Make Accumulated Data in Companies Eloquent by SQL Statement Constructors

Toshiyuki Shimono
DG Lab, Digital Garage
Tokyo, Japan
Email: shimono@dglab.com

***Abstract*—In recent years, many organizations have started to incorporate vast amounts of information into their databases as they transition into the era of Big Data. Though having this data available for analysis presents exciting opportunities, the way in which this data is made available to researchers sometimes leaves much to be desired. When databases are first designed, they often fail to account for the way in which the data will actually be analyzed later. It is very challenging and tedious for researchers to analyze poorly laid-out databases, which occasionally contain dozens of tables or hundreds of columns. In response to these challenges, we have created a new tool that enables researchers to understand the underlying database structure very quickly, so that they are able to extract useful information in short order. Our tool excels at building organized, sometimes very long SQL statements from a list of database tables and columns. From there, researchers can use our tool to quickly extract representative sample data from columns they wish to understand, extract row counts from target tables in one shot, and so on.**

***Keywords*-Knowing data contents; Data understanding; Data pre-processing; Data analysis; SQL queries;**

I. Introduction

Many organizations, ranging from private businesses to government and research organizations, are accumulating operational data into their database (DB) systems. However, analyzing this data (using human researchers) to produce meaningful results is challenging work. One of the largest reason is that these DBs are often not designed with future analysis in mind. Because qualified labor is limited, it is unusual for DBs to be well maintained by the organization unless they are critical for ordinary business operations.

Analyzing accumulated data presents many challenges. Firstly, researchers need to understand the overall structure of the new data. Then, they must select a meaningful sample of the data, and construct an environment suitable for analysis. One encounters in the real world the disorganized nature of the databases often makes overcoming these first steps very difficult. While [1] comprehensively deals in effectively performing big data projects from broad aspects, this paper presents a way to deal with this problem in a technical way. [2] and [3] handle data largely in text format, this paper focuses on data stored in an SQL-type data server.

A. Manipulating tables and columns by SQL

SQL (Structured Query Language) is widely used for maintaining and querying DBs, characterized by the concept of relational database management system [4]. Several vendors and open-source organizations provide SQL implementations more or less following the SQL standard (ISO/IEC 9075) [1]. A SQL-type DB contains tables described by its schema, which specifies column names and their data types (character, numeric, date, and so on). SQL statements obey SQL syntax. The **insert** sentence is issued to add a record (a row) into a table, and the **select** sentence is issued to see the contents of a table. For example, if we have a table named customer with a column named gender, we can count customers by gender with the SQL statement: **select gender, count(*) from customer group by gender**. The syntax is slightly different between providers and version of each SQL implementation, which is called SQL *dialect*.

B. Assumed DB environment

The following DB and analysis environment are assumed: (i) The DB is queried by issuing SQL statements (SQL sentences), (ii) A *comman-line interpreter*[1] is used to produce SQL statements and to store the results of SQL queries.

C. Disclaimer

Operational data provided by a credit card company is used to exhibit how the author's software works so concrete values such as DB names, table names, column names and data values are slightly changed or masked. The SQL sentences in this article are tried to be written to work generally on any SQL-type database systems but they are checked solely on Microsoft SQL Server 2008. For simplicity, the null value often used in SQL-type data is not cared well in this article unless it is especially necessary.

II. How to understand DB contents

In this section, we introduce what one would do on a given DB to know about the DB, following explicating why one needs to understand the DB before the analysis in earnest. To know about DB here contains knowing about each table, and knowing about each column of a table, knowing about the relation between columns as well.

The items (a), (b)..., (l) in this section are what one would want to retrieve from the DB. The SQL query sentences contained (a), (d), (e) and (g) possibly as well as (b) are

[1]The author like to suggest to use Unix or Linux with a programmable environment where Perl, Ruby and/or Python are also available.

trivial but the others are conceived by the author. Some of these items will be "integrated" in the next section § III.

A. Understanding an unfamiliar DB

Understanding a DB involves a number of steps, several of which seems not have computer-automated solutions essentially. Even if one is given documentation which ostensibly specifies the tables and columns within a DB, it may be incorrect or difficult to understand.

1) Effectiveness check for the analysis purpose
   One first must determine if the database contains the information/clues that one are looking for. One should also check for any invalid or malformed data.
2) Checking concrete data values
   One need to sample typical values and special/abnormal values. The name of a column alone most always does not clarify whether its values are substance names or encoded or encrypted values.
3) Handling column connections over tables
   Having multiple tables in a DB often indicates that there are connections between some columns by sharing the same coding systems. The ER (*entity-relation*) diagram clarifies these connections but is not always available.
4) For building a new environment for analysis
   There are several reasons to build a new environment. One may need to narrow the whole data, to combine or transform columns, or to avoid unintended information leakages.

Note that according to statistics today without sufficient human understanding of data, almost no meaningful insights can be derived; one should know the difficulty of cause-effect direction, confounding factors, multicollinearity problems, difficulties of time series analysis and so on. To be more fundamental, designing how to collect data is also critically important.

B. Knowing unfamiliar tables

(a) Counting the number of rows:
   select count(1) from table. Often the row number is a good memorable identifier of a table rather than the table name. It tends to be a small or large number if the table is a so-called *master* table or *transaction log* table, respectively.

(b) Random sampling of rows:
   select * from (select * , rand (cast (newid() as varbinary)) RandValue from TargetTable) T where RandValue < 1.0 * 20 / (select count(1) from TargetTable) [2] to take approximately 20 rows from *TargetTable*. Relevances among multiple columns can often easily be detected with human eyes by random sampling of rows in a table such as date order relations among date columns. Non-random sampling can easily cause misunderstanding in data.

(c) Row difference between two tables:
   There are good techniques for completely comparing the sameness regarding two files: cryptographic hash functions work well for this. We can also compare tables. Let us consider a function using the SHA-1 hash value of each row to the range of $[-\sqrt{3}, \sqrt{3}]$ whose output is uniformly distributed on different values, which has the average of 0 and the variance of 1. The sum of this function calculated by **select sum(-4294967296/2 + convert (bigint , convert (varbinary(4) , hashbytes ('sha1' , CombinedColumns + 'SomeCharStr')))) / 4294967296 * sqrt(12) from TargetTable** (the 10-digit numbers here are $2^{32} = 256^4$). While this acts as a

Table I
THE SAMPLED 4 VALUES FROM EACH COLUMN BY (f)

Col. num.	minLeft1min **s1.v11**	minLeft1max **s1.v12**	other_min **s2.v21**	other_max **s2.v22**
1	0000003e9..	0ffffcb27..	100001061..	fffffff2d..
2	000000d78..	0fffff1d5..	1000008fe..	fffffe7d0..
:	:	:	:	:
116	00000	00375	79901	79903
117	1992	1999	2000	2026
118	1	12	2	9
119			1	1
120			00000	79901
:	:	:	:	:
180	Aug 22 2013	Aug 31 2017	May 7 2013	Oct 11 2017
181	-1	-9990	0	9999
182	-1	-5	0	99
:	:	:	:	:
415	-32825000	-1	1	72342610
416	0	0	1	8
417	ABW	AZE	BA	ZZZ
418	1	1999999	2	8557773
419	10001	190002	20001	690021
:	:	:	:	:
435	0001-01-01	0001-01-01	2014-07-07	2017-06-06

Each row above was retrieved by **select * from (select min(C) v11, max(C) v12 from T where left(C,1) = (select min(left(C,1)) from T)) s1, (select min(C) v21 , max(C) v22 from T where left(C,1) != (select min(left(C,1)) from T)) s2** for table and column pair **T** and **C** (one can exchange **min** and **max**, **left** and **right**, the argument 1 into 2 or 3 in this SQL query). Note that the above values are mixture of numeric, character and date so the meaning of **min** and **max** change according to them.

[2]Some SQL implementations allow much simpler statements. On *Microsoft SQL Server 2008*, function from **rand** without the **newid** function does not give varying values, and **rand** following the **where** phrase is not functional.

Table II
A VARIATION OF TABLE I FOCUSING ON THE DIGIT LENGTHS

Seq.	minLen_min	minLen_max	other_min	other_max
1	0000003e9..	fffffff2d..	(null)	(null)
2	000000d78..	fffffe7d0..	(null)	(null)
:	:	:	:	:
116	00000	79903	(null)	(null)
117	1992	2026	(null)	(null)
118	1	9	10	12
119			1	1
120			00000	79901
:	:	:	:	:
180	May 7 2013	Aug 24 2017	(null)	(null)
181	0	9	-1	9999
182	0	9	-1	99
:	:	:	:	:
415	1	9	-32825000	72342610
416	0	8	(null)	(null)
417	AD	ZW	ABW	ZZZ
418	1	9	10	8557773
419	10001	90002	100001	690021
:	:	:	:	:
435	0001-01-01	2017-06-06	(null)	(null)

A variation of the previous. Each row above was retrieved by **select * from (select min(C) v11, max(C) v12 from T where len(C) = (select min(len(C)) from T)) s1, (select min(C) v21 , max(C) v22 from T where len(C,1) != (select min(len(C)) from T)) s2** for table and column pair **T** and **C**. **len** may be replaced by **length**. Another variation: **min(len(C))** can be replaced by **max(len(C))**, which is useful to observe the digit lengths of coded column values.

good fingerprint of the table, one can estimate the number of rows different between two tables by varying the **SomeCharStr** value using the statistical method of *estimating the popular variance*[3]. The detail is described in §APPENDIX-A.

C. Knowing unfamiliar columns of each table

Two of the following calculations, (d) and (e), may require a lot of memory because the computer has to store all of the different values it encounters. One should ensure that the system has enough memory. (f) is a good way to fulfill 1) and 2) of §II-A.

(d) How many different values appear on a column:

This may be used to check if the columns have the correct different values such as to see the prefecture column (in Japan) has 47 different values. **select count(distinct TargetColumn) from GivenTable** is the simplest SQL statement for this.

(e) Most frequent values of a column:

select TargetColumn, count(1) freq from Table group by TargetColumn order by freq desc with the output row number limitation by an additional phrase such as **limit 5** or **top 5**.

[3]R language is applicable to calculate the 95% confidence interval by `sum(x*x)/qchisq(c(39,1)/40,length(x))` where `x` is the vector whose elements are repeated results of the previous SQL statement.

(f) A *concise* way to see concrete values of a column:

Instead of just only seeing **min** and **max** of column values, the four characteristic values from a column greatly help understanding column values by: **select * from (select min(C) v11, max(C) v12 from T where left(C,1) = (select min(left(C,1)) from T)) s1, (select min(C) v21 , max(C) v22 from T where left(C,1) != (select min(left(C,1)) from T)) s2** for the table T and the column C, nonetheless the data type is any of numeric, date or character. The function **left** takes the most left character from its argument. See Table I and its variation Table II.

(g) How values of two columns overlap:

One probably wants to know how two sets of values of corresponding columns overlap. Assume a value set S_A comes from a column C_A of a table T_A and the other value set S_B comes form a column C_B of a table T_B, one can get the number of cardinalities $\#(S_A - S_B), \#(S_A \cap S_B), \#(S_B - S_A)$ by a one-shot (but lengthy!) SQL statement: **select * from (select count(distinct v10) i10 from (select** C_A **v10 from** T_A **except select** C_B **v10 from** T_B**) T) t10, (select count(distinct v11) i11 from (select** C_A **v11 from** T_A **intersect select** C_B **v11 from** T_B**) T) t11, (select count(distinct v01) i01 from (select** C_B **v01 from** T_B **except select** C_A **v01 from** T_A**) T) t01**. This kind of SQL statements should be generated automatically.

D. Knowing column connections over different tables

Determining all of the column connections over different tables seems like an endless task to fulfill 2) of §II-A if the total number of columns is in the hundreds. One solution the author found is as follows:

(1) Assemble a table as in Table I by (f) for the all columns of the all tables in the DB.
(2) Identify and collect all the coded columns. They tend to have fixed-length digits when special values are excluded.
(3) Group them based on their digit lengths.
(4) Find out the relation between columns sharing the same coding systems. Approach the process like solving an easy puzzle, as the columns sharing the same code system have similar values.

In Table III shows how the six relationships #1, #2... and #6 were found as a example written above.

E. Knowing column relations inside a table

One often encounters a muddy step to know column relations inside a table between fulfilling 2) and 3) of §II-A and nearly compulsory before 4). This subsection tries to explore what kind of functions would be necessary.

Table III
COLUMNS SHARING THE SAME CODING SYSTEMS FOUND THROUGH (f)

Seq. num.	Table name	Column name	Type of value	minLeft1min **s1.v11**	minLeft1max **s1.v12**	other_min **s2.v21**	other_max **s2.v22**	The found relationships labelled by #X
46	T01	mCode	varchar	0036	0036	1212	1212	#1 – but only 2 values
62	T01	iCompany	varchar	0000	0426	1284	1888	#2
85	T01	bCode	varchar			0002	9980	#3 + a blank value
107	T02	cCode	varchar	0000	0400	1284	1888	#2
126	T02	sbCode	varchar			0002	9980	#3 + a blank value
130	T02	cmCode	varchar	1212	1212	(null)	(null)	#1 – probably
244	T06	cCode	varchar	0020	0426	1004	8888	#4
265	T07	cCode	varchar	0020	0426	1004	8888	#4
314	T10	hCode	varchar	000000	020002	900004	zzzzzz	#5 + special value
319	T11	gCode	varchar	3000	3880	4020	9986	#6
333	T15	hCode	varchar	000320	020320	900014	901458	#5
345	T19	cCode	varchar	0000	0400	1284	1888	#2
346	T19	rcCode	varchar	0000	0000	(null)	(null)	#2 – maybe
375	T19	dType	varchar	4021	4121	(null)	(null)	#6 – probably

To see how columns share the same coding systems, the 14 columns with the fixed-length in 4 or 6 were selected from the original table, each represented as a row in the table above. By observing the 4 values **s1.v11**, **s1.v12**, **s2.v21** and **s2.v22** for every row in the table above, one can easily see whether the same coding system shared by two or more columns of the original table, numbered as #1,#2,.. and #6 in the far right column in the above table.

When a table is given which has b columns (the tables recording the customer of a company often has many columns such as $b \approx 100$), it is useful to have a way to find out rules how the columns are related by detecting:

(h) multiple columns in which every row has the same values.
(j) a column determined by other p column(s) ($p \geq 1$)
(k) a frequent value combination of p columns ($p \geq 2$)
(l) a relation between two columns such as $v_i(n) < v_j(n)$ for any row n where v_i, v_j indicate the values of i-th and j-th columns, respectively (magnitude relation)
(m) a relation of $v_i(n_1) < v_i(n_2) \Rightarrow v_j(n_1) < v_j(n_2)$ for any n_1 and n_2 (magnitude relation preservation)

For (h), occasionally some pairs of columns, say, client_cd and customer_id, seems to has the same values on every row and one can check it (if it is actually not, to know the cause some rows have different values on the two columns may arise), and seeking all of such candidate pairs costs computationally proportional to $1+2+\ldots+(b-1) = (b^2-b)/2$ putting aside the multiplication factor of the row number of the table. For (j), the counterpart is $b \times (b-1) \times \ldots \times (b-p) = b\,!/(b-p-1)! \approx b^{p+1}$. Concise and generalizable method that is also computationally light is (always!) desirable, but it may be difficult for the functions introduced here, so we have not yet developed the SQL statement constructor for those functions. A two-step method consists of exploring and confirming, or sample-check and minute-inspection may work well.

The author made prototype programs which provides $b \times b$ matrices for the purpose of (h) and (j), (k) with $p \leq 2$ with functions as follows. Here i and j means (i, j)-th element of the $b \times b$ matrix to be displayed on one's computer screen.

Sameness: The number of rows whose i-th row and j-th row have exactly same values for $i \leq j$ (upper-right) and the number of rows whose j-th row and i-th row have different values for $i > j$ (lower-left).

Combinations: The number how many different combination of i-th and j-th column appear. (This number is compared to the possible number range according to the number pair how many different i-th column values appear and how many different j-th column values appear).

Frequency ranges: Assume a *two-way contingency table*[4] for combination of the values of i-th and j-th columns. The minimum value and the maximum value of frequencies and their corresponding combination values. It is useful to detect abnormal behavior of user or system, and to check k-anonymity.

Non-determinability: How many different values of i-th column cannot uniquely determine j-th column value.

Determinable-column: Which one among the b columns can be uniquely determined by the combination values of i-th and j-th columns. $O(p^3)$ computation cost putting aside the row number.

The above functions often give sufficient answers for (h), (j) and (k) to find "rules" in a table, also coping with a commonly inescapable difficulty that is very small percentage of rows break the rules, although (j), (k) for $p \geq 3$, (l) and (m) have not yet solved. The functions introduced in this subsection is summarized in Table IV.

[4] A p-way contingency table is a table of showing the occurrence numbers of p variables combinations. The table consists of p *variable* columns for the variable values and a *frequency* column which indicates how many times every existing combination of p values appears.

Table IV
FUNCTIONS TO KNOW COLUMN RELATIONS INSIDE A TABLE

Ref. name	(i,j)-th element of the output matrix	For	Cost
Sameness	$\#\{n \mid v_i(n) = v_j(n)\}$	(h)	$O(b^2)$
Combinat.	$\#\{v_i(n) \oplus v_j(n) \mid 1 \le n \le N\}$	(j)$^{\text{partly}}_{p=1}$	$O(b^2)$
Frequency -ranges	Most and least frequencies of the frequency table from $(\, v_{i\oplus j}(n)\,)_{n=1}^{N}$ and their corresponding values	(k)	$O(b^2)$
Non-determinability	$\#\{x \mid \#\{y \mid x\,R\,y\} \ge 2\}$ where $x\,R\,y \Leftrightarrow \exists\, n : x \oplus y = v_{i\oplus j}(n)$	(j) in $p=1$	$O(b^2)$
Determinable column	k s.t. $x\,R\,y$ for $^{\forall}x$, $^{\exists 1}y$ where $x\,R\,y$ $\Leftrightarrow \exists\, n : x = v_{i\oplus j}(n)$ and $y = v_k(n)$	(j) in $p=2$	$O(b^3)$

b is the number of columns of the given table. "Cost" is the computation cost but in this table the term regarding the number of rows N is omitted for simplicity. $v_i(n), v_j(n), v_k(n)$ are the values of i-th, j-th, k-th column, respectively, on the n-th row in the table. $\#$ means the cardinality to a given set, or the number how many different values has in the set. $\oplus$ means combination of two values, which can be regarded as the concatination of the both sides of $\oplus$. $v_i(n) \oplus v_j(n)$ is shortened to $v_{i\oplus j}(n)$. R stands for the relation of *the relational databases* of which idea comes from [4]. O is the Landau symbol.

F. Knowing how "distinctive" rows occur in a table

After executing the functions described in the two previous subsections, one would probably be interested in what the characteristics are in a group of distinctive rows which have a special value, an error value, a rare value or the null value. By referring to the other columns, one can see that it would be caused by merging different systems in the past, or the input system changes, or remaining obsolete records. To survey this from the data, say, to seek the column in which its specific value causes another column to have null value on the same row, tidy methods desirable.

When the above is solved, 4) of §II-A would be highly fulfilled.

G. Complement: principles of tidy recording of findings

The intermediate steps described above produce many files (documents, spreadsheets, slides) that one should record in an organized way. We advocate the following procedure (below *note* means a description of a finding about a DB).

1) Recording any note with the date and the processing time.
2) Each note should desirably be concise, individually eloquent.
3) Notes should be connected to some other notes. And any bunch of notes should not be isolated from others.
4) When grouping multiple bunches of notes, each bunch should have similar complexity or size.
5) Discarding the notes that cannot be appropriately administered in the way described above – in order to keep appropriately administered notes from being buried.

To attain the principles above, the following is highly desirable:

1) Utilizing spreadsheet software[5] well.
2) Numbering and hierarchizing of the documents to keep all of the documents accessible.
3) Not having to keeping all the scripts to reproduce the output. It is not realistic. Intermediate results which eloquently recall the way to reproduce them would suffice.
4) Collect, organize and split the bunches of documents with the viewpoints of how to backup the documents and how to give the access rights to other persons.

III. NEWLY CREATED SOFTWARE

Each SQL query sentence appeared in the previous section § II largely just outputs only a single row because it considers only a single DB table or only a single column of some tables. But in practical situations, one probably like to consider all of the tables of the given DB, and also all of the columns of the tables. To understand the DB, especially for the purpose of (a), (c), (d) and (f), it is desirable that a one-shot SQL query outputting many rows each corresponding to each table/column on the DB.

This section explains why new software which yield SQL query sentence(s) is necessary, and the requirements to the software and how it is designed and where it is available on the Internet. And then Fig. 1 shows how the integrated function of (a) works as the yielded SQL sentences is inserted into the white-backed rectangular area.

A. New software is necessary for understanding a DB

A human data analyst probably enters SQL statements by hand (on the other side, many SQL statements are internally generated and issued in dynamic web pages and online games by programs). To understand a DB as in § II-A, the ways presented in § II-B and § II-C may not be enough. For example, assume there are N tables denoted as $(T_i)_{i=1}^{N}$. Listing the number of rows of every table requires either:

- issuing each SQL statement **select count(*) from** T_i for N times, or
- issuing only an SQL statement **select '**T_1**', count(*) from** T_1 **union all select '**T_2**', count(*) from** T_2 **union all .. union all select '**T_N**', count(*) from** T_N,[6]

but this is tedious. The former requires summarizing human handiwork to produce a table from N output of SQL, and the latter requires the user to input a statement that may consist of $9N - 2$ words. And the number of tables N can be enormous in practical situations. Therefore, a software package of SQL generator is desirable. See Fig. 2.

[5]Such as Google Drive, Microsoft Excel, and open office software.

[6]Sorting by **order by** phrase is necessary in practical. In that case, the **select** phrase should be changed such as **select** k **sequence_number, '**T_k**' table_name, count(1) row_number** in order to specify the name of which column to be sorted. This kind of transformation is formalized into (ii) and (iii).

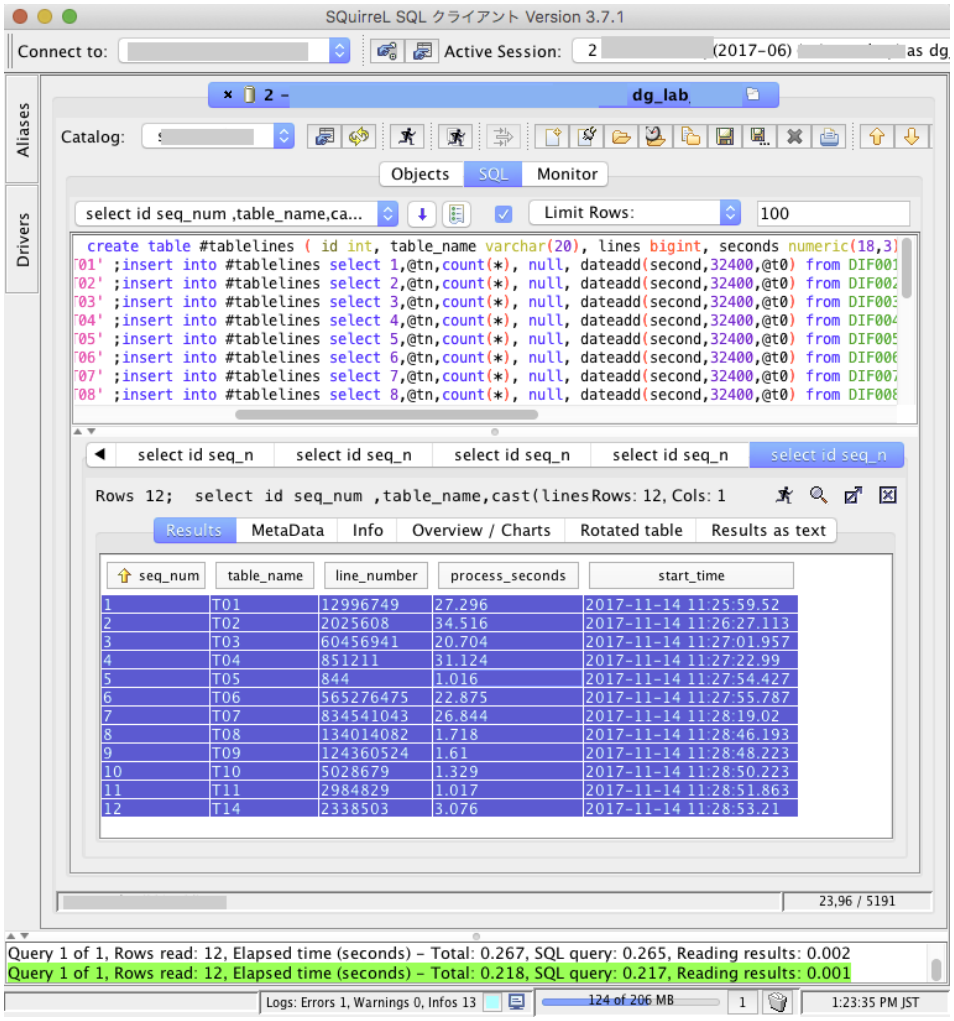

Figure 1. The dark blue-backed table (12 rows, 5 columns) shows the row numbers for given tables (T01, T02...) with the processing time information. The white-backed rectangular area shows part of the whole SQL sentences that is yielded by our new software. Listing up and appending useful information orderly requires very complicated SQL sentences while **select count(1) from T** for each table **T** to count its row number is the essential element of the whole SQL sentences. The GUI software shown is SQuirrel SQL [8], an open-source SQL client software, which here connects to a DB server built on Microsoft SQL Server 2008. Our new software may yield SQL sentences in thousands of lines depending on purpose.

B. Necessary features the software should have

To conduct the functions in §II over all of the table or over all of the columns especially for (a), (c), (d) and (f), the following features are necessary or recommendable. Many of the followings are motivated by purely technical/operational reasons.

(i) The output of functions over each table or column should be *assembled* (integrated) into a table to be displayed. Each row of the assembled table corresponds to each of the original tables or the columns of the original tables.

(ii) Although the output of simple queries like (a) - (g) need not contain the table names or column names, each row of the assembled output table must contain them. If not, for example, just the list of the numbers of rows of all the tables, without the names of the DB tables, does not make sense for (a).

(iii) For easy viewing, a sequence number is desirable on each row of the assembled table so that it may be sorted.

(iv) A timestamp for each row of the assembled table is desirable. It indicates when the DB was accessed. A DB system may return an error value because of the instability of itself or the network connection and the timestamp is useful information to have.

(v) The calculation time in seconds for each row of the assembled table is desirable. A DB response may take a long time depending both on the query and the DB server status. The recorded calculation time helps one understand how the query was conducted, and how to

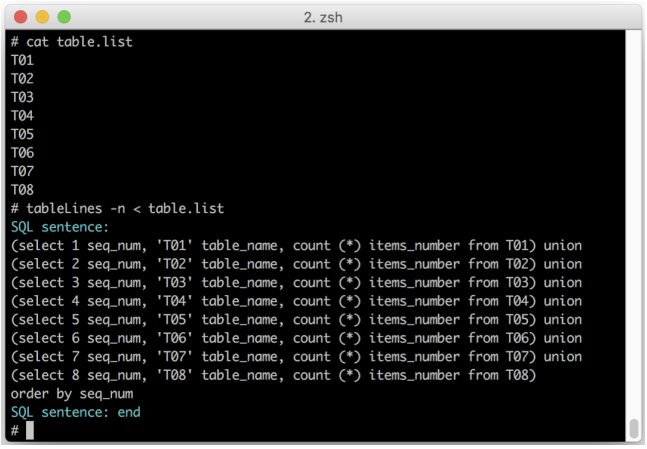

Figure 2. The file `table.list` contains table names. Using this to the *standard input* of a command, an SQL sentence is yielded to the *standard output* as shown the 9 lines between the 2 colored lines having the phrase "SQL sentence:" here. This command can yield SQL sentences with many other extended functionality by replacing the *options* indicated `-n` above.

estimate the calculation time of similar calculations in the future.

(vi) The applicability to many different SQL implementation is better although the development may be costly. The *environmental variable* rather than *command-line options* (also known as *flags* or *switches*) might be desirable to switch over this SQL dialects.

C. The user-interface of the software

Considering the requirement described above, the author designed commands about the interface in the following ways:

1) The *standard input* to each commands is newline-delimited data each of whose element is either:
   - table name (to see every table)
   - a sequence of table name, a tab character, and column name (to see every column)
2) The name of each command is basically combination of two English words for convenient in recalling.
3) Each of the commands receives command-line options to abundantly expand its functionality.

The commands providing following are also included[7], which is necessary at the beginning stage of touching the DB by the software:

4) To know the SQL implementation version
5) To list up the table names
6) To list up both the table and column names

[7]4), 5) and 6) just output **select @@version**, **select name from sysobjects where xtype = 'U'**, and **select table_name, column_name from information_schema.columns**, respectively.

D. Where the newly created software is available?

Part of the tools mentioned above in this paper are available on GitHub at https://github.com/tulamili/sql_generator. The software only covers for Microsoft SQL Server 2008, and other SQL system such as PostgreSQL, MySQL, Oracle, SQLite, BigQuery and so on have not considered so far, therefore (v) of §III-B has not achieved. It is much desirable to build similar software by the hands of the readers based on what is written in §III-B, §III-C and their references.

E. Comparison issues

It is natural to be asked how much the newly created software significantly speeds up the data analyzing project, as well as the comparison with other tools. Let us assume an illustrative situation that one is given a DB with 20 tables with 500 columns in total and some table contains millions or billions of rows.

To acquire meaningful and valid business findings such as monthly sales totals, user usage rankings or such business-related calculations, sufficient understanding of the DB is necessary (see §II-A). This may take anything from one month to years because the DB may contain confusable columns with cryptic special values (note that a project period of an ERP or *Enterprise Resource Planning* system replacement often takes years). Our new tool would reduce the time needed for the DB understanding process within a week possibly.

Comparable software does not seem to exist as multiple survey articles [6][7] have yet to identify any similar tools. However, a similar software creation concept was shared by [3] in 2016.

IV. Conclusion

Existing SQL may not be enough to conduct the analysis of a given DB, so new tools and how to make them are necessary. To meet this demand we presented some function examples, which integrates a lot of SQL statements that are beyond the ability of manual entry. Some of these kinds of functions should be implemented and natively run [9] on SQL-type DB servers, whereas further expanding the functionality of the newly created program would probably make good proposals of desirable function groups and systematizing such functions.

Our new tool is a set of command-line interface programs which enables researchers or data scientists to analyze an enormous database more easily than with SQL alone. When presented with an unknown database, our tool greatly helps researchers to understand the database contents prior to any data pre-processing, data transformation, mathematical modeling, or business-related calculation. Through an easy use of command-line interface, our tool is able to generate long, complex SQL expressions that would be unrealistic to write manually.

ACKNOWLEDGMENT

The author would like to thank CREDIT-SAISON Co., Ltd. who provided us their business operation data.

REFERENCES

[1] Saltz, J. (2015). "Big Data Team Process Methodologies: A Literature Review and the Identification of Key Factors for a Project?s Success", IEEE International Conference on Big Data.

[2] Janssens, J.H.M. (2014), Data Science at the Command Line, O'Reilly.

[3] Shimono, T. (2016). "A Hacking Toolset for Big Tabular Files", IEEE International Conference on Big Data.

[4] E. F. Codd (1990), The Relational Model for Database Management: Version 2, Addison-Wesley.

[5] ISO/IEC 9075 "Information technology - Database languages - SQL", https://www.iso.org/committee/45342.html

[6] Saltz, J., Shamshurin, I. (2016). "Big Data Team Process Methodologies: A Literature Review and the Identification of Key Factors for a Project's Success", IEEE International Conference on Big Data.

[7] Kumar, V., Alencar, P. (2016). "Software Engineering for Big Data Projects: Domains, Methodologies and Gaps", IEEE International Conference on Big Data.

[8] SQuirrel SQL, http://squirrel-sql.sourceforge.net/.

[9] Native to a system, https://en.wikipedia.org/wiki/Native_(computing).

APPENDIX

A. Usage of variance estimation to see differences in tables.

In II-B (c), in order to compare two (seemingly same or slightly different) tables, a cryptographic hash function is used to see the complete sameness or else how many records (rows, lines) different between the two by essentially calculating

$$f_\lambda(t) = \sum_{i=1}^{N_t} \sqrt{3}\,(-1 + 2\,h(r_{t,i} \oplus \lambda)) \tag{1}$$

for each table t with records $r_i\,(i = 1, \ldots, N_t)$, where $\oplus\lambda$ here means concatenation by some byte sequence λ and h means some function whose range is $[0, 1)$. We actually employ the function h where $h(x)$ equals $0.b_1b_2 \ldots b_{32}$ in binary, or

$$h(x) = \frac{b_1}{2} + \frac{b_2}{2^2} + \frac{b_3}{2^3} + \cdots + \frac{b_{32}}{2^{32}} \tag{2}$$

where $b_1, \ldots, b_{32}$ are the beginning 32 bits of the SHA-1 function return value for x. By the transfomation

$$\bullet \to \sqrt{3}(-1 + 2 \times \bullet) \tag{3}$$

Table V
CLUES Δ_λ TO ESTIMATE HOW TWO TABLES t_1, t_2 DIFFER

λ	$f_\lambda(t_1)$	$f_\lambda(t_2)$	Δ_λ
"Sun"	-525.43	-510.21	-15.22
"Mercury"	-153.47	-143.69	-9.78
"Venus"	-2066.50	-2042.78	-23.72
"Earth"	-593.07	-576.22	-16.85
"Moon"	601.36	613.11	-11.75
"Mars"	2764.32	2775.64	-11.32
"Ceres"	304.56	311.21	-6.65
"Jupiter"	-1165.96	-1175.84	9.88
"Saturn"	2279.93	2280.89	-0.96
"Uranus"	1967.68	1977.39	-9.71
"Neptune"	1537.59	1543.95	-6.36
"Pluto"	1472.29	1487.23	-14.95
$s = \{\overline{\bullet^2}\}^{1/2}$	1526.95	1528.99	12.72
$12s^2/\chi^2_{0.95}(12)$	1198919.	1202127.	83.2
$12s^2/\chi^2_{0.05}(12)$	6353341.	6370342.	440.8
True value of $\#''$	2810122	2810116	unknown

Note that Δ_λ drifts as a variable from $\mathcal{N}(0, d'')$ when $d'' \gg 1$, and $\sum_{i=1}^k z_i^2$ distributes $\chi^2(k)$ when $z_i \sim \mathcal{N}(0, 1)$ with the i.i.d. condition. The 95% confidence interval of d'' from the 12 repetitions ("Sun" to "Pluto") is $[83.2, 440.8]$ according to the method of *estimating the popular variance* (the 2.5% and 97.5% points of $\chi^2(12)$ are 4.403.. and 23.33.. while $\sum_\lambda \Delta_\lambda^2 = 1942.057$ referring to the table above). One can conclude that probably 83 to 441 records differ between t_1 and t_2 if every distinct record value appear only once in each table.

the inner term of (1) behaves as a good uniform random generator with the range $[-\sqrt{3}, \sqrt{3}]$ unless $r_{t,i} \oplus \lambda$ takes some completely same values by varying i and λ.

Then how can we estimate how many records are different between two given tables? Here is a example. t_1, t_2 have 2,810,122 and 2,810,116 records, respectively, which means they seemingly has (only) 6 records difference in number. $\Delta_\lambda = f_\lambda(t_1) - f_\lambda(t_2)$ for various λ is shown in Table V. If t_1 and t_2 were completely same then $\Delta_\lambda = 0$ holds for any concatenated λ. If not, by varying λ, Δ_λ^2 behaves as a random variable distributed approximately with the *chi-squared distribution* with $d'' = \sum_\iota d_\iota^2$ *degrees of freedom* where d_ι means the absolute value of the difference in occurrence number of each distinct record value ι between tables t_1 and t_2. According to the caption of Table V, d'' is estimated as between 83 and 441 in 95% confidence level. Although the difference in row numbers of t_1 and t_2 is 6, the record can differs in hundreds in this case.

To shrink the width of this confidence interval $[83, 441]$, the number of different λ, which is 12 here, must be increased, but it may require much computation resources.

Note that a variable from the uniform distribution $[-\sqrt{3}, \sqrt{3}]$ has the mean zero and the variance one and if such variable are repeatedly summed up k times ($k \gg 1$), say 3 or more, the sum is almost approximately distributed with the normal distribution with the mean zero and the standard deviation $\sqrt{k}$, or $\mathcal{N}(0, k)$. We this time employed the almost uniform distribution obtained by (2) and (3) because of the calculation ease.