram matrix calculation in matrix format*

Assuming we choose Formula 3.7 to calculate the Gram matrix, and we use a table *data* to store the matrix $\mathbf{X}$:

```
create table data
(
        x matrix[][]
);
```

Then, the SQL query to calculate the Gram matrix $\mathbf{G}$ from $\mathbf{X}$ can be written as follows:

```
select matrix_multiply(trans_matrix(x), x)
from data;
```

In the SimSQL code above, two built-in functions are employed: *trans_matrix* and *matrix_multiply*. The function *trans_matrix* is applied to get $\mathbf{X}^T$, which is the transpose matrix of $\mathbf{X}$. And the function *matrix_multiply* is applied on $\mathbf{X}^T$ and $\mathbf{X}$ to get their multiplication $\mathbf{X}^T\mathbf{X}$, and the result of this multiplication is just the Gram matrix $\mathbf{G}$. More examples of using functions with vectors and matrices will be presented in the following sections and chapters.

3.3 Construction and Deconstriuction for Vectors and Matrices

In section 3.2, I have talked about some basic operations and functions on vectors and matrices. However, there are two important operations I have not covered yet: the *construction* and *deconstruction* for vectors and matrices. Overall, there are two types of such construction/deconstruction: construct a vector/matrix from data; construct/deconstruct a vector/matrix by calling functions.

3.3.1 Vectors and Matrices Construction from Data

If a table has vector and matrix attributes, then vectors and matrices can be constructed by loading the data into these attributes. For example, assuming a table *vector_matrix* has a vector attribute v and a matrix attribute m:

```
create table vector_matrix
(
        v vector[3],
        m matrix[2][3]
);
```

Then, a data file *data.tbl* for this table can be written as follows:

```
[1, 2, 3] | [1.1, 1.2, 1.3][2.1, 2.2, 2.3] |
[4, 5, 6] | [1.4, 1.5, 1.6][2.4, 2.5, 2.6] |
[7, 8, 9] | [1.7, 1.8, 1.9][2.7, 2.8, 2.9] |
```

In the data file *data.tbl*, each line represents a record. In each record, the data before the first | is for the vector v (such as "$[1, 2, 3]$"), and the data before the second

| is for the matrix m (such as "$[1.1, 1.2, 1.3][2.1, 2.2, 2.3]$"). With this data file, the data can be loaded into the table *vector_matrix* with the following command:

```
load vector_matrix from data.tbl;
```

And finally we obtain three vectors and matrices from the data.

3.3.2 Vectors and Matrices Construction/Deconstruction from Functions

This section discusses how to construct and deconstruct vectors and matrices by calling functions, including how to construct a vector from scalars, how to construct a matrix from vectors, how to deconstruct a vector into scalars and how to deconstruct a matrix into vectors. In the end this section, the Gram matrix calculation is revisited, and it is computed with the Formula 3.6.

(1) Construct a Vector from Scalars

Assuming a vector has already been stored in a pure relational way (As discussed in section 3.1):

```
create table vector_table
(
        pos integer,
        value double
);
```

and its data is:

```
0 | 1.0 |
```

```
1 | 2.0 |
2 | 3.0 |
```

How might one transfer this representation to the vector type based representation? Or, more specifically, given a list of (pos, value) tuples, how to construct a vector from them so that those tuples will be the vector's elements? My solution is to introduce a new data type: *scalar*. Basically, a scalar type attribute has two fields: a *value* field, which is a double; and a *label* field, which is an integer. The *value* field stores the value of this attribute, and the *label* field indicates the position of this scalar in the future vector. With such value and label information, we can construct a vector from a list of scalars by calling an aggregate function *vectorize*.

Recall the *vector_table* example. Now assume we want to transfer the data in this table to a vector, and store it to a vector attribute *vector_table_new.v*. Then we can use the following code to do the work:

```
create table vector_table_new (v) as
        select vectorize(label_scalar(value, pos))
        from vector_table;
```

Firstly, a built-in function *label_scalar* is called to construct a scalar based on the (value, pos) tuples. The attribute *vector_table.value* is used to form the *value* field, and the attribute *vector_table.pos* is used to form the *label* field. Then, an aggregate function *vectorize* is applied to construct a vector based on these newly-created scalars. Lastly, a new table *vector_table_new* is created, and the newly-constructed vector is stored in the attribute *vector_table_new.v*.

If we print out the table *vector_table_new*, its data will look as follows:

```
[1.0, 2.0, 3.0]
```

(2) Construct a Matrix from Tuples and Vectors

Assuming a matrix has already been stored in a pure relational way (as discussed in section 3.1):

```
create table matrix_table
(
        row_index integer,
        col_index integer,
        value double
);
```

and its data is:

```
0 | 0 | 1.0 |
0 | 1 | 2.0 |
0 | 2 | 3.0 |
1 | 0 | 4.0 |
1 | 1 | 5.0 |
1 | 2 | 6.0 |
2 | 0 | 7.0 |
2 | 1 | 8.0 |
2 | 2 | 9.0 |
```

How to transfer this representation to the matrix type based representation? Or, given a list of (row_index, col_index, value) tuples, how to construct a matrix by using these tuples as its elements? Two steps can be applied to fulfill this task. First, a

list of vectors can be constructed based on the given tuples. Second, a matrix can be built based on those newly-created vectors.

To create a matrix from a list of vectors, a new field of a vector attribute is needed: a **label** field. The label field is used to indicate the position of a vector in the matrix. In SimSQL, there are two aggregate functions for matrix construction: the *rowmatrix* function and the *colmatrix* function. The *rowmatrix* function uses the input vectors as rows to construct the output matrix, while the *colmatrix* function uses the input vectors as columns to construct the output matrix.

The following example shows the process of matrix construction. The function *rowmatrix* is used, and the new constructed matrix is stored in a matrix attribute *matrix_table_new.m*:

```
create table matrix_table_new (m) as
        select rowmatrix(label_vector(
                        vectorize(label_scalar(value, col_index)), row_index))
        from matrix_table
        group by row_index;
```

The code above implements the two-step process of matrix construction. First, the function *vectorize* is applied to construct the row vectors from scalars. The scalars that make up a row vector are obtained from the (row_index, column_index, value) triples, and they have the same *row_index*, and their labels are their *column_index*. Second, the function *rowmatrix* is applied to construct a matrix from the vectors produced in the first step. Note that before those vectors are passed to the *rowmatrix* function, another built-in function *label_vector* is called to assigns the labels to those vectors, and the labels are their *row_index*. After the vectors are labeled, the function *rowmatrix* can use them to form a new matrix, and this new matrix is finally stored

in the attribute *matrix_table_new.m*.

If we print out the table *matrix_table_new*, its data is as follows:

```
[1.0, 2.0, 3.0] [4.0, 5.0, 6.0] [7.0, 8.0, 9.0]
```

We can also use the function *colmatrix* to construct a matrix from the table *matrix_table*. The code is shown below:

```
create table matrix_table_new (m) as
        select colmatrix(label_vector(
                        vectorize(label_scalar(value, row)), col))
        from matrix_table
        group by col;
```

Generally, the code is similar to the one using the *rowmatrix* function.

A matrix can be constructed not only from tuples, but also from vectors. For example, we can create the table *matrix_table* in a different way:

```
create table matrix_table
(
        pos integer,
        value vector[]
);
```

This time, the matrix in the table *matrix_table* is stored in a row-based way. The attribute *value* stores the row vector of the matrix, and the attribute *pos* stores the position of the corresponding row vector. Then the data of *matrix_table* is in the following format:

```
0 | [1.0, 2.0, 3.0] |
```

```
1 | [4.0, 5.0, 6.0] |
2 | [7.0, 8.0, 9.0] |
```

With the new *matrix_table*, the code for matrix construction is simple:

```
create table matrix_table_new (m) as
        select rowmatrix(label_vector(value, pos))
        from matrix_table;
```

Firstly, the function *label_vector* is applied to assign labels to the vectors in the *matrix_table.value* attribute, using *matrix_table.pos* as the values of those labels. Then, the aggregate function *rowmatrix* is applied to construct a matrix from the labeled vectors. And finally, this new matrix is stored into the attribute *matrix_table_new.m*. The data of this attribute will be:

```
[1.0, 2.0, 3.0] [4.0, 5.0, 6.0] [7.0, 8.0, 9.0]
```

which is same as the matrix constructed from tuples.

(3) Deconstruct a Vector into Scalars

In section (1) and (2), I have described how to construct a vector from scalars, and how to construct a matrix from tuples and vectors. In section (3) and (4), I will discuss how to conduct deconstructions in SimSQL. Particularly, in this section, I will describe how to deconstruct a vector into scalars.

Suppose the vector to be deconstructed is stored in the attribute *vector_table_new.v*:

```
create table vector_table_new
(
        v vector[]
```

```
);
```

and the data of this vector is:

```
[1.0, 2.0, 3.0] |
```

How to deconstruct this vector into a list of scalars? The key is to use a built-in function called *get_scalar*. The two inputs of this function are a vector and the position of the element to be extracted from this vector. The output is a scalar. Its value is obtained from the element siting in the input position, and its label is just the input position. For example, the following code shows how to extract the first element of the vector stored in *vector_table_new.v*:

```
select get_scalar(v, 0)
from vector_table_new;
```

And the result will be:

```
label: 0, value: 1.0
```

Before starting the deconstruction, we also need another table, *index(i integer)*, which stores the indexes of the elements in the vector *vector_table_new.v*. The data of the table *index* is shown below:

```
0 |
1 |
2 |
```

Now with these two tables, the vector *vector_table_new.v* can be deconstructed with the following code:

```
select get_scalar(vt.v, id.i)
from vector_table_new as vt, index as id
```

The result of this query will be:

```
label: 0, value: 1.0
label: 1, value: 2.0
label: 2, value: 3.0
```

(4) Deconstruct a Matrix into Vectors and Tuples

Similar to the deconstruction process of a vector, a matrix in can also be deconstructed. In this section, I will talk about how to deconstruct a matrix into a list of vectors, and further, deconstruct those vectors into tuples.

Assuming the matrix for deconstruction is stored in the attribute *matrix_table_new.m*:

```
create table matrix_table_new
(
        m matrix[][]
);
```

and the data of this table is:

```
[1.0, 2.0, 3.0] |
[4.0, 5.0, 6.0] |
[7.0, 8.0, 9.0] |
```

Then the following code extracts the first row of the matrix *matrix_table_new.m*:

```
select get_rowvector(m, 0)
```

```
from matrix_table_new
```

And the result of the above query is:

```
label: 0, value: [1.0, 2.0, 3.0]
```

In the SQL code above, a built-in function *get_rowvector* is applied to extract a row from a matrix. The two inputs of this function are a matrix and the index of the row to be extracted from this matrix. The output is a vector, which is a row of the input matrix, and is extracted from the position indicated by the input index. The input index is also stored as the label of the output vector.

Likewise, the first column of the matrix *matrix_table_new.m* can also be extracted:

```
select get_colvector(m, 0)
from matrix_table_new
```

And the result is:

```
label: 0, value: [1.0, 4.0, 7.0]
```

The code is similar to the one for the row vector extraction. The only difference is that this time, another built-in function *get_colvector* is employed, and this function is designed to extract column vectors from a matrix, according to the input column index.

With the function *get_rowvector* and *get_colvector*, the code for matrix deconstruction is straightforward. For example, assuming we have another table *index(i integer)* that stores the indexes of the rows in the matrix *matrix_table_new.m*, and the data of this table is

```
0 |
```

```
1 |
2 |
```

Then the matrix *matrix_table_new.m* can be deconstructed with the following code:

```
select get_rowvector(mt.m, id.i)
from matrix_table_new as mt, index as id
```

The result of the query above will be:

```
label: 0, value: [1.0, 2.0, 3.0]
label: 1, value: [4.0, 5.0, 6.0]
label: 2, value: [7.0, 8.0, 9.0]
```

Apart from deconstructing a matrix into a list of vectors, we can also deconstruct it into a list of (row_index, col_index, value) tuples, which is the relational way to represent a matrix in a RDBMS. Such deconstruction process is simple. The first step is to deconstruct a matrix into (row_index, row_vector) pairs. Then, those *row_vectors* will be further deconstructed into (col_index, value) pairs, and we get the (row_index, col_index, value) triples for the original matrix.

For example, the following SQL code shows how to deconstruct the matrix *matrix_table_new.m* into tuples. Firstly, a table *row_vector* is created to deconstruct this matrix into (row_index, row_vector) pairs:

```
create table row_vector (row_index, vec) as
        select id.i, get_rowvector(mt.m, id.i)
        from matrix_table_new as mt, index as id;
```

Then, the vector *row_vector.vec* is further deconstructed into (col_index, value)

pairs, and combining with the attribute *row_vector.row_index*, the final (row_index, col_index, value) triples are obtained:

```
create table matrix_table (row_index, col_index, value) as
        select rv.row_index, id.i, get_scalar(rv.vec, id.i)
        from row_vector as rv, index as id;
```

Most part of the code above has been discussed before, so the code analysis for this piece of code will be ignored. If we print out the table *matrix_table*, its data will be shown as follows:

```
0 | 0 | label: 0, value: 1.0
0 | 1 | label: 1, value: 2.0
0 | 2 | label: 2, value: 3.0
1 | 0 | label: 0, value: 4.0
1 | 1 | label: 1, value: 5.0
1 | 2 | label: 2, value: 6.0
2 | 0 | label: 0, value: 7.0
2 | 1 | label: 1, value: 8.0
2 | 2 | label: 2, value: 9.0
```

(5) Revisiting the Gram matrix example

This section revisits the Gram matrix example to illustrate how to construct/deconstruct vectors and matrices in the context of a real application. This time, the formula 3.6 is used to calculate the Gram matrix.

First of all, a table *data* is needed to hold the data vector x and its *id*:

```
create table data
```

```
(
        x vector[],
        id integer
);
```

The values of the attribute *id* are $0, 1, 2, \ldots, n$, which indicate the sequence of the data points. With the data table, the value of $\mathbf{G}_i$, i.e., the ith row vector of the Gram matrix $\mathbf{G}$ can be calculated with the following code:

```
create table data_vector (id, gi) as
        select d.id, vectorize(label_scalar(inner_product(d.x, dt.x), dt.id))
        from data as d, data as dt
        group by d.id
```

The code above can be decided into several parts. First, a built-in function *inner_product* is applied on $\mathbf{x}_i$ and $\mathbf{x}_j$ to calculate their inner product $\mathbf{x}_i^T\mathbf{x}_j$. According to the Formula 3.6, this inner product is the value of $\mathbf{G}_{ij}$. Next, the value of $\mathbf{G}_{ij}$ and its *id* i can be used to construct the vector $\mathbf{G}_i$, by calling the built-in function *label_scalar* and the aggregate function *vectorize*. Such calculation is just the process of vector construction that has been described before. Finally, the vector $\mathbf{G}_i$ as well as its position ID i are stored in the table *data_vector*.

With the table *data_vector* at hand, the Gram matrix $\mathbf{G}$ is easy to compute:

```
select rowmatrix(label_vector(gi, id))
from data_vector
```

The two functions used in the code are *rowmatrix* and *label_vector*, which have been discussed before. Overall, they construct the Gram matrix $\mathbf{G}$ from the vector

$\mathbf{G}_i$ (i.e., *data_vector.gi*), with the hep of the row ID i (i.e., *data_vector.id*). The result of this query is just the Gram matrix $\mathbf{G}$.

So far, the basic definitions, operations, and functions of vectors and matrices have been discussed. In the following two sections, I will use vectors and matrices to solve a real machine learning problem.

3.4 Gaussian Mixture Model

In the next two sections, a model called *Gaussian Mixture Model* (GMM) (Reynolds [9]) is introduced. It is very natural to use vectors and matrices to describe and learn this model. Thus, this model is a good example to illustrate how to use vectors and matrices in practice. This section focuses on the mathematical description of GMM, while the next section will provide and analyze the real SimSQL code for learning GMM.

A Gaussian Mixture Model is a probabilistic clustering model. The assumption of this model is that the data points come from a mixture of Gaussian distributions with different means and covariances [61]. More formally, we assume that there are K multi-dimensional Gaussian distributions. Each distribution, say, the kth distribution, is parameterized by a mean vector μ_k and a covariance matrix Σ_k. The mixture probability of these distributions is represented by a vector π. Hence, π_k represents the probability of sampling a data point from kth distribution.

Then, the data can be generated by sampling from these distributions. More specifically, suppose we want to produce jth data point x_j for a data set x. First of all, a vector c_j is needed to indicate which Gaussian distribution we should use to produce x_j. This vector is computed by sampling from the distribution Multinomial $(\pi, 1)$. $c_{j,k}$ is 1 if and only if we will use the kth Gaussian to produce x_j, and it is zero otherwise. Then, x_j is generated by sampling from kth Gaussian distribution,

that is, the Normal (μ_k, Σ_k) distribution (given $c_{j,k} = 1$).

Moreover, before generating the data, we need to have all required GMM parameters at hand. Under the Bayesian assumption, these parameters are produced by sampling from some prior distributions. We assume π is sampled from the distribution Dirichlet (α), μ_k is sampled from the distribution Normal (μ_0, Σ_0), and Σ_k is sampled from the distribution InvWishart (v, Ψ).

Now the goal of learning a Gaussian Mixture Model is clear: given a data set x, infer the parameters of a mix of Gaussian distributions that are assumed to produce this data set. More specifically, according to the description above, the parameters to learn are the c_j for each data point x_j, and the μ_k, Σ_k and π_k for each Gaussian distribution k.

In this study, I apply *Markov Chain Monte Carlo* (**MCMC**) (Andrieu et al. [62]) and *Gibbs sampling* (Geman and Geman [63]) to learn the GMM. In the MCMC method, a Markov Chain will be constructed, whose states are the parameters of the model to learn. The states satisfy the Markov property. That is, the conditional probability distribution of the current state only depends on the state before it. In MCMC theory, the equilibrium distribution of this Markov Chain is the desired posterior distribution of those parameters.

Gibbs sampling is a common MCMC algorithm. It works in the following way: first, with a Markov chain of parameters, one parameter is sampled conditioning on other parameters with fixed values. Then, another parameter is sampled in the same way, so on so forth. After several iterations, the equilibrium distribution of this parameter Markov Chain will be achieved, which consists of the correct values of the parameters. In SimSQL, such Markov Chain is made up of tables which represent the states of parameters, and the simulation will be stipulated by sampling the parameters iteration by iteration, using the Gibbs sampling method. Each sampling is conducted

by calling the stochastic functions for the corresponding distributions.

Before I give the detailed MCMC process for GMM, I will introduce a new unit-vector parameter $p_j^{(i)}$, which can make the final formula easier to read. $p_j^{(i)}$ is a probability vector for data point x_j at iteration i. Each element of this vector, $p_{j,k}^{(i)}$, is proportional to $\pi_k^{(i)} \times \text{Normal}\left(x_j|\mu_k^{(i)}, \Sigma_k^{(i)}\right)$. Thus, $p_{j,k}^{(i)}$ represents the posterior probability with which the point x_j comes from the kth Gaussian distribution at the ith MCMC iteration. Then, a Markov Chain to learn the GMM can be described as follows (Cai et al. [64]):

$$
\begin{aligned}
\mu_k^{(i)} \sim \text{Normal}\Bigg(& \left(\Sigma_0^{-1} + n\left(\Sigma_k^{(i-1)}\right)^{-1}\right)^{-1} \times \\
& \left(\Sigma_0^{-1}\mu_0 + \left(\Sigma_k^{(i-1)}\right)^{-1} \sum_j c_{j,k}^{(i-1)} \mathbf{x}_j\right), \\
& \left(\Sigma_0^{-1} + n\left(\Sigma_k^{(i-1)}\right)^{-1}\right)^{-1}\Bigg) \\
\Sigma_k^{(i)} \sim \text{InvWish} & \left(n + v, \Psi + \sum_j c_{j,k}^{(i-1)} (\mathbf{x}_j - \mu_k^{(i)})(\mathbf{x}_j - \mu_k^{(i)})^T\right) \\
\pi_k^{(i)} \sim \text{Dirichlet} & \left(\alpha + \sum_j c_j^{(i-1)}\right) \\
c_j^{(i)} \sim \text{Multinom} & \left(\mathbf{p}_j^{(i)}, 1\right)
\end{aligned}
\tag{3.8}
$$

Formula 3.8 : *The sampling process for GMM*

In the next section, I will describe the SimSQL implementation of GMM, according to above sampling process.

3.5 Implementing the GMM in SimSQL

This section provides the complete SimSQL code for learning the Gaussian Mixture Model.

3.5.1 Preprocessing

Before giving the implementation of the Gibbs sampling process, some preparation work is needed.

First of all, a *cluster* table and a *data* table are created to store the cluster information and the data points (A *cluster* here means a *Gaussian distribution* in GMM):

```
create table cluster
(
        id integer,
        primary key (id)
);


create table data
(
        data_id integer,
        point vector[],
        primary key (data_id)
);
```

The table *cluster* stores the *id* for each cluster, and the table *data* stores each data point and its *data_id*. Note that the type of the attribute *point* is *vector[]*. If there was no vector type, we had to use two attributes *position* and *value* to represent

each data point.

Next, a table *meta* is created to store some super-parameters, including the α in the Dirichlet (α) prior for π, and the v and Ψ in the InvWishart (v, Ψ) prior for Σ_k, referring to the Formula 3.8:

```
create table meta
(
        alpha vector[],
        prior_v integer,
        psi matrix[][]
);
```

The type of the attribute *psi* is *matrix[][]*. If there was no matrix type, we would need to use three attributes *row_index*, *col_index* and *psi_value* to represent the matrix *psi*.

We also need the other two super-parameters for the prior distributions: the μ_0 and the Σ_0 in the Normal (μ_0, Σ_0) prior for μ_k. Different from the super-parameters in the *meta* table, μ_0 and Σ_0 are calculated from data. μ_0 is computed as the mean of all data points, and Σ_0 is a diagonal matrix whose values are from the diagonal of the covariance matrix of input data:

```
create view gmm_prior(mean, covar) as
        select m.data_mean, diag_matrix(get_matrix_diag(
                avg(outer_product(data.point - m.data_mean,
                data.point - m.data_mean))))
        from data, (select avg(point) as data_mean from data) as m
        group by m.data_mean;
```

In the code above, a view *gmm_prior* is created, with two attributes: *mean* and *covar*, corresponding to μ_0 and Σ_0, respectively. The attribute *mean* is calculated as $\frac{1}{n}\sum_j x_j$, as the following code shows:

```
select avg(point) as data_mean from data
```

In this piece of code, an aggregate function, *avg* is applied on the *data.point* to get the average value of all data points. If the attribute *data.point* does not have a vector type, the calculation of its average value is not easy.

Another attribute *covar*, in the view *gmm_prior*, is calculated according to this expression: $diag\_matrix(get\_diag(\frac{1}{n}(\mathbf{x}_j - \overline{\mathbf{x}})(\mathbf{x}_j - \overline{\mathbf{x}})^T))$. $\overline{\mathbf{x}}$ is the mean of input data points. $\frac{1}{n}(\mathbf{x}_j - \overline{\mathbf{x}})(\mathbf{x}_j - \overline{\mathbf{x}})^T$ is the covariance matrix of input data points. *diag_matrix* and *get_diag* are two built-in functions, and they are applied together to construct a diagonal matrix, using the diagonal of the covariance matrix $\frac{1}{n}(\mathbf{x}_j - \overline{\mathbf{x}})(\mathbf{x}_j - \overline{\mathbf{x}})^T$. The corresponding code is as follows:

```
diag_matrix(get_matrix_diag(
                avg(outer_product(data.point - m.data_mean,
                data.point - m.data_mean))))
```

First, a built-in function *outer_product* is called to get the outer product between the vector (*data.point - m.data_mean*) and itself. This part of code is written according to the formula $(\mathbf{x}_j - \overline{\mathbf{x}})(\mathbf{x}_j - \overline{\mathbf{x}})^T$. Then, the aggregate function *avg* is applied on that outer product to get the covariance matrix of the input data, as the formula $\frac{1}{n}(\mathbf{x}_j - \overline{\mathbf{x}})(\mathbf{x}_j - \overline{\mathbf{x}})^T$ shows. Third, a built-in function *get_matrix_diag* is called to obtain the diagonal of that covariance matrix. Finally, after feeding this diagonal to another built-in function *diag_matrix*, a diagonal matrix is constructed, and this matrix is just the value of the attribute *covar*. Note that if there were no vector

and matrix types, the calculation of *covar* would be all tuple-based, which would be tedious and complex, and we could not apply those bulit-in functions on vectors and matrices directly.

So far, we have finished most parts of the preparation work. The last part is to provide some initialization data for the GMM parameters. More specifically, a table *modelfix* is created to store the initial values of the mean and the covariance matrix of each cluster:

```
create table modelfix
(
        mean vector[] random,
        covariance matrix[][] random,
        cluster_id integer
);
```

In this table, the data type of the attribute *mean* is *vector[] random*, and the data type of the attribute *covariance* is *matrix[][] random*. The word "random" here means that these two attributes are random variables. Hence, these attributes may have different values in different runs and iterations.

Up to now, we have finished all preparation work. In the next section, I will give the full code for learning the GMM by employing the Gibbs sampling.

3.5.2 GMM Implementation

This section provides the full Gibbs sampling process for learning the GMM. The code is based on the mathematical description of GMM in section 3.4.

First of all, an initial value is assigned to the cluster mixture probability π. It is stored in the attribute *probability* of the table *mix_prob[0]*. The value of π is calculated

by applying the VG function *Dirichlet_VM* on the prior *meta.alpha*, referring to the formula $\pi_0 \sim \text{Dirichlet}(\alpha)$. The code below shows this calculation:

```
create table mix_prob[0] (probability) as
        with prob as Dirichlet_VM(
                select alpha
                from meta
        )
        select prob.probability
        from prob;
```

The next step is to assign the initial values to the means and covariance matrices for the clusters. These values are obtained from the*modelfix* table, which is created in section 3.5.1:

```
create table model[0](cluster_id, mean, covariance) as
        select mf.cluster_id, mf.mean, mf.covariance
        from modelfix as mf;
```

With the initial values at hand, the real sampling process can start. Firstly, the membership $c_j^{(i)}$ for each data point is sampled, based on the Formula 3.8:

```
create table membership[i](data_id, cluster_id, member_vector) as
        for each d in data
                with mem as multinomial_membership_VM(
                        (select probability
                         from mix_prob[i]
                        ),
                        (select dd.point
```

```
                    from data as dd
                    where dd.data_id = d.data_id
                  ),
                  (select mean, covariance, cluster_id
                   from model[i]
                  )
        )
select d.data_id, get_nonzero_pos(mem.cluster_count),
            mem.cluster_count
from mem;
```

The table *membership[i]* consists of three attributes: *data_id* is the ID of a data point, which has the same value of *data.data_id*; *cluster_id* is the ID of the cluster that this data point belongs to; *member_vector* is the membership indicator vector, i.e., the $c_j^{(i)}$ in Formula 3.8. The value of *member_vector[cluster_id]* is 1, and the values of the other entries of *member_vector* are 0. Tthe attribute *member_vector* is calculated by calling the VG function *multinomial_membership_VM*. The inputs of this function are the cluster mixture probability (obtained from the table *mix_prob[i]*), a data point (obtained from the table *data*), and the parameters of all clusters (obtained from the table *model[i]*). The output is the membership vector of the input data point, and its name is *cluster_count*. Moreover, a built-in function, *get_nonzero_pos*, is employed to extract the position of the first non-zero value in the *member_vector*, which is just the *cluster_id* of the cluster that the in-processing data point belongs to.

With the membership information, we are able to sample the new cluster mixture probability for the next iteration, i.e., the $\pi_k^{(i)}$ in the Formula 3.8:

```
create table mix_prob[i](probability) as
```

```
        with prob as Dirichlet_VM(
                select sum(m.member_vector) + meta.alpha
                from membership[i-1] as m, meta
                group by meta.alpha
        )
        select prob.probability
        from prob;
```

The code is straightforward by following the formula.

Next, we introduce two assistant tables. These tables can make the sampling process easier and clearer to read and understand. The first assistant table stores a vector to indicate the number of data points in each cluster, and this vector is calculated as $\sum_j c^{(i)}$:

```
create table member_number[i](mem_num) as
        select sum(m.member_vector)
        from membership[i] as m;
```

The second table calculates some small parts in the Formula 3.8:

```
create table assist[i](cluster_id, partial_sum, sum_sqr) as
        select me.cluster_id, sum(data.point),
               sum(outer_product(
               data.point - mo.mean, data.point - mo.mean))
        from data, membership[i-1] as me, model[i-1] as mo
        where data.data_id = me.data_id
              and
              mo.cluster_id = me.cluster_id
```

```
group by me.cluster_id;
```

In the code above, a calculation is conducted on the data points that come from the same cluster. More specifically, the attribute *partial_sum* calculates the sum of these data points, with the code"`sum(data.point)`", and its value is the $\sum_j c_{j,k}^{(i-1)} \mathbf{x}_j$ in the Formula 3.8. Another attribute *sum_sqr* calculates the covariance matrix of those data points, with the code "`sum(outer_product(data.point - mo.mean, data.point - mo.mean))`", and its value is the $\sum_j c_{j,k}^{(i-1)} (\mathbf{x}_j - \mu_k^{(i)})(\mathbf{x}_j - \mu_k^{(i)})^T$ in the Formula 3.8.

With all of the tables we have created, the final step is to sample the model parameters $\mu_k^{(i)}$ and $\Sigma_k^{(i)}$ for each cluster. These values are the final results we want to obtain from learning a GMM. The code is given below.

```
create table model[i](cluster_id, mean, covariance) as
  for each c in cluster
    with sampleMean as MultiNormal_VM(
      (select matrix_vector_multiply(
                matrix_inverse(matrix_inverse(p.covar) +
                get_scalar(mn.mem_num, c.id) *
                matrix_inverse(mo.covariance)),
                matrix_vector_multiply(matrix_inverse(p.covar), p.mean) +
                matrix_vector_multiply(matrix_inverse(mo.covariance),
                                       at.partial_sum)
              ),
              matrix_inverse(matrix_inverse(p.covar) +
                get_scalar(mn.mem_num, c.id) *
                matrix_inverse(mo.covariance))
       from gmm_prior as p, membe_number[i-1] as mn,
```

```
            assist[i] as at, model[i-1] as mo
       where c.id = at.cluster_id and
             c.id = mo.cluster_id
    )
    )
  with sampleVar as inverse_wishart_VM(
    (select meta.psi + at.sum_sqr, get_scalar(mn.mem_num, c.id) +
            meta.prior_v
     from assist[i] as at,
          member_number[i-1] as mn, meta
     where c.id = at.cluster_id
    )
    )
  select c.id, sampleMean.out_mean, sampleVar.out_variance
  from sampleMean, sampleVar;
```

The mean and covariance matrix of each cluster are calculated in the last part of the GMM code. The mean is computed by sampling from a multi-normal distribution, and the VG function *MultiNormal_VM* fulfills this sampling. The covariance matrix is computed by sampling from an inverse-Wishart distribution, and the VG function *inverse_wishart_VM* conducts this sampling. All input parameters of these functions are calculated according to the Formula 3.8, and the calculation is very straightforward.

Note that many VG functions and built-in functions are used in the GMM example, such as the VG function *MultiNormal_VM* for the multi-normal distribution, and the built-in function *matrix_vector_multiply* for the matrix-vector multiplication.

These functions are very common and useful in machine learning applications. The next chapter, Chapter 4, provides more details about these functions, and lists out their general programming interfaces.

Chapter 4

For SimSQL Developers: Vector and Matrix Type

4.1 Aggregate Functions, VG Functions and Built-in Functions

In this section, I will describe the programming interfaces of aggregate functions, VG functions, and built-in functions, and explain how to manipulate vectors and matrices with these functions.

4.1.1 Aggregate functions

In a database system, an aggregate function is used to conduct aggregation operations on the records. It takes a group of row records as input, and outputs a certain value with some calculation on the those records. In SimSQL, there are two types of aggregate functions: regular aggregate functions, including *sum*, *avg*, *count*, etc, and special aggregate functions, including *vectorize*, *rowmatrix* and *colmatrix*, which are designed for vector/matrix construction. In previous chapters, we have seen some aggregate functions. In this section, I will describe the general programming interfaces for aggregate functions, and my examples will focus on the those functions with vector

and matrix as their input/output parameters.

Basically, the programming interface of an aggregate function is as follows:

```
agg_function(parameter 1, parameter 2, ..., parameter n)
return parameter n+1
```

And an example of such interface is given below:

```
agg_vector_matrix(vector[a], vector[3], matrix[a][b], matrix[5][5])
return vector[a]
```

The example shows the programming interface of an aggregate function called *agg_vector_matrix*. It has four input parameters, which have types *vector[a]*, *vector[3]*, *matrix[a][b]* and *matrix[5][5]*, correspondingly. Its output type is *vector[a]*. Generally, in the programming interface of an aggregate function, a vector/matrix parameter is represented as a vector/matrix type attribute. The variable or the positive integer in the bracket indicates the dimension requirement for this parameter.

If the bracket in a parameter contains a variable, that variable will be bound to a real value when the argument is passed to this parameter. If such a variable appears multiple times in the programming interface, the value of this variable must be consistent for all of its appearances during the argument passing. For example, in the function *agg_vector_matrix*, the variable "a" appears in two input parameters, *vector[a]* and *matrix[a][b]*, and it is also used in the output parameter, *vector[a]*. Now, assuming a vector attribute, *vector[6]*, is passed to the input parameter *vector[a]* as its argument. Then the argument for the parameter *matrix[a][b]* must be "matrix[6][b]" (b can be any integer or just empty), and the output of this function will have type *vector[6]*.

If the arguments passed to a programming interface do not meet its require-

ments, SimSQL will throw out an error. The only exception is the empty bracket. The empty bracket is compatible with all variables. For instance, in the function *agg_vector_matrix*, *matrix[][]* will be an acceptable argument for the input parameter *matrix[a][b]*, no matter what arguments are passed to other input parameters.

If the bracket in a parameter contains a positive integer, this integer represents the dimension requirement for this vector/matrix parameter. The argument for such parameter must have the same dimension value as that positive integer indicates. Otherwise, SimSQL will throw out an error. Again, the only exception is the empty bracket. The empty bracket is compatible with all positive integers. For example, in the function *agg_vector_matrix*, the second input parameter has type *vector[3]*. Thus, the argument for this parameter must have type *vector[3]*, or *vector[]*.

The programming interfaces of the aggregate functions *vectorize*, *rowmatrix* and *colmatrix* are listed below, and these functions are designed for vector and matrix construction.

```
vectorize(scalar) return vector[a]
rowmatrix(vector[a]) return matrix[b][a]
colmatrix(vector[a]) return matrix[a][b]
```

The function *vectorize* is used to construct a vector from scalars. The function *rowmatrix* is used to construct a row-based matrix from vectors. The input vectors for this function are the rows for the output matrix. The function *colmatrix* is used to construct a column-based matrix from vectors. The input vectors for this function are the columns for the output matrix.

4.1.2 VG functions

Variable Generation functions (VG functions) are pseudo-random functions for sampling values from a list of pre-defined distributions. Some common distributions supported by the VG functions are the categorical distribution, the inverse-wishart distribution, the multinomial distribution, etc. The programming interface of a VG function is given below:

```
VG_function(parameter 1, parameter 2, ..., parameter i)
return parameter i+1, parameter i+2 random, ..., parameter n
```

Compared with an aggregate function, a VG function can have multiple outputs, instead of one output, and at least one output has the keyword **random** in its type definition. A "random" output represents a random variable. Its value is obtained by sampling, and may have different values in the different calls of the VG function.

The dimension requirement for a VG function's vector/matrix parameters is similar to that for an aggregate function. The bracket in a vector/matrix parameter contains a variable or a positive integer, and the type checking for a vector/matrix parameter during the argument passing is the same as the one for an aggregate function, which is described in section 4.1.1.

4.1.3 Built-in functions

The built-in functions are pre-defined libraries provided in SimSQL. Currently, they are mainly designed for linear algebra operations. The programming interface of a built-in function is as follows:

```
Buil_in_function(parameter 1, parameter 2, ..., parameter n)
return parameter n+1
```

This programming interface is the same as the one for an aggregate function. Also, the type checking for a built-in function's vector and matrix parameters follows the same rules for an aggregate function, which are discussed in section 4.1.1.

4.2 Type and Size Inference for Vector and Matrix Type Attribute

In section 4.1, we have considered the programming interfaces for aggregate functions, VG functions and built-in functions, especially how the "parameterized types" are used for the vector and matrix parameters in these functions. In this section, I will discuss why we design such "parameterized types", and how we can utilize them to infer the output sizes of vector/matrix related operations.

4.2.1 Why Parameterized Types for Vector and Matrix Parameters: an Optimization Consideration

Current relational database systems pay little attention to the syntaxes and semantics of linear algebra operations (as Chapter 2 shows). Without that information, those systems cannot have an accurate estimation of the cost of those operations. Therefore, when those systems try to optimize the query plan that executes those operations, they may omit some optimization opportunities.

This optimization problem can be illustrated by a simple example. Assume we have three tables defined as below:

```
R (r_rid integer, r_matrix matrix[1000][1000]), 100 tuples
S (s_sid integer, s_matrix matrix[1000][1000]), 100 tuples
T (t_rid integer, t_sid integer), 1000 tuples
```

The table R has two attributes: *r_rid*, with type *integer*, and *r_matrix*, with type *matrix[1000][1000]*. There are 100 tuples in this table. The table S has the similar attributes with the table R, and it also has 100 tuples. The third table T has two attributes: *t_rid* and *t_sid*. Both of them have *integer* type. There are 1000 tuples in this table.

Now, suppose we want to calculate the inner product between the diagonal of the matrix *r_matrix* and the diagonal of the matrix *s_matrix*. We can write the following query to perform such calculation:

```
SELECT inner_product(diag(r_matrix), diag(s_matrix))
FROM R, S, T
WHERE r_rid = t_rid and s_sid = t_sid
```

When a relational database system executes the code above, it will generate a plan to guide the execution. In this example, the following plan is usually produced:

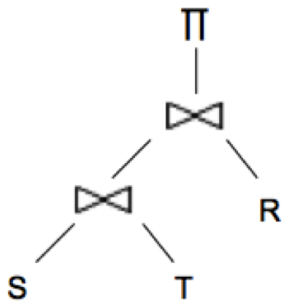

Figure 4.1: Plan1 of "R, S, T" example

The plan is straightforward. Firstly, it does a join between the table S and the table T. Then, it conducts another join between the previous join result and the table R. Finally, it performs a projection on the top of the second join, and this projection calculates the inner product between the diagonal of the selected *r_matrix* and the

diagonal of the selected *s_matrix*. Note that the join between the table S and T produces about 1000 tuples (estimated as 1000 * 100 / 100), and each tuple is a 8MB matrix (estimated as 8 * 1000 * 1000 Byte; each element in a matrix is 8 Byte). Thus, the total data produced in this join is about 8 GB.

Can we find a plan with less intermediate data? The answer is "yes". Look at the plan below:

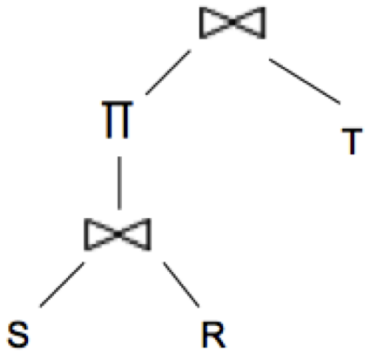

Figure 4.2: Plan2 of "R, S, T" example

The new plan arranges the operators in a different order. Firstly, it does a join between the table S and R. Then, it performs a projection on the join result to calculate the inner product between the diagonal of the *r_matrix* and the diagonal of the *s_matrix*. Finally, it conducts another join between the projection result and the table T, which picks up the inner product for the matrices specified by the query. Note that this time, the join between the table S and R produces about 10000 tuples (estimated as 100 * 100), and each tuple is a 8 Bytes double. Thus, the total data produced in this join is about 80 KB. Compared with the previous plan, this new plan generates much less data for the first join (see Figure 4.3), and the intermediate data produced by the whole plan is also reduced. Thus, in respect of space and speed consideration, the new plan is a better choice.

Why are current database systems not likely to produce plan2? The answer is that

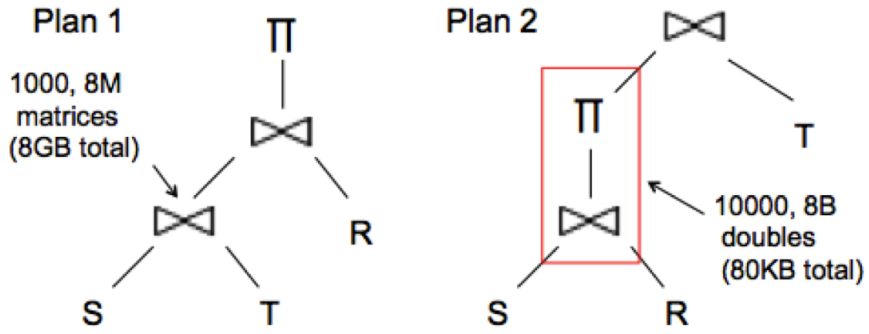

Figure 4.3: Comparison of Plan1 and Plan2

they are not aware of the syntax information of linear algebra operations. Therefore, they do not realize that the inner_product function (a.k.a, the projection part in the plan) will shrink the vector inputs to a double output, and they should execute this function as early as possible (a.k.a, push the projection down). SimSQL, however, is aware of the syntax of linear algebra operations. With such information, the optimizer of SimSQL can estimate the output size of those operations, and utilizes this size information to optimize the plan.

For example, in SimSQL, the function signature of the built-in function *diag* is:

```
diag(matrix[a][a]) -> vector[a]
```

This signature indicates that the input of the function *diag* is a matrix, and the output is a vector. Also, the first and the second dimension of the input matrix, as well as the length of the output vector, must have the same value. It is because they are all represented by the same letter, a, in the function signature.

Similarly, the syntax of the built-in function *inner_product* is as follows:

```
inner_product(vector[a], vector[a]) -> double
```

The input parameters are two vectors that have the same length, and the output is

a double. Now, combining with the syntax of the function *diag*, SimSQL can infer that the output of this piece of code, *inner_product(diag(r_matrix), diag(s_matrix))*, is a double. When the optimizer gets this information, it will try to perform such calculation as early as possible, because this calculation can reduce the intermediate data produced during the execution. Hence, for the query plan for our example, the optimizer of SimSQL will recommend plan2, instead of plan1. This example shows that the syntaxes of linear algebra operations can help us estimate the output size of each operation, and with this estimation, the optimizer can explore more optimization opportunities.

4.2.2 Size Estimation of Vector/Matrix Related Operations

In section 4.2.1, we have seen the importance of knowing the syntaxes and semantics of linear algebra operations, and the advantages of using such information to estimate the output sizes of those operations. In this section, I will talk about how we implement this idea in SimSQL. More specifically, I will describe how to utilize the syntax information of functions and the "parameterized types" to infer and estimate the output sizes for vector/matrix related operations.

I will start with an example. Suppose we want to use the built-in function *matrix_multiply* to multiply two matrices. The function signature of this function is given below:

```
matrix_multiply(matrix[a][b], matrix[b][c]) -> matrix[a][c]
```

The inputs of this function are two matrices. The first matrix has a rows, b columns, and the second matrix has b rows, c columns. The output is a matrix with a rows and c columns. As described in section 4.1, such function signature actually provides some requirements for the dimensions of the input and output matrices. The

number of columns of the first input matrix must be the same as the number of rows of the second input matrix, and the output matrix has the same number of rows with the first input matrix, and the same number of columns with the second input matrix. Such requirements are naturally derived from the mathematical definition of matrix multiplication.

Now, assume we want to multiply two matrices, *matrix[10][100]* and *matrix[100][1000]*. The following code fulfills this task:

```
matrix_multiply(matrix[10][100], matrix[100][1000])
```

To execute this piece of code, SimSQL first needs to parse and compile it. During the compilation, an important task is to conduct a type checking for the function parameters. To facilitate such type checking, we will build a map between the dimension variables and the dimension values. Such map can be defined as *parameter_value_map(dimension variable: dimension value)*. The key of this map is the dimension variable in the function parameter, and the value of this map is the corresponding value for this variable conveyed by the argument.

In this matrix multiplication example, when the system reads in the first argument, *matrix[10][100]*, it will bind this argument to the first parameter, *matrix[a][b]*. Then, the value 10 will be assigned to the variable *a*, the value 100 will be assigned to the variable *b*, and the pairs (a: 10) and (b: 100) will be added to the map *parameter_value_map*.

Next, the system will read in the second argument, *matrix[100][1000]*, and bind it to the second parameter, *matrix[b][c]*. This time, the system will firstly check whether the dimension variables, *b* and *c*, have been assigned values by the map parameter_value_map. If not, the system will fetch the dimension values from the arguments, bind them to those dimension variables, and put the new pair (dimension

variable: dimension value) into the map. If the dimension variables already exist in the map, the system will check if the dimension values in the coming argument are the same as those implied by the map. If the new values and the existing values are different, the system will throw out an error, indicating that this argument does not satisfy the dimension requirement. If the new values and the existing values are the same, the code will pass this type checking, and the checking will move to the next argument.

In the matrix multiplication example, when the system does the type checking for the second argument, it will find that the dimension variable b already exists in the map, and its value in the map is 100. Then, the system will check the value of b in the second argument, *matrix[100][1000]*. This value is also 100. Then the system can conclude that the value of b in the second argument is valid. Moreover, if the second input is the matrix *matrix[][1000]*, the system will also treat it as a valid input. However, if the second input is *matrix[50][1000]*, the system will treat it as an invalid input. The compiler will find that the value of b in this argument is 50, which is different from its current value in the parameter_value_map. Hence, the compiler will report an error which indicates that the input *matrix[50][1000]* does not meet the dimension requirement for this function. For the dimension variable c, there is no record in the parameter_value_map. Therefore, a new pair (c:1000) will be added to the map.

In the type checking phase, we will also be able to infer the output size of a function. Still take the matrix multiplication as an example. The function signature of *matrix_multiply* indicates that the output of this function is *matrix[a][c]*. Meanwhile, the map *parameter_value_map* indicates that the value of a is 10, and the value of c is 1000. Thus, the output of the function *matrix_multiply* in this example is a 10×1000 matrix, and the size of the output is about $8 \times 10 \times 1000$ bytes, i.e., 80KB

(each element of a matrix has 8 bytes). Such size information will be passed to the optimizer, and it can help the optimizer to explore more optimization opportunities when it optimizes the query plan. If there is unknown size dimension in the output, e.g., matrix[][100], the default value of the unknown size is 50. So the size estimation of the matrix[][100] is $8 \times 50 \times 100$ bytes, i.e., 4KB.

4.3 Serialization and Deserialization for Vectors and Matrices

Serialization is the conversion of an in-memory object to a series of bytes. Deserialization is the reverse process. Serialization and deserialization are used when we need to write an object to a file and read an object from a file, or to transfer an object across the network. When we write/read user-generated input to/from files, the contents usually need to be human readable (e.g., storing the calculation results to a file), but there is no such requirement when we transfer objects across the network. Thus, for these two scenarios, we use different formats to serialize and deserialize the objects. In this section, I will focus on how to serialize and deserialize vectors and matrices in SimSQL.

4.3.1 Vector/Matrix Serialization and Deserialization in Text File Format

A database system usually consists of a set of tables. Before we write any queries for these tables, we need to load data into them. Also, sometimes we need to store the query results to external places for future analysis. In SimSQL, both the data for the tables and the query results for external use are saved in the text files. Thus, to provide the full vector and matrix support for SimSQL, we must specify how to store

them in a text file.

More specifically, a vector $[val\,1, val\,2, ..., val\,n]$ will be stored in a text file as follows:

```
[val 1, val 2, ..., val n]
```

The pair of brackets "[]" indicates that this data is a vector. The variables *val 1*, *val 2*, ..., *val n* are numbers with double type, and they represent the elements of this vector. For example, the vector $[1.0, 2.0, -3.0]$ is stored as follows:

```
[1.0, 2.0, -3.0]
```

Similarly, a $m \times n$ matrix

$$\begin{bmatrix} val\,11 & val\,12 & ... & val\,1n \\ val\,21 & val\,22 & ... & val\,2n \\ ... & & & \\ val\,m1 & val\,m2 & ... & val\,mn \end{bmatrix}$$

will be stored In a text file as follows:

```
[val 11, val 12, ..., val 1n][val 21, val 22, ... val 2n] ... [val m1,
val m2, ..., val m3]
```

The matrix above is stored in a row-based way. The elements in the first pair of brackets are from the first row of the matrix; the elements in the second pair of brackets are from the second row of the matrix; so on so forth. For example, the matrix

$$\begin{bmatrix} 1.0 & 2.0 & -3.0 \\ 4.0 & 5.0 & -6.0 \\ 7.0 & 8.0 & -9.0 \end{bmatrix}$$

will be stored as:

```
[1.0, 2.0, -3.0] [4.0, 5.0, -6.0] [7.0, 8.0, -9.0]
```

After introducing the format of storing vectors and matrices in a file, their serialization and deserialization are straightforward. The serialization process is just taking out the elements of a vector or a matrix attribute, and write them into a text file in the above formats. The deserialization process is reading the data from a text file, parse it into a vector or a matrix according to its format, and store the parsed data in a vector or matrix attribute.

4.3.2 Vector/Matrix Serialization and Deserialization in Stream Format

SimSQL is a distributed database system, and it can be run on a computer cluster. Thus, it must provide support for data transfer across several computers. In Java (the implementation language for SimSQL), data transfer is conducted through streams. Suppose a Java object needs to be copied from a computer to another one. First, this object is written to an output stream, and this output stream is then saved to a file. (This process is deserialization.) Next, the file is transmitted across the network, and received by the targeted computer. Then, the targeted computer reads the file with an input stream, and finally reconstructs the Java object from the input stream. (This process is serialization.) In this section, I will describe the format for storing

a vector and a matrix in a stream. Once the format is at hand, the serialization and deserialization are straightforward.

The format of storing a vector in a stream is as follows:

```
label length_of_vector elements_of_vector
```

Firstly, the label of the vector is stored. Then, the length of the vector is appended. Finally, each element of the vector is added one by one. More specifically, given a vector $[val\,1, val\,2, ..., val\,n]$ with size n and label m, it can be stored in the following format:

```
m n val 1 val 2 ... val n
```

For example, a vector $[1.0, 2.0, -3.0, -4.0]$ with label `1` will be stored as:

```
1 4 1.0 2.0 -3.0 4.0
```

The way to store a matrix in a stream is a little more complex. First, we use different ways to store a row-based matrix and a column-based matrix. The storage formats for these two types of matrices are similar, but the variables in the formats have different meanings. The reason is that the arrays to store a matrix will have different meanings for row-based matrix and column based matrix. Internally, a matrix will be stored as an array of basic arrays. If the matrix is stored in a row-based way, each basic array will represent a row of the matrix. If the matrix is stored in a column-based way, each basic array will represent a column of the matrix. We make this decision because sometimes a matrix has sparse rows, while sometimes a matrix has sparse columns. Thus, we should store a matrix according to its sparsity so that we can save the space of storage. Second, we do not store every element in a matrix to the stream. The reason is also from the consideration of sparsity. For a

sparse matrix, storing a lot of 0s is useless and space-consuming.

The format of storing a row-based matrix in a stream is given below:

```
ifRow num_of_row num_of_column num_of_non_zero_row
index_of_non_zero_row elements_of_non_zero_row
```

ifRow is a boolean variable, which indicates whether the matrix is a row-based matrix (true) or column-based matrix (false). *num_of_row* stores the number of rows. *num_of_column* denotes the number of columns. *num_of_non_zero_row* stores the number of non-zero rows. A "non-zero row" is a row with at least one non-zero element. *index_of_non_zero_row* indicates the positions of non-zero rows (the count of rows starts from 0). Finally, *elements_of_non_zero_row* lists out every element in the non-zero rows, and they are arranged according to those rows' positions.

I will use an example to give a clearer view of this format. Assume this row-based matrix waits to be stored in the stream:

$$\begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 2 \\ 0 & 0 \\ 3 & 4 \end{bmatrix}$$

As it is a row base matrix, *ifRow* should be *true*. It is a 5×2 matrix, so the value of *num_of_row* is *5*, and the value of *num_of_column* is *2*. There are two non-zero rows: the row $[1, 2]$ in position 2, and the row $[3, 4]$ in position 4. Thus, the value of *num_of_non_zero_row* is *2*, and the values of *index_of_non_zero_row* are *2 4*. Finally, the elements of those two non-zero rows are store in *elements_of_non_zero_row*, and the values of this variable will be *1 2 3 4*. In sum, the numbers to store the example

matrix in a stream will be:

```
true 5 2 2 2 4 1 2 3 4
```

The format of storing a column-based matrix is very similar:

```
ifRow num_of_column num_of_row num_of_non_zero_column
index_of_non_zero_column elements_of_non_zero_column
```

Again, *ifRow* is a boolean variable to indicate whether the matrix is a row-based matrix. This time, its value is *false* because this matrix is a column-based matrix. *num_of_column* stores the number of columns, and *num_of_row* stores the number of rows. *num_of_non_zero_column* is the number of non-zero columns. A "non-zero column" is a column with at least one non-zero element. *index_of_non_zero_column* indicates the positions of non-zero columns (the count of columns starts from 0). Finally, the *elements_of_non_zero_column* gives every element in the non-zero columns, and they are arranged according to those columns' positions.

I will use an example to illustrate the format. Assume we want to store this matrix:

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 4 \\ 0 & 2 & 0 & 0 & 5 \\ 0 & 3 & 0 & 0 & 6 \end{bmatrix}$$

Three columns in this matrix have all-zero elements. Thus, it is more space-efficient to store it in a column-based way. The variable *ifRow* will be set to *false*. It is a 3×5 matrix, so the value of *num_of_column* is *5*, and the value of *num_of_row* is *3*. There are two non-zero columns: the column $[1, 2, 3]$ in position 1, and the column $[4, 5, 6]$ in position 4. Thus, the value of*num_of_non_zero_column* is *2*, and the values

of *index_of_non_zero_column* are *1 4*. Finally, the elements of those two non-zero columns are store in *elements_of_non_zero_column*, and the values of this variable will be *1 2 3 4 5 6*. In sum, the numbers to store the example matrix in a stream will be:

```
false 5 3 2 1 4 1 2 3 4 5 6
```

With the formats introduced above, serialization and deserialization of vectors and matrices are straightforward. The serialization of a vector is just writing it to an output stream in its stream format. The serialization of a matrix is a little more complex. First, we need to judge if the matrix is stored in a row-based format or is stored in a column-base format. Then, we will scan the matrix to find the non-zero rows or columns, as well as the elements stored in them. Finally, we will write those data to an output stream, which will be arranged in the stream format of a matrix.

The deserialization of a vector/matrix is the process of reconstructing a vector/-matrix attribute from an input stream, using its stream format as a guide. For the deserialization of a vector, a new vector with length *length_of_vector* will be created, and its elements are obtained from the variable *elements_of_vector*. For the deserialization of a matrix, a new matrix with *num_of_row* rows and *num_of_columns* columns will be created. Then, the spaces for non-zero rows/columns will be allocated according to the variable *num_of_non_zero_row*/*num_of_non_zero_column*, and the variable *index_of_non_zero_row*/*index_of_non_zero_column*. Finally, the elements of the non-zero rows/columns will be extracted from the variable *elements_of_non_zero_row* or *elements_of_non_zero_column*, and those elements will be added to the new matrix.

Chapter 5

Experiments

We have conducted several experiments to test our implementation for vectors and matrices in SimSQL. The first experiment tests Gram matrix computation. The second experiment estimates the parameters for the linear regression model. In the third experiment, four more complicated machine learning models are tested. In all experiments, we write two versions of SimSQL code for the model: a tuple-based version and a vector/matrix-based version. In the tuple-based version, all data is stored and operated as relational tuples. While in the vector/matrix-based version, we mainly use vectors and matrices to store and operate the data. By running these two versions of code, we would like to see whether the introduction of vectors and matrices can bring us any performance benefits.

In this chapter, I will describe these three experiments in detail. For each experiment, I will firstly give the description of the model, and then provide our two versions implementation for the model. Next, I will list out the experiment setup, and then give the results of the experiment. Finally, I will analyze the results, and describe my findings and insights behind the experiment.

5.1 Gram Matrix Computation

The Gram matrix computation is widely used in the mathematical calculation of algorithms, such as covariance matrix computation and kernel function computation. I have introduced the computation of Gram matrix in Chapter 3. In this section, I will calculate this model in two different ways, and give the full SimSQL implementations for them. I have also conducted an experiment to test my implementations. The results, as well as some findings, will be presented at the end of this section.

5.1.1 Model Description

The Gram matrix computation can be defined as follows: Given a set of column vectors $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$, their Gram matrix $\mathbf{G}$ is defined as:

$$\mathbf{G} = \mathbf{X}^T\mathbf{X} = \sum_{i}^{n} \mathbf{X}_i^T\mathbf{X}_i \tag{5.1}$$

$$\text{where } \mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n]$$

Formula 5.1 : *Gram matrix calculation*

Another way to write the calculation is:

$$\mathbf{G}_{i,j} = \sum_{k}^{n} \mathbf{X}_{k,i}\mathbf{X}_{k,j} \tag{5.2}$$

Formula 5.2 : *Gram matrix calculation* : *An alternate way*

5.1.2 Model Implementation

This section gives the two implementations of the Gram matrix computation: pure tuple-based, and vector/matrix-based.

5.1.2.1 Tuple-based Implementation

First, a table is needed to store each element of the matrix **X**:

```
CREATE TABLE x
(
        row_index integer,
        col_index integer,
        value double
);
```

Then, according to Formula 5.2, the Gram matrix can be calculated with the following code:

```
SELECT x1.col_index, x2.col_index, sum(x1.value * x2.value)
FROM x as x1, x as x2
WHERE x1.row_index = x2.row_index
GROUP BY x1.col_index, x2.col_index;
```

5.1.2.2 Vector/Matrix-based Implementation

Similarly, a table is firstly created to store the matrix **X**:

```
CREATE TABLE x_vm
(
        id integer,
```

```
        value vector[]
);
```

Then, according to Formula 5.1, the Gram matrix can be calculated with the following code:

```
SELECT sum(outer_product(x.value, x.value))
FROM x_vm as x;
```

5.1.3 Experiment Setup

Both pure tuple-based implementation and vector/matrix-based implementation are tested. For the matrix $\mathbf{X}$, its row number is kept constant as 5×10^5, and its column number is changed from 10, 100 to 1000. The experiment is conducted on a 5-machine Amazon EC2 cluster. The machine's type is c3.2xlarge, with 8 CPU cores, 15 GB memory, and 2×80 GB SSD. SimSQL v0.4 is used as the experiment platform.

5.1.4 Query Plan and Output Estimation

The query plan of the tuple-based implementation is depicted in Figure 5.1, and the output size estimation of each job is shown in Figure 5.2. The query plan of the vector/matrix-based implementation is depicted in Figure 5.3, and the output size estimation of each job is shown in Figure 5.4

5.1.5 Experiment Result

The experiment' results are listed in Table 5.1, and they are depicted in Figure 5.5. The numbers on the top of bars in Figure 5.5 represent how many times the tuple-based version is slower than the vector/matrix-based version.

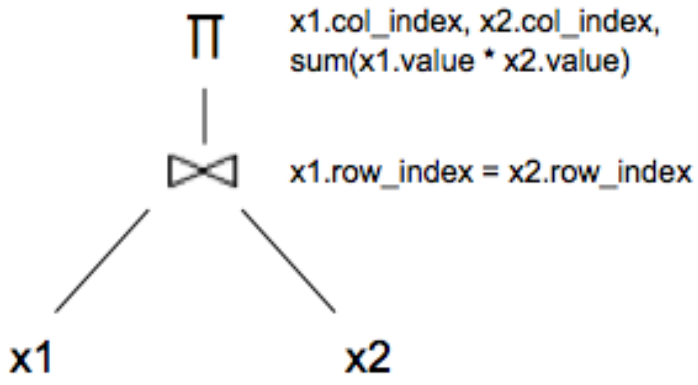

Figure 5.1: Query Plan of Tuple-Based Implementation of Gram Matrix Computation.

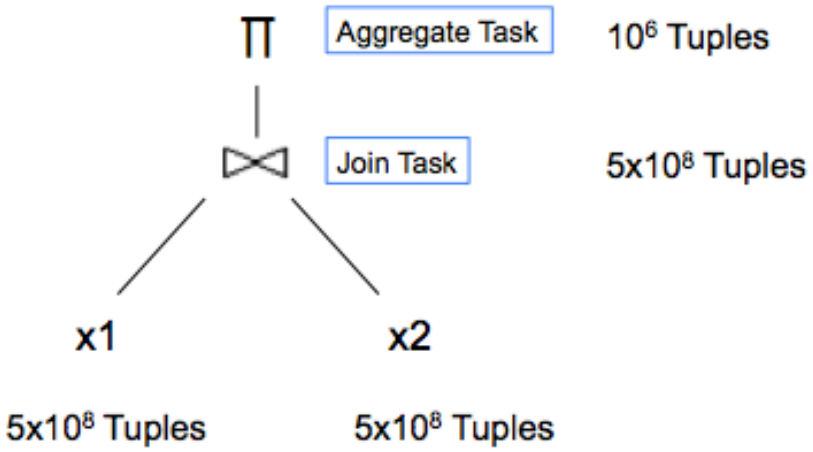

Figure 5.2: Output Size Estimation of Tuple-Based Implementation of Gram Matrix Computation.
(The Column number of X is 1000)

Gram Matrix Computation			
Column numbers of **X**	10	100	1000
Tuple Based Version	00:01:17	00:03:37	05:25:49
Vector/Matrix Based Version	00:00:22	00:00:33	00:06:58

Table 5.1: Running Time of Gram Matrix Computation. Format is HH:MM:SS.

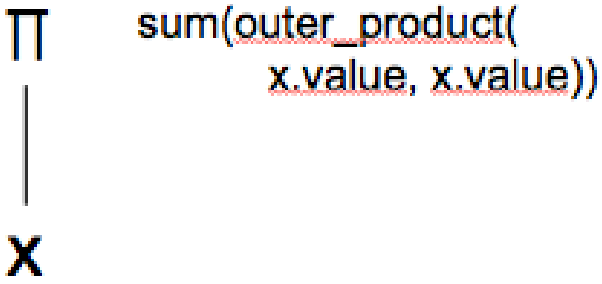

Figure 5.3: Query Plan of Vector/Matrix-Based Implementation of Gram Matrix Computation.

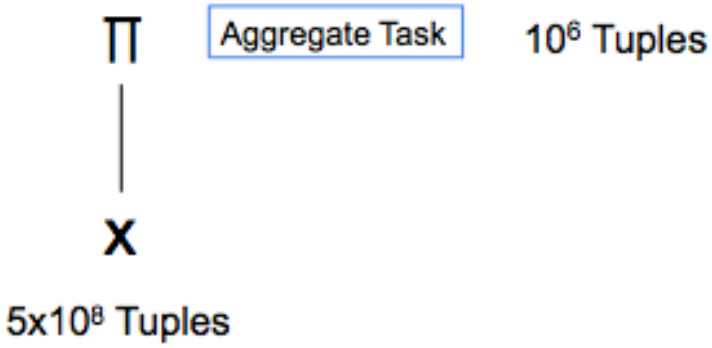

Figure 5.4: Output Size Estimation of Vector/Matrix-Based Implementation of Gram Matrix Computation.
(The Column number of X is 1000)

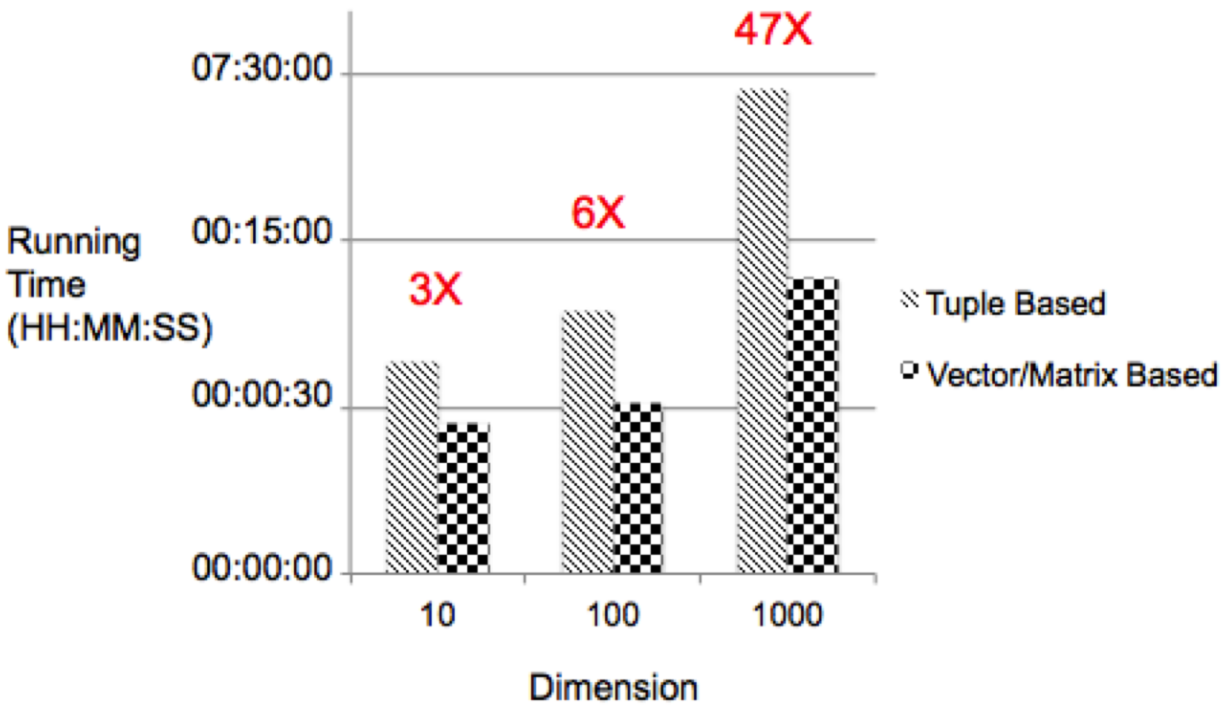

Figure 5.5: Results of Gram Matrix Computation. Drawn in Log-Scale.

5.1.6 Experiment Analysis

The results show that the vector/matrix-based version is faster than the tuple-based version in all experiments. By analyzing the experimental log, we found that the join was the most time consuming operation. However, the system pipelined the aggregation at the end of the join. So we did a supplemental experiment to figure out where exactly the time was spent. The following query is written for the supplemental test. The matrix $\mathbf{X}$ has 5×10^5 rows and 1000 columns:

```
SELECT x1.col_index, x2.col_index, sum(x1.value * x2.value)
FROM x as x1, x as x2
WHERE x1.row_index = x2.row_index
and x1.row_index = x2.row_index + 600000
GROUP BY x1.col_index, x2.col_index;
```

In the query above, we add a new predicate in the WHERE clause: *x1.row_index = x2.row_index + 600000*. Because the max value of *x1.row* is 500000, this predicate will always be evaluated as false. Therefore, the new query should not conduct any aggregation after it finishes the join. Hence, we can get the pure running time for the join by running this query. The result shows that for the pure join operation, the running time is 10min 52sec. Thus, the pure aggregation time can be obtained by subtracting the pure join time from the overall time of join and aggregation, which is 5hour 15min 3sec. With this supplemental experiment, we can conclude that the reason why the vector/matrix-version is faster is that it avoids aggregation on a large number of records.

The pure running time for each operation is listed in Table 5.2, and they are drawn in Figure 5.6.

Also, the experiment results in Table 5.1 show that when the dimension size of

Gram Matrix Computation ($\mathbf{X}$ with 1000 columns)			
Operation	Join	Aggregate	Other Operations
Tuple Based Version	00:10:52	05:15:03	00:00:33
Vector/Matrix Based Version	00:00:00	00:06:38	00:00:20

Table 5.2: Running Time of Each Operation for Gram Matrix Computation. Format is HH:MM:SS.

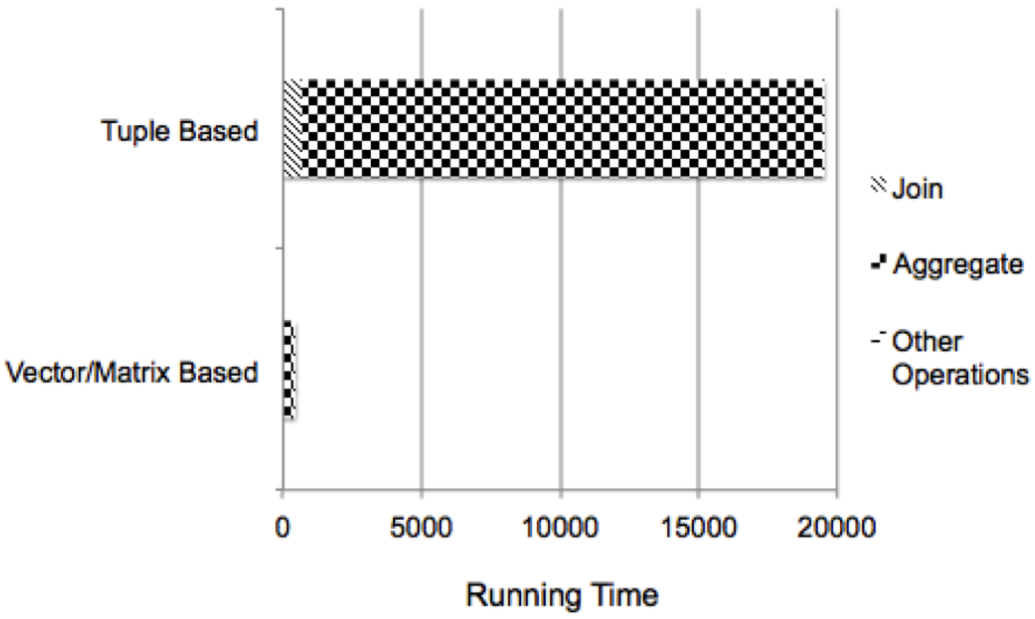

Figure 5.6: Operation Time Analysis of Gram Matrix Computation. ($\mathbf{X}$ with 1000 columns)

the matrix $\mathbf{X}$ increases, the speed gain of the vector/matrix version becomes more significant. The reason is that when the column number increases, the number of tuples in the tuple-based version will also increase on the same scale as the column number changes. However, in the vector/matrix version, the number of tuples will stay constant. Hence, the vector/matrix version will show more speed gains when the data size becomes larger.

5.2 Linear Regression

This section considers the linear regression model. I have described the definition and implementation of this model in Chapter 3. In this section, I will discuss how I conduct the experiment on the this model, and give some findings obtained from the experiment.

5.2.1 Experiment Setup

The linear regression experiment is conducted in a similar manner as the Gram matrix experiment. The row number of the matrix $\mathbf{X}$ is kept constant as 5×10^5, and its column number is changed from 10, 100 to 1000. The data in the matrix $\mathbf{X}$ is generated with a pre-defined mean and covariance. Then, this matrix, together with a random $\boldsymbol{\beta}$ and a random $\boldsymbol{\epsilon}$, is employed to generate the vector $\mathbf{y}$. I run my experiment on the SimSQL v0.4 on a 5-machine Amazon EC2 cluster, with machine type c3.2xlarge.

5.2.2 Query Plan and Output Estimation

The query plan of the tuple-based implementation is depicted in Figure 5.7, and the output size estimation of each job is shown in Figure 5.8. The query plan of the

vector/matrix-based implementation is depicted in Figure 5.9, and the output size estimation of each job is shown in Figure 5.10

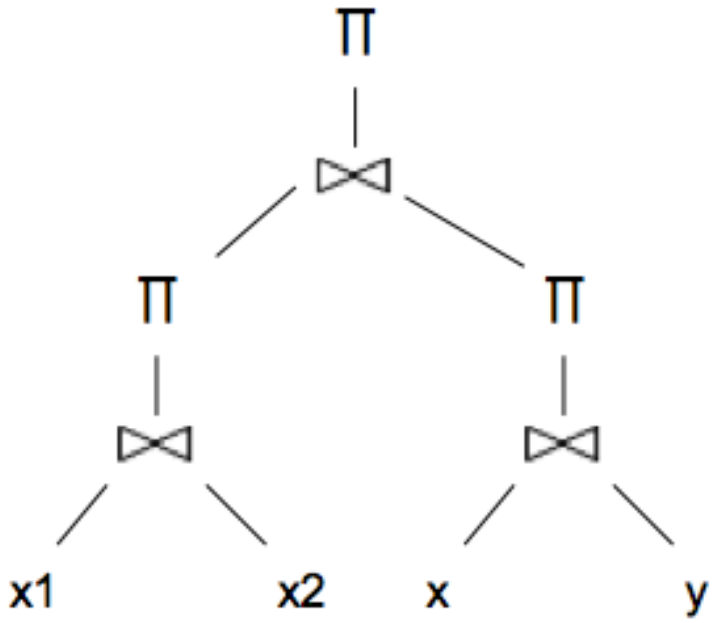

Figure 5.7: Query Plan of Tuple-Based Implementation of Linear Regression.

5.2.3 Experiment Result

The experimental results are given in Table 5.3, and they are depicted in Figure 5.11. The numbers on the top of bars in Figure 5.11 represent how many times the tuple-based version is slower than the vector/matrix-based version.

Linear Regression			
Column numbers of **X**	10	100	1000
Tuple Based Version	00:03:22	00:05:52	5:14:59
Vector/Matrix Based Version	00:00:28	00:00:43	00:06:03

Table 5.3: Running Time of Linear Regression Estimation. Format is HH:MM:SS.

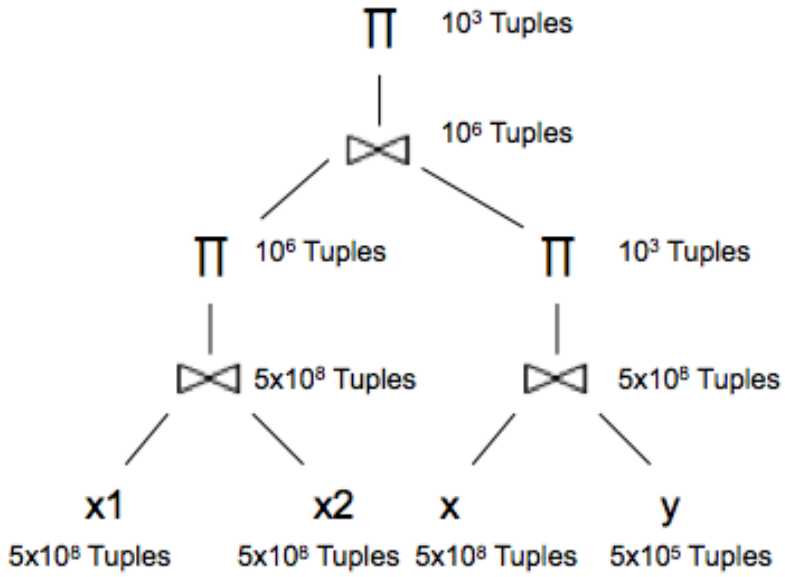

Figure 5.8: Output Size Estimation of Tuple-Based Implementation of Linear Regression.
(The Column number of X is 1000)

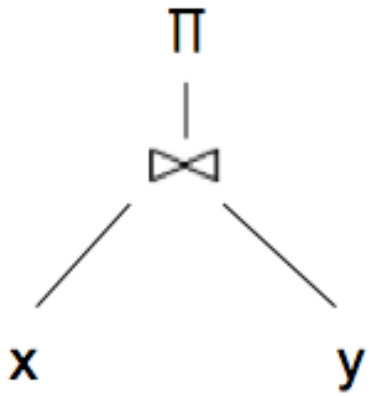

Figure 5.9: Query Plan of Vector/Matrix-Based Implementation of Linear Regression.

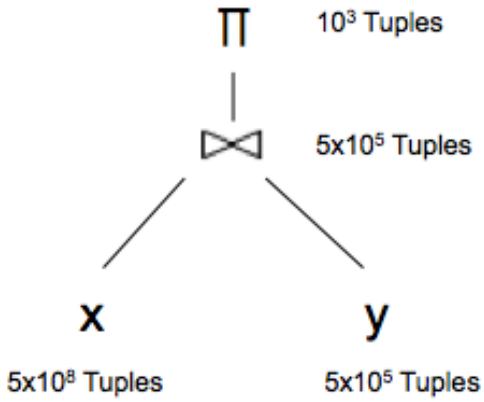

Figure 5.10: Output Size Estimation of Vector/Matrix-Based Implementation of Linear Regression.
(The Column number of X is 1000)

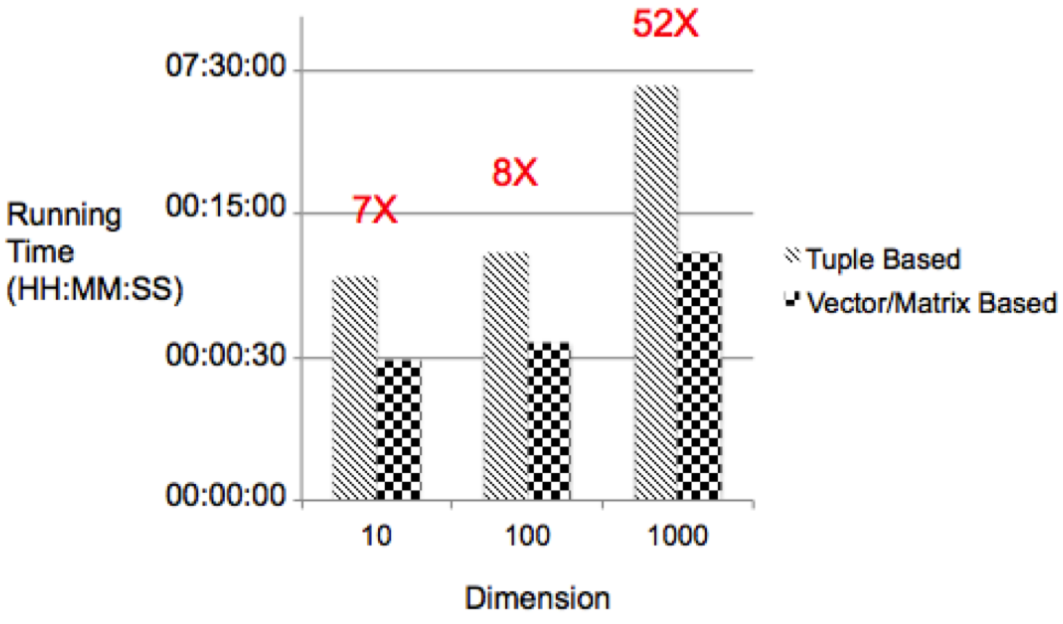

Figure 5.11: Results of Linear Regression Estimation. Drawn in Log-Scale.

5.2.4 Experiment Analysis

The results for the linear regression estimation are very similar to the Gram matrix computation. The vector/matrix-based version is faster than the tuple-based version in all experiments. For the tuple-based version, the calculation of the table *xtx_inv* is the most time-consuming job, because this job requires join between big tables, as well as aggregation on a large number of records. However, the vector/matrix-based version avoids such task. So it can be much faster than the tuple-based one. Also, with the dimension size of the matrix $\mathbf{X}$ increasing, the speed advantage of the vector/matrix version becomes more pronounced, for reasons similar to those in the Gram matrix computation.

5.3 Machine Learning Models

This section considers four more *real* and *complicated* models: Bayesian Lasso (Lasso), Gaussian Mixture Model (GMM), Latent Dirichlet Allocation (LDA), and Hidden Markov Model (HMM). They are widely used machine learning models, and are way more complicated than the previous two models. Similar to the previous two sections, I write two versions SimSQL code for each model: a pure tuple-based version and a vector/matrix-based version. We would like to see in more complicated applications, how these two implementations perform differently from each other, and obtain some insights from this comparison. I use Markov Chain Monte Carlo (MCMC) and Gibbs sampling to learn these models.

The next several sections are organized as follows. First, I give the mathematical definition of each model, and describe how to use MCMC and Gibbs sampling to learn each model. Then, I present how I conduct the experiment for each model. Lastly, the experiment results are provided. I will analyze the results in deep, and give some

insights from the experiments. As the page limits, I will not provide the full SimSQL implementations of these models in this thesis.

5.3.1 Model Description

This section provides the mathematical definitions as well as the MCMC simulation processes for the four models.

5.3.1.1 Bayesian Lasso

The Bayesian Lasso is a Bayesian version of Lasso, which is a regularized regression model [10]. This model can be defined as follows: Given n p-dimension regressor $\mathbf{x}$ and n centered response $\tilde{y}$, the Bayesian Lasso model proposes that $\mathbf{x}$ and $\tilde{y}$ satisfies this relationship: $\tilde{y} \sim \text{Normal}(\beta \cdot \mathbf{x}, \sigma^2)$, where β is the regression coefficients, and σ is the error item.

If we use a vector $\tilde{\mathbf{y}}$ to represent the centered responses, a matrix $\mathbf{X}$ to denote the regressors, and p auxiliary variables $\tau_1^2, \tau_2^2, ...\tau_p^2$ to control the variance of the regression coefficients, the MCMC process for learning the Bayesian Lasso can be described with the following formula (Referring to [64]):

$$
\begin{aligned}
1/(\tau_j^{(i)})^2 \sim & \text{InvGaussian}(\sqrt{\frac{\lambda^2(\sigma^{(i-1)})^2}{(\beta_j^{(i-1)})^2}}, \lambda^2) \\
\beta^{(i)} \sim & \text{Normal}\left((\mathbf{A}^{(i)})^{-1}\mathbf{X}^T\tilde{\mathbf{y}}, \sigma^2(\mathbf{A}^{(i)})^{-1}\right), \text{ where} \\
& \mathbf{A}^{(i)} = \mathbf{X}^T\mathbf{X} + (\mathbf{D}_\tau^{(i)})^{-1} \text{ and} \\
& \mathbf{D}_\tau^{(i)} = diag\left((\tau_1^{(i)})^2, (\tau_2^{(i)})^2, ...(\tau_p^{(i)})^2\right) \\
(\sigma^{(i)})^2 \sim & \text{InvGamma}(\frac{1+n+p}{2}, \\
& \frac{2 + \sum_{\langle \mathbf{x}, y\rangle \in D}\left(\tilde{y} - \beta^{(i)} \cdot \mathbf{x}\right)^2 + \sum_j (\beta_j^{(i)})^2/(\tau_j^{(i)})^2}{2})
\end{aligned}
$$

5.3.1.2 Gaussian Mixture Model

The definition and MCMC simulation for Gaussian Mixture model have been described in Chapter 3, so I will omit them here.

5.3.1.3 Latent Dirichlet Allocation

Latent Dirichlet Allocation is a classic model for text mining [11]. It assumes that in a document j, the kth word $w_{j,k}$ has topic t, with the probability $\Pr[w_{j,k} = \omega] = \phi_{t,\omega}$, where ϕ_t is the word distribution of topic t. Each document also has a topic distribution. Thus, a document j has topic t for the kth word with the probability $\Pr[z_{j,k} = t] = \theta_{j,t}$, where θ_j is the topic distribution of document j. The model also assigns a Dirichlet(β) prior for ϕ_t and a Dirichlet(α) prior for θ_j.

To learn this model, we introduce two new functions: $f(j,t) = \sum_k one(z_{j,k}^{(i)} = t)$, and $g(t,\omega) = \sum_{j,k} one(w_{j,k} = \omega \text{ and } z_{j,k}^{(i)} = t)$, where $one()$ returns 1 if the Boolean argument is true and 0 otherwise. Then the MCMC simulation of LDA can be described as follows (Referring to [64]):

$$\Pr[z_{j,k}^{(i)} = t] \propto \theta_{j,t}^{(i-1)} \times \phi_{t,w_{j,k}}^{(i-1)}$$

$$\theta_j^{(i)} \sim \text{Dirichlet}(\alpha + \langle f(j,1), f(j,2), f(j,3), ...\rangle)$$

$$\phi_t^{(i)} \sim \text{Dirichlet}(\beta + \langle g(t,1), g(t,2), g(t,3), ...\rangle)$$

5.3.1.4 Hidden Markov Model

Hidden Markov Model is used to model a Markov process with hidden states [12]. In our experiment, we will apply HMM to identify the topics for words (i.e., the hidden states) in a list of documents. More specifically, this model assumes that the probability of a topic s producing a word w is $\Psi_{s,w}$, where Ψ_s is the emission distribution of topic s. Also, each topic s has the probability of $\delta_{s,s'}$ to transfer from topic s to topic s', where δ_s is the transition distribution of topic s. There are also a Dirichlet(β) prior on each Ψ_s, and a Dirichlet(α) prior on each δ_s.

If we use a vector $\mathbf{y}_j$ to store the topics in document j, and use a vector $\mathbf{x}_j$ to store the words in document j, then according to HMM, these variables are generated in the following way: the first topic $y_{j,1}$ is sampled from a Categorical distribution Categorical(δ_0); For each k in $2...n$, the topic $y_{j,k+1}$ is sampled with the probability $\Pr[y_{j,k+1}|y_{j,k}] = \delta_{y_{j,k},y_{j,k+1}}$. Each word $x_{j,k}$ is generated by sampling from a Categorical distribution Categorical($\Psi_{y_{j,k}}$).

In the MCMC simulation for learning HMM, the core part is to update the topic assignment for each word. We use the following formula to update the topic $y_{j,k}$ for the word $x_{j,k}$ in iteration i (Referring to [64]):

$$\begin{aligned}\Pr[y_{j,k}^{(i)} = s] &\propto \delta_{0,s}^{(i-1)} \times \Psi_{s,x_{j,k}}^{(i-1)} \times \delta_{s,y_{j,k+1}^{(i-1)}}^{(i-1)} \text{ if } k = 1 \\ &\propto \delta_{y_{j,k-1}^{(i-1)},s}^{(i-1)} \times \Psi_{s,x_{j,k}}^{(i-1)} \text{ if } k \text{ ends the document} \\ &\propto \delta_{y_{j,k-1}^{(i-1)},s}^{(i-1)} \times \Psi_{s,x_{j,k}}^{(i-1)} \times \delta_{s,y_{j,k+1}^{(i-1)}}^{(i-1)} \text{ otherwise.}\end{aligned}$$

To update other parameters, three new functions $f(w, s)$, $g(s)$ and $h(s, s')$ are employed to simplify the representation. Their definitions are as follows: $f(w, s) = \sum_{j,k} one(x_{j,k} = w \text{ and } y_{j,k}^{(i)} = s)$; $g(s) = \sum_j one(y_{j,0}^{(i)} = s)$; $h(s, s') = \sum_{j,k} one(y_{j,k}^{(i)} = s \text{ and } y_{j,k+1}^{(i)} = s')$ ($one()$ returns 1 if the Boolean argument is true and 0 otherwise). With these functions, the updates of the remaining parameters are given as follows (Referring to [64]):

$$\begin{aligned}\Psi_s^{(i)} &\sim \text{Dirichlet}(\beta + \langle f(1, s), f(2, s), f(3, s)...\rangle) \\ \delta_0^{(i)} &\sim \text{Dirichlet}(\alpha + \langle g(1), g(2), g(3)...\rangle) \\ \delta_s^{(i)} &\sim \text{Dirichlet}(\alpha + \langle h(s, 1), h(s, 2), h(s, 3)...\rangle)\end{aligned}$$

5.3.2 Experiment Setup

We have implemented two versions code for each learning algorithm: a pure tuple-based version and a vector/matrix version. Both versions are tested on a 5-machine Amazon EC2 cluster. The machine's type is m2.4xlarge, with 8 CPU cores, 68 GB memory, and 2×840 GB disk. We use SimSQL v0.4 as our experiment platform. The data we use to learn each model is given in Table 5.4. The process to generate these datasets is discussed in [64].

Model	Data Specification
Bayesian Lasso	Number of (response, regressor) pairs: 5×10^5 Dimension of response: 1 Dimension of regressor: 1000
Gaussian Mixture Model	Number of data points: 5×10^7 Dimension of each data point: 10 Number of clusters: 10
Latent Dirichlet Allocation	Number of documents: 1.25×10^7 Number of topics: 100 Dictionary size: 10^4
Hidden Markov Model	Number of documents: 1.25×10^7 Number of topics: 20 Dictionary size: 10^4

Table 5.4: Data for Experiments of Four Machine Learning Models.

5.3.3 Experiment Result

We run each piece of code for six iterations. The first iteration is the initialization iteration, which assigns the initial values for the parameters and triggers the simulation. The other five iterations imitate the actual simulation process, and calculate the actual values for the parameters. The average running time of these five iterations is used to measure the actual running time for learning the model.

The experiments results are listed out in Table 5.5, and are depicted in Figure 5.12 and Figure 5.13, which display the initialization time and running time separately. The numbers on the top of bars represent how many times the tuple-based version is slower than the vector/matrix-based version.

Machine Learning Model Experiment				
Model	Lasso	GMM	LDA	HMM
Tuple Based	00:07:28 (02:38:46)	00:06:39 (00:13:08)	04:12:37 (03:59:26)	03:36:47 (00:17:51)
Vector/Matrix Based	00:04:04 (00:04:44)	00:09:15 (00:08:09)	00:29:28 (00:01:13)	00:28:17 (00:26:28)

Table 5.5: Running Time of Four Machine Learning Models. Time in Parens is for the Initialization Iteration. Format is HH:MM:SS.

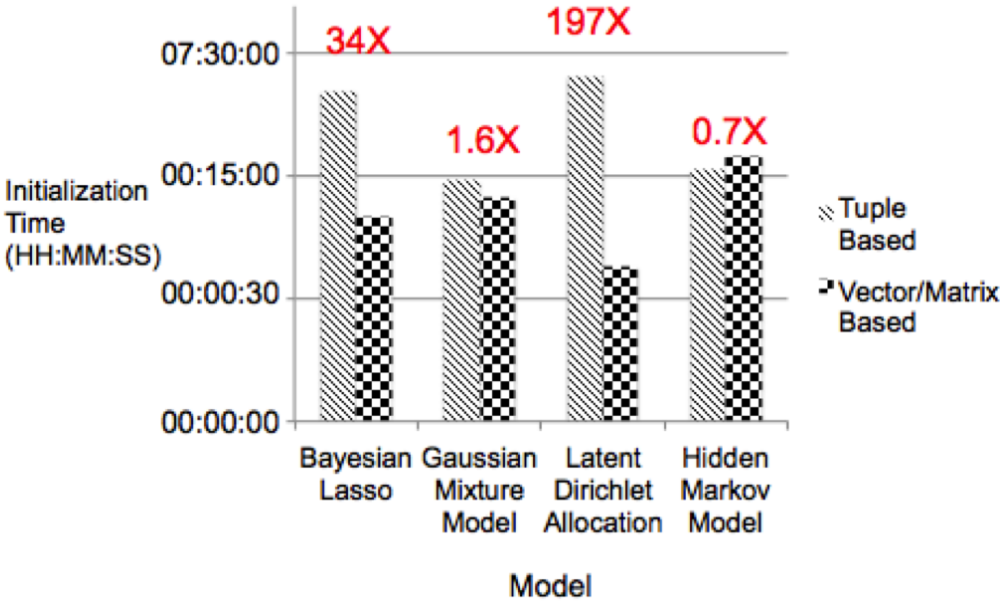

Figure 5.12: Initialization Time of Four Machine Learning Models. Drawn in Log-Scale.

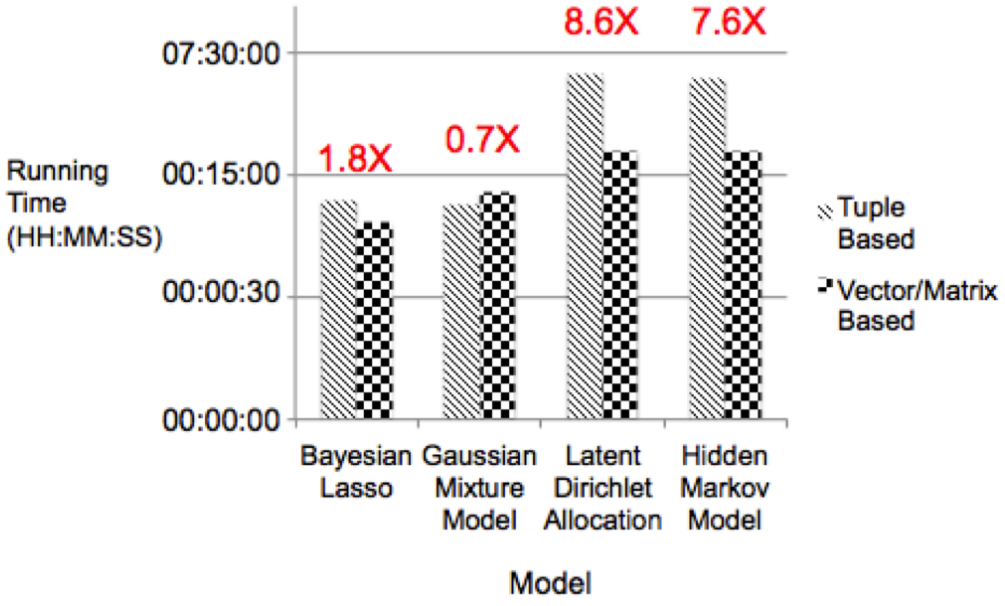

Figure 5.13: Running Time of Four Machine Learning Models. Drawn in Log-Scale.

5.3.4 Experiment Analysis

Compared with the pure-tuple based implementation, the vector/matrix-based implementation is more straightforward to write and easier to read. The experiment results show that the vector/matrix-based version is faster than the tuple-based version in almost all experiments. The only exception is the Gaussian Mixture Model, where the running time of the vector/matrix-based version is about 1.5 times slower than the tuple-based version. The reason is that the tuple-based version is optimized to process the data in a block-based manner. It groups data into a set of blocks, and conducts computation on these blocks in parallel. Such optimization can increase the data locality, and it provides more pipeline opportunities. In the future, we can also apply such optimization to the vector/matrix-based version, and check if this version can also benefit from this optimization.

For Bayesian Lasso, a notable result is that the vector/matrix-based version is 34 times faster than the tuple-based version for the initialization time. Such time difference comes from the Gram matrix computation for all data points in the initialization phase. What's more, the result shows that the initialization time is much more than the running time. Because Bayesian Lasso usually converges within several iterations, the reduction of the initialization time is important, and it is helpful to reduce the whole running time.

For LDA and HMM, the results are similar. The running time of the vector/matrix-based version is about eight times faster than the tuple-based version. The reason is that it is very natural to use vectors and matrices to store the variables in these two models, and such storage can avoid join and aggregation on a large number of tuples, compared with the pure tuple-based implementation. For example, in LDA, we store each document as a vector of words, and use two matrices to represent the doc-topic distribution and the topic-word distribution. In HMM, we also store each

document as a vector, and employ two matrices to represent the topic-word emission distribution and the topic-topic transition distribution. Such vector/matrix-based representation makes the implementation of these two models straightforward and efficient.

Chapter 6

Conclusion

In this thesis, I have discussed how to add vector and matrix support to a relational database system (SimSQL) to make linear algebra operations easy and efficient to use in this system. First, I explained the motivation for this study, and described my solution to provide vector and matrix support to a database system: add vector and matrix data types to the system. Second, I reviewed how current database systems handled vectors and matrices, and discussed possible problems in their solutions. Third, I detailed how to define and use vector and matrix data types as a SimSQL user, and how to conduct constructions and deconstructions for vectors and matrices. Fourth, I described the vector and matrix data types from a SimSQL developer's perspective, including how to write functions with vector and matrix parameters, how to infer type and size information for vector and matrix attributes, how to utilize such information to help the query optimization, and how I implemented the serialization and deserialization for vector and matrix attributes. Finally, my colleague and I implemented our ideas in SimSQL, and I conducted some experiments to verify the efficacy of our implementation. The experiment results showed that the introduction of vectors and matrices not only made the code simpler but also brought a significant performance improvement for SimSQL, compared with the pure

tuple-based implementation.

Many topics can be investigated based on this work. For example, with the vector and matrix data structure, it becomes feasible to integrate other languages into SimSQL, such as Matlab, R, Python, etc. Such integration can increase the usability of SimSQL. Moreover, with the semantic and size information of linear algebra operations, the optimizer can explore more optimization opportunities, and generate a more efficient plan for the quires. As a SimSQL user or developer, he/she will have more flexibility and ease in writing code in SimSQL, and bring SimSQL to a wider range of applications. Finally, more experiments can be conducted on SimSQL. We can test it on more models, and explore more applications. We can also compare SimSQL with other systems, such as RDBMSs with array support, array database systems, or even analytic, cluster computing systems such as Spark [65] and SystemML [66]. This comparison can reveal the advantages and disadvantages of different solutions, and provides the directions to improve SimSQL further. After all, how to integrate vectors and matrices into a database system is still an open question, and it deserves us to have more discussions and studies to obtain more insights into it.

Bibliography

[1] E. F. Codd, "A relational model of data for large shared data banks," *Commun. ACM*, vol. 13, no. 6, pp. 377–387, Jun. 1970. [Online]. Available: http://doi.acm.org/10.1145/362384.362685 1.1, 2.2

[2] S. Sumathi and S. Esakkirajan, *Fundamentals of Relational Database Management Systems (Studies in Computational Intelligence)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007. 1.1

[3] IBM Knowledge Center, "Structured Query Language (SQL)," 2006, [Accessed 14-December-2015]. [Online]. Available: http://www-01.ibm.com/support/knowledgecenter/SSEPGG_9.1.0/com.ibm.db2.udb.admin.doc/doc/c0004100.htm 1.1

[4] J. Melton, *Advanced SQL, 1999: Understanding Object-relational and Other Advanced Features*, ser. The Morgan Kaufmann series in data management systems. Morgan Kaufmann Pub., 2003. [Online]. Available: https://books.google.com/books?id=QRtDyDU9g2YC 1.1

[5] Wikipedia, "Vector (mathematics and physics) — wikipedia, the free encyclopedia," 2015, [Online; accessed 7-January-2016]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Vector_(mathematics_and_physics)&oldid=690380646 1.1

[6] Wikipedia, "Matrix (mathematics) — wikipedia, the free encyclopedia," 2016, [Online; accessed 7-January-2016]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Matrix_(mathematics)&oldid=698298736 1.1

[7] Z. Cai, Z. Vagena, L. Perez, S. Arumugam, P. J. Haas, and C. Jermaine, "Simulation of database-valued markov chains using simsql," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. New York, NY, USA: ACM, 2013, pp. 637–648. [Online]. Available: http://doi.acm.org/10.1145/2463676.2465283 1.2

[8] SimSQL Home Page, "Simsql overview," 2016, [Accessed 08-January-2016]. [Online]. Available: http://cmj4.web.rice.edu/SimSQL/SimSQL.html 1.2

[9] D. A. Reynolds, "Gaussian mixture models," in *Encyclopedia of Biometrics*, 2009, pp. 659–663. [Online]. Available: http://dx.doi.org/10.1007/978-0-387-73003-5_196 1.2, 3.4

[10] T. Park and G. Casella, "The bayesian lasso," *Journal of the American Statistical Association*, vol. 103, no. 482, pp. 681–686, 2008. [Online]. Available: http://dx.doi.org/10.1198/016214508000000337 1.2, 5.3.1.1

[11] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *J. Mach. Learn. Res.*, vol. 3, pp. 993–1022, Mar. 2003. [Online]. Available: http://dl.acm.org/citation.cfm?id=944919.944937 1.2, 5.3.1.3

[12] Z. Ghahramani, "Hidden markov models." River Edge, NJ, USA: World Scientific Publishing Co., Inc., 2002, ch. An Introduction to Hidden Markov Models and Bayesian Networks, pp. 9–42. [Online]. Available: http://dl.acm.org/citation.cfm?id=505741.505743 1.2, 5.3.1.4

[13] ISO/IEC 9075-1:2008, "Information technology — Database languages — SQL — Part 1: Framework," International Organization for Standardization, Geneva, Switzerland, 2008. 2.1

[14] ISO/IEC 9075-2:1999, "Information technology — Database languages — SQL — Part 2: Foundation," International Organization for Standardization, Geneva, Switzerland, 1999. 2.1

[15] ISO/IEC 9075-2:2003, "Information technology — Database languages — SQL — Part 2: Foundation," International Organization for Standardization, Geneva, Switzerland, 2003. 2.1

[16] ISO/IEC 9075-2:2008, "Information technology — Database languages — SQL — Part 2: Foundation," International Organization for Standardization, Geneva, Switzerland, 2008. 2.1

[17] ISO/IEC 9075-2:2011, "Information technology — Database languages — SQL — Part 2: Foundation," International Organization for Standardization, Geneva, Switzerland, 2011. 2.1

[18] G. Leopold, "Big Data Forcing Update of SQL Standard," 2014, [Accessed 16-December-2015]. [Online]. Available: http://www.datanami.com/2014/06/25/big-data-forcing-update-sql-standard/ 2.1

[19] R. Chirgwin, "SQL fights back against NoSQL's big data cred with SQL/MDA spec," 2014, [Accessed 16-December-2015]. [Online]. Available: http://www.theregister.co.uk/2014/06/26/sql_to_worlddog_we_doing_big_data_too/ 2.1

[20] M. Widenius and D. Axmark, *Mysql Reference Manual*, 1st ed., P. DuBois, Ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2002. 2.2

[21] MySQL Home Page, 2015, [Accessed 28-December-2015]. [Online]. Available: https://www.mysql.com/ 2.2

[22] Microsoft SQL Server Home Page, 2015, [Accessed 28-December-2015]. [Online]. Available: http://www.microsoft.com/en-us/server-cloud/products/sql-server/ 2.2

[23] SQLite Home Page, 2015, [Accessed 17-December-2015]. [Online]. Available: https://www.sqlite.org/index.html 2.2

[24] Wikipedia, "Binary large object — wikipedia, the free encyclopedia," 2015, [Accessed 29-December-2015]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Binary_large_object&oldid=682123194 2.2

[25] L. Dobos, A. Szalay, J. Blakeley, T. Budavári, I. Csabai, D. Tomic, M. Milovanovic, M. Tintor, and A. Jovanovic, "Array requirements for scientific applications and an implementation for microsoft sql server," in *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, ser. AD '11. New York, NY, USA: ACM, 2011, pp. 13–19. [Online]. Available: http://doi.acm.org/10.1145/1966895.1966897 2.2

[26] Oracle Database 12c Home Page, "Database 12c — oracle," 2015, [Accessed 29-December-2015]. [Online]. Available: https://www.oracle.com/database/index.html 2.2

[27] R. Greenwald, R. Stackowiak, and J. Stern, *Oracle essentials*. O'Reilly, 2013. 2.2

[28] IBM DB2 Home Page, "Ibm - db2 database software," 2015, [Accessed 29-December-2015]. [Online]. Available: http://www-01.ibm.com/software/data/db2/ 2.2

[29] R. F. Chong, *Understanding DB2*. IBM Press/Pearson, 2005. 2.2

[30] PostgreSQL Home Page, "Postgresql: The world's most advanced open source database," 2015, [Accessed 29-December-2015]. [Online]. Available: http://www.postgresql.org/ 2.2

[31] M. Stonebraker and L. A. Rowe, "The design of postgres," in *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '86. New York, NY, USA: ACM, 1986, pp. 340–355. [Online]. Available: http://doi.acm.org/10.1145/16894.16888 2.2

[32] "Oracle holds #1 dbms market share worldwide for 2013," [Accessed 31-December-2015]. [Online]. Available: http://www.oracle.com/us/corporate/features/number-one-database/index.html 2.2

[33] PL/SQL User's Guide and Reference Release 2 (9.2), "Pl/sql collections and records," 2015, [Accessed 31-December-2015]. [Online]. Available: https://docs.oracle.com/cd/B10501_01/appdev.920/a96624/05_colls.htm 2.2

[34] Oracle Help Center, "Database pl/sql packages and types reference," 2015, [Accessed 31-December-2015]. [Online]. Available: http://docs.oracle.com/cd/B19306_01/appdev.102/b14258/u_nla.htm#CIABEFIJ 2.2

[35] Netlib Repository, "Blas (basic linear algebra subprograms)," 2015, [Accessed 31-December-2015]. [Online]. Available: http://www.netlib.org/blas/ 2.2

[36] Netlib Repository, "Lapack - linear algebra package," 2015, [Accessed 31-December-2015]. [Online]. Available: http://www.netlib.org/lapack/ 2.2

[37] M. Stonebraker and L. A. Rowe, "The design of postgres," in *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '86. New York, NY, USA: ACM, 1986, pp. 340–355. [Online]. Available: http://doi.acm.org/10.1145/16894.16888 2.2

[38] M. Stonebraker and G. Kemnitz, "The postgres next generation database management system," *Commun. ACM*, vol. 34, no. 10, pp. 78–92, Oct. 1991. [Online]. Available: http://doi.acm.org/10.1145/125223.125262 2.2

[39] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar, "The madlib analytics library: Or mad skills, the sql," *Proc. VLDB Endow.*, vol. 5, no. 12, pp. 1700–1711, Aug. 2012. [Online]. Available: http://dx.doi.org/10.14778/2367502.2367510 2.2

[40] M. Jarke and J. Koch, "Query optimization in database systems," *ACM Comput. Surv.*, vol. 16, no. 2, pp. 111–152, Jun. 1984. [Online]. Available: http://doi.acm.org/10.1145/356924.356928 2.2

[41] P. Baumann, "Management of multidimensional discrete data," *The VLDB Journal*, vol. 3, no. 4, pp. 401–444, Oct. 1994. [Online]. Available: http://dl.acm.org/citation.cfm?id=615204.615207 2.3

[42] P. Baumann, "A database array algebra for spatio-temporal data and beyond," in *In Next Generation Information Technologies and Systems*, 1999, pp. 76–93. 2.3

[43] A. G. Gutierrez and P. Baumann, "Modeling fundamental geo-raster operations with array algebra," in *Data Mining Workshops, 2007. ICDM Workshops 2007. Seventh IEEE International Conference on*, Oct 2007, pp. 607–612. 2.3

[44] Rasdaman wiki, "Welcome to rasdaman – the world's leading array database," 2016, [Accessed 04-January-2016]. [Online]. Available: http://www.rasdaman.org/wiki 2.3

[45] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann, "The multidimensional database system rasdaman," in *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '98. New York, NY, USA: ACM, 1998, pp. 575–577. [Online]. Available: http://doi.acm.org/10.1145/276304.276386 2.3

[46] M. Stonebraker, J. Becla, D. J. DeWitt, K. Lim, D. Maier, O. Ratzesberger, and S. B. Zdonik, "Requirements for science data bases and scidb," in *CIDR 2009, Fourth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2009, Online Proceedings*, 2009. 2.3

[47] P. G. Brown, "Overview of scidb: Large scale array storage, processing and analysis," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 963–968. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807271 2.3

[48] MonetDB Home Page, "The column-store pioneer," 2016, [Accessed 05-January-2016]. [Online]. Available: https://www.monetdb.org/Home 2.3

[49] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten, "Monetdb: Two decades of research in column-oriented database architectures," *IEEE Data Eng. Bull.*, vol. 35, no. 1, pp. 40–45, 2012. 2.3

[50] P. Brown, "Big data and big analytics: Scidb is not hadoop," 2016, [Accessed 05-January-2016]. [Online]. Available: http://conferences.oreilly.com/strata/stratany2011/public/schedule/detail/21376 2.3

[51] D. J. Abadi, P. A. Boncz, and S. Harizopoulos, "Column-oriented database systems," *Proc. VLDB Endow.*, vol. 2, no. 2, pp. 1664–1665, Aug. 2009. [Online]. Available: http://dx.doi.org/10.14778/1687553.1687625 2.3

[52] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik, "C-store: A column-oriented dbms," in *Proceedings of the 31st International Conference on Very Large Data Bases*, ser. VLDB '05. VLDB Endowment, 2005, pp. 553–564. [Online]. Available: http://dl.acm.org/citation.cfm?id=1083592.1083658 2.3

[53] M. Kersten, Y. Zhang, M. Ivanova, and N. Nes, "Sciql, a query language for science applications," in *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, ser. AD '11. New York, NY, USA: ACM, 2011, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/1966895.1966896 2.3

[54] C. Ordonez and J. García-García, "Vector and matrix operations programmed with udfs in a relational dbms," in *Proceedings of the 15th ACM International Conference on Information and Knowledge Management*, ser. CIKM '06. New York, NY, USA: ACM, 2006, pp. 503–512. [Online]. Available: http://doi.acm.org/10.1145/1183614.1183687 2.4

[55] C. Ordonez, "Building statistical models and scoring with udfs," in *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '07. New York, NY, USA: ACM, 2007, pp. 1005–1016. [Online]. Available: http://doi.acm.org/10.1145/1247480.1247599 2.4

[56] C. Ordonez, "Statistical model computation with udfs," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 22, no. 12, pp. 1752–1765, Dec 2010. 2.4

[57] D. Kernert, F. Köhler, and W. Lehner, "Bringing linear algebra objects to life in a column-oriented in-memory database," in *Proceedings of the 1st International Workshop on In Memory Data Management and Analytics, IMDM 2013, Riva Del Garda, Italy, August 26, 2013.*, 2013, pp. 37–49. [Online]. Available: http://www-db.in.tum.de/other/imdm2013/papers/Kernert.pdf 2.4

[58] D. Kernert, F. Köhler, and W. Lehner, "Slacid - sparse linear algebra in a column-oriented in-memory database system," in *Proceedings of the 26th International Conference on Scientific and Statistical Database Management*, ser. SSDBM '14. New York, NY, USA: ACM, 2014, pp. 11:1–11:12. [Online]. Available: http://doi.acm.org/10.1145/2618243.2618254 2.4

[59] N. R. Draper, H. Smith, and E. Pownell, *Applied regression analysis*. Wiley New York, 1966, vol. 3. 3.1.1

[60] H. Sorenson, "Least-squares estimation: from gauss to kalman," *Spectrum, IEEE*, vol. 7, no. 7, pp. 63–68, July 1970. 3.1.1

[61] Scikit-learn, "Gaussian mixture models," 2015, [Accessed 14-December-2015]. [Online]. Available: http://scikit-learn.org/stable/modules/mixture.html 3.4

[62] C. Andrieu, N. de Freitas, A. Doucet, and M. Jordan, "An introduction to mcmc for machine learning," *Machine Learning*, vol. 50, no. 1-2, pp. 5–43, 2003. [Online]. Available: http://dx.doi.org/10.1023/A%3A1020281327116 3.4

[63] S. Geman and D. Geman, "Stochastic relaxation, gibbs distributions, and the bayesian restoration of images," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 6, no. 6, pp. 721–741, Nov. 1984. [Online]. Available: http://dx.doi.org/10.1109/TPAMI.1984.4767596 3.4

[64] Z. Cai, Z. J. Gao, S. Luo, L. L. Perez, Z. Vagena, and C. Jermaine, "A comparison of platforms for implementing and running very large scale

machine learning algorithms," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14. New York, NY, USA: ACM, 2014, pp. 1371–1382. [Online]. Available: http://doi.acm.org/10.1145/2588555.2593680 3.4, 5.3.1.1, 5.3.1.3, 5.3.1.4, 5.3.2

[65] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863103.1863113 6

[66] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan, "Systemml: Declarative machine learning on mapreduce," in *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ser. ICDE '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 231–242. [Online]. Available: http://dx.doi.org/10.1109/ICDE.2011.5767930 6