面向 MAX/MIN 优化的 SQL Window 函数处理

马建松 王科强 宋光旋 张 凯 王晓玲 金澈清

(华东师范大学数据科学与工程研究院 上海市高可信计算重点实验室 上海 200062)

摘 要 Window(窗口)函数作为关系数据库领域中数据分析技术的一种解决方案,其精妙的语义特征使其能代替自连接(Self Join)和相关子查询(Sub Queries)等完成传统复杂查询功能,现已被广泛应用到互联网应用的数据管理和分析中.在目前互联网应用步入大数据时代的背景下,针对高吞吐和实时响应等需求,已有的 Window(窗口)函数的处理性能已经出现了瓶颈.文中首先介绍了关系数据库中窗口函数在执行器中的两阶段执行框架,然后基于 PostgreSQL 数据库中原有 MAX/MIN Window(窗口)函数执行框架,提出了一种基于临时窗口的优化方法,来优化 SQL Window 查询针对 MAX/MIN 函数的处理,并给出了查询代价的分析模型,从理论上分析了该算法的性能.通过与现有商业数据库 SQL Server 进行性能上的对比,验证了该方案的有效性.

关键词 Window 函数;查询处理;性能优化;MAX/MIN;PostgreSQL

中图法分类号 TP311 **DOI 号** 10.11897/SP.J.1016.2016.02149

Optimizing the MAX/MIN of SQL Window Functions

MA Jian-Song WANG Ke-Qiang SONG Guang-Xuan

ZHANG Kai WANG Xiao-Ling JIN Che-Qing

(*Shanghai Key Laboratory of Trustworthy Computing, Institute for Data Science and Engineering, East China Normal University, Shanghai* 200062)

Abstract With the growing number of Internet users, the Internet application gradually enters the era of big data. How to store and analyze the big data becomes a problem in the Internet application. Window Function, as a solution of data analytics in the field of relational database, makes it take the place of Self Join and Sub Queries to complete the traditional complex queries with its subtle semantic characteristics and is widely used in the current enterprise data management and analysis in the Internet application. Under the background of big data, Window Function has the bottleneck in the face of such demand as high throughput and real-time response. In this paper, we detail the two-phase evaluation framework from the perspective of executor in the Relational Database Management System and design a new algorithm dependent on Temporary Window for the MAX/MIN Window functions contraposing the original framework in PostgreSQL. We design a query cost analysis model and theoretically proved the performance of the algorithm. We conduct the performance comparison between the new algorithm and the existing commercial database SQL Server and verify the effectiveness of the proposed algorithm.

Keywords Window function; query processing; performance optimization; MAX/MIN; PostgreSQL

收稿日期:2015-10-18;在线出版日期:2016-03-02.本课题得到国家自然科学基金(61532021,61170085,61472141)、上海市可信物联网软件协同创新中心(中心代号:ZF1213)资助.**马建松**,男,1990 年生,硕士,主要研究方向为数据库、数据挖掘. E-mail: ecnumjs@gmail.com. **王科强**,男,1990 年生,博士,主要研究方向为数据库、数据挖掘、机器学习. **宋光旋**,男,1992 年生,硕士,主要研究方向为数据库、信息检索. **张 凯**,男,1991 年生,硕士,主要研究方向为推荐系统. **王晓玲**,女,1975 年生,博士,教授,博士生导师,主要研究领域为数据管理技术、数据服务及应用. **金澈清**,男,1977 年生,博士,教授,博士生导师,主要研究领域为数据流管理、基于位置的服务.

1 引 言

随着互联网的普及和互联网用户数量的不断增加,互联网应用逐渐进入了大数据时代.大数据时代的来临,使得互联网应用面临着数据的爆发式增长.由于用户数据中包含了丰富的用户行为模式信息,因此,大数据量的用户数据对互联网应用来说变得越来越重要.如何存储和分析这些大数据成为了互联网应用中的难题.随着用户对数据处理效率的要求不断提高,融合数据存储和数据分析的内数据库分析(In-Database Analytics)技术,受到了越来越多企业和研究者的关注.Window(窗口)函数作为关系数据库领域中内数据库分析技术的一种解决方案,最初以扩展文档的形式被引入到 SQL:1999,之后不久,SQL:2003 就正式规范了 Window(窗口)函数的标准,并在后续的标准版本中有所丰富与扩展.Window 函数拥有 SQL 语句惯有的简洁构成模式,但是其精妙的语义特征使其能代替自连接(Self Join)和相关子查询(Sub Queries)等完成传统复杂查询的功能.

在数据处理中,窗口概念的引入使得计算被应用到特定的数据集合(data set)上,或者说是一个窗口所包含的元组(tuple)之上.在此基础上,随之产生了一系列常规分析函数.例如,平均值(average)、累积求和(sum)、最大值(max)、最小值(min)、排序(ranking)、百分比(percentile)等.这些函数都可以用一条 SQL 语句精确、直观、有效地表达出来.

现今,主流商业数据库系统中都实现了 Window 函数以支持数据分析任务,比如 DB2 的联机分析处理(Online Analytical Processing,OLAP)函数,Oracle 的解析函数(analytic function)和 SQL Server 的 Window 函数等.

随着互联网应用逐渐步入大数据时代,Window 函数也逐渐被应用于各类互联网应用的数据管理和数据分析中,如商务智能的查询报表和各类分析应用.越来越多的企业级数据的查询和分析逐渐将旧的查询替换成 Window 函数的方案.Window 函数在数据查询和分析中的应用使得查询处理更加高效,尤其是可以有效地消除效率低下的自连接(Self Join)和相关子查询(Sub Queries)[1-2],而且查询处理过程中可以尽可能地减少临时表的使用.

尽管 Window 函数在数据查询和分析中可以尽可能地提高查询效率,其执行框架也设计的足够精简,但由于其应用场景的多样性使其在实现时并没有得到足够的优化.在目前互联网应用步入大数据时代的背景下,针对高吞吐和实时响应等需求,已有的 Window 函数的处理性能已经出现了瓶颈.因此,针对特定的数据分析场景,需要设计出更适合 Window 函数执行的优化方法.

1.1 Window 函数

Window 函数由一个分析函数和一个窗口定义子句构成.在标准 SQL 的定义中,使用窗口定义函数所作用于的上下文区间,窗口的具体规范则由一个 OVER 子句来定义.

例 1. 给定一张员工工资表 empsalary(如表 1 所示),包含 3 个属性,empno、depno 和 salary(其中 empno 是主键).empno 指的是员工的员工编号(唯一的),depno 指的是部门编号,salary 指的是员工工资.下面是一条带 Window 函数的 SQL 查询语句.表示将所有员工按部门号划分,按员工编号排序,将每位员工的工资与前后 20 位比较,得到工资最高的员工的工资.

```
SELECT
  empno, depno, salary, MAX(salary)
  OVER(PARTITION BY depno ORDER BY empno ROWS
  BETWEEN 10 PRECEDING AND 10 FOLLOWING)
        From empsalary;
```

表 1 员工工资表(empsalary 表)

empno	depno	salary
1	1	2000
2	1	3000
3	1	9000
4	1	3000
5	1	4000
6	1	5000
7	1	2000
8	1	3000
9	1	4000
10	1	5000
11	1	6000
12	1	8000
13	1	5000
14	1	2000
15	1	3000
16	1	4000
17	1	5000
18	1	6000
19	1	7000
20	1	3000
21	1	2000
22	1	4000
23	1	5000
24	1	6000
25	1	2000
…	…	…

如例 1 中的 SQL 语句所示，Window 函数包含一个分析函数（例 1 的 SQL 语句中为 MAX 函数）和一个规定窗口大小的 OVER 子句. 其中 OVER 子句中包含了 3 个组成部分：

(1) PARTITION BY 子句. PARTITION BY 子句用来定义数据分区，形式为 PARTITION BY *expr_list*. 后面的属性值 *expr_list* 决定了数据表按照哪些属性进行划分，*expr_list* 的值相同的元组属于同一个分区.

(2) ORDER BY 子句. ORDER BY 子句用来定义数据排序模式，形式为 ORDER BY *order_list*. 后面的属性值 *order_list* 决定了数据表按照哪些属性进行排序.

(3) ROWS 子句. ROWS 子句用来定义 Window 函数中一个窗口的大小，形式为 ROWS BETWEEN *pre_value* and *post_value*. 其中 *pre_value* 包含了 UNBOUNDED PRECEDING（窗口的起始位置为分区的第一个元组）、*value* PRECEDING（窗口的起始位置为当前行的前 *value* 个元组）和 CURRENT ROW（窗口的起始位置为当前行）3 种，*post_value* 包含了 UNBOUNDED FOLLOWING（窗口的终止位置为分区的最后一个元组）、*value* FOLLOWING（窗口的终止位置为当前行的后 *value* 个元组）和 CURRENT ROW（窗口的终止位置为当前行）3 种.

数据表中的每一个元组作为当前行都对应一个窗口，窗口的大小由 OVER 子句确定，其定义跟当前行相关的一个元组的集合（元组的集合通常包括一个或者多个元组，也可以是整张数据表）.

例 1 中的 SQL 语句部分查询结果如表 2 所示.

表 2　SQL 语句部分查询结果表

salary	depno	salary	max(salary)
1	1	2000	9000
2	1	3000	9000
3	1	9000	9000
4	1	3000	9000
5	1	4000	9000
6	1	5000	9000
7	1	2000	9000
8	1	3000	9000
9	1	4000	9000
10	1	5000	9000
11	1	6000	9000
12	1	8000	9000
13	1	5000	9000
14	1	2000	8000
15	1	3000	8000
…	…	…	…

窗口函数的计算过程包括 3 个过程：分区、排序和确定窗口大小. 如图 1 所示.

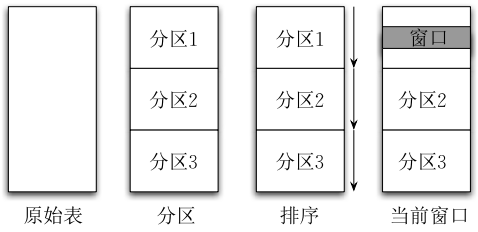


图 1　窗口函数概念模型

(1) 分区是将输入元组分成相互独立的组，不同组的元组在计算时不会出现在同一窗口内，接下来的排序和确定窗口大小的工作都是在各个分区内进行，分区间相互不影响. 基于这种特性，窗口函数其实非常适合于并行计算，在多核计算机上，我们可以将每个分区的任务放到不同的核上独立运行，从而提高计算效率.

(2) 排序则比较简单，在每个划分内部运用 ORDERY BY 子句指定的属性列进行排序即可，整个流程与通常使用的排序子句并无不同.

(3) 确定窗口大小这一步主要是在分区内为当前的元组确定窗口函数所能作用的范围，这个范围是以当前元组为中心与它相邻的某些元组构成. SQL 中有两种确定范围的模式：ROW 和 RANGE. ROW 模式比较简单，只需指定当前元组与在它之前或者之后的元组数量即可. RANGE 模式则是根据当前元组的数值来确定，凡是满足数值范围的与之相邻的所有元组都构成窗口. 如图 2 所示，当前元组为灰色部分的元组，采用 ROW BETWEEN 2 PRECEDING AND 2 FOLLOWING 子句确定的窗口是从当前元组往前取两个相邻的元组和往后取两个相邻元组构成的，也就是[5，5.5，6，6.5，7]. 而采用 RANGE BETWEEN 2 PRECEDING AND 2 FOLLOWING 子句确定窗口大小时，首先要取出当前元组的值，然后往前或者往后读取相邻元组的值，如果两个值的差不超过 2，那么这个元组就包含在

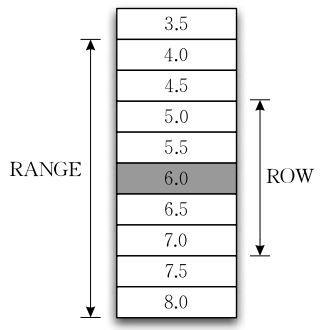


图 2　当前元组（图中的灰色部分）的取值为 6
（当窗口范围为 2 时，row 模式的窗口为[5，5.5，6，6.5，7]；而 range 模式对应的窗口为[4，4.5，5，5.5，6，6.5，7，7.5，8]）

窗口之中，直到差值超过 2 也就确定了窗口的边界，也就是[4,4.5,5,5.5,6,6.5,7,7.5,8]. 从中我们可以看出来，ROW 模式窗口数量相对固定，RANGE 模式则与元组的取值紧密相关，因此窗口大小不确定.

1.2 相关工作

当前，主流的数据仓库厂商，如 Teradata、Greenplum 等，针对高吞吐量、实时响应（比如作弊检测、风险控制）等应用都提供了不同程度的内数据库分析支持. Window 函数针对每个窗口都会计算出一列额外的属性. GROUP BY 及其 3 类扩展：GROUP SETS、ROLLUP 和 CUBE，是将数据表中的数据按照不同的划分方式进行分组，然后对每个组内的元组进行各种数据操作，其操作方式跟 Window 函数类似. 但是，如文献[3-5]等关于 GROUP BY 及其扩展的优化工作，并不适用于 Window 函数的执行框架. 主要有两方面的原因：(1) GROUP BY 及其扩展的计算模式和 Window 函数的计算模式并不一样. Window 函数在保留原始数据的基础上，计算出额外属性列，输出结果可以包含原始表的详细信息，而 GROUP BY 及其扩展是在数据表分组的基础上得出数据操作的信息，因此 GROUP BY 及其扩展的输出最多保留分组信息；(2) GROUP BY 及其扩展中，数据操作是针对整个分组的，而 Window 函数的语意特性更强，在分组的基础上可以指定任意物理上或逻辑上的窗口大小.

数据流领域也引入了窗口的概念. 在数据流领域，由于数据是持续不断产生的，因此，处理的数据具有数据量大和持续更新等特性. 在数据流领域，由于数据其特有的特征，将整个数据作为操作的对象是基本不可能的，常用的数据操作方法是使用滑动窗口（sliding window）来处理最新到达的数据，并在窗口范围内进行数据上的操作. 针对数据流上的窗口操作，尤其是聚合（aggregate）操作，已经有了大量的研究工作. 如文献[6]对经典的 top-k 问题做了研究，文献[7]则在滑动窗口中加入了语义信息. 由于在数据流处理过程中，其数据访问和结果要求与关系数据库存在差异，因此许多优化策略在关系数据库中并不适用. 文献[8]则针对滑动窗口计算过程中存在的很多重叠部分，提出了一种基于计算共享思想的优化方法. 在连续查询领域（continuous query），文献[9-12]也是基于这种计算共享思想对计算过程进行了相应的优化.

重排序和顺序调用是 Window 函数执行过程中的两个重要阶段. 文献[13-17]是重点关注 Window 函数执行优化工作的几篇文献. 其中，文献[13]提出了一种基于全排序（full sort）的重排序方法，文献[14]则在此基础上提出了更高效的哈希排序（hash sort）和分段排序（segment sort）. 文献[15]则从排序顺序（sort order）和排序共享（sort share）的角度，提出了一种协同排序技术，其排序代价极为接近全局最优. 文献[16]基于窗口分组共享的思想针对顺序调用阶段进行了优化，文献[17]基于 segment 树共享的思想针对顺序调用阶段进行了优化，都是一个通用框架，但不能保证每种类型的函数处理都能达到最优.

1.3 知识预定义

为了尽可能全面、细致地量化 Window 函数在数据库中的执行过程的消耗，在描述 Window 函数的消耗模型时需要用到以下几个参数，如表 3 所示.

表 3 参数表

参数	参数释义
n	数据表中元组个数
W_i	一个分区中的第 i 个窗口
w_i	窗口的大小
s	窗口 W_i 的起始位置
e	窗口 W_i 的终止位置
r_i	分区中第 i 个元组的位置
R_i	分区中第 i 个元组的值
C	一个元组的数据读取和计算消耗
C_t	更新临时窗口信息的消耗

2 Window 函数执行过程

2.1 两阶段执行框架

Window 函数的执行过程被分为两个部分（如图 3 所示）：(1) 重排序阶段：根据 PARTITION BY 子句和 ORDER BY 子句将表进行划分和重排序；(2) 顺序调用阶段：得到重排序后的表，对窗口内的元组依次调用转移函数（transition function），依次得到转移值（transition value），拥有最终计算函数（final function）的 Window 函数通过调用最终计算函数得到每个窗口的最终结果值. 如果一条查询中含有多个窗口函数，且每个函数的窗口定义都不一样，则会形成一条窗口函数链，顺序的执行窗口函数，当前窗口函数的输出结果，可以作为下一个窗口

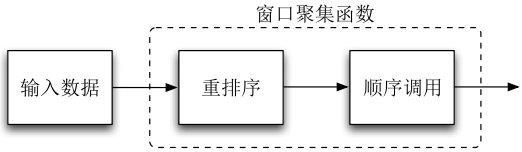


图 3 窗口函数的执行过程

函数的输入，每个窗口函数执行前都会有一个相应的重排序操作.

通过分析发现，重排序阶段对于所有类型的聚集函数来说基本一致. 主要的不同体现在执行的第 2 阶段，也就是顺序调用阶段. 顺序调用实际也包含两个过程，第 1 个过程是顺序地接收处理每一个划分，第 2 个过程则顺序地将窗口函数作用于划分中的每一行对应的窗口上，这两个过程对于所有窗口函数都是一样的. 但不同类型的函数在窗口上具体如何执行又各有不同，主要分为 3 类：分布型函数、偏移类函数和聚集类函数.

(1) 分布型函数，边框不会起作用，即在窗口函数的具体运算时，其窗口大小为 1，也就是只包含当前行，因此计算过程非常简单，只需遍历一遍数据. 即使是求百分比的函数，最多只需要再对每个分割提前做一次统计，然后这个统计值就可用于当前分割中所有行的计算，所占百分比通过当前行的值与这个统计值相除即可求出. 这样对于每个分割，最多只需要扫描两遍数据，一遍用于计算统计值，一遍用于获取当前行.

(2) 偏移类函数，其函数意义是将划分内的所有元组向前或者向后移动一段距离，对于这类函数，只需要在计算第一个元组时，确认其位置，然后顺序从该位置往后读取即可.

(3) 聚集类函数计算的对象是一个集合里的所有行，与传统的聚集函数类似，只不过聚集的范围被限定在当前窗口，每个窗口聚集函数都有一个转移函数和一个可选最终计算函数与之对应. 计算的过程会维护一个转移值，转移值本身可以是基本数据类型也可以是抽象类型.

在重排序阶段，MAX/MIN Window 函数跟其他 Window 函数一样，主要是针对 PARTITION BY 子句和 ORDER BY 子句对表进行划分和重排序. 由于 MAX/MIN Window 函数并没有最终计算函数，因此每个窗口得到的最终转移值即为该窗口的最终结果值. 因此，对于 MAX/MIN Window 函数，在顺序调用阶段，主要是对重排序后的表中的每一个窗口中的元组去顺序调用转移函数求转移值的过程.

2.2　顺序调用阶段的执行过程

本文是针对窗口聚集函数中的 MAX/MIN 进行优化. 在顺序调用阶段，主体的计算过程是由转移函数完成的. 对于像 AVG 等拥有最终计算函数的 Window 函数，最终结果是由最终计算函数获得. 对于像 MAX/MIN 等函数，并不拥有最终计算函数，其最终结果就是最终获得的转移值.

定义 1.　窗口(Window). 由 OVER 子句定义的包含一系列元组的集合(包含了一个起始位置和一个终止位置，并且拥有其对应的转移值)，$W_i(s,e,TV)$. 其中 W_i 指一个分区中的第 i 个窗口，s 指窗口 W_i 的起始位置，e 指窗口 W_i 的终止位置，TV 指窗口 W_i 的转移值.

对于例 1 中的 SQL 查询，PostgreSQL 中的 Window 函数在顺序调用阶段的执行过程如图 4 所示(有色的元组表示此窗口的当前行，黑色实体箭头表示转移值的计算过程)：对于第一个窗口 W_1，窗口的起始位置为 r_1，终止位置为 r_{11}. 转移函数依次作用于窗口中的每一个元组之上，获得的最终转移

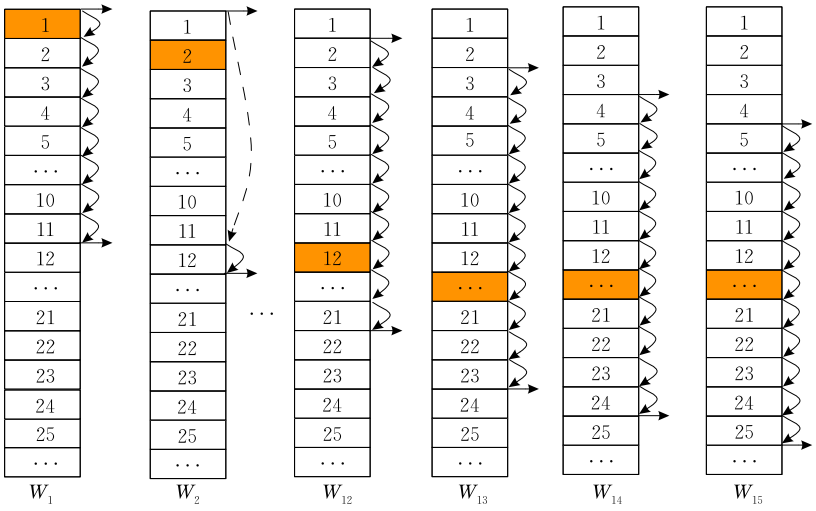


图 4　PostgreSQL 中 MAX/MIN Window 函数在顺序调用阶段的执行过程

值即是窗口 W_1 要求的最终结果值. 对于窗口 W_2，起始位置为 r_1，终止位置为 r_{12}. 相对于窗口 W_1 来说，只比其多出了位置 r_{12} 处的这一个元组. 因此，对于窗口 W_2 来说，并不需要重新计算 r_1 至 r_{11} 的元组的转移值. 只需将窗口 W_1 的转移值 $W_1.TV$ 赋予窗口 W_2 的转移值 $W_2.TV$，并计算 r_{12} 处的元组的转移值，得到的最终转移值即是窗口 W_2 要求的最终结果值. 接下来的计算过程都是如此直到窗口 W_{12}. 从窗口 W_{12} 开始，窗口的起始位置不再是 r_1，开始逐渐依次增加. 对于窗口 W_{12}，计算过程与窗口 W_1 类似，转移函数依次作用于窗口中的每一个元组之上，最终的转移值即是要求的最终结果值. 后面的 W_{13}、W_{14} 等窗口的计算过程同窗口 W_{12} 类似.

算法 1. PostgreSQL 顺序调用算法.

输入：经过重排序的表 T'

输出：每一个元组所对应的窗口的 MAX/MIN 函数值

```
1.  FOR 表 T'中的每一个分区 P DO
2.    FOR 分区 P 中的每一个窗口 W_i DO
3.      初始化 W_i.s, W_i.e, W_i.TV;
4.      IF W_i.s==W_{i-1}.s THEN
5.        W_i.TV←W_{i-1}.TV;
6.        FOR each row in (W_{i-1}.e, W_i.e] DO
7.          W_i.TV←transfunc(W_i.TV, R_m);
8.      ELSE
9.        FOR each row in [W_i.s, W_i.e] DO
10.         W_i.TV←transfunc(W_i.TV, R_m);
11.     RETURN W_i.TV;
```

PostgreSQL 中的 MAX/MIN Window 函数，在顺序调用阶段，转移函数执行过程的具体算法如算法 1 所示：首先，初始化当前窗口的参数 (s, e, TV)，并将读指针置于窗口的起始位置. 当前窗口的起始位置与上一个窗口的起始位置比较，如果位置不相同，则遍历窗口内的所有元组，并计算其相应的转移值(第 8～10 行). 如果位置相同，则只需遍历与上一个窗口相比新增的元组，并计算其相应的转移值(第 4～7 行).

2.3 消耗模型

为了更好的发现 MAX/MIN Window 函数在顺序调用阶段执行过程中存在的瓶颈，我们建立一个量化模型去刻画其在数据库执行过程中的消耗.

首先，我们假设表中共有 n 个元组($n\geq 1$)，n 个元组处在同一分区中，且在 Window 函数顺序调用阶段的执行过程中，每一个元组进行数据读取和转移函数计算的总消耗为 C.

由于表中的每一个元组作为当前行时都会唯一的确定一个窗口，因此，包含有 n 个元组的表，在 Window 函数执行过程中共有 n 个窗口的函数值需要计算. 在此，我们假设 n 个窗口的大小都是 w，且每个窗口的起始位置相对于上一个窗口都向下平移一个元组. 则传统计算框架中，Window 函数在顺序调用阶段的消耗为

$$Cost_{pg}=nwC.$$

从中可以看出，在每一个元组的数据读取和计算消耗固定的情况下，执行过程中的瓶颈主要存在于元组重复地进行数据读取和计算，也就是在窗口头部发生变化时，上一个窗口的转移值无法重新利用，需要从头重新计算，而两个相邻窗口转移值计算时的数据大部分都是重叠的，因而计算效率非常低.

当然这种默认的方式在有些情况下也是有比较不错的效率，之前求取的消耗代价 nwC 是在假设窗口大小固定，依次下移的前提下计算的. 如果窗口定义时采用类似 BETWEEN UNBOUNDED PRECEDING and CURRENT ROW 的方式，即窗口大小是从第一行到当前行，那么默认的这种执行方式除了第一次需要遍历所有元组之外，每次只需要额外计算一个新的转移值即可，计算消耗为

$$Cost_{pg}=nC.$$

这时消耗代价仅为 $O(n)$级别. 只不过在实际应用当中采用这种定义方式的情况相比 BETWEEN *value* PRECEDING and *value* FOLLOWING 的定义方式要少很多，不具有代表性，我们着重讨论更为一般的情况，以后不做特殊说明都是指最为一般的情况.

3 基于临时窗口的 MAX/MIN Window 函数优化

MAX/MIN Window 函数是不拥有最终计算函数的 Window 函数. 因此，每个窗口的最终转移值即是其要求的 Window 函数值. 优化 MAX/MIN Window 函数的核心思想是在函数执行过程中维持一个临时窗口，其中包含了此临时窗口对应的临时转移值，用于共享计算. 以此来减少在顺序调用过程中重复进行的数据元组的读取和计算消耗.

3.1 优化执行过程

定义 2. 临时窗口(Temporary Window). 临

时产生的包含一系列元组的集合(包含了一个起始位置和一个终止位置，并且拥有其对应的临时转移值)，TW(h,t,TTV). 其中 h 为临时窗口 TW 的起始位置，t 为临时窗口 TW 的终止位置，TTV (Temporary Transition Value)为临时窗口 TW 的临时转移值.

对于例 1 中的 SQL 查询，其优化算法的执行过程如图 5 所示(图中有色元组为每个窗口的最大值所处位置)：对于第一个窗口 W_1，转移函数依次作用于窗口中的每一个元组之上，与此同时，并记录下最大转移值的位置(r_3)来作为临时窗口的起始位置(h)，记录下窗口的终止位置(r_{11})作为临时窗口的终止位置(t)以及记录下最大转移值作为临时窗口的临时转移值(TTV). 对于窗口 W_2，起始位置为 r_1，终止位置为 r_{12}. 相对于窗口 W_1 来说，只比其多出了位置 r_{12} 处的这一个元组. 因此，对于窗口 W_2 来说，并不需要重新计算 r_1 至 r_{11} 处的元组的转移值. 只需将窗口 W_1 的转移值 $W_1.TV$ 赋予窗口 W_2 的转移值 $W_2.TV$，并计算 r_{12} 处的元组的转移值，得到的最终转移值即是窗口 W_2 要求的最终结果值.

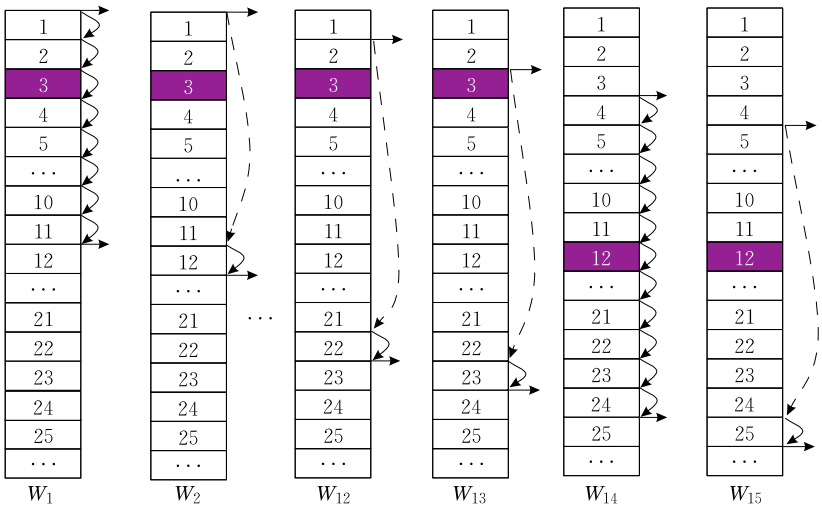


图 5　计算过程

由于 R_{12} 小于 TTV，因此我们只需将临时窗口的终止位置(t)的值由 r_{11} 更新为 r_{12} 即可. 后面的窗口的执行过程与窗口 W_2 的计算过程类似. 对于窗口 W_{12}，其起始位置与上一个窗口 W_{11} 相比向下移动了一个元组，但是并没有超过记录的临时窗口的起始位置，因此决定了，窗口 W_{12} 中的前 20 个元组中的最终转移值为记录下来的临时窗口的临时转移值(TTV). 对于前 20 行元组，我们无需重复的进行数据读取和调用转移函数，只需将临时窗口的临时转移值(TTV)赋给窗口 W_{12} 的转移值($W_{12}.TV$)即可. 接下来，我们只需对 r_{22} 处的元组进行数据读取和调用转移函数，并更新临时窗口的终止位置. 同理，窗口 W_{13} 的计算过程也与窗口 W_{12} 的计算过程类似. 但是，对于窗口 W_{14} 来说，其起始位置(r_4)超过了临时窗口的起始位置(r_3)，因此，临时窗口已经不再适用于窗口 W_{14} 的计算过程. 因此，窗口 W_{14} 的计算过程与窗口 W_1 的类似，转移函数依次作用于窗口中的每一个元组之上，与此同时，并记录下最大转移值的位置(r_{12})作为临时窗口的起始位置(h)，记录下窗口的终止位置(r_{24})作为临时窗口的终止位置(t)，记录下最大转移值作为临时窗口的临时转移值(TTV). 对于窗口 W_{15}，由于其起始位置(r_5)没有超过临时窗口的起始位置(r_{12})，因此，窗口 W_{15} 的计算过程与窗口 W_{12} 类似. 后面的窗口的计算过程以此类推.

计算过程中，不仅需要更新临时窗口的终止位置，同时还需要更新临时窗口的临时转移值和其起始位置. 例如，对于例 1 中的 SQL 查询，我们假设前 11 个窗口的计算过程如图 6 中所示一样(前 21 行中的最大值为 R_3 的值)，并且 R_{22} 的值大于 R_3 的值. 因此，对于窗口 W_{12} 来说，由于其起始位置(r_2)并没有超过临时窗口的起始位置(r_3)，因此，r_2 处到 r_{21} 处的元组并不需要重复的进行数据读取和调用转移函数，只需对 r_{22} 处的元组进行数据读取和调用转移函数即可. 由于 R_{22} 的值大于 R_3 的值，因此我们需要将临时窗口的临时转移值更新为 R_{22} 的值，并且将临时窗口的起始位置由 r_3 更新为 r_{22}. 对于窗口 W_{13} 来说，其起始位置(r_3)没有超过临时窗口的起始位置(r_{22})，因此，计算过程类似于窗口 W_2. 同理，后面的窗口的计算过程以此类推.

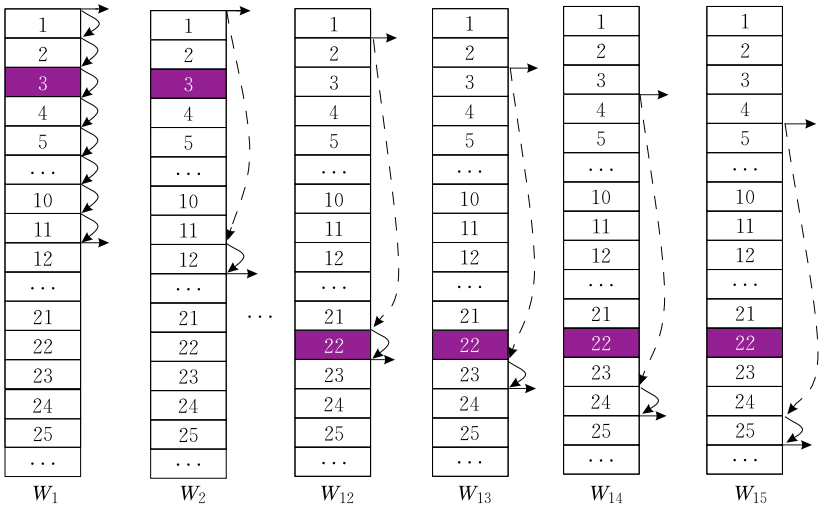


图 6 更新过程

算法 2. TW(Temporary Window)顺序调用优化算法.

输入:经过重排序的表 T'

输出:每一个元组所对应的窗口的 MAX/MIN 函数值

1. FOR 表 T'中的每一个分区 P DO
2. 初始化 h,t,TTV;
3. FOR 分区 P 中的每一个窗口 W_i DO
4. 初始化 $W_i.s,W_i.e,W_i.TV$;
5. IF $W_i.s==W_{i-1}.s$ THEN
6. $W_i.TV \leftarrow W_{i-1}.TV$;
7. FOR each row in $(W_{i-1}.e,W_i.e]$ DO
8. $W_i.TV \leftarrow$ transfunc$(W_i.TV,R_m)$;
9. IF $W_i.TV!=TTV$ THEN
10. $h \leftarrow r_m$;
11. $TTV \leftarrow W_i.TV$;
12. $t \leftarrow W_i.e$;
13. ELSE IF $W_i.s \leq h$ THEN
14. $W_i.TV \leftarrow TTV$;
15. FOR each row in $(t,W_i.e]$ DO
16. $W_i.TV \leftarrow$ transfunc$(W_i.TV, R_m)$;
17. IF $W_i.TV!=TTV$ THEN
18. $h \leftarrow r_m$;
19. $TTV \leftarrow W_i.TV$;
20. $t \leftarrow W_i.e$;
21. ELSE
22. FOR each row in $[W_i.s, W_i.e]$ DO
23. $W_i.TV \leftarrow$ transfunc$(W_i.TV, R_m)$;
24. IF $W_i.TV!=TTV$ THEN
25. $h \leftarrow r_m$;
26. $TTV \leftarrow W_i.TV$;
27. $t \leftarrow W_i.e$;
28. RETURN $W_i.TV$;

MAX/MIN Window函数 TW(Temporary Window)顺序调用优化算法在顺序调用阶段执行过程的具体算法如算法 2 所示:对于数据表中的每一个分区,初始化临时窗口的参数(h,t,TTV)(第 2 行).对于分区中的每一个窗口,初始化当前窗口的参数(s,e,TV),并将读指针置于窗口的起始位置(第 4 行).当前窗口的起始位置与上一个窗口的起始位置相比较,如果位置相同,则只需遍历与上一个窗口相比新增的元组,并计算其相应的转移值.与此同时,如果转移值不等于临时窗口的临时转移值,就更新临时窗口的起始位置和临时转移值.所有元组遍历结束后更新临时窗口的结束位置(第 5~12 行).如果位置不相同,则比较当前窗口的起始位置与临时窗口的起始位置.如果当前窗口的起始位置不大于临时窗口的起始位置,首先将临时窗口的临时转移值赋予当前窗口的转移值,并遍历临时窗口的终止位置后直到当前窗口的终止位置处的元组及计算相应的转移值.与此同时,如果转移值不等于临时窗口的临时转移值,就更新临时窗口的起始位置和临时转移值.所有元组遍历结束后更新临时窗口的结束位置(第 13~20 行).如果当前窗口的起始位置大于临时窗口的起始位置,遍历窗口内的所有元组,并计算其相应的转移值.与此同时,如果转移值不等于临时窗口的临时转移值,就更新临时窗口的起始位置和临时转移值.所有元组遍历结束后更新临时窗口的结束位置(第 21~27 行).

3.2 消耗模型

与 PostgreSQL 原有顺序调用阶段的消耗模型类似,我们假设表中共有 n 个元组($n \geq 1$),n 个元组处在同一分区中,且在 Window 函数的执行过程中,每一个元组的数据读取和计算消耗为 C.假设 n 个

窗口的大小都是 w，且每个窗口的起始位置都相对于上一个窗口向下平移一个元组. 更新一次临时窗口信息的消耗为 C_t.

优化后的方法计算效率和数据分布有一定的关系，在最好情况下是不需要重新遍历窗口，也就是说在当前窗口起始位置快要逼近临时窗口起始位置时，正好更新临时转移值，因为 $C \geq C_t$ 这种情况下的消耗接近于 nC. 而在最坏情况下，每一次都需要重新选择临时转移值，也就是退化为没有优化的情况，消耗变成 $nw(C+C_t)$.

最好和最坏不具有代表性，下面我们讨论更为一般的情况. 对于第一个窗口，我们需要将从窗口起始位置到窗口终止位置处的所有元组依次进行数据的读取和计算，并且更新临时窗口的信息. 因此，对于第一个窗口的消耗为

$$Cost_{W_1} = wC + C_t + (w-1)p_tC_t,$$

其中 p_t 指的是每次遍历一个元组时需要更新临时窗口信息的平均概率.

对于后面的窗口，当窗口的起始位置没有超越临时窗口的起始位置时，共用临时窗口的信息(同时更新临时窗口的信息). 当窗口的起始位置超过临时窗口的起始位置时，无可避免的需要重新遍历窗口内的所有元组. 因此. 剩余窗口的消耗为

$$Cost_{w_2 \to w_n} = (n-1)p(wC + C_t + (w-1)p_tC_t) + (n-1)(1-p)(C+C_t),$$

其中 p 指的是一个窗口需要重新遍历全部元组的平均概率.

总消耗为

$$\begin{aligned} Cost = {} & (1+(n-1)p)wC + (1+(n-1)p)C_t + \\ & (1+(n-1)p)(w-1)p_tC_t + \\ & (n-1)(1-p)(C+C_t). \end{aligned}$$

令 $f = Cost - Cost_{pg}$，则得

$$\begin{aligned} f = {} & (n-1)(w-1)C_tp_tp + (n-1)(w-1)Cp + \\ & (w-1)C_tp_t + nC_t + (n-1)(1-w)C. \end{aligned}$$

当 $w=1$ 并且 $p=1$ 时，$f = nC_t > 0$.

当 $w>1$ 时，函数 f 简写形式如下：

$$f = \theta_1 p_t p + \theta_2 p + \theta_3 p_t + \theta_4.$$

由于 $C \gg C_t$，很显然 $\theta_2 > |\theta_4|$，$\theta_1 > \theta_3$. 令 $|\theta_4| - \theta_1 > 0$，化简得 $\dfrac{C}{C_t} > 1 + \dfrac{n}{(n-1)(w-1)}$，当 $n>1$，$w>1$ 时，$1 + \dfrac{n}{(n-1)(w-1)} < 3$，实际上 n 和 w 都比较大时 $1 + \dfrac{n}{(n-1)(w-1)} \to 1$，而我们前提条件 $C \gg C_t$(尤其是数据量足够大，致使数据库工作内存已经无法存放所有数据，使得部分数据不得不存放在磁盘上)，即 $|\theta_4| - \theta_1 > 0$ 成立，从而得到 $\theta_2 > |\theta_4| > \theta_1 > \theta_3$. 令 $\theta_1 = 200$，$\theta_2 = 1110$，$\theta_3 = 22$，$\theta_4 = -1100$ 可得如下函数图像(图 7). 其中横轴为 p，纵轴为 p_t，竖轴为 f.

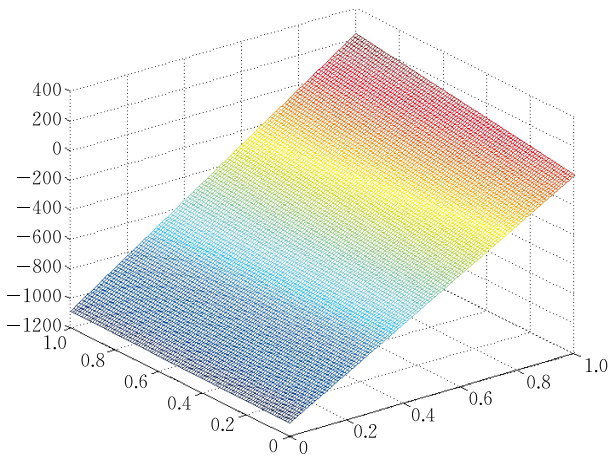


图 7　消耗差函数图像

随着窗口 w 和数据量 n 的逐渐增加，θ_4 对整个函数下移的影响要远比其他 3 个参数对函数上移的影响大. 因此，窗口越大，数据量越多，优化效果越明显.

由于 p_t 指的是每次遍历一个元组时需要更新临时窗口信息的概率，p 指的是一个窗口需要重新遍历全部元组的概率，因此 p_t、p 反映了一种数据分布趋势. 由以上分析可以看出，数据分布与取值函数越趋于一致，优化效果会越好.

3.3　总　结

本节主要介绍 MAX/MIN 窗口函数的优化算法并分析其计算效率. 我们的 MAX/MIN 窗口函数优化算法简单而有效，仅仅在 PostgreSQL 默认执行算法的基础之上增加一个结构用于存放临时转移值，从额外空间消耗的角度上来说几乎可以忽略. 但是从时间消耗的角度上，通过我们前两部分的介绍可以知道，优化后的算法在绝大多数情况下是远好于默认执行方法，即便在最差情况下也仅仅是退化为与默认算法同等的时间复杂度.

表 4　默认执行方式与优化执行方式消耗代价比较

执行情况	默认执行	优化执行
最好情况	nC	nC
最差情况	nwC	$nw(C+C_t)$
一般情况	nwC	$(1+(n-1)p)wC+(1+(n-1)p)C_t+(1+(n-1)p)(w-1)p_tC_t+(n-1)(1-p)(C+C_t)$

对于默认执行方式最好情况所采用的窗口定义形式(BETWEEN UNBOUNDED PRECEDING and

CURRENT ROW),我们的优化策略依然能够保证最少消耗为 nC. 而我们优化策略在最坏情况下,只比默认执行策略多消耗 nwC_t,多出的开销是为了维护临时窗口信息.

最为重要的是,上述两种极端情况在实际查询当中所占的比例很低,也就是说大多数情况下,优化策略能够发挥其作用,对于系统效率的提高大有裨益. 更为具体的优化的效果我们将在下一部分通过实验加以说明.

4 实验结果与分析

4.1 实验环境

我们更改了 PostgreSQL 9.3.6 的内核,实现了基于临时窗口的 MAX/MIN Window 函数优化算法. 我们将其与 Microsoft SQL Server 2012(Express Edition)共同部署在一台 ThinkPad X220i 电脑上. CPU 型号是 Intel(R) Core(TM) i3-2310M CPU@ 2.10 Hz,内存 4 GB 1333 MHz DDR3. 所有数据库的工作内存(work_mem)设置为 500 MB.

4.2 实验数据

本文的实验数据是使用 TPC-H DBGEN 生成的. 指令如下:

1. dbgen−S 1−T O
2. dbgen−S 10−T O

生成的是 TPC-H 中的表"order",指令 1 生成的数据集大小为 170 MB,包含 1 500 000 条元组,指令 2 生成的数据集大小是 1.7 GB,包含 15 000 000 条元组.

4.3 SQL 查询语句

本文实验采用的 SQL 查询如下所示:

```
SELECT
  o_totalprice, MAX(o_totalprice)
  OVER (ORDER BY o_orderkey ROWS BETWEEN
  frameoffset PRECEDING AND frameoffset
  FOLLOWING)
FROM
  order;
```

我们通过更改参数 $frameoffset$ 来改变实验中窗口的大小.

4.4 实验对比

(1) PG: PostgreSQL 中原有的顺序调用算法;

(2) TW: TW(Temporary Window)顺序调用优化算法;

(3) SQL Server: MS SQL Server 2012(Express Edition).

4.5 实验结果

本文的实验对比方法有 3 种:(1) PostgreSQL 数据库,原有执行框架;(2) 实现了基于临时窗口的 MAX/MIN Window 函数优化算法的 PostgreSQL 数据库;(3) Microsoft SQL Server 2012(Express Edition).

图 8 是本文实验采用的 SQL 查询在 170 MB 数据集上的 SQL 查询执行时间对比图. 其中横轴为 $frameoffset$,大小从 10 到 500,纵轴为 SQL 查询执行时间,单位为秒(s). 由图中可以看出,采用优化算法 TW 的执行效率要远好于 PG 和 SQL Server. 而且,随着窗口的增大,TW 的执行效率的提升越来越明显,其基本稳定在 5 s 之内,相比之下,PG 和 SQL Server 的执行时间已经增长到几百秒.

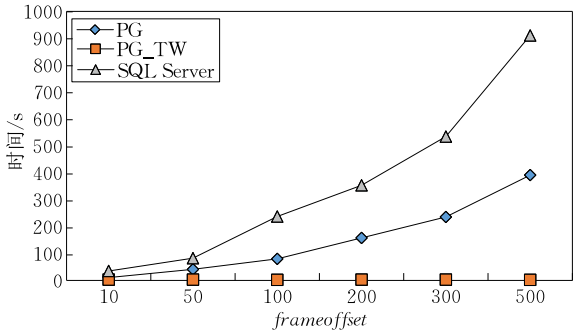


图 8 MAX Window 函数执行时间(170 MB)

图 9 是本文实验采用的 SQL 查询在 1.7 GB 的数据集上的 SQL 查询执行时间对比图. 其中横轴为 $frameoffset$,大小从 10 到 500,纵轴为 SQL 查询执行时间,单位为秒(s). 与图 8 结果类似,采用优化算法 TW 的执行效率要远好于 PG 和 SQL Server. 图 9 中 3 种数据库的 SQL 查询执行时间,相比于图 8 都要有所上升,这是因为数据集的增大导致了查询处理过程中相应窗口的数量也随之增加,使得查询处理时间随之上升.

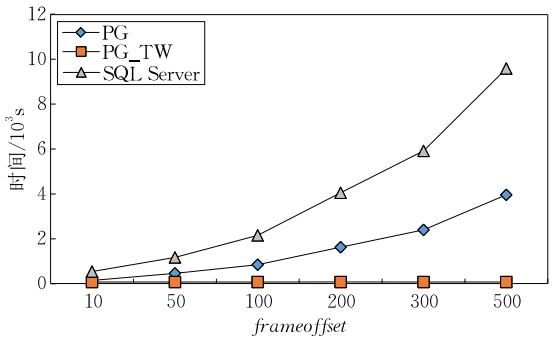


图 9 MAX Window 函数执行时间(1.7GB)

图 10 是 TW 在不同数据集和不同的窗口大小下 SQL 查询执行时间结果图，其中横轴是 $frameoffset$，纵轴是 SQL 查询执行时间. 由图 7 可以看出，基于临时窗口的 MAX/MIN Window 函数优化算法 TW，在数据集大小一定的情况下，SQL 查询的执行时间随着窗口大小的变化基本维持稳定，并不会出现较大幅度的改变. 相比于 PG、SQL Server 在数据集一定的情况下，SQL 查询执行时间随着窗口的增大而呈指数级增长，TW 的这一特性，在数据处理中会更加有优势.

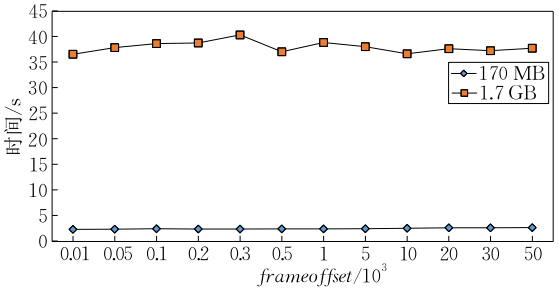


图 10　MAX Window 函数执行时间

5　结束语

针对 MAX/MIN SQL Window 函数在顺序调用阶段的执行，本文提出了一种基于临时窗口的优化算法. 利用执行过程中产生的一些临时结果，极大地避免了数据库中元组的重复读取和调用转移函数，从而大大提高了 MAX/MIN SQL Window 函数在顺序调用阶段的执行时间和效率. 与现有的算法相比，该算法在保证结果正确的基础上极大提高了运行效率，其独有的特性在处理大数据时也有明显的优势.

致　谢　北京 EMC 实验室的曹逾博士对本文的模型完善和实验设计提出了有益的建议. 评审老师对本文提出了宝贵的修改建议. 在此表示感谢！

参　考　文　献

[1] Zuzarte C, Pirahesh H, Ma Wenbin, et al. WinMagic: Subquery elimination using window aggregation//Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data. San Diego, USA, 2003: 652-656

[2] Bellamkonda S, Ahmed R, Witkowski A, et al. Enhanced subquery optimizations in oracle. Proceedings of the VLDB Endowment, 2009, 2(2): 1366-1377

[3] Agarwal S, Agrawal R, Deshpande P, et al. On the computation of multidimensional aggregates//Proceedings of the 22nd International Conference on Very Large Data Bases. Bombay, India, 1996: 506-521

[4] Chatziantoniou D, Ross K A. Querying multiple features of groups in relational databases//Proceedings of the 22nd International Conference on Very Large Data Bases. Bombay, India, 1996: 295-306

[5] Chen Zhimin, Narasayya V. Efficient computation of multiple group by queries//Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data. Baltimore, Maryland, 2005: 263-274

[6] Jin Cheqing, Yi Ke, Chen Lei, et al. Sliding-window top-k queries on uncertain streams. The International Journal on Very Large Data Bases, 2010, 19(3): 411-435

[7] Li Jin, Maier D, Tufte K, et al. Semantics and evaluation techniques for window aggregates in data streams//Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data. Baltimore, Maryland, 2005: 311-322

[8] Li Jin, Maier D, Tufte K, et al. No pane, no gain: Efficient evaluation of sliding-window aggregates over data streams. SIGMOD Record, 2005, 34(1): 39-44

[9] Arasu A, Babu S, Widom J. The CQL continuous query language: Semantic foundations and query execution. The International Journal on Very Large Data Bases, 2006, 15(2): 121-142

[10] Guirguis S, Sharaf M A, Chrysanthis P K, Labrinidis A. Optimized processing of multiple aggregate continuous queries//Proceedings of the 20th ACM International Conference on Information and Knowledge Management. Glasgow, United Kingdom, 2011: 1515-1524

[11] Arasu A, Widom J. Resource sharing in continuous sliding-window aggregates//Proceedings of the 30th International Conference on Very Large Data Bases. Toronto, Canada, 2004: 336-347

[12] Law Yan-Nei, Wang Haixun, Zaniolo C. Relational languages and data models for continuous queries on sequences and data streams. ACM Transactions on Database Systems, 2011, 36(2): 8:1-8:32

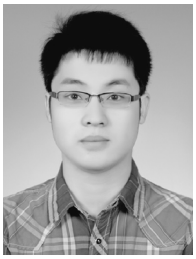
[13] Bellamkonda S, Bozkaya T, Ghosh B, et al. Analytic functions in oracle 8i. Oracle, San Francisco, USA: Technical Report, 2000

[14] Cao Yu, Chan Chee-Yong, Li Jie, Tan Kian-Lee. Optimization of analytic window functions. Proceedings of the VLDB Endowment, 2012, 5(11): 1244-1255

[15] Cao Yu, Bramandia R, Chan Chee-Yong, Tan Kian-Lee. Sort-sharing aware query processing. The International Journal on Very Large Data Bases, 2012, 21(3): 411-436

[16] Ma Jiansong, Cao Yu, Wang Xiaoling, et al. PGWinFunc: Optimizing window aggregate functions in PostgreSQL and its application for trajectory data//Proceedings of the 31st IEEE International Conference on Data Engineering. Seoul, Korea, 2015: 1448-1451

[17] Leis V, Kundhikanjana K, Kemper A, Neumann T. Efficient processing of window functions in analytical SQL queries. Proceedings of the VLDB Endowment, 2015, 8(10): 1058-1069



MA Jian-Song, born in 1990, M. S. candidate. His major research interests include database and data mining.

WANG Ke-Qiang, born in 1990, Ph. D. candidate. His major research interests include database, data mining and machine learning.

SONG Guang-Xuan, born in 1992, M. S. candidate. His major research interests include database, information retrieval.

ZHANG Kai, born in 1991, M. S. candidate. His major research interest is recommendation system.

WANG Xiao-Ling, born in 1975, Ph. D., professor, Ph. D. supervisor. Her major research interests include data management technology and data service.

JIN Che-Qing, born in 1977, Ph. D., professor, Ph. D. supervisor. His major research interests include streaming data management and location-based services.

Background

With the growing number of Internet users, the Internet application gradually enters the era of big data. How to store and analyze the big data becomes a problem in the Internet application. In-Database Analytics as a technology that integrate data storage with data analysis, has attracted the attention of many enterprises and researchers. Window Function, as a solution of In-Database Analytics in the field of relational database, makes it take the place of Self Join and Sub Queries to complete the traditional complex queries with its subtle semantic characteristics and is widely used in the current enterprise data management and analysis in the Internet application. Under the background of big data, Window Function has the bottleneck in the face of such demand as high throughput and real-time response.

Most of optimization works on Window Function are concentrated on the step of resorting. In the aspect of data stream, there are many works to optimize the aggregate operation of sliding window. In database, there are two main optimization ideas, group sharing and segment sharing.

In this paper, we design a new algorithm dependent on Temporary Window for the MAX/MIN Window functions contraposing the original framework in PostgreSQL. We design a query cost analysis model and theoretically proved the performance of the algorithm. We did the performance comparison between the new algorithm and the existing commercial database SQL Server and proved the effectiveness of the proposed algorithm.

This research is supported by the National Natural Science Foundation of China under Grant Nos. 61532021, 61170085, and 61472141, and the Shanghai Knowledge Service Platform Project (No. ZF1213).