Percentage Aggregation Functions by Extending SQL

Jie Xiao
Department of Information Science & Engineering,
Hunan First Normal University, Changsha, 410205, China
Email: xiaoflyfox@163.com

Yan Zhu
Department of Electronics and Information Engineering,
Loudi Vocational and Technical College, Loudi, 417000, China

JinLing Luo
Department of Electronics and Information Engineering,
Loudi Vocational and Technical College, Loudi, 417000,China

FenShi Zeng
Department of Information Science & Engineering,
Hunan First Normal University, Changsha, 410205, China

***Abstract*—Current SQL aggregation functions have evident limitations for computing percentages, for which this paper proposes two SQL aggregation functions. The two novel aggregation functions are easy to use, which have wide applicability and can be efficiently evaluated. They may be used as a framework to study percentage queries and to generate efficient SQL code. Experiments compare our proposed percentage aggregations against queries using OLAP aggregations. The results show that both proposed aggregations are significantly faster than existing OLAP aggregate functions.**

***Index Terms*—Relation database, SQL, Query process, Aggregate function, Percentage aggregation**

I. INTRODUCTION

As a standard interface accessing relational database, SQL is a very easy understanding relational database query language. SQL offers users some aggregation functions such as sum, avg, count, max and min functions. The percentage is often used in data analysis to help understand and compare the number of statistical information. However, when calculate the percentage, current SQL aggregation functions are very cumbersome and inefficient.

Aggregation function has a wide range of applications in the OLAP and data mining environment. Some of current documents have been extended the computing of aggregation functions, and one of the important extensions is the cube operation [1]. At present, the Transact-SQL of SQL Server proposes new relationship operators: the PIVOT and UNPIVOT, which have improved communication skills. PIVOT lines out for the rotation, and implements polymerization at the same time. Based on given pivot tables, it generates an output table with a pivot out of the only value that corresponds to each set of the output table. While UNPIVOT is contrary to the implementation of the operation. The literature [2-4] is the first study researched on the percentage of aggregation function, but its syntax form is not easy to understand, of which the gathering function has a poor scalability; what' more, when gathered on the property and a second gathering, it adopts a number of properties not in line with the percentage of the characteristics of aggregation, but it can not be carried out on the property for more than three times of gathering. The literature [5] gave an OLAP solution, but it can not effectively extend the slice data. The literature [6] proposed a new SQL generator based on current SQL generator lacking of comprehensive error detection and good scalability, but it ignored to consider the aggregation functions.

In this paper, the proposed two kinds of percentage aggregation functions can be used to generate SQL code; it is also an expansion of current SQL aggregation functions, which provides a new train of thought for commercial databases.

II. AGGREGATION FUNCTIONS

Assume that T is a relation, id is T's primary key, there are *n*-classification of property, *agg* is the property value. That for $T\ (id, d_1, d_2,..., d_n, agg)$, classification of property $d_1, d_2, ..., d_n$ group gathered for the numerical expressions A. Take T as an *n*-dimensional cube, the classification of property (*Victoria*) for the group

A Project Supported by Scientific Research Fund of Hunan Provincial Education Department(09C232)

Biography: Jie Xiao(1974-), male, lecturer, postgraduate, Main Field: database

E-mail:xiaoflyfox@163.com

gathered numerical attribute *agg*. As the percentage emphasizes on proportion, common aggregation mainly use the sum (). (The following will only relate to sum ()). Here, *T* is also a provisional list table to view or query.

This article will use the following example of a table for analysis. Table *sales* (id, province, city, sale_date, quantity) respectively mean the main keys, provinces, cities, quantity and the date of sale.

Table 1 Sales

id	province	city	sale_date	quantity
1	p1	c1	2005/01/01	80
2	p1	c1	2005/02/01	50
3	p1	c1	2006/01/01	40
4	p1	c2	2006/01/01	30
5	p2	c3	2005/01/01	50
6	p2	c3	2006/01/01	80
7	p2	c4	2006/01/01	70

A. Vertical gathered percentage

This article employs the following form to show aggregation functions query:

SELECT sum (*agg*), $d_1, d_2, \ldots, d_{k-1}$, V_{pct} (d_k)
FROM *T* WHERE <*condition*> GROUP BY $d_1, d_2, \ldots, d_k$;

Which different from general SQL is that the above statement has a "V_{pct} (dk)" , of which $1 \le k \le n$ said the group out, it can be used to calculate the percentage dk in *d1, d2, ..., dk-1* . V_{pct} *()* aggregation function has the following rules:

- Because of 2 aggregation groups, the GROUP BY statement must be exists;
- $d_1, d_2, \ldots, d_k$ and V_{pct} *(Dk)* of the group set out in the order must be strictly limited, because the percentage function is calculated based on d_k' properties;
- Could base on the same GROUP BY statements with other SQL aggregation functions;
- The return interval of Vpct () is in the range of values [0,1] or null (in addition to being 0 or when be operated by blank value);
- If ($d_1, d_2, \ldots, d_k$) has only an element, then the above statement equivalent to:

SELECT sum (*agg*), $V_{pct}(d_k)$ FROM *T* WHERE <*condition*> GROUP BY d_k;

To deal with the null operation and in addition to the operation were 0 of *Vpct ()*, the method used is to make *Vpct ()* in line with sum (). If sum () = 0, then Vpct () = 0; such sum () is null, Vpct () is also empty.

As the user needs a small result sets in general, so this paper employs putting the result setting into the middle of the provisional table that calculated to produce the final result set. The specific calculation method is shown in Figure 1.

Because there is connection operation to the properties of $d_1, d_2, \ldots, d_n$, so property index to $d_1, d_2, \ldots, d_n$ could improve the executive performance. However, to maintain the index will consume more system resources.

Therefore, in order to get each county (district) in the city's proportion of sales inquiries of Table 1, e.g.. *Q1*, thus can get the results as in Table 2.

```
Input: SELECT sum (agg), $d_1,d_2,\ldots,d_{k-1}$ ,$V_{pct}(d_k)$
        FROM T
        WHERE <condition>
        GROUP BY $d_1,d_2,\ldots,d_k$;
1. Establish a provisional table $T_k$ ($d_1, d_2, \ldots , d_k$, agg),
   computing division agg's gathering value of $d_1, d_2, \ldots, d_k$ in
   table T, and insert the gathered results into $T_k$.
2. If $k = 1$, no group gathered outcome called total, the
   provisional table update agg = agg / total, which will be the
   final set, thus end the algorithm; or establishing the
   provisional table $T_{k-1}$ ($d_1, d_2, \ldots, d_{k-1}$, agg), computing
   division agg's gathering value of $d_1, d_2, \ldots, d_k$ in table T, and
   insert the gathered results into $T_{k-1}$.
3. Considering the value of 0, make the two different results
   that born in the above calculation process divide and get
   percentage. To avoid the creation of provisional table, there
   must be update operation of provisional table $T_k$. It can
   employ the following statement form:
        UPDATE $T_k$ SET agg =(
              CASE WHEN $T_{k-1}$.agg <> 0 THEN
                    $T_k$.agg/$T_{k-1}$.agg
              ELSE NULL END)
           WHERE $T_{k-1}.d_1= T_k.d_1$, $T_{k-1}.d_2= T_k.d_2$,…,
                 $T_{k-1}.d_{k-1}= T_k.d_{k-1}$;
Output: percentage of $d_k$ in vertical division property.
```

Figure 1 Percentage of vertical aggregation

Q1: SELECT sum (*quantity*), *province*, V_{pct} (*city*)
FROM *sales*
GROUP BY *province, city*;

Similarly, to inquiry to sales ratio of each province can use the fifth rule, thus get result {(*p1*, 0.5), (*p2*, 0.5)}.

Table 2 the results gathered of V_{pct}(*city*)

province	city	quantity
p1	c1	85%
p1	c2	15%
p2	c3	65%
p2	c4	35%

B. The aggregation of the level of percentage

B.1 Basic form

This paper employs the following forms to represent the level of aggregation function query:

SELECT sum (*agg*), $d_1,d_2,\ldots,d_{k-1}$,$H_{pct}(d_k)$
FROM T
WHERE <*condition*>
GROUP BY $d_1,d_2,\ldots,d_k$;

H_{pct} () is similar to V_{pct} (), it has the following rules:

- Because of 2 aggregation groups, the GROUP BY statement is necessary;
- Could base on the same GROUP BY statements with other SQL aggregation functions;
- The return interval of Vpct () is in the range of values [0,1] (the sum of tuple attribute value is 1), or null (in addition to being 0 or when be operated by blank value);
- Limit ($d_1, d_2, \ldots, d_k$) more than one element

Similarly, as the user needs a small result sets in general, so it also employs putting the result setting into the middle of the provisional table that calculated to produce the final result set. The main problem of the percentage of the level of aggregation is that *dk* has only one element in $(v_1, v_2,..., v_i)$. However, if the value of the match in CASE statement has been made, then they should stop comparison. In fact, database query optimizer does not optimize the mechanism to identify and stop comparison. Therefore, using another method to avoid the CASE statement's invalid comparison is necessary. The specific calculation method is shown in Figure 2.

```
Input： SELECT sum (agg), $d_1,d_2,…,d_{k-1}$,H_pct($d_k$)
        FROM T
        WHERE <condition>
        GROUP BY $d_1, d_2,…,d_k$;
1.  If (d1, d2, ..., dk) has only one element, then end the
    algorithm,   otherwise, eliminate duplication of dk's
    properties that meets the tuple of condition  <condition> in
    Table T, so next there are i non –repetition values($v_1, v_2$ , ..., $v_i$),
    of which $i \geq 1$. In theory, the maximum value of i is the
    total groups of tuple. But in fact, under limited conditions,
    non- repetition values are generally very small, take the
    annual 12 months for instance.
2.  Establish a provisional table $T_{agg}(d_1,d_2,…,d_{k-1},v_1,v_2,…,v_i,total)$,
    computing division agg's gathering value of d1, d2, ..., dk in
    table T,  make the i non-repetition values be the different
    conditions cycle i times, then insert the gathered results into
    Tk, calculate the percentage of value at last.
    FOR m=1 TO i DO
        INSERT INTO $T_{agg}(d_1,d_2,…,d_{k-1},v_m)$
        SELECT $d_1,d_2,…,d_{k-1}$,sum (
            CASE WHEN agg IS NULL THEN 0
                     ELSE agg END),
        FROM T
        WHERE $d_k = v_m$ AND <condition>
        GROUP BY $d_1,d_2,…,d_{k-1}$ ;
    ENDDO
    INSERT INTO $T_{agg}$
    SELECT $d_1,d_2,…,d_{k-1}$, sum ($v_1$)/$\sum_{m=1}^{i}$ sum ($v_m$),…,

           sum ($v_i$) / $\sum_{m=1}^{i}$ sum ($v_m$),  $\sum_{m=1}^{i}$ sum ($v_m$)

    FROM $T_{agg}$ ;
Output: the percentage of $d_k$ in level division property.
```

Figure 2 Percentage of horizontal aggregation

This method has overcome the shortcoming of re-comparison, but it need to repeatedly scan Table *T*, so disk I/O will have a heavier loading. However, as for mass data, the queried middle-result set (temporary table) usually is very small; so the loading of disk I/O can be ignored.

Therefore, in order to get each province's annual sales ratio of Table 1, e.g..*Q1*, thus can get the results as Table 3.

Q2: SELECT sum (*quantity*), H_{pct}(*province*)
FROM *sales*
GROUP BY *province*;

Table 3 the results gathered of H_{pct} (*city*)

province	2005	2006	total
p1	65%	35%	200
p2	25%	75%	200

B.2 Extending form

It often needs comparison over the same period in practical statistics, such as over the same period last year, over the same period last month and so on. This article employs the following forms to represent the level of aggregation function extending:

SELECT *AGG* (*v*), $d_1,d_2,…,d_{k-1}$,C_{ps}(E(d_k),*n*)
FROM *T*
WHERE <*condition*>
GROUP BY $d_1, d_2,…, E(d_k)$;

Among which *AGG* is a general aggregation function, C_{ps} is the same period-comparison function, $E(d_k)$ is a time expression of d_k, n is the number of compared times over the same period.

C_{ps} () aggregation function has the following rules:

- Because of 2 aggregation groups, the GROUP BY statement is necessary;
- Could base on the same GROUP BY statements with other SQL aggregation functions;
- *dk* limit in $\{d_1,d_2,…,d_k\}$ can be said the date or time.

The calculation method is shown in the following:

```
Calculate the $v_{max}$ of E ($d_k$) in Table T (T meets the
conditions of <condition>), and establish provisional Table
Tc ($d_1,d_2,…,d_k,v_1,…, v_{max},total$), then calculate the gathered
value of $d_1,d_2,…,d_k$ in Table T in groups, inset the gathered
results into the provisional table at last.
    INSERT INTO $T_{agg}$
    SELECT $d_1,d_2,…,d_{k-1}$,
           AGG (CASE WHEN $E(d_k)=v_{max}-n+1$
              THEN agg ELSE 0 END),
              …
           AGG (CASE WHEN $E(d_k)= v_{max}$
              THEN agg ELSE 0 END)
    FROM T
    WHERE <condition>
 GROUP BY $d_1, d_2,…,d_{k-1}$ ;
```

Figure 3 Expanding of the level of aggregation

III. Experiment Evaluation

The experiment environment is: Intel Celeron 2.53 Ghz, Memory 512MB, Windows XP professional, SQL Server2005. Based on the percentage of the proposed gathering function, it can adopt JAVA for a given query to generate SQL code. As for the given sales table, it can use generate mass date Data factory 5.2 to generate 0.5G data. Among which the number of non-repetition of the attribute is *province*:32,*city*:100,*year*(*sale_data*):5.

Because each query use the same parameters to produce the same results, so their difference is the generated SQL query code leads to a different evaluation of the inquiry. SQL / OLAP query will adopt sum () function and window OVER/PARTITION BY statement [7], under these circumstances optimizer group gathered data can use the provisional table and index, and then adopt the same group of properties and property values to compare with SQL / OLAP. Inquiries are divided into 3 kinds: case 1 (d_1 ... d_n: *sale_date;* d_1 ... d_k: *sale_date*); case 2 (d_1 ... d_n: *province, sale_date;* d_1 ... d_k: *sale_date*);

case 3 (d_1 ... d_n: *province, city, sale_date; d_1 ... d_k: sale_date*).

Figure 4 shows the comparison between the method which this paper employed and the query execution time of SQL/OLAP. It can be seen that the gathering method this paper given is significantly better than that of SQL/OLAP. As a result, even if the SQL/OLAP can be a simple form to calculate percentage, the implementation of the performance is inefficient.

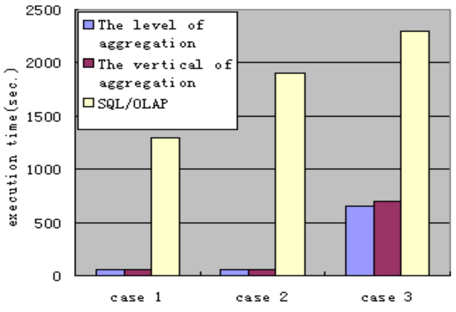

Figure 4 Comparison of the execution time

VI. CONCLUSION

This paper gives two sorts of aggregation functions being used to calculate the percentage (horizontal and vertical gathered percentage). They can be used as a framework to study the percentage of inquiries to give method to generate SQL code. Experiments have studied the percentage gathering methods and the executive performance of SQL/OLAP gathering methods. They show that these two methods in terms of performance have significant improvement than SQL/OLAP gathering methods. In order to get a better optimization, the next step of the work need combine the horizontal and vertical percentage aggregations under a same query. In addition, it is necessary to make a research that under the circumstance users parallel to submit the percentage of inquiries; and then organize different physical storage and index design to optimize the query.

REFERENCES

[1] Zaharioudakis M, Cochrane M, Lapis R, et al. Answering complex SQL queries using automatic summary tables. In ACM SIGMOD Conference[C]. New York: ACM Press, 2000, 105-116.

[2] C. Ordonez. Statistical model computation with UDFs. IEEE Transactions on Knowledge and Data Engineering (TKDE), 2010,22.

[3] Ordonez C.Vertical and horizontal percentage aggregations.In ACM SIGMOD conference[C]. New York: ACM Press, 2004, 866-871.

[4] C. Ordonez and S. Pitchaimalai. Bayesian classifiers programmed in SQL. IEEE Transactions on Knowledge and Data Engineering (TKDE), 2010,22(1):139-144.

[5] Xiang Jun,Lei Ying jie. Design and implementation of medical care data processing based on online analytical processing. Computer Engineering and Design, 2006,27(3):485-489.

[6] Hu Hong Yin,He Cheng Wan,Yao Feng. Design and implementation of SQL builder. Computer Engineering and Design, 2006,27(11):2024-2027.

[7] ISO-ANSI [EB/OL]. Amendment 1: On-Line Analytical Processing, ANSI:SQL/OLAP[S], 2003.