Similarity Aware Shuffling for the Distributed Execution of SQL Window Functions

Fábio Coelho[1(✉)], Miguel Matos[2], José Pereira[1], and Rui Oliveira[1]

[1] INESC TEC, Universidade do Minho, Braga, Portugal
fabio.a.coelho@inesctec.pt, {jop,rco}@di.uminho.pt
[2] INESC-ID/IST, Lisboa, Portugal
mm@gsd.inesc-id.pt

Abstract. Window functions are extremely useful and have become increasingly popular, allowing ranking, cumulative sums and other analytic aggregations to be computed over a highly flexible and configurable sliding window. This powerful expressiveness comes naturally at the expense of heavy computational requirements which, so far, have been addressed through optimizations around centralized approaches by works both from the industry and academia. Distribution and parallelization has the potential to improve performance, but introduces several challenges associated with data distribution that may harm data locality. In this paper, we show how data similarity can be employed across partitions during the distributed execution of these operators to improve data co-locality between instances of a Distributed Query Engine and the associated data storage nodes. Our contribution can attain network gains in the average of 3 times and it is expected to scale as the number of instances increase. In the scenario with 8 nodes, we were to able attain bandwidth and time savings of 7.3 times and 2.61 times respectively.

1 Introduction

Nowadays, the scalability of database engines is paramount, specifically when it is targeted at large scale analytical processing. Systems must be able to support several computing nodes, enabling component scalability to possibly reach hundreds or thousands of nodes. However, reaching such scale introduces several challenges associated with data and request distribution and balance. Cloud computing infrastructures offer a nearly transparent environment where computation is available as virtually infinite computing nodes. However, commercial relational database engines (RDBMS) do not conform to such paradigm, typically offering a monolithic structure. Legacy-type servers are usually considered for running RDBMSs, limiting system scalability from the purchase moment or until they become economically unacceptable.

Window Functions (WF) define a sub-set of analytical operations that enable the formulation of analytical queries over a derived view of a given relation R. They are also known as OLAP Analytical Functions and are part of the SQL:2003

© IFIP International Federation for Information Processing 2017
Published by Springer International Publishing AG 2017. All Rights Reserved
L.Y. Chen and H.P. Reiser (Eds.): DAIS 2017, LNCS 10320, pp. 3–18, 2017.
DOI: 10.1007/978-3-319-59665-5_1

standard. All major database systems like Oracle [7], IBM DB2 [14], Microsoft SQL Server [6], SAP Hana [21], Cloudera Impala [17] or Postgresql [20] have the ability to execute a sub-set of the available WFs.

WF are widely used by analysts as they offer a highly configurable environment together with a straightforward syntax. In fact, SQL WF are used in at least 10% of the queries in TPC-DS [22] benchmark, a benchmark suite aimed to evaluate data warehouse systems. Despite their relevance, parallel implementations and optimizations considering this operator are almost non existing in the literature. While [4,18,23] are notable exceptions, these works are targeted at many-core CPU centralized architectures that are substantially different from distributed architectures.

The nature of current centralized architectures do not typically take into account data distribution. This eases their processing models, but prevents them to scale beyond the limitations of the hardware that hosts them. The massively parallel nature that distribution approaches enable requires, however, to carefully address data distribution. Having the right grasp on data placement allows to improve data movement, but requires additional mechanisms to maximize network efficiency.

In this paper we focus on WF, particularly exploring opportunities for their distributed execution. We propose a technique that exploits *similarity* between partitions as a metric that can be used to judiciously improve the affinity of data and computing nodes, consequently minimizing the data movement between computing nodes.

Contributions: First, we demonstrate that it is possible to improve data forwarding by using partition *similarity* to chose the forwarding mechanism between Distributed Query Engine (DQE) workers. Second we present an experimental evaluation that confirms the merit of our approach. **Roadmap:** The remainder of this paper is organized as follows: Sect. 2 introduces WF. Section 3 introduces Distributed WF, describing their query execution plans and cost models. Section 4 presents our similarity technique, improving affinity between data and computing nodes. Section 5 evaluates our proposal. Section 6 presents related work and Sect. 7 concludes our work.

2 Window Functions

WF started to be largely adopted by database vendors from the 2011 revision of the SQL standard. These are powerful analytical operators that enable complex calculations such as moving, cumulative or ranking aggregations to be computed over data. WF are expressed in SQL semantics by the keyword `OVER` as shown in Fig. 1. In the next Sections we will analyze each part of the query.

```
select analytical_function() OVER( PARTITION BY A ORDER BY B ROWS BETWEEN
        UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) from R
```

Fig. 1. Example of SQL query with WF.

Like other analytical operators, WF are required to reflect several concepts, namely: the processing order, the partitioning of results or the notion of the current row being computed. These design constraints are clearly translated from the syntax as seen in the previous example, and configure two main considerations as foundation for the WF environment. Firstly, WF are computed after most of the remaining clauses in the query (e.g., such as `JOIN`, `WHERE`, `GROUP BY` or `HAVING`), but immediately before any required final ordering (e.g., `ORDER BY`). Secondly, the analytical operator to be computed with the WF environment will create an output attribute that reflects, but does not modify or filter the input data present in the source relation. Therefore, the result-set will present the same cardinality of rows as in the source relation, but will have an additional attribute mapping the result.

2.1 Partitioning, Ordering and Framing

The WF environment can be decomposed into three stages, as depicted in Fig. 2, defining the processing order: the partitioning (1), ordering (2) and framing (3) stages. Each stage is defined by specific clauses namely: the `PARTITION BY` and the `ORDER BY` that respectively create logical partitions of distinct data elements and afterwards develop an intra-partition sorting. The logical partitions are regulated by the mandatory argument of the `PARTITION BY` clause, defining the column attribute or expression that controls the partitioning. The partition clause resembles the behavior of the `GROUP BY` clause, but does not collapse all group members into a single row.

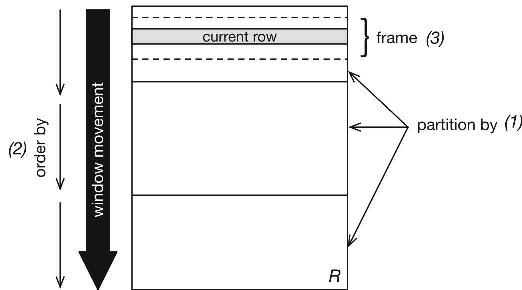

Fig. 2. Stages of the Window operator: partitioning (1), ordering (2) and framing (3).

The intra-partition ordering follows the partitioning stage and is also regulated by the mandatory column attribute or expression considered as argument for the `ORDER BY` clause. The ordering stage is very important for a set of non-cumulative analytical functions, that are the focus of our contribution, but also as it is the costliest operation in the environment [4].

Finally, the framing stage builds on the provided ordering, taking into account the current row being considered to introduce the concept of window or frame. The frame is built from a group of adjacent rows surrounding the current

row and changes as the current row moves towards the end of the partition. The framing is set by either the `ROWS BETWEEN` or the `RANGE BETWEEN` clauses. The former considers n rows before and after the current row, while the latter restricts the window by creating a range of admissible values and, the current row is considered if the stored values fit in the provided range[1].

The WF environment allows to combine different clauses, enabling the inclusion or exclusion of each clause type. For instance, it is possible to declare a WF with just a partitioning or ordering clause. If no partitioning clause is declared, the entire relation is considered as a single partition. If no ordering clause is declared, then the natural ordering of the relations key, or partitioning clause (if present) is considered. Moreover, each available analytical function may or not change the computation logic. Due to space constraints we do not characterize all the possible configurations of the WF environment. The interested reader should consult the 2003 and 2011 revisions of the ANSI SQL standard for further information [1].

2.2 Cumulative and Ranking Analytical Functions

The analytical set of functions currently available in most Query Engines (QE) can be classified into Cumulative or Ranking. Cumulative analytical functions or aggregates, are a group of functions that are not order-bound. That is, when they are computed within a WF, an `ORDER BY` clause is not required. The $sum(x)$, $avg(x)$ or $count()$ are just some examples of this category of functions. Figure 3(a) depicts the result of computing a WF structured as "`select analytical_function() OVER (PARTITION BY A ORDER BY D) FROM table`", but immediately before applying the requiring analytical function to a given relation. Figure 3(b) depicts the result of computing the previous WF with the $sum(D)$ function. The result of a cumulative function is the same for all the members belonging in the same partition.

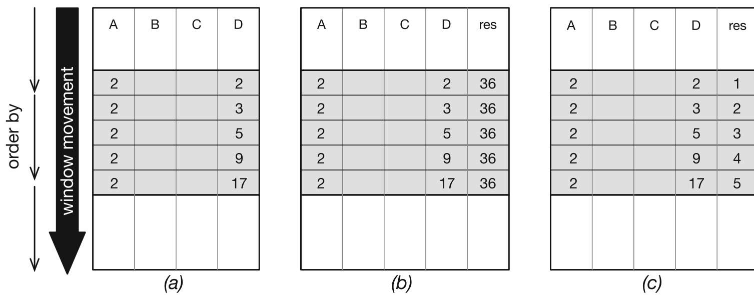

Fig. 3. WF query as: `select analytical_function() OVER (PARTITION BY A ORDER BY D) FROM table`. (a) WF where the partition by clause generated 1 partitions. (b) Cumulative (sum) analytical function over WF in (a). (c) Ranking (rank) analytical function over WF in (a)

[1] Typically, the use of this clause is restricted to numeric types.

Ranking analytical functions, on the other hand, are order-bound. That is, the function requires the data to be ordered according to some criteria in order to output a deterministic result, and thus, the ordering clause is always required. The *rank*(), *dense_rank*() or *ntile*() are just some examples of this category of functions. Figure 3(c) outputs the result of computing the previous WF with the *rank*() function, outputting a different result for each row in the partition.

The ordering requirement for the latter category of functions implies data co-locality in order to minimize the number of sorting steps needed to achieve intra-partition ordering [4]. In the remainder of this paper, we consider a WF computing a ranking analytical function with a single partition and ordering clause and no framing clause, since the rank function implicitly defines framing constraints.

3 Distributed Window Functions

RDBMS are built from several components, namely the QE and the Query Optimizer (QO). The former translates SQL syntax into a set of single operators. The latter considers several statistical techniques to improve the query execution plan of a query. In a nutshell, QEs split the execution of a query in two separate stages: the query planning and the query execution. During the first stage, the QE decides how the query is executed during the second stage, and which operators are used in such a query plan. This builds a complex multi-optimization problem that has to be executed in polynomial time.

The QO uses hints about data in the form of statistical approximations, allowing the query engine to optimize query execution based on the approximation cost of each individual operator in a given data set. When scaling from a single QE to a DQE, data partitioning techniques are necessary in order to distribute data among instances. The number of available computing nodes configures the installed Degree of Parallelism (DOP). However, non-cumulative analytical algorithms are order-bound, thus requiring that logical data partitions are co-located (i.e., they should live in the same storage node). If elements of a given logical partition are spread in a group of nodes it becomes impossible to sort each logical partition in just one step. The sorting in each data partition would induce a partial sorting that is not deterministic and that would prevent inter-partition parallelism. The QOs therefore need to adapt their cost models to reflect the data movement required in order to ensure co-locality of partitions during execution time of the operator.

3.1 Distributed Query Engines

The DQE takes advantage of data distribution in order to scale query execution. The present architecture is provided by a Highly Scalable Transactional PaaS [16]. Each node in the system is split in two layers, the DQE itself and the storage layer, holding the data partitions to be manipulated by a given DQE instance. Particularly, the considered DQE is based on the Apache derby

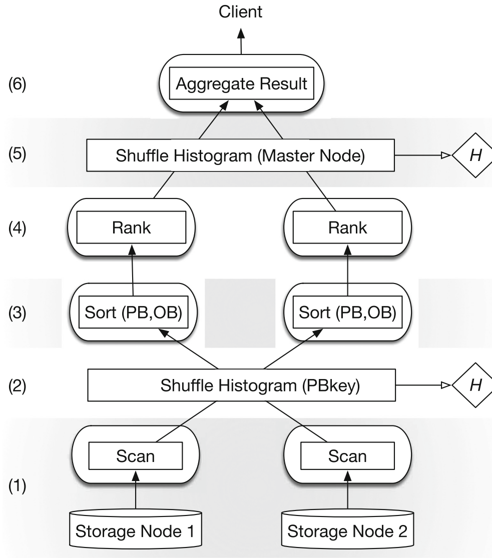

Fig. 4. Distributed Query Plan for Ranking WF. Round boxes represent individual stages of the WF environment. Arrows represent data flow in a process or over the network. PB and OB respectively represent Partition By and Order By attributes. H describes a statistical histogram. Numbers represent process execution order.

project [2] and the storage layer is provided by Apache HBase [11], working over the Hadoop Distributed File Systems (HDFS) [13].

The DQE instances are able to accept client query requests through a JDBC connection and generate the distributed query plan. This plan is then shared with all participating DQE instances. The data distribution in each Storage Node is typically accomplished by means of an Hash function, considering a single or a collection of attributes as key. The distribution of keys lies within the inner characteristics of the considered hash function, usually producing uniform distributions that evenly place tuples across all available storage nodes. Poorly chosen hash functions may result in data skew and should be tailored to each specific workload providing adjusted table splitting [8].

Figure 4 presents the simplified distributed query plan for a ranking analytical function. The following stage numbers resemble the ones depicted in Fig. 4. With data partitioned in several nodes, each one will scan (1) its local partition. The partial results found in each node derive from the data partitioning required to distributed data. Data movement is then required in order to ensure that each logical partition created by the partitioning clause will reside in a single node for computation. This is achieved by the shuffle mechanism (2). Afterwards, data is sorted according to the partitioning and ordering clauses (3), and results are submitted to the rank function (4). At this stage, each computing node holds partial results from each logical partition. The results from each logical partition are then reunited in a single location (5) before being delivered to the client (6).

Ranking aggregation algorithms are dependent on having full disclosure of the entire logical partitions. If the first shuffling stage (2) is not performed, the partial results in each partition will produce incorrect results. That is, if members of a logical partition are processed in the same node where they are stored (therefore in distinct DQE instances), the partial aggregation results produced will not be able to sort the entire logical partition. Thus, when the partial results of logical partitions are merged in the final result-set (6), they will need to be entirely recomputed. By considering the first shuffle stage (2), the results produced by stage (3) in each logical partition are globally correct since, independence from logical partitions ensures inter-partition parallelism, allowing computation to be distributed through several computing locations. The partition strategy considered depends on the mandatory argument of the partitioning clause. It is thus impractical to adjust the table splitting of the workload to a specific partitioning clause, since the ideal configuration may change with each query. Moreover, the environment allows the use of expressions as the arguments of the inner clauses, posing an extra hurdle to this abstraction.

3.2 Data Shuffling

Data movement during the execution of a WF query is required, ensuring that all the elements of each logical partition are in the same location. In order to judiciously forward data while minimizing at the same time the transfer cost, in [5] we introduced a mechanism that works together with the data transfer mechanism, a shuffler, promoting co-location of logical partitions. This is achieved by considering an histogram, characterizing the universe of elements present in each partition. Briefly, the histogram should hold the cardinality of each different element in each different column qualifier. The histograms referring to each node are then combined into a global histogram. The introduction of this mechanism along with the shuffler, allowed to forward data to the specific node that should process a given partition.

Consider Fig. 5 where a table similar to the table in Fig. 3(a) was split in two partitions on the storage layer. This initial partitioning is defined by hashing the value of the nodes ol_w qualifier and performing the arithmetic modulo between the hash result and the number of computing instances ($Hash(value\ in\ ol_w)\ \%\ \#Nodes$). Guided by the query in Fig. 1, the results were then ordered according to the qualifier ol_d (the partitioning clause). Both nodes of the storage layer hold elements from the available three partitions in ol_d ($p1, p2, p3$). According to the previously introduced, ol_d partitions ($p2$) and ($p3$) in instance $DQE\ w1$ will be relocated to instance $DQE\ w2$ and, ol_d partition ($p1$) will be relocated from instance $DQE\ w2$ to instance $DQE\ w1$.

On the one hand, hash forwarding a single row at a time prevents batching several rows in a single request. On the other hand, due to the asynchronous nature of DQEs, latency is usually not the bottleneck and thus, data movement can be delayed until network usage can be maximized [12]. This enables the use of batching in order to improve network usage. A batch payload is formed by grouping rows that need to be forwarded to a common destination and it is

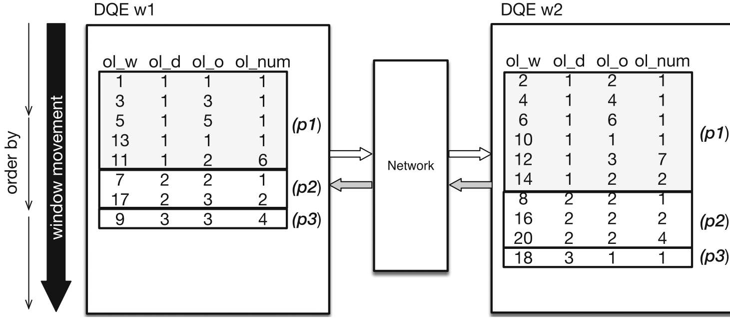

Fig. 5. Shuffling instances partitioned by *ol_w*. In WF context, they were partitioned by *ol_d* and Ordered by *ol_num*. The DQE instances will use the network to combine partitions during execution time. Instance w_1 will hold partitions $ol_d = 1$, instance w_2 will hold partitions $ol_d = 2$ and $ol_d = 3$, respectively.

regulated by a buffer within the shuffling mechanism, whose size and delivery timeout are configurable. Nevertheless, the use of this mechanism can prove to be a misfit in cases where workloads do not benefit from grouping data (i.e., logical partitions with reduced number of rows). Therefore, not having to delay data transmission reduces execution time. To understand up to what level a given logical partition may or not benefit from batching, we considered a correlation mechanism to guide such decision, identifying the logical partitions that are good candidates for forwarding data in batch.

4 Similarity

QOs found in modern QEs use several statistical mechanisms to explore data features in order to improve query execution performance. Without them, independence assumptions between attributes are preserved, which commonly leads to under or over provisioned query plans, which is particularly undesirable in DQEs. As in real-world data, correlations between relation attributes are the rule and not the exception, the array of correlation or other algebraic extraction mechanisms in the literature is vast, namely [3,9,19]. Correlations can also be used in DQEs to improve how data distribution is handled. When logical data partitions need to be relocated in order to improve co-locality, the correlation between qualifiers in different locations of the storage layer can be explored to minimize the required data movement.

In this paper, we introduce a *similarity measure* to quantify to what level the partitions of a given attribute held by different storage nodes are alike. Data partitions with high *similarity* are good candidates to be shuffled within a batch payload. This is so as a high *similarity* implies a high common number of partitions. On the other hand, data partitions with low *similarity* are better candidates to be immediately shuffled for their destination. This is so as they share a low number of common partitions. This is efficiently achieved through Algorithm 1. The *similarity measure* quantifies in a universe between 0 (not

Algorithm 1. Similarity Aware Shuffling Mechanism

```
1: P(r) =< r_0, r_1, r_2, r_n >← partition
2: r_i ← current_row
3: pbk ← partition_by_key
4: w_id ← worker_id
5: H ← histogram
6: t ← similarity_threshold
7: procedure SIMILARITY(attr_A, attr_B)
8:     Sim ← unique(attr_A ∩ attr_B) / unique(attr_A ∪ attr_B)
9: procedure BATCHSHUFFLING(P(r), dest)
10:     send P(r) to dest
11: procedure HASHSHUFFLING(r_i, dest)
12:     send r_i to dest
13: function SHUFFLER
14:     dest ← H(r_i.pbk)
15:     if w_id ≠ dest then
16:         Sim ← SIMILARITY(w_id_pbk,dest_pbk)
17:         if Sim > t then
18:             BATCHSHUFFLING(P(r), dest)
19:         else
20:             HASHSHUFFLING(r_i, dest)
```

similar) and 1 (similar) how similar two attributes are, by considering the number of unique values in each attribute to compute the metric. The data required to compute this metric is already provided by the histogram introduced in previous work [5], bypassing the need to collect additional statistical data. This structure is characterized by a small memory footprint (few KB) and the update period dictated by the DQE administrator. This algorithm will be considered during the first shuffling stage (stage (2) of Fig. 5). It will consider each logical partition ($P(r)$), the previously introduced Histogram (H) and a configurable *similarity* threshold. Three auxiliary procedures are considered. The `SIMILARITY` procedure computes the *similarity measure* from the set of unique values in the qualifiers considered as arguments. The `BATCHSHUFFLING` procedure marshals all the rows of partition $P(r)$ and sends it to the destination worker *dest*. The `HASHSHUFFLING` procedure marshals a single row r_i and sends it to destination *dest*.

When the shuffler action is required, it consults the Histogram H to verify what is the optimal destination (DQE instance) from row r_i. When the destination is a remote instance (line 15), the shuffling mechanism computes the *similarity measure* between the local ($attr_A$) and destination ($attr_B$) qualifiers (line 16). The partition $P(r)$ is marshaled to the appointed destination when the observed *similarity* is above threshold t (line 18) (`BATCHSHUFFLING`), or each row r_i is otherwise sent to destination (line 20) (`HASHSHUFFLING`). The parameter t sets a threshold above which rows are forwarded in batch to the destination instance. This parameter defaults to 0.5 meaning that if not modified, rows are batch forwarded if the origin contains at least half the number of unique partition values of the destination.

5 Evaluation

We validated that by batch shuffling tuples between DQE instances we would save bandwidth, improving execution time of the shuffling stage. We considered a synthetic data set and shuffled rows between distinct DQE instances. The data set used was extracted from the TPC-DS [22], a benchmark suite tailored for data analytics. We extracted a single relation (`web_sales`) which is composed of 35 distinct attributes, configuring TPC-DS with a scale factor of 50 GB. This resulted in a relation with 9.4 GB corresponding to 36 million rows.

The outcome of the mechanism we propose is directly related with the data distribution considered. In order to bound the outcome of our contribution in terms of the lower and upper performance bounds, we statistically analyzed the considered relation. The lower bound is set by not using the *similarity* mechanism. The upper bound is set by considering the relation attributes that would favor data distribution. This was achieved by identifying the placement key attribute, but also a candidate attribute to be the partitioning clause or shuffling key (`PBK`) of the WF. The placement key attribute will define the data distribution in each DQE Storage Node through the use of an Hash function, and the `PBK` will define the runtime partitioning within the WF environment.

The results are depicted in Fig. 6. The top plot presents the number of partitions in each single attribute in the considered relation. That is, the number of unique values in each attribute. The bottom plot depicts the average cardinality of each partition. That is, the average number of elements in each group of unique values in each attribute. The ideal candidate attribute to become the relation placement key is the attribute that displays the highest partition number and at the same time holds the smallest cardinality, ensuring an even data distribution and reduced data skew. Observing both plots leads us to consider attribute with index 17 (`ws_order_number`), displaying the highest number of

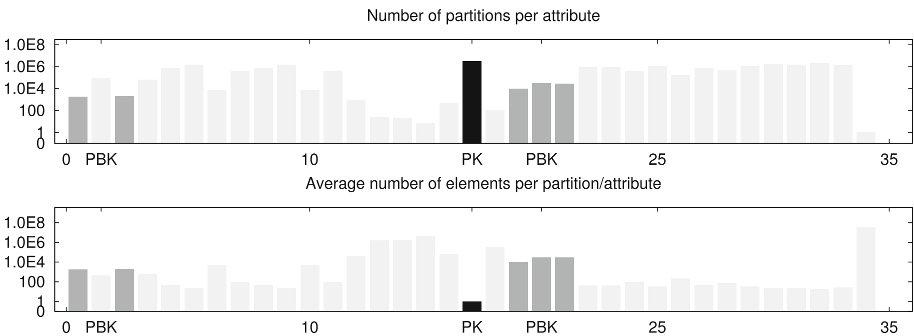

Fig. 6. Number of partitions per attribute (top) and the average number of elements per partition/attribute (bottom). The horizontal axis represents the attribute index. The vertical axis quantifies each measure in logarithmic scale. The attribute considered for placement key (PK) is shown in black and the candidates for WF Partition By key (PBK) are shown in dark gray.

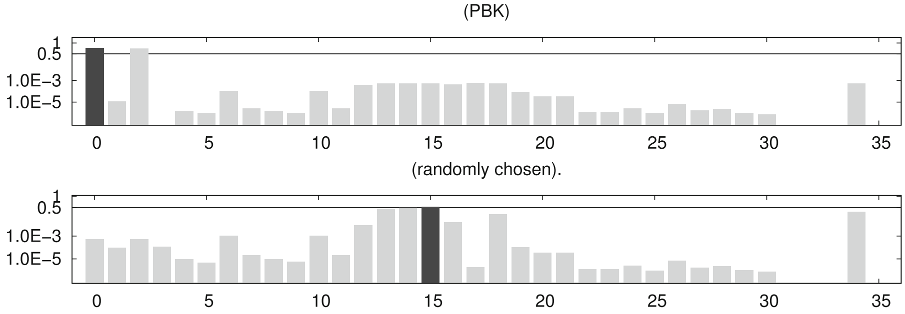

Fig. 7. Similarity between attributes in two data nodes. Horizontal axis represents the attribute index. Vertical axis represents the *Similarity measure* in logarithmic scale.

partitions, each one with a single element. On the other hand, the candidate attributes to be selected as WF `PBK` are the attributes that would hold at the same time a high number of partitions and high partition cardinality. These are good `PBK` candidates since they will induce a number of logical partitions that is above the configured DOP. The observation of the plots leads to identify as candidates the attribute indexes depicted in dark gray, from which we select attribute 0 (`ws_sold_date_sk`) as `PBK`.

After the election for the `PK` and `PBK` keys, we conducted a second experiment to verify the computed *similarity measure*. Figure 7 depicts the results of applying the metric in two scenarios. In both cases, we consider our scenario to be built from several DQE instances and corresponding Storage Nodes. On all experiments, we considered only the communication layer of the DQE where our contribution is, thus avoiding the SQL parsing and optimization stages. Each data partition was computed by applying an Hash function with the elected `PK` dividing the data into as many partitions as configured DQE instances. We first considered the configuration with 2 instances A and B. In the experiment in the top plot we computed the *similarity measure* between the `PBK` of location A and each distinct attribute in location B. It is possible to observe that attribute 0 in location B presents the highest similarity, followed by attribute two. These are also the only attributes that are above the set up threshold of 0.5 denoted by the horizontal line. The remaining attributes have a residual similarity measure. The bottom plot depicts a different configuration where attribute 15 was randomly chosen among all non candidate attributes. The similarity measure in this attribute is lower than our threshold, even though it seems to be equal given the logarithmic scale required to observe the remainder attributes. Therefore, the results achieved during the first configuration would induce the shuffler to use batching mechanisms to forward partitions among DQE instances, instead of hash forwarding. The latter would culminate in sending a single row at a time.

In order to verify the impact of our contribution regarding network usage, we conducted an experiment to assess the magnitude of the network savings promoted. Namely, we considered configurations with 2, 4 and 8 DQE and Storage

instances. The computing nodes were only set up with the communication layer responsible for the shuffling in the WF environment. Each node is comprised of commodity hardware, with an Intel i3-2100-3.1 GHz 64 bit CPU with 2 physical cores (4 virtual), 8 GB of RAM memory and one SATA II (3.0 Gbit/s) hard drive, running Ubuntu 12.04 LTS as the operating system and interconnected by a switched Gigabit Ethernet network. During execution, each computing node acts as a DQE instance shuffler, forwarding data to the remainder instances. In a distributed deployment, the DQE instance will be co-located with other services (e.g., storage node) which will typically restrict the available memory to the DQE instance.

We evaluated two configurations where the first represents a baseline comparison, forwarding all data by hash shuffling, and a second where data is forwarded according to our *similarity* mechanism.

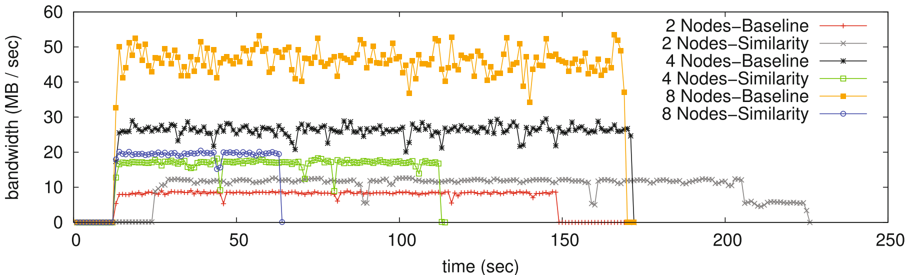

Fig. 8. Bandwidth (outbound) registered during shuffling between instances.

The results depicted in Fig. 8 are twofold. The *similarity measure* registered both a decrease in bandwidth and it also promoted a shorter execution period for the shuffling technique. This is the result of pairing the batch shuffling mechanism together with the proposed *similarity measure*. The savings induced come at a residual cost, since the statistical information is not collected for the single purpose of this improvement, nor it has to be updated in each query execution. The *similarity measure* technique only proved effective from the configuration with 4 instances onward, since it is only from that configuration that both bandwidth and execution time are lower than the baseline. For the configuration with only two nodes, the baseline technique proved to be better by both shortening the shuffling time and registered bandwidth. However, in the configurations with 4 and 8 nodes, the *similarity measure* was able to reduce the bandwidth and execution time when compared with the baseline approach. As the number of partitions in the system increase, each single partition becomes responsible for a shorter set of data, promoting bandwidth savings up to 7.30 times for the 8 node configuration.

The previous experiment evaluated the shuffling mechanism by considering an attribute with ideal *similarity measure* and partitioning on the storage layer.

Table 1. Total Bandwidth (sent) and execution time registered for each configuration.

	2 nodes	4 nodes	8 nodes
Baseline (MB)	1,132.45	4,172.59	7,237.56
Similarity (MB)	2,365.34	1,695.24	991.72
Bandwidth Gain (x)	−0.48	2.46	7.30
Shuffle time			
Baseline (sec)	149	172	170
Similarity (sec)	226	114	65
Speed up (x)	−0.48	1.55	2.61

In order to demonstrate the impact of selecting an attribute that do not favor an uniform distribution of data among data partitions, we conducted a second experiment that considered an attribute with poor partitioning properties (i.e., reduced number of partitions). The results consider the same component configuration, but selected attribute 15 (`ws_warehouse_sk`) for the partitioning. When selecting an attribute that lacks the desirable distribution, the logical partitions will present an imbalance, thus promoting a low *similarity measure*. Therefore, the shuffling mechanism will not be able to maximize network usage and will end up having to consider the `HASHSHUFFLING` mechanism to forward data. The results are not thoroughly presented due to space constraints. However, we point out that they are in line with the considered baseline results presented in Table 1, registering a bandwidth variance of ±4%. Moreover, even though we do not consider it, the use of compression techniques may further increase the observed savings.

6 Related Work

Window Functions were introduced in the 2003 SQL standard. Despite its relevance, parallel implementations and optimizations considering this operator are almost non existing. Works such as [4] or [23] fit in the first category, respectively tackling optimization challenges related with having multiple window functions in the same query, and showing that it is possible to use them as a way to avoid sub-queries and lowering quadratic complexity. However, such approaches do not offer parallel implementations of this operator. A vast array of correlation mechanisms have been so far deeply studied in the literature. Nonetheless, most of the conducted studies focus on efficient ways to discover and exploit soft and hard correlations [15], allowing to find different types of functional dependencies. Works like [18] introduced mechanisms to improve the performance of the WF environment when many-core architectures are used. Distinct approaches and algorithm improvements are introduced, enabling to parallelize the distinct stages of the operator.

When addressing WFs, a common misconception generally brings a comparison between SQL WF (in which our contribution focuses) and CEP windowing.

Differences are both semantical and syntactical. On the one hand, the CEP environment is characterized by an incoming and infinite stream of events. From there, a configurable, but constant sample (e.g., window) builds a sketch [10] where aggregations are derived. On the other hand, SQL WF are computed over finite sets built from SQL relations. While the former windows are fixed and the data moves through, in the latter, the data is fixed and the window performs the movement. Moreover each approach considers distinct SQL keywords (e.g., `OVER`, `RETAIN`) and subsequent syntax.

7 Conclusion

WF with ranking analytical functions are required to have full disclosure of a given logical partition. Data partitioning is required to enable systems to scale, but harms data locality, which poses added difficulties when trying to parallelize these functions.

In this paper we motivate and validate how similarity between partitions can be used to promote efficient data forwarding among instances of a DQE. We introduced an algorithm to choose whether to batch or to hash forward rows between such instances by understanding how the *similarity measure* between distinct partitions of a DQE can be used towards the effectiveness of the WF environment.

The WF environment changes how analytical functions are computed, requiring specific implementation details for each functions. We therefore plan to leverage such parallelization opportunities to other analytical functions.

Acknowledgments. The research leading to these results was part-funded by (1) the European Union's Horizon 2020 - The EU Framework Programme for Research and Innovation 2014–2020, under grant agreement No. 732051; (1) Project TEC4Growth - Pervasive Intelligence, Enhancers and Proofs of Concept with Industrial Impact/NORTE-01-0145-FEDER-000020 is financed by the North Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement, and through the European Regional Development Fund (ERDF) and by (1) 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. ANSI: Information technology - database languages - SQL multimedia and application packages. Technical report, ANSI (2003). http://webstore.ansi.org/RecordDetail.aspx?sku=ISO%2fIEC+13249-2%3a2003
2. Apache: The apache derby project. Technical report, Apache Foundation (2016). https://db.apache.org/derby/derby_charter.html

3. Brown, P.G., Hass, P.J.: BHUNT: automatic discovery of fuzzy algebraic constraints in relational data. In: Proceedings of the 29th International Conference on Very Large Data Bases, vol. 29, pp. 668–679. VLDB Endowment (2003)
4. Cao, Y., Chan, C.Y., Li, J., Tan, K.L.: Optimization of analytic window functions. Proc. VLDB Endowment **5**(11), 1244–1255 (2012)
5. Coelho, F., Pereira, J., Vilaça, R., Oliveira, R.: Holistic shuffler for the parallel processing of SQL window functions. In: Jelasity, M., Kalyvianaki, E. (eds.) DAIS 2016. LNCS, vol. 9687, pp. 75–81. Springer, Cham (2016). doi:10.1007/978-3-319-39577-7_6
6. Microsoft Corporation: Transact-SQL. Technical report, Microsoft Corporation (2013). https://msdn.microsoft.com/library/ms189461(SQL.130).aspx
7. Oracle Corporation: SQL analysis and reporting. Technical report, Oracle Corporation (2015). http://docs.oracle.com/database/121/DWHSG/analysis.htm#DWHSG8659
8. Cruz, F., Maia, F., Oliveira, R., Vilaça, R.: Workload-aware table splitting for NoSQL. In: Proceedings of the 29th Annual ACM Symposium on Applied Computing, SAC 2014, pp. 399–404. ACM, New York (2014). http://doi.acm.org/10.1145/2554850.2555027
9. Fan, W., Geerts, F., Jia, X., Kementsietsidis, A.: Conditional functional dependencies for capturing data inconsistencies. ACM Trans. Database Syst. (TODS) **33**(2), 6 (2008)
10. Garofalakis, M., Keren, D., Samoladas, V.: Sketch-based geometric monitoring of distributed stream queries. Proc. VLDB Endowment **6**(10), 937–948 (2013). http://dx.doi.org/10.14778/2536206.2536220
11. George, L.: HBase: The Definitive Guide: Random Access to Your Planet-Size Data. O'Reilly Media, Inc., USA (2011)
12. Gonçalves, R.C., Pereira, J., Jiménez-Peris, R.: An RDMA middleware for asynchronous multi-stage shuffling in analytical processing. In: Jelasity, M., Kalyvianaki, E. (eds.) DAIS 2016. LNCS, vol. 9687, pp. 61–74. Springer, Cham (2016). doi:10.1007/978-3-319-39577-7_5
13. Hadoop Apache: Hadoop (2009)
14. IBM: OLAP specification. Technical report, IBM (2013). http://www.ibm.com/support/knowledgecenter/SSEPGG_10.5.0/com.ibm.db2.luw.sql.ref.doc/doc/r0023461.html
15. Ilyas, I.F., Markl, V., Haas, P., Brown, P., Aboulnaga, A.: CORDS: automatic discovery of correlations and soft functional dependencies. In: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, pp. 647–658. ACM (2004)
16. Jimenez-Peris, R., Patiño-Martinez, M., Magoutis, K., Bilas, A., Brondino, I.: Cumulonimbo: a highly-scalable transaction processing platform as a service. ERCIM News **89**(null), 34–35 (2012)
17. Kornacker, M., Behm, A., Bittorf, V., Bobrovytsky, T., Ching, C., Choi, A., Erickson, J., Grund, M., Hecht, D., Jacobs, M., et al.: Impala: a modern, open-source SQL engine for hadoop. In: CIDR, vol. 1, p. 9 (2015)
18. Leis, V., Kundhikanjana, K., Kemper, A., Neumann, T.: Efficient processing of window functions in analytical SQL queries. Proc. VLDB Endowment **8**(10), 1058–1069 (2015). http://dx.doi.org/10.14778/2794367.2794375
19. Liu, H., Xiao, D., Didwania, P., Eltabakh, M.Y.: Exploiting soft and hard correlations in big data query optimization. Proc. VLDB Endowment **9**(12), 1005–1016 (2016). http://dx.doi.org/10.14778/2994509.2994519

20. Postgresql: Advanced features - window functions. Technical report, Postgresql (2015). https://www.postgresql.org/docs/9.4/static/tutorial-window.html
21. SAP: SAP HANA SQL reference (2014). https://help.sap.com/hana/SAP_HANA_SQL_and_System_Views_Reference_en.pdf?original_fqdn=help.sap.de
22. Transaction Processing Performance Council: TPC Benchmark DS (2012). http://www.tpc.org/tpcds/spec/tpcds_1.1.0.pdf
23. 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)