PGWinFunc: Optimizing Window Aggregate Functions in PostgreSQL and Its Application for Trajectory Data

Jiansong Ma[1], Yu Cao[2], Xiaoling Wang[1], Chaoyong Wang[1], Cheqing Jin[1], Aoying Zhou[1]

[1]Shanghai Key Laboratory of Trustworthy Computing,
East China Normal University, Shanghai, China
ecnumjs@gmail.com, {xlwang, cqjin, ayzhou}@sei.ecnu.edu.cn

[2]EMC Labs, Beijing, China
yu.cao@emc.com

***Abstract*—In modern cities, more and more people drive the vehicles, equipped with the GPS devices, which create a large scale of trajectories. Gathering and analyzing these large-scale trajectory data provide a new opportunity to understand the city dynamics and to reveal the hidden social and economic phenomena. This paper designs and implements a tool, named as PGWinFunc, to analyze trajectory data by extending a traditional relational database. Firstly we introduce some efficient query process and optimization methods for SQL Window Aggregate Functions in PostgreSQL. Secondly, we present how to mine the LBS (Location-Based Service) patterns, such as the average speed and traffic flow, from the large-scale trajectories with SQL expression with Window Aggregate Functions. Finally, the effectiveness and efficiency of the PGWinFunc tool are demonstrated and we also visualized the results by BAIDU MAP.**

I. INTRODUCTION

Trajectory data generated by moving vehicles becomes more and more important, and it provides us an unprecedented opportunity to understand the city dynamics and reveal the hidden social and economic phenomena. In this paper, we focus on storing and analyzing these large-scale real digital trajectory data by the traditional RDB(Relational Database Systems), PostgreSQL.

SQL Window Aggregate Functions perform common analyses such as ranking, percentiles, moving averages and cumulative in a flexible, intuitive and efficient manner, overcoming shortcomings of the traditional alternatives such as grouped queries, correlated subqueries and self-joins [1], [2]. As one of the most useful standardized extensions to SQL since the SQL:2003 standard, Window Aggregate Functions have been widely implemented in most of the major commercial and open-source relational database systems (e.g. Oracle, DB2, SQL Server, Teradata, Pivotal Greenplum and PostgreSQL), as well as in some emerging Big Data systems (e.g. Google Tenzing, SAP HANA, Amazon Redshift, Pivotal HAWQ and Cloudera Impala). With the Window Aggregate Functions, we can easily tackle with the large-scale trajectories to mine the hidden social patterns and phenomena.

In current database systems [3], [4], in principle a Window Aggregate Function is evaluated over the windowed table in a two-phase manner. In the first phase, the windowed table is reordered into a set of physical window partitions, each of which has a distinct value of the `PARTITION BY` key and is sorted on the `ORDER BY` key. The generated window partitions are pipelined into the second phase, where the Window Aggregate Function is sequentially invoked for each row over its frame within each window partition.

While existing techniques [3], [5], [6], [7] are available to optimize the table reordering operation in the first phase, there exist rare previous studies on how to save the costs of window function calls in the second phase, which is exactly what we want to set off in this system.

In this demo, we illustrate our design of a new data analysis system, PGWinFunc, which is implemented by extending PostgreSQL. In this system, the user submits his SQL query with Window Aggregate Function and obtains the analysis results by BAIDU MAP [8] visualization technology. In PGWinFunc, user gets the intuitive visual insights of the analysis results instead of the traditional relational tables. These visualizations help users quickly understand the city dynamics and reveal the traffic circumstances.

II. FRAMEWORK

The framework of the PGWinFunc system is illustrated in Figure 1. The PGWinFunc system is divided into two components: online part and offline part.

Some preprocess work, including map matching and loading data into the PostgreSQL table, are conducted in the offline mode. The component of map-matching is to map the LBS data onto the road-network with the most simple map matching algorithm (we map the GPS point onto the nearest road).

For this application, we design the trajectory schema with four basic attributes: $date$,$roadid$,$speed$ and $carnum$, where $date$ is the time attribute, i.e. $2013/11/05/12:05$, $roadid$ is the road label in the road network, $speed$ is the average speed of the cars in the $road$ at the time $date$ and $carnum$ is the

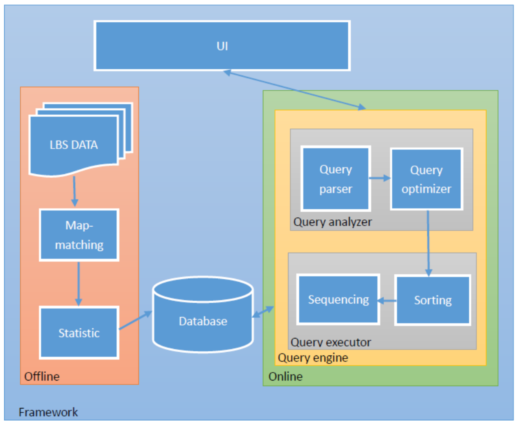

Fig. 1: The Framework of PGWinFunc

traffic flows in the $road$ at the time $date$. Finally, we load the trajectory data into the table in PostgreSQL database.

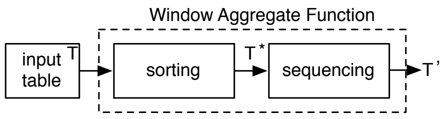

Fig. 2: The execution of Window Aggregate Function

In the online part, we execute users' SQL query and return the results by using BAIDU MAP visualization. The PGWinFunc system accepts the request from the UI(User Interface) and creates the SQL query. The Query Engine has two components, i.e. Query Analyzer and Query Executor. The Query Analyzer parses the query by the Query Parser and generates the query plan by the Query Optimizer. Query Analyzer sends the plan to Query Executor, which performs the sorting by executing the *PARTITION BY* clause and *ORDER BY* clause and gets the intermediate table T^*. Then the generated temporary table T^* is pipelined into the second step named *sequencing*, where the Window Aggregate Function is executed and the final result *T'* is outputted, as shown in Figure 2.

In order to improve the efficiency of processing the SQL query in the process of sequencing within the database, two optimization approaches: **TF(Temporary Frame)** and **SDA(Sharing Data Access)**, are designed and implemented. The detailed algorithms and optimization technology are discussed in [9].

In PGWinFunc, we focus on an important class of *window query*, which is an SQL query Q with the window aggregate functions defined in the `SELECT` clause. The general form of a Window Aggregate Function is as follows:

```
   Agg_func(expression) OVER(
  [PARTITION BY expr_list]
  [ORDER BY order_list [frame_clause]])
```

`Agg_func` is a normal aggregate function (e.g. SUM, AVG and COUNT), except that it operates on a partition or "window" of a (base or derived) table, and returns a value for every row in that window. Unlike group functions that aggregate result rows, all rows in the table are retained. The values returned are calculated by invoking function calls over the sets of rows in that window. A window is defined using a window specification (the `OVER` clause), and is based on three main concepts:

- *Window partitioning*, which forms groups of rows (`PARTITION BY` clause);
- *Window ordering*, which defines an order or sequence of rows within each partition (`ORDER BY` clause);
- *Frames*, which are defined relative to each row to further restrict the set of rows when using `ORDER BY` (`frame_clause` specification)

The frame identifies two bounds of a window, and only the rows in the window are calculated for the aggregate function. There are two frame models, ROWS and RANGE. In ROWS model, the bounds of the frame can be expressed as offsets in terms of the number of rows of difference from the current row; in RANGE model, the offsets are more dynamic and expressed as a logical value. In this paper, we only consider the ROWS model.

To evaluate the *window query*, `PARTITION BY` clause and `ORDER BY` clause are first executed to derive an intermediate table, on which `Agg_func` functions are then invoked.

Example 1: The following example window query calculates the average speed over a frame covering the rows inclusively between the 10^{th} row preceding it and 10^{th} row following it.

```
SELECT
  road,date,carnum
  AVG(speed) OVER (PARTITION BY road ORDER BY
        date ROWS BETWEEN 10 PRECEDING AND 10
       FOLLOWING) as avgspeed
FROM lbsdata;
```

In our trajectory data, because the data are sampled each minute, the above *window query* aims at finding the average speed of cars in some road with the sliding window size of 21 minutes.

Figure 3 shows the original implementation of *sequencing* in PostgreSQL. For frame f_1, all rows in [1,11] are aggregated to the transition value of f_1. Compared with the size of f_1, the frame of f_2 has just one more row, i.e., row 12. Thus, f_2 does not need to calculate from the start position and only adds the value of row 12 to the transition value of f_1, we call this calculation method as the *incremental strategy*. From f_{12}, the start positions of all frames change, so the last frame's transition value cannot be used. Thus, for each frame from f_{12}, we need to calculate from the start position of each frame.

From Figure 3, we observe that there are much redundant computations among different frames, i.e. they may contain common rows to compute repeatedly. Based on this initial observation, we design **TF(Temporary Frame)** approach to reduce this redundant computation among the frames.

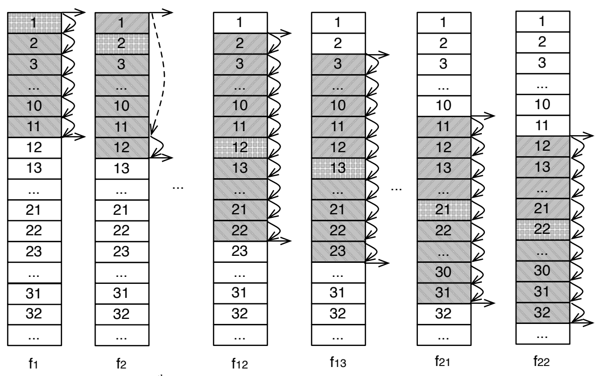
f_i means the i^{th} frame. The row with cell is the current row. i.e. for f_2, the current row is row 2. The black curve arrows show the calculation process for the transition value of frame.

Fig. 3: PostgreSQL

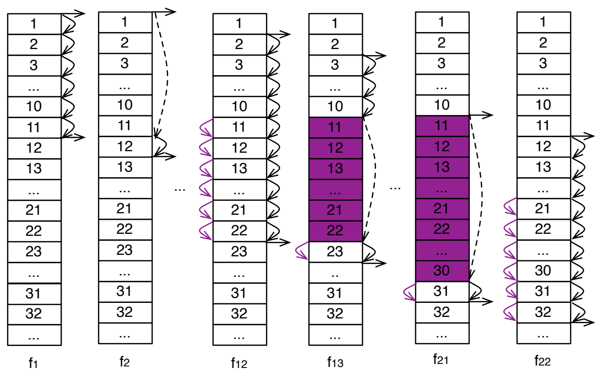
The black and purple curve arrows show the calculation process for the transition value of frame and *TF* respectively. The black dashed arrows means the corresponding region is skipped. The purple shaded area is the rows which the *TF* contains.

Fig. 4: TF

A. TF(Temporary Frame) approach

The basic optimization idea works as follows. For a set of rows S shared among two or more overlapping frames of the aggregate function `Agg_func`, we first apply `Agg_func` to the rows in S and derive a *TTV(the transition value of temporary frame)*. To calculate `Agg_func` over a frame f whose rows are denoted by $[f]$, we just need to directly take the *TTV* as the current transition value and then apply `Agg_func` to rows in $[f] - S$, i.e., we only calculate those rows in the frame which are not been shared among frames. By avoiding repeatedly calculating the *TTV* once for each frame, redundant data accesses and computations are eliminated and thus the total Window Aggregate Function cost is reduced. When the start position of the current frame exceeds the start position of *TF*, we abort the *TTV*, recalculate and get another new *TTV*. Then, we use the new *TTV* to calculate for the following frames.

In one partition, if the frame head for the following frames doesn't change comparing with the last one, there is no need to traverse from the frame head again, the previous frame's transition value can be reused. As a result, there is no need to calculate with *TF*. If we use *TF* to calculate the transition value instead of the previous frame's transition value, some rows will be calculated twice or even more. So, the procedure for evaluating aggregates for the frames whose frame head does not change is the *incremental strategy*: the transition function is invoked for each row added to the frame. When the frame head has changed comparing with the last frame, the previous transition value is not applicable for the current frame, and then we compare the start position of current frame with *TF*:

- If it is bigger than *TF*, the TTV of *TF* can not be used, which means the recalculation is inevitable: all rows in the current frame have to be accessed and aggregated. Then, we reassign the start and end position of *TF*, so that the rows in *TF* are aggregated to TTV.
- If the start position of current frame is smaller than *TF*, TTV can be reused. When calculating the aggregation value of current frame, TTV is assigned to the transition value of current frame firstly. For these rows which are not in *TF*, the rerunning strategy has to be applied, at the same time the end position of *TF* is updated incrementally.

Figure 4 illustrates the *TF* approach for Example 1. Before f_{12}, the *incremental strategy* is used. When calculating f_{12}, the start position changes compared with f_{11}, no appropriate *TTV* can be used, the whole frame has to be aggregated. Meanwhile, we reassign the start and end position of *TF*. TTV is calculated when calculating f_{12}, which means each row in the set of $[11, 22]$ is aggregated to both the transition value of f_{12} and TTV. For frame f_{13}, TTV is assigned to the transition value of f_{13} firstly. Then each row in $[3, 11)$ is aggregated to the transition value of f_{13}. The overlap region is skipped. Finally, the new rows which don't belong to *TF* are aggregated to both the transition value of the current frame f_{13} and TTV of *TF*. f_{21} is a special frame in which the start position is the same as *TF*, so the transition value of f_{21} only needs to add the value of row 13 into the TTV of *TF*. When calculating f_{22}, the start position exceeds the start position of *TF*, so the start position of *TF* is reassigned and needs to be recalculated. The process is similar to f_{12}.

The challenge lies how to dynamically identify the optimal number of frames of sharing a single *TTV* during the execution of a Window Aggregate Function, especially in the complicated runtime environment of RDBMSs. We discuss this optimal problem in [9].

The above optimization does not guarantee the minimum of the total Window Aggregate Function cost, as the row sets covered by the TTVs may still overlap and thus calculating these TTVs may still incur redundant data accesses and computations. In order to further reduce the redundant data accesses, we develop another advanced optimization technique, *Sharing Data Access (SDA)*.

B. SDA(Sharing Data Access) approach

In the SDA approach, multiple *TF*s are designed. We maintain a pool of *TF*s, where *TF*s are arranged by their start

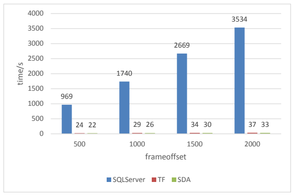

Fig. 5: Execution time of MAX aggregation function

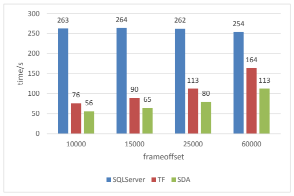

Fig. 6: Execution time of SUM aggregation function

position in ascending order. These *TF*s are maintained and how to choose one *TF* to reuse depends on the current frame. We have analyzed this problem and shown that only one *TF*, whose start position is the smallest, is active for the calculation of the current frame. Although most of *TF*s are inactive, we still update those *TTV*s. According to our experiments, SDA approach further reduces the data access.

C. Efficiency

We have compared the two optimization algorithms with SQLServer on a 170MB dataset under different frame sizes. The working memory is set to 128MB(the minimal server memory of SQLServer). From Figure 5 and 6, the two proposed optimization algorithms behave significantly better than SQLServer.

III. Demonstration Details

The PGWinFunc system is implemented by extending PostgreSQL and the UI is based on the BAIDU MAP API. The source data set is a month of trajectory data from taxis. The data size is about 50GB.

We design and implement the two proposed methods in PostgreSQL in order to execute Window Aggregate Function. The effectiveness and efficiency of PGWinFunc are demonstrated in Section 2. In the PGWinFunc system, we convert the request of trajectory analysis into the window query with parameters. This simplifies the process of writing codes or SQL clauses with subquery. In the PGWinFunc system, the users can choose different analysis tasks by UI(User Interface), such as the traffic flow and average speed through one day.

In the demonstration, we focus on two kinds of queries:

A. Traffic time series

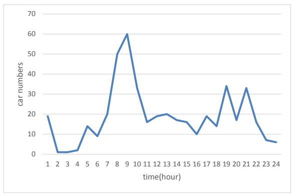

Fig. 7: The car numbers

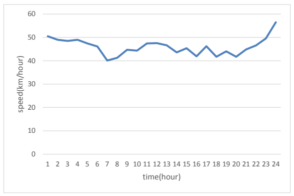

Fig. 8: The average speed

The PGWinFunc system shows the change of traffic conditions for particular road. The change of traffic conditions is

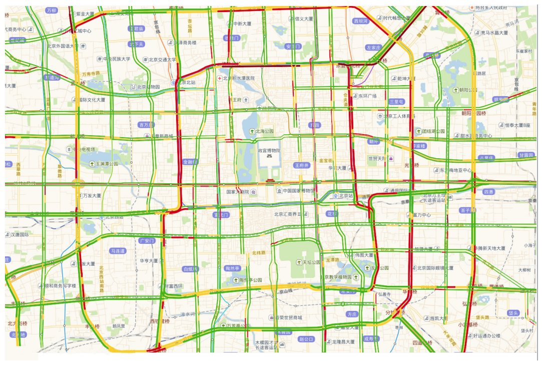

Fig. 9: Traffic circumstances.

the analysis results of continuous time by using the sliding window of Window Aggregate Function. The PGWinFunc system accepts the road name and shows the change of traffic conditions in a day. Figure 7 shows the change of the car numbers and Figure 8 shows the average speed of cars in a road. From Figure 7 and 8, we can observe the rush hours in this road. It is also obviously to get the inverse relation between the average speed and car numbers.

B. Traffic Circumstances

The system shows the traffic circumstances on a specific day, which can be regarded as a parameter by adding the WHERE clause in SQL query. From Figure 9, the PGWinFunc system shows the traffic circumstance by BAIDU Map, which provides a friendly visualization approach.

Acknowledgement

This work was supported by 973 plan (No. 2012CB316203) the NSFC grants (No. 61033007, 61170085 and 61021004), Shanghai Leading Academic Discipline Project (No. B412), and Shanghai Knowledge Service Platform Project (No. ZF1213).

References

[1] C. Zuzarte, H. Pirahesh, W. Ma, Q. Cheng, L. Liu, and K. Wong. Winmagic: subquery elimination using window aggregation. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, pages 652–656, New York, NY, USA, 2003. ACM.
[2] S. Bellamkonda, R. Ahmed, A. Witkowski, A. Amor, M. Zait, and C.-C. Lin. Enhanced subquery optimizations in oracle. *Proc. VLDB Endow.*, 2(2):1366–1377, Aug. 2009.
[3] I. Ben-Gan. *Microsoft SQL Server 2012 High-Performance T-SQL Using Window Functions*. Microsoft Press, 2012.
[4] S. Bellamkonda, T. Bozkaya, B. Ghosh, A. Gupta, J. Haydu, S. Subramanian, and A. Witkowski. Analytic functions in oracle 8i. Technical report, 2000.
[5] Y. Cao, R. Bramandia, C. Y. Chan, and K.-L. Tan. Optimized query evaluation using cooperative sorts. In *Proceedings of the 26th International Conference on Data Engineering, ICDE, Long Beach, California, USA*, pages 601–612. IEEE, 2010.
[6] Y. Cao, C.-Y. Chan, J. Li, and K.-L. Tan. Optimization of analytic window functions. *Proc. VLDB Endow.*, 5(11):1244–1255, July 2012.
[7] Y. Cao, R. Bramandia, C.-Y. Chan, and K.-L. Tan. Sort-sharing-aware query processing. *The VLDB Journal*, 21(3):411–436, June 2012.
[8] BAIDUAPI http://developer.baidu.com/map/
[9] J. Ma, Y. Cao, X. Wang, C. Wang, and A. Zhou. Optimizing Window Aggregate Functions in Relational Database Systems. *Technical report*, 2015.