HAL open science

Holistic Shuffler for the Parallel Processing of SQL Window Functions

Fábio Coelho, José Pereira, Ricardo Vilaça, Rui Oliveira

To cite this version:

Fábio Coelho, José Pereira, Ricardo Vilaça, Rui Oliveira. Holistic Shuffler for the Parallel Processing of SQL Window Functions. 16th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems (DAIS), Jun 2016, Heraklion, Crete, Greece. pp.75-81, 10.1007/978-3-319-39577-7_6 . hal-01434801

HAL Id: hal-01434801

https://inria.hal.science/hal-01434801v1

Submitted on 13 Jan 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Holistic Shuffler for the Parallel Processing of SQL Window Functions

Fábio Coelho, José Pereira, Ricardo Vilaça, and Rui Oliveira

INESC TEC & Universidade do Minho, Braga, Portugal
facoelho@inesctec.pt {jop,rmvilaca,rco}@di.uminho.pt

Abstract. Window functions are a sub-class of analytical operators that allow data to be handled in a derived view of a given relation, while taking into account their neighboring tuples. Currently, systems bypass parallelization opportunities which become especially relevant when considering Big Data as data is naturally partitioned. We present a shuffling technique to improve the parallel execution of window functions when data is naturally partitioned when the query holds a partitioning clause that does not match the natural partitioning of the relation. We evaluated this technique with a non-cumulative ranking function and we were able to reduce data transfer among parallel workers in 85% when compared to a naive approach.

1 Introduction

Window functions (WF) are a sub-group of analytical functions that allow to easily formulate analytical queries over a derived view of a given relation *R*. They allow operations like ranking, cumulative averages or time series to be computed over a given data partition. Listing 1.1 presents one window function that is expressed in SQL by the operator OVER, together with a partition by (PC) and an order by (OC).

Listing 1.1: Window Function example.

```
select rank() OVER(Partition By A Order By B) from R
```

The growing Big Data trend is shifting the processing of these functions to cloud environments deploying computation to a distributed mesh of computing nodes, where data and processing are naturally partitioned. The distributed execution of queries leverages on data partitioning as a way to attain gains associated with parallel execution. Nevertheless, the partitioning strategies are typically governed by a primary table key, which only benefits the cases where the partitioning of a query matches that same key as depicted in Figure 1. When the query has to partition data according to a different attribute in a relation, it becomes likely that the members of each partition will not all reside in the same node, which compromises the final result for a subgroup of non-cumulative analytical operators, such as `rank`, since all members of a distinct partition need to be handled by a single entity, in order not to incur in unnecessary sorting steps, which is the most costly operation [3].

In this paper we propose an Holistic shuffler which according to the partitioning considered by the ongoing window function will instruct workers to handle specific

select rank() OVER (partition by PK) from table

PK	A	B
1	1	1
1	1	2
1	2	1
1	3	1

node #1

PK	A	B
2	1	1
2	2	3
2	2	3
2	3	1

node #2

PK	A	B
3	2	1
3	2	1
3	2	3
3	2	3

node #3

Fig. 1: Data partitioning according to the primary key (PK).

partitions according to the data sizes they hold, minimizing data transfer among workers. We present the design and action of the shuffler which is based on prior knowledge of the data size distribution of each column in the relation, reflecting the data size held in each partition rather than considering the actual data value as seen in common use of database indexes. The preliminary evaluation of our mechanism shows that our approach is able reduce data transfer by 85% when compared with a naive approach.

Roadmap: In the remainder of this paper, Section 2 introduces the distribution considered and Section 3 presents the design and architecture of the Holistic Shuffler we propose. Section 4 accesses our approach. Section 5 briefly reviews related work and overviews our contributions.

2 Data Transfer Statistics

The use of indexes [5], histograms [6] and other heuristics are now a staple feature in modern database systems, as they allow to expedite operations, avoiding full scan operations over relations. More recently, these strategies started to be present in cloud infrastructures [4], allowing for processing on primary and secondary attributes.

key	PK	A	B
1	4	2	3
2	0	1	1
3	0	1	0

(a) (p1)

key	PK	A	B
1	0	1	2
2	4	2	0
3	0	1	2

(b) (p2)

key	PK	A	B
1	0	0	2
2	0	4	0
3	4	0	2

(c) (p3)

key	PK	A	B
1	p1	p1	p1
2	p2	p3	p1
3	p3	p1 / p2	p2 / p3

(d) Global

Fig. 2: Partition and Global Histogram construction.

Histograms are commonly used by query optimizers as they provide a fairly accurate estimate on the data distribution, which is crucial for a query planner. An histogram allows to map keys to their observed frequencies. Database systems use these structures to measure the cardinality of keys or key ranges. Without histograms, the query planner would have to assume uniform distribution of data, leading to incorrect partitioning, particularly with skewed data [7], a common characteristic of non synthetic workloads.

When a query engine has to generate parallel query execution plans to be dispatched to distinct workers, each one holding a partition of data; having histograms like the aforementioned ones is an asset, but it does not completely present an heuristic that could be used to enhance how parallel workers would share preliminary and final results. This is so as such histograms only introduce and insight about the cardinality of each partition key. In order to improve bandwidth usage, thus reducing the amount of traded information, the histogram also needs to reflect the volume of data existing in each node, instead of just considering the row cardinality.

2.1 Histogram Construction

The cornerstone of the contribution we present is based on merging the knowledge for row cardinality and average row size for each partition. Both could be seen as global metrics that a given query engine may be able to produce and maintain, as this type of information is already used for similar purposes. The cadence at which the histogram should be updated was left outside of the scope of this paper due to space constraints. Nevertheless we note it as a relevant topic, since the optimal performance of any heuristic based approach is entirely connected with its own representativity. Figure 1 presents the result of hash partitioning a relation in 3 workers, according to key PK. The histogram to be built will consider the cardinality of each value in each attribute of the relation for each single partition. Since the construction of the histogram should not be done during query planning time, it cannot know beforehand the partitioning clauses induced by queries. As such, we consider all distinct groups of values in each attribute. Each partition will contribute to the histogram with the same number of attributes as the original relation, plus a key, reflecting the data in that partition. Afterwards, each worker should be able to share its partial histogram with the remainder workers in order to produce the global histogram.

Algorithm 1 Histogram Construction in Partition n

1: **procedure** COUNT_DISTINCT_KEYS($attr$)
2: $\quad$ *foreach key* : $attr$
3: $\quad$ $count \leftarrow count(distinct)$
4: $\quad$ $size \leftarrow size(key)$
5: $\quad$ $hist_P_n(key, attr) \leftarrow (count, size)$
6: $P_n \leftarrow [attr_1, attr_2, attr_n]$
7: $hist_P_n \leftarrow [key, attr_1, attr_2, attr_n]$
8: **function** GLOBAL HISTOGRAM(P_n)
9: $\quad$ *for each attr* : P_n
10: $\quad$ COUNT_DISTINCT_KEYS($attr$)

Algorithm 1 governs how each partition histogram ($hist_P_n$) should be built. Briefly, each attribute ($attr$) is traversed and for each key, the total number of distinct occurrences of that key is computed, together with its size. The pair of values is then added to the histogram. The tables in Figures 2(a), 2(b) and 2(c) present the resulting histograms for each partition according to Figure 1.

When all workers have completed computing the histogram regarding its own physical partition, they need to share it with a designated worker, so that the global histogram is also computed. The global histogram will traverse each physical partition histogram and evaluate, for each key, which is the physical partition that holds the largest volume (in size, evaluating the $cardinality \times average_row_size$). The table in Figure 2(d) depicts the final result of the global histogram. The global histogram will have the same number of attributes of each partition histogram. Please note that the keys for both the physical partition and the global histograms are not the primary keys of the relation, but rather the distinct values found in each attribute during the construction of each partition

histogram. Therefore, we provide a brief example on how to read this histogram. Consider that a given query requires data to be partitioned according to attribute A. Then, the histogram informs that key 1 and 2 have the largest volume of data respectively in partitions $p1$,$p3$ and, and regarding key 3, partitions $p2$ and $p3$ both hold the same volume.

3 Holistic Shuffler

The Holistic Shuffler leverages on the data distribution data collected by the Global Histogram in order to expedite shuffling operations. The Shuffle operator can be translated into a *SEND* primitive that forwards a bounded piece of data to a given destination, considering the underlying network to be reliable. During the workflow for processing a window operator, there are two different moments where data needs to be shuffled. The first moment occurs immediately after the operator starts, and its goal is to reunite partitions, thus fulfilling the locality requirement. The second moment occurs in the end of the operator and it is intended to reconcile partial results in order to produce the final result.

Both operators define distinct goals regarding the destinations that need to be chosen for each forwarding operation. Therefore, we establish two shuffle operators, the local shuffle and the global shuffle contemplating each set of requirements. The Local Shuffle will dispatch rows of a given partition to the worker responsible for that partition as dictated by the Global histogram. Algorithm 2 depicts the behavior of the operator. As each row is read from scanning the partition, the value contained in that row for the attribute that dictates the partition clause is collected ($partition$). This value is then used together with the partitioning attribute to obtain the destination worker from the global histogram. If this row is not meant to be handled by the ongoing worker, then it is forwarded to the correct worker.

Algorithm 2 Local Shuffle Operation

```
worker_id
row ← [key, attr_1, attr_n]
hist_P_n ← [key, row]
partition_by ← attr_1
function LSHUFFLE(local_partition)
    for each row : local_partition
    partition ← row[attr_1]
    destination ← hist_p_n[partition, attr_1]

    if worker_id ≠ destination then
        SEND(destination, row)
```

Algorithm 3 Global Shuffle Operation

```
worker_id
master_worker ← hist_P_n
function GSHUFFLE(aggregated_data)
    foreach row : aggregated_data

    if worker_id ≠ master_worker then
        SEND(master_worker, row)
```

The Global Shuffle will forward all aggregated rows to the worker that will hold the overall largest data volume, which will from now on designate as master worker. By instructing the workers that hold the least volume of data, we are promoting the minimal usage of bandwidth possible. Algorithm 3 reflects the behavior for the Global

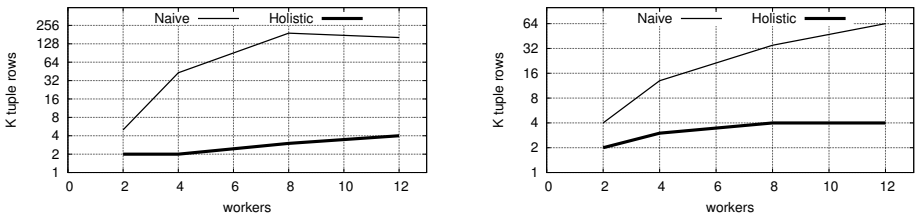
Shuffle operator. The input data considered by the Global Shuffler is composed by the ordered and aggregated rows, both produced by earlier stages of the worker work flow. Such rows will now have to be reconciled by a common node, which for this case will be dictated by the master node, as previously stated. Upon start, the Global Shuffle operator will interrogate the histogram regarding the identity of the master node. Afterwards, as each aggregated row is handled by the operator, it is forwarded to the master worker, if the master node is not the current one.

4 Evaluation

Along the current Section, we present the preliminary evaluation for the contributions we propose. In order to evaluate our contribution, we used RX-Java [2] to simulate the parallel execution of the window operator in several workers. This framework establishes bindings to the Java language, enabling it to use the semantics of Reactive Programming [1]. We selected this framework as it allows to establish a series of data streams, mimicking the window operator data flow. Throughout the evaluation, we employed a single ranking query (listing 4) holding a window function over a synthetically-generated relation as in TPC-C's Order Line relation, holding 10 distinct attributes.The values considered for each of these attributes were distributed according to TPC-C's specification. The generated data composes 100 distinct partitions, each one with 500 rows. Globally, the Order Line relation held 500 K tuples.

```
select rank() OVER (partition by OL_D_ID order by OL_NUMBER)
    from Order Line
```

The experiments were performed on a system with an Intel i3-2100-3.1GHz 64 bit processor with 2 physical cores (4 virtual), 8GB of RAM memory and SATA II (3.0Gbit/s) hard drives, running Ubuntu 12.04 LTS as the operating system.



(a) Average Forwarded Rows for both stages (b) Forwarded Rows for the Shuffle Local

Fig. 3: Comparison results between the Naive and Holistic approach.

For comparison purposes, we report the results by using a naive approach and our Holistic Shuffler. The naive approach, instead of using any knowledge to forward data, disseminates all data among all participating workers. The results in both pictures are depicted according to a logarithmic scale, in the average of 5 independent tests for each configuration.

The Holistic technique we propose required in average only 14.7% of the rows required for the Naive approach to reunite all the partition in each computing node, as depicted in Figure 3(a). The large difference is justified by the fact that the naive approach reunites partitions by forwarding data among all participating nodes, which intrinsically creates duplicates in each node, growing in proportion to the number of nodes. The Local Shuffling stage is depicted in Figure 3(b), in which we varied the number of computing nodes that participate in the computation of the ranking query, verifying the number of rows that were forwarded according to each technique.

5 Related Work and Conclusion

Despite its relevance, optimizations considering this operator are scarce in the literature. The work by [3] or [8] are some of the exceptions. Respectively, the first overcomes optimization challenges related with having multiple window functions in the same query, while the second presents a more broad use of window functions, showing that it is possible to use them as a way to avoid sub-queries and reducing execution time down from quadratic time.

In this paper we proposed an Holistic Shuffler, tailored to be used for the efficient parallel processing of queries with non-cumulative window functions. The design is based on a statistical method that can be used to reduce the amount of data transfered among computing nodes of a distributed query engine, where data is naturally partitioned. Moreover, the preliminary evaluation we present shows that by applying this methodology we were to reduce in average 85% of data transferred among computing nodes. As future work, we plan to translate this approach to a real query engine.

Acknowledgments

This work was part-funded by project LeanBigData : Ultra-Scalable and Ultra-Efficient Integrated and Visual Big Data Analytics (FP7-619606), and by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme within project ≪POCI-01-0145-FEDER-006961≫, and by National Funds through the FCT – Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) as part of project UID/EEA/50014/2013.

References

1. Reactive programming. http://reactivex.io (2015)
2. Reactive programming for java. https://github.com/ReactiveX/RxJava (2015)
3. Cao, Y., Chan, C.Y., Li, J., Tan, K.L.: Optimization of analytic window functions. Proceedings of the VLDB Endowment 5(11), 1244–1255 (2012)
4. Chen, G., Vo, H.T., Wu, S., Ooi, B.C., Özsu, M.T.: A framework for supporting dbms-like indexes in the cloud. Proceedings of the VLDB Endowment 4(11), 702–713 (2011)
5. Garcia-Molina, H.: Database systems: the complete book. Pearson Education India (2008)
6. Poosala, V., Ganti, V., Ioannidis, Y.E.: Approximate query answering using histograms. IEEE Data Eng. Bull. 22(4), 5–14 (1999)
7. Poosala, V., Haas, P.J., Ioannidis, Y.E., Shekita, E.J.: Improved histograms for selectivity estimation of range predicates. In: ACM SIGMOD Record. vol. 25, pp. 294–305. ACM (1996)
8. Zuzarte, C., Pirahesh, H., Ma, W., Cheng, Q., Liu, L., Wong, K.: Winmagic: Subquery elimination using window aggregation. In: Proceedings of the 2003 ACM SIGMOD international conference on Management of data. pp. 652–656. ACM (2003)