Improving the Processing of DW Star-Queries under Concurrent Query Workloads

João Pedro Costa[1] and Pedro Furtado[2]

[1] Polytechnic Institute of Coimbra, ISEC, DEIS, Portugal
jcosta@isec.pt
[2] University of Coimbra, Portugal
pnf@dei.uc.pt

Abstract. Currently, Data Warehouse (DW) analyses are extensively being used not only for strategic business decisions by a few, but also for feedback to a wider audience and into daily operational decisions. As a result, there's an increase in the number of aggregation star-queries that are being concurrently submitted. Although such queries require similar processing patterns, they are stressing the database engine ability to deliver timely execution, due to the fact that each query executes independently from the others (query-at-time processing model). Recently, there's an increasing interest in approaches that cooperate to manage large numbers of concurrent aggregation star-queries. We have proposed SPIN in a previous paper [1]. It is a data processing model that shares data and computation in order to handle large concurrent query loads, and its data organization provides almost constant and predictable execution times for all submitted queries. It has a data reader that reads data in circular loop, placing it in a pipeline, before being processed by branches that combine common processing computations. SPIN is IO dependent, i.e. a query is only be answered after a full circular loop, even though tuples and similar predicates have been evaluated in the past. In this paper we propose data processing approach that uses a set of *bitsets*, built on-the-fly, to significantly reduce the query processing time, the tuple evaluation cost and the number of predicates and tuples evaluated, without sacrificing its predictability features. The data read from storage is reduced to the minimum needed by the current query load.

1 Introduction

Common database engines process every query independently without data and processing sharing considerations. In this model (query-at-time) each competes for resources, and thus the execution time increases with the number of queries that are concurrently being executed. While this may not raise performance issues for most operational systems, it is a performance killer when dealing with large Data Warehouses (DW). In this context, large fact and dimension relations are concurrently scanned by each query and tuples are independently filtered, joined and aggregated. This lack of data and processing sharing results in the system inability to provide predictable execution times for scalable data volumes and query workloads.

We have proposed SPIN [1], a data and processing sharing model that delivers predictable execution times to a large set of concurrently running aggregation star queries. SPIN physically stores the star schema as a single de-normalized relation as

L. Bellatreche and M.K. Mohania (Eds.): DaWaK 2014, LNCS 8646, pp. 245–253, 2014.

proposed in [2], [3] to avoid costly join operations. A data reader is continuously spinning, sequentially reading data chunks in a circular loop, placing the data in a base pipeline shared by all running queries. Common filters and computations (with the same operators) from different queries are combined into common pipelines with data switches added to end to share its results. Subsequent data pipelines are then connected as logical branches of this common data pipeline, consuming its output. As a result, pipelines of running queries are split, merged and organized into a workload data processing tree (*WPTree*), with the base pipeline as root. Any query q, Each query starts processing tuples that flow along the pipelines and only stops when all the tuples have being considered for evaluation (after a complete loop). While this improves IO sharing, the query execution is constrained by the time needed to fully read the data. SPIN performance can also be limited by the evaluation costs related to redirecting and filtering tuples as they go along branches, even though most of them already has been evaluated in the past.

The query execution time is constrained by the data reading time (t_{read}) and the query processing time ($t_{process}$), particularly t_{read} since all tuples must be considered for evaluation. While t_{read} is constant and shared among queries, the $t_{process}$ time is influenced by the computation (e.g. aggregation operators) and evaluation of predicates of each query (t_{eval}), and how these can be shared among queries. Since reading and processing is done in parallel, the query execution time is constrained by the largest of these times ($\max(t_{process}, t_{read})$). For wide WPTree (large number of simultaneous queries), the processing time ($t_{process}$) can be larger than the reading time (t_{read}), and thus endangering the objective of execution time predictability.

In this paper we propose a *bitset*-based data processing approach that uses a set of *bitsets* to reduce the overall execution time, by reducing the evaluation time (t_{eval}), the number of evaluated tuples (n_{eval}), and also the time required to read the data (t_{read}).*Bitsets* are built on-the-fly with the results of previous executions, and when built they deliver faster processing times (by using a bit lookup instead of the actual evaluation), and also reduce the number of evaluations and the data read from storage.

2 Related Work

The usage pattern of DWs is changing from the traditional, limited set of simultaneous users and queries, mainly well-known reporting queries, to a more dynamic and concurrent environment, with more simultaneous users and ad-hoc queries. DW query patterns are mainly composed by star aggregation queries, which contain a set of query predicates (filters) and aggregations. The query-at-a-time execution model of traditional RDBMS systems, where each query is executed following its own execution plan, does not provide a scalable environment to handle much larger, concurrent and unpredictable workloads. Analyzing the execution query plan, we observe that the low-level data access methods, such as sequential scan, represent a major weight in the total query execution time. One way to reduce such a burden is to store relations in memory. However, the amount of available memory is limited, insufficient to hold large DW, and is also required for performing join and sort operations. Recently there is increasing interest in approaches that share data and processing among queries.

Cooperative scans [4] enhances performance by improving data sharing between concurrent queries, by performing dynamic scheduling of queries and their data requests taking into account with the current executing actions. While this minimizes the overall IO costs, by mainly using sequential scans instead of a large number of costly random IO operations, and the number of scan operations (since scans are shared between queries), it introduces undesirable delays to query execution and does not deliver predictable query execution times.

QPipe [5] applies on-demand simultaneous pipelining of common intermediate results across queries, avoiding costly materializations and improving performance when compared to tuple-by-tuple evaluation.

CJOIN[6] [7] applies a continuous scan model to the fact table, reading and placing fact tuples in a pipeline, and sharing dimension join tasks among queries, by attaching a bitmap tag to each fact tuple, one bit for each query, and attaching a similar bitmap tab to each dimension tuple referenced by at least one of the running queries. Each fact tuple in the pipeline goes through a set of filters (one for each dimension) to determine if it is referenced by at least one of the running queries. It not, the tuple is discarded. Tuples that reach the en d of the pipeline (tuples not discarded in filters) are them distributed to dedicated query aggregations operators, one for each query. However, its usefulness is limited to small dimensions that can fit entirely in memory and, as recognized in [7], large dimensions may severely impact performance.

SPIN [1] is conceptually related to CJoin, and QPipe in what concerns the continuous scanning of fact data, but it uses a simpler approach with minimum memory requirements and does not have the limitations of such approaches. SPIN uses a de-normalized model, as proposed in [2] as a way to avoid the join costs, at the expense of additional storage costs. Since it has fully data and processing scalability, ONE allows massive parallelization [3], which provides balanced data distribution, scalable performance and predictable query execution times.

3 The *bitset* Branch Processing Approach

The evaluation time (t_{eval}) is constrained by the number of predicates and branches of the current *WPTree*. A submitted query has to process and evaluate each of its predicates, even though similar queries, with common predicates, have been previously processed in the past. As predicates of common queries are associated to common branches, a branch with a given predicate can be repeatedly built or may persist over time while at least one running query uses it. To avoid subsequent evaluation of unchanged data tuples, we propose to maintain the result of the predicate evaluation as tuples flow through data branches. This is particularly relevant for predicates with high evaluations costs. We build a branch *bitset* (bitmap) according to the branch' predicates, where each bit represents the result of the predicate evaluation (true/false) applied to a corresponding tuple index.

As SPIN processes tuples in a circular fashion, when the relation reaches the end, it restarts reading from the beginning. Therefore, future evaluations of a tuple can take advantage of the existence of this *bitset*, since the selection operator that evaluates the predicate can be replaced by a fast lookup operator that look up the corresponding position in the *bitset* to gathers the result. *Bitsets* are small and will be in memory in order to avoid introducing overhead at IO level.

3.1 Creation of Bitsets

A *bitset* is built on-the-fly, as tuples go through the branches and are evaluated, with minimum overhead, since these results are stored in an in-memory data structure. A branch evaluates each tuple and stores the result of the predicate evaluation in the *bitset*, at the tuple index position. This is very important, since it makes it possible in every future evaluation of each row, to decide whether that row should proceed to the next step in the branch, or not, based on the simple lookup of the *bitset*. This avoids most of the evaluation costs. A *bitset* can be built for each value (e.g. 11, 12, 13 …), sets of values, or ranges of the attribute domain (e.g. [10;13]). To avoid additional overhead, *bitsets* are built according to the selection predicates of the submitted queries. For a new selection operator deployment in the *WPTree*, without a matching *bitset*, a new *bitset* is built with the result of the selection predicate. This bitmap is kept in memory and shared by all branches that can use it, allowing future evaluations of these tuples to be replaced by a fast *bitset* lookup operator.

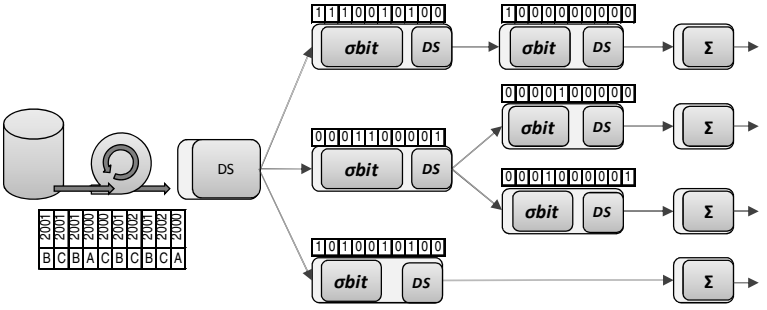

Fig. 1. Branch processing using equivalent BitSets

3.2 Bitset Lookup Operators

Branches can be built, or reorganized, to use *bitsets*. Selection operators (***σ***) with a matching *bitset* are replaced with a bit-selection operator (***σbit***), which performs bit lookups to the corresponding index position in the *bitset,* and thus avoiding the evaluation of these tuples. Selection operators without a matching *bitset* can still take advantage of *bitset* evaluation by combining the existing *bitsets* for other values of the attribute domain, by using a bit-selection not operator (***σ!bit***) evaluates as true all the index positions in the *bitset* that are 0. ***σ!bit*** is equivalent to NOT ***σbit.***

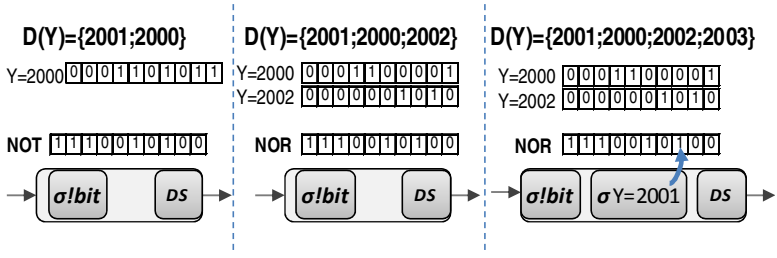

Fig. 2. a) NOT b) NOR c) NOR and selection operator

For instance, consider that the domain of the attribute Y (year) is 2000 and 2001, `D(Y)={2000,2001}`. If there's a *bitset* for 2000, ***bitset*(y=2000)**, the selection operators **σ (y=2001)** can be replaced with a ***σ!bit* (y=2000)** operator that applies a bitwise **NOT** to ***bitset*(y=2000)**. The result is equivalent to **NOT(*σbit* (y=2000)).** Fig.2 shows how *bitset* processing can be employed for boost performance even when a perfectly matching *bitset* does not exist. Fig.2-a) depicts the domain complement.

Fig.2-b) depicts the scenario for a wider domain, where a *bitset* exists for each of the values of the attribute domain except for the value Y=2001. In this case, the ***σ!bit*** operator applies a **NOR(*bit* (y=2000), *bit* (y=2002)).** The domain complement (Fig.2-a)) is a particular case, where the domain has only two distinct values.

Bitset processing can still be used when the number of values, of the attribute domain, without a matching *bitset* is greater than 1 (Fig.2-c). In this case, the selection operator **σ** is maintained in the branch, but it is preceded by a ***σ!bit*** that applies a **NOR** to the existing *bitsets*, and thus obtaining a *bitset* with all the index positions that are certainly false, marked as 0. The goal of this ***σ!bit*** is to avoid the **σ** operator from evaluating these tuples that are known to be false. The remaining index positions, which can be evaluated as true or false, are market as 1. As the **σ** operator evaluates these remaining tuples, it the updates and completes the *bitset* with the result of the evaluation (illustrated in the figure with a blue arrow).

3.3 Mixed Branch Processing: Branches with and without *bitsets*

At any given time, SPIN may have branches with *bitset* operators and other branches that have to evaluate the predicates, because there isn't a *bitset* that matches the selection predicate. Branches that use *bitsets* are pushed forward and connected directly to the base data pipeline to maximize the sharing costs, and to reduce the overall number of tuples that have to be evaluated with selection operators (**σ**). New branches without ***σbit*** operators are connected as usual to existing pipelines, regardless if they have a *bitset* or not, and the corresponding branch predicate is associated with a *bitset* filled with 1's. This *bitset* can be updated with existing *bitsets* related with the branch' predicate, when exists. Whenever a branch ends building a *bitset*, it is replaced by an equivalent branch with a corresponding *bitset* operator (***σbit***). Since *bitset* processing is faster than the tuple predicate evaluation, branches that contain *bitset* operators are reorganized and pushed forward to the base pipeline.

Over time, predicates more frequently used will have a corresponding *bitset*, and therefore will deliver faster query processing times.

3.4 Merging *bitsets* along the Query Logical Path

When in a logical data path, two or more branches use exclusively *bitsets* to evaluate tuples, then these branches are merged into the later one, composed with a single *bitset* that is a logical AND of all these branches. The resulting branch replaces the merged branches, or is directly connected to the base pipeline. The main goal is to filter as soon as possible the data tuples that are relevant to a query, before reaching the branches that evaluate tuples using selection operators. Fig. 3 depicts the new deployment, where the query logical data paths are built with less data branches,

where the last branch of each query path in the previous deployment (Fig. 1) is substituted with a branch that evaluates tuples using single *bitset* that represents the logical evaluation of all the *bitsets* of the logical data path. In the figure, the *bitset* of query 1 (the topmost branch, which has only the first bit set to 1) was built by the selection operator (σ) of the last branch of the logical path, but it could also be built by applying a AND to *bitset(y=2000)* and *bitset(p=b)*. The four queries depicted in the figure are evaluated with dedicated branches, each with a single ***σbit*** operator. These *bitsets* can be pushed forward to the base pipeline and be used by the data reader to reduce the cost of getting and forwarding the data.

3.5 Pushing forward *bitsets* to the Data Reader

When all the branches connected to the base pipeline use ***σbit*** operators to filter relevant tuples for each branch, then a *WPbitset* is created by applying a logical OR to all the branches' *bitsets.* The data reader can use this *WPbitset* to control the data to gather from storage and optimize the IO reading cost.

In a mixed environment, with some branches using ***σbit*** operators and others not, an all 1's *bitset* is considered in the bitwise OR, when exists at least one first level branch that does not use a ***σbit*** operator to evaluate tuples. For the previous deployment, the result of the merging OR is depicted in Fig. 3.

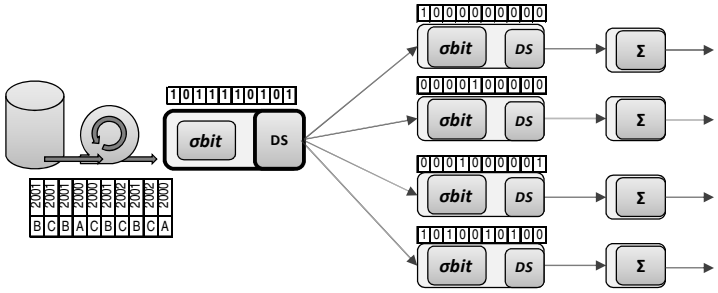

Fig. 3. Data Reader *Bitset* computed as a bitwise OR of the branches bitsets

WPbitset allows the Data Reader to control the data to gather from storage and optimize the IO reading cost, by skipping data chunks that are not relevant for the workload processing tree. For relevant chunks, the data reader uses the *WPbitset* to decide which tuples to place in the pipeline (only tuples pinpointed by the *WPbitset).* Queries, with predicates that are evaluated exclusively with ***σbit*** operators, can early end its execution and return the result, without having to wait for the full loop to be completed. As soon as the number of processed tuples reaches the *bitset count* (hamming distance), then the query can stop execution since all the tuples relevant for the query (which satisfies the selection predicates) were processed. When that occurs, the query execution time fall below the barrier imposed by the IO cost of reading the full relation (in a circular loop), since with *bitsets* a query can end as soon as all the set positions of the *bitset* as been processed.

4 Evaluation

We extended our SPIN engine (release 1.9.1), which is implemented in Java, and incorporated the *bitset*-based processing approach, presented in this paper, and used the TPC-H benchmark with a scale factor (SF) of 10 to evaluate its performance and scalability capabilities. We used an Intel i5 processor, with 8GB of RAM and 3 SATAIII disks with 2Terabytes each, running a default Linux Server distribution. An additional server submits 1000 random variations of Q5, with different selectivity, generated by a varying number of simultaneous concurrent clients. For this setup, we compare the number of evaluations carry out to tuples as they flow and are switched along the branches in the *WPTree* when using the base SPIN setup (SPIN), the *bitset* operators (***σbit***) (SPIN-WP bitset), and when the Data Reader uses the *WPbitset* (SPIN-DR bitset). Fig. 4 depicts the results for a *WPTree* with 10 branches.

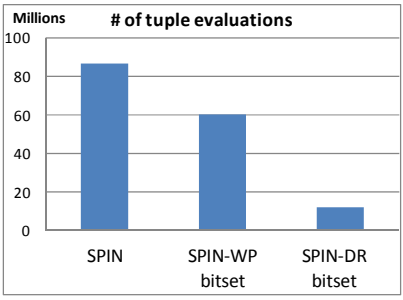

Fig. 4. Number of evaluations

The number of evaluations with SPIN-WP bitset, because *bitsets* are merged along the logical path, is utmost equal to the number of tuples, while the base SPIN requires more evaluations to filter and redirect tuples to appropriated branches. With SPIN-DR bitset, the number of evaluates drops significantly, with some chunks aren't read from storage and uninteresting tuples are not placed into pipelines, and thus reducing the number of evaluations, and improving the average execution time (depicted in Fig.5).

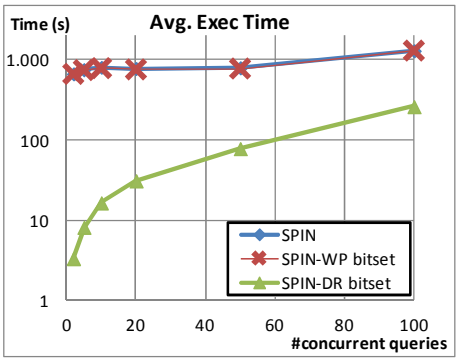

Fig. 5. Average execution time

The results shows that even though *bitset* evaluation is faster than tuple evaluation, the SPIN-WP bitset setup only yields a slightly improvement in the average execution time in comparison with SPIN (in the graph they are almost the same). This is because this query workload results in a *WPTree* with utmost 50 branches, with low selectivity operators and the IO reading costs represents a large percentage of the overall execution cost. Therefore, for more complex, and more processing intensive evaluations, SPIN-WP bitset will deliver better results that the base SPIN. The results also show that when we apply the *WPbitset* to the data reader (SPIN-DR bitset) the execution time is significantly lower since it avoids reading large number of tuples, and consequently reduces the number of evaluations and the evaluation time.

5 Conclusions

We present a *bitset*-based data processing approach that extends the SPIN processing model, a data and processing sharing model that deliver predictable execution times to star-join queries even in the presence of large concurrent workloads. It replaces selection operators, or sets of selection operators, with fast *bitset* selection operators.

Bitset speeds up SPIN by minimizing the processing costs, replacing selection operators with fast bit-selectors, and thus reducing the number of evaluated tuples and consequently the overall processing cost.

Since the number of bitsets is limited by the available memory, we currently are employing run-length encoding compression algorithms, such as the Byte-aligned Bitmap Code (BBC), and Enhanced Word-Aligned Hybrid (EWAH) which require very little effort to compress and decompress and can be used in bitwise operations without decompression. We are evaluating in algorithms that take into account factors such as usage, rebuildability, predicate evaluation costs and hamming distance for managing the *bitsets* that are maintained in memory. We are also working on a column-oriented data organization that can be combined with *bitset* processing for further improvement in IO reading costs and query execution times.

Acknowledgments. This work was partially financed by iCIS – Intelligent Computing in the Internet Services (CENTRO-07- ST24 – FEDER – 002003), Portugal.

References

[1] Costa, J., Furtado, P.: SPIN: Concurrent Workload Scaling over Data Warehouses. In: Proc. of 15th International Conference on Data Warehousing and Knowledge Discovery - DaWaK 2013, Prague, Czech Republic (2013)

[2] Costa, J.P., Cecílio, J., Martins, P., Furtado, P.: ONE: a predictable and scalable DW model. In: Proceedings of the 13th International Conference on Data Warehousing and Knowledge Discovery, Toulouse, France, pp. 1–13 (2011)

[3] Costa, J.P., Martins, P., Cecílio, J., Furtado, P.: A Predictable Storage Model for Scalable Parallel DW. In: 15th International Database Engineering and Applications Symposium (IDEAS 2011), Lisbon, Portugal (2011)

[4] Zukowski, M., Héman, S., Nes, N., Boncz, P.: Cooperative scans: dynamic bandwidth sharing in a DBMS. In: Proceedings of the 33rd International Conference on Very Large Data Bases, Vienna, Austria, pp. 723–734 (2007)
[5] Harizopoulos, S., Shkapenyuk, V., Ailamaki, A.: QPipe: A Simultaneously Pipelined Relational Query Engine. In: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, pp. 383–394 (2005)
[6] Candea, G., Polyzotis, N., Vingralek, R.: A scalable, predictable join operator for highly concurrent data warehouses. Proc. VLDB Endow. 2, 277–288 (2009)
[7] Candea, G., Polyzotis, N., Vingralek, R.: Predictable performance and high query concurrency for data analytics. The VLDB Journal 20(2), 227–248 (2011)