Reducing Data Transfer in Parallel Processing of SQL Window Functions

Fábio Coelho, José Pereira, Ricardo Vilaça and Rui Oliveira
INESC TEC & Universidade do Minho, Braga, Portugal

Keywords: Window Functions, Reactive Programming, Parallel Systems, OLAP, SQL.

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. We propose a technique that can be used in the parallel execution of this operator when data is naturally partitioned. The proposed method benefits the cases where the required partitioning is not the natural partitioning employed. Preliminary evaluation shows that we are able to limit data transfer among parallel workers to 14% of the registered transfer when using a naive approach.

1 MOTIVATION

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. Each window function is expressed in SQL by the operator OVER, which is complemented with a partition by (PC), an order by (OC) and a grouping clause (GC). A given analytical query may hold several analytical operators, each one bounded by a given window function. Each partition or ordering clause represents one, or a combination of columns from a given relation *R*.

Despite its relavance, optimizations considering this operator are almost nonexisting in the literature. The work by (Cao et al., 2012) or (Zuzarte et al., 2003) are some of the execeptions. 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.

Listing 1 depicts a SQL query where the analytical operator rank is bounded by a window function, which holds a partition and ordering clause respectively for columns A and B. The induced partitioning is built from each group of distinct values in the partition clause and ordered through the order by clause. A single value corresponding to the analytical operator is added to each row in the derived relation.

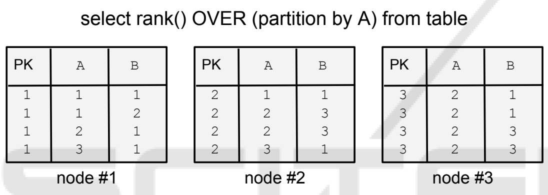
```
select rank() OVER(Partition By A
        Order By B) from table
```

Listing 1: Window Function example.

With the Big Data trend and growing volume of data, the need for real-time analytics is increasing, thus requiring systems to produce results directly from production data, without having to transform, conform and duplicate data as systems currently do. Therefore, parallel execution becomes the crux of several hybrid databases that fit in the category of Hybrid Transactional and Analytical Processing (HTAP). These systems typically need to leverage all parallelization and optimization opportunities, being usually deployed in a distributed mesh of computing nodes, where data and processing are naturally partitioned. In one of the methods to query such systems, each node computes the results for the data partitions they hold, contributing to the overall final result. Nonetheless, the data partitioning is usually achieved by means of a single primary column in a relation. This impacts mainly cases where the partitioning clause does not match the natural partitioning, thus compromising the final result for a sub group of non-cumulative analytical operators, as all members of a distinct partition need to be handled by a single entity. A naive solution would be to forward the partition data among all nodes, but the provision of such a global view in every node would compromise bandwidth and scalability.

Data distribution is among one of the cornerstones of a new class of data substrates that allows data to be sliced into a group of physical partitions. In order to split a relation, there are usually two main trends, namely: Vertical (Navathe et al., 1984) and Horizontal (Sadalage and Fowler, 2012) partitioning. To do so, the relation needs to be split either by using (e.g.) range or hash partitioning. Hashing algorithms are used to split a given domain – which is usually achieved through a primary key in database notation – into a group of buckets with the same cardinality of the number of computing nodes. An hash algorithm is defined by an hash function (H) that makes each computing node accountable for a set of keys, allowing to map which node is responsible for a given key.

The distributed execution of queries leverages on data partitioning as a way to attain gains associated with parallel execution. Nevertheless, the partitioning strategies typically rely on a primary table key to govern the partitioning, which only benefits the cases where the partitioning of a query matches that same key. 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.


select rank() OVER (partition by A) 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

Figure 1: Data partitioning among 3 workers.

Figure 1 presents the result from hash partitioning a Relation into 3 workers according to the primary key (PK). The query presented holds a window operator that should produce a derived view induced by partitioning attribute A.

Non-cumulative aggregations such as rank require all members of a given partition to be collocated, in order not to incur in the cost of reordering and recomputing the aggregate after the reconciliation of results among nodes. However, different partitions do not share this requirement, thus enabling different partitions to be processed in different locations. To fulfill the data locality requirement, rows need to be forwarded in order to reunite partitions.

The shuffle operator arises as way to reunite partitions and to reconcile partial computations originated by different computing nodes. The shuffler in each computing node has to be aware of the destination where to send each single row, or if it should not send it at all. Typically, this is achieved by using the modular arithmetic operation of the result of hashing the partition key over the number of computing nodes. However, this strategy is oblivious to the volume of data each node holds of each partition. In the worst case, it might need relocate all partitions to different nodes, producing unnecessary use of bandwidth and processing power.

In this position paper we show that if data distribution of each column in a relation is approximately known beforehand, the system is able to adapt and save network and processing resources by forwarding data to the right nodes. The knowledge needed is the cardinality and size (in bytes) in each tuple partition rather than considering the actual tuple value as seen in common use of database indexes. This knowledge would then be used by an Holistic shuffler which according to the partitioning considered by the ongoing window function would instruct workers to handle specific partitions, minimizing data transfer among workers.

2 STATISTICS

Histograms are commonly used by query optimizers as they provide a fairly accurate estimate on the data distribution, which is crucial for the query planner. An histogram is a structure which allows to map keys to their observed frequencies. Database systems use these structures to measure the cardinality of keys or key ranges. Relying on statistics such as histograms takes special relevance in workloads where data is skewed (Poosala et al., 1996), a common characteristic of non synthetic data, as their absence would induce the query optimizer to consider uniformity across partitions.

While most database engines use derived approaches of the previous technique, they only allow to establish an insight regarding the cardinality of given attributes in a relation. When considering a query engine that has to generate parallel query execution plans to be dispatched to distinct workers, each one holding a partition of data; such histograms do not completely present a technique that could be used to enhance how parallel workers would share preliminary and final results. This is so as they only introduce and insight about the cardinality of each partition key. In order to minimize bandwidth usage, thus reducing the amount of traded information, the histogram also needs to reflect the volume of data existing in each node. To understand the relevance of having an intuition regarding the row size, please consider that we have 2 partitions with exactly the same cardinality of rows that need to be shuffled among workers. From this point of view the cost of shuffling each row is the same. However, if the first row has an average size of 10 bytes, and the second 1000 bytes, then shuffling the second implies transferring 100 times more data over the network. This will be exacerbated as

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

(a) physical partition #1 (p1)

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

(b) physical partition #2 (p2)

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

(c) physical partition #3 (p3)

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

(d) Global

Figure 2: Partition and Global Histogram construction.

the difference in row size and the number of workers grows. Figure 1 presented the result of hash partitioning a relation in 3 workers according to key PK. The histogram to be built would consider the cardinality and size of each value in each attribute of the relation for each single partition. The construction of the histogram should not be done during query planning time as it cannot know beforehand the partitioning clauses induced by queries. Therefore, prior to execution, 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 would share its partial histogram with the remainder workers in order to produce the global histogram. The global histogram will map a given key to the partition that should handle it, by having the largest volume (in bytes) for that given key. Figure 2 depict the partial an global histograms produced over the paritions in Figure 1.

3 HOLISTIC SHUFFLER

Non-cumulative aggregations such as rank require the processing of partition members to be done under the same worker, in order not to incur in further unnecessary ordering stages (which present to be one of the most costly operations (Cao et al., 2012)). The Holistic Shuffler leverages the data distribution 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. We consider the underlying network to be reliable.

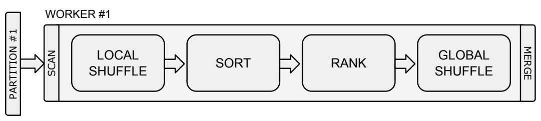


Figure 3: Parallel worker design.

During the work flow for processing a window operator, as depicted in Figure 3, there are two different moments where data needs to be shuffled. The first moment occurs immediately after the operator start, and its goal is to reunite partitions, thus fulfilling the locality requirement. The second moment occurs in the end of the operator and 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 each contemplating each set of requirements.

The Local Shuffle operator will be used whenever the window operator needs to reunite partition members between parallel workers. It will dispatch rows of a given partition to the worker that holds the largest volume of data for that partition. Considering the configuration in Figure 3, when for example, worker 1 retrieves one row from scanning its physical partition, it must assess whether or not it should hold that row for later computing the analytical function, or by other means forward it to the responsible worker.

The information collected in the Global Histogram will enable each worker to know if it should hold or forward the row to the the node holding the largest volume for that given partition.

The Global Shuffler operator will be used whenever the window operator needs to reconcile partial results from workers. It will forward all aggregated rows to the worker that will hold the overall largest data volume, the master worker. By instructing the workers that hold the least volume of data to forward rows, we are promoting the minimal usage of bandwidth possible.

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 have to be reconciled by a common node, which for this case will be dictated by the master node. Upon start, the Global Shuffle 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 it is not the current one.

4 PRELIMINARY ASSESSMENT

Reactive Programming (RXJ, 2015) was used in the undertaken micro-benchmark; allowing to establish a series of data streams from which entities can be constructed. We used this idea to map to the individual components that build a window operator.

We employed a single query holding a window function over the synthetically-generated relation from the TPC-C (Council, 2010) bechmark, Order Line. This relation holds 10 attributes, all uniformely distributed. The generated data composes 100 distinct partitions, each one with 500 rows. Globally, the Order Line relation held 500 K tuples, totaling 3906 Mb. To highlight the properties of our proposal, we considered the following ranking query:

```
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.

For comparison purposes, we report the results by using a naive approach and our Holistic Shuffler. The naive approach, disseminates 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 projected gain achieved by our contribution stems from the reduction of rows that need to be forwarded among nodes to reunite and reconcile results among workers, thus promoting shorter bandwidth consumption. Therefore, the evaluation results we present highlight the registered differences between the naive and the holistic approach for both shuffling stages. According to Figure 4, the Holistic

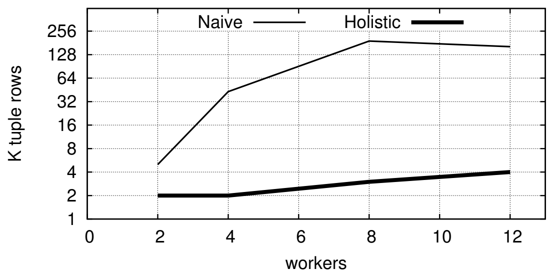


Figure 4: Average number of shuffled rows in both shuffle operations.

technique we propose induced a significantly smaller number of rows required to reunite and reconcile all the partitions in each computing node. Specifically, the Holistic technique required in average only 14.7% of the rows required by the Naive approach. The large difference is directly justified by the fact that the naive approach reunites partitions by forwarding data among all participating nodes, which intrinsically creates duplicates in each node. Moreover, it is also possible to verify that the row cardinality required by the Naive approach is proportional to the number of nodes. On the contrary, the Holistic technique discriminates each row, so that it is forwarded to the node responsible for it, as dictated by the proposed technique.

5 CONCLUSION

In this paper, we proposed a technique to reduce the amount of data transfered among computing nodes of a distributed query engine. It is based on the observation that the information regarding the existing data distribution on a set of computing nodes (each one with a disjoint partition of data) can be used to enable improovements on bandwidth consumption. We show how to implement it, which we tailored to be used for the efficient parallel processing of queries with non-cumulative window functions. We show that by conceptually applying this methodology, we were able to project an improvement, requiring only 14% (in average) of rows in bandwidth consumption, when compared with the naive technique. The research path to be followed will translate this methodology to a real distributed query engine.

ACKNOWLEDGEMENTS

This work was part-funded by project LeanBigData : Ultra-Scalable and Ultra-Efficient Integrated and Visual Big Data Analytics (FP7-619606).

REFERENCES

(2015). Reactive programming for java. https://github.com/ReactiveX/RxJava.

Cao, Y., Chan, C.-Y., Li, J., and Tan, K.-L. (2012). Optimization of analytic window functions. *Proceedings of the VLDB Endowment*,5(11):1244–1255

Council, T. P. P. (2010). *TPC Benchmark C*.

Navathe, S., Ceri, S., Wiederhold, G., and Dou, J. (1984). Vertical partitioning algorithms for database design. *ACM Transactions on Database Systems (TODS)*, 9(4):680–710.

Poosala, V., Haas, P. J., Ioannidis, Y. E., and Shekita, E. J. (1996). Improved histograms for selectivity estimation of range predicates. In *ACM SIGMOD Record*, volume 25, pages 294–305. ACM.

Sadalage, P. J. and Fowler, M. (2012). *NoSQL distilled: a brief guide to the emerging world of polyglot persistence*. Pearson Education.

Zuzarte, C., Pirahesh, H., Ma, W., Cheng, Q., Liu, L., and Wong, K. (2003). Winmagic: Subquery elimination using window aggregation. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 652–656. ACM.