Noname manuscript No.
(will be inserted by the editor)

Cooking DBMS Operations using Granular Primitives

An overview on a primitive-based RDBMS query evaluation

Bala Gurumurthy · David Broneske · Tobias Drewes · Thilo Pionteck · Gunter Saake

Received: date / Accepted: date

Abstract The increasing heterogeneity of the underlying hardware forces modern database system engineers to implement multiple variants of a single database operator (e.g., join, selection). With increasing heterogeneity, these variants become too complex to maintain and tune for different devices. To overcome these disadvantages, developers use an alternative, primitive-based operator design. This design paradigm splits the database operators into granular functions or primitives and executes a given operator by combing the necessary primitives. Hence, we require only a limited set of these primitives as we reuse them for multiple database operations. Thus, tuning a single primitive improves efficiency of all the database operations using it.

In this survey, we provide an overview of a primitive-based database engine. First, we list different primitives from literature and place them in a hierarchy from the finest granular level to a complete database operator. Second, for each of primitive we list its possible tuning opportunities. Finally, we discuss the significance of primitive-based execution on the query engine. Overall, this survey aims to serve as a general reference for implementing a primitive-based query engine and possible strategies to tune it for specific processors.

Keywords Primitives · RDBMS operation · Operation variant · Granular functions

1 Introduction

Modern database machines overcome their bandwidth and scaling limitations using a variety of heterogeneous (co-)processors [8,15,16]. Hence, current database management systems (DBMSs) make use of heterogeneous compute units for processing (e.g. CoGaDB [6], Ocelot [16], GPUQP [14], DoppioDB [35]). These systems mostly port the necessary database operations to the underlying devices creating multiple variants for a single operation. Implementing the variants is time consuming and complex. To overcome these disadvantages, a primitive-based execution model is used. This model divides the database operators into granular functions, which are called *primitives*, where multiple operators share common primitives.

The name *primitive* is used in a broad sense and hence, a primitive in literature can range from describing a database operator to the finest granular function (e.g., a parallel loop with a single instruction).

In this work, we aim to shed some light on the current state of the art in the area of primitive-based database engines. We explore the design space of a primitive-based execution model and provide a comprehensive survey on various primitives available for implementing an operator. Furthermore, we provide a set of primitives necessary for implementing a given database operator and present different tuning opportunities for these primitives to the underlying hardware. Finally, our proposed classification of primitives spans multiple levels of granularity, which requires changes in the query execution engine. We discuss different implementation strategies than can be used for primitive-based evaluation by the query engine and its components.

Our main contributions in this work are:

- A hierarchical model of the primitives based on their composition level (Section 2).
- A comprehensive set of DBMS primitives generalized from several papers and systems (Section 3).
- A list of tuning opportunities available for these primitives (Section 3).
- Finally, the implications of using primitives on a query engine (Section 6).

Otto-von-Guericke Universität, Magdeburg, Germany

2 Primitives

Modern CPU architectures provide multiple code optimization opportunities to fine tune a given database operation. Based on efficiency criteria (e.g., cache utilization, response time), different variants are available for a single operation. In addition to these criteria, the code optimization strategies also vary according to the used processor. Thus, we have a plethora of different variants available for evaluating a single operation. Implementation of all these variants is time-consuming and not manageable. As an alternative, these operations are evaluated by combining database primitives.

A database primitive is a basic building block that can be re-arranged to execute a part of a given DBMS operation. Primitives aim to minimize the implementation time and also improve the re-usability of code. Furthermore, they usually are easily parallelizable. Exploiting this level of parallelism improves the throughput of overall computation. Based on the level of granularity, multiple primitives have been proposed for DBMS operations [14,23,5]. On a conceptual level, the DBMS operations itself serve as primitives that are combined to form the user-defined query. We detail this hierarchy in Figure 1.

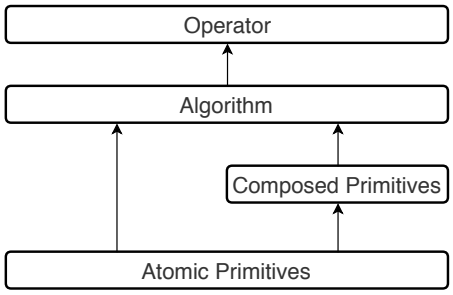

Fig. 1: Hierarchy of Primitives

As shown in the figure, primitives are present on multiple levels. At the lowest granular level, primitives are just stand-alone functions with multiple variants for efficient execution. These atomic primitives are combined with operation-specific components forming coarse-granular or *composed primitives*, which provide additional functionality by adding a tailor-made component to the atomic primitives. Finally, the desired algorithm for executing an operation is built using these primitives. In the next sections, we provide an overview of the different primitives available and discuss how they are used to form a DBMS operator.

3 Atomic Primitives

Atomic primitives are fine-granular functions that can be combined to form a larger operation. These primitives are data parallel in nature and can be tuned for efficiency based on the underlying processor. Furthermore, an atomic primitive consists of a single loop with a body that cannot be split further (i.e., it contains a single operation executed on a vector of data). In this section, we present the atomic primitives *map, scan, reduce* and *scatter & gather*, as well as the code optimization strategies available for each of them. An overview of their properties can be found in Table 1.

3.1 Map

The map primitive applies a function f (e.g., a *user-defined function (UDF)*) to all values in the given input vector. This is a non-blocking primitive as the function applies f to each input independent of other.

3.1.1 Naive Implementation

In a basic implementation, a map is simply a loop applying f to all input values. This is depicted in Algorithm 1.

```
foreach Value in Vector do
    Result_Vector[Index] = f(Value)
    Index++
end
return Result_Vector
```

Algorithm 1: Map Algorithm

Depending on the definition of the function f, a map either takes two columns, or one column plus an additional constant as input. Frequently used functions include:

- **Arithmetic operations:** Arithmetic operations perform an arithmetic function (e.g., addition, multiplication) over vector pairs or a vector-constant value pair.
- **Logical operation:** Logical operations combine vectors on a logical/bitwise operation on the vector values.
- **Comparison:** A comparison tests whether an input value in the vector passes a given condition. It returns boolean vectors of `true` or `false` as a result, which can also be bit-packed into larger words.
- **Bit-shift:** A bit-shift takes two inputs – value and shift count – and shifts bits inside the given value by the shift count value.

Table 1: Properties of atomic primitives

Primitive	Input Size	Working Space	Output Size	Access pattern	Multi-step
MAP	N	-	N	Sequential	N
REDUCE	N	$\leq N$	1	Random	Y
SCAN	N	$W + N$	N	Random	Y
SHUFFLE	N	-	N	Random	N

N - Vector Size; W - Work group size;

– **Zip:** A zip pairs values from two vectors into a single vector.
– **Project:** A project forwards the required set of values from the given vector to the next function in the process.

3.1.2 Code Optimization on Map

Since a map is implemented in a loop, loop-based code optimization can be applied. Loop unrolling is used to reduce pipeline stalls. Stalls are reduced by unrolling a tight loop to the desired depth. This depth is decided based on the underlying hardware [17]. Another optimization commonly used is loop fission or fusion. Depending on the execution environment, a single map function can be divided into multiple smaller loops or multiple maps can be combined into single loop [22].

3.2 Scan

A scan is proposed as a fundamental primitive for vector processing by Guy Blelloch [4]. In contrast to the notion in DBMSs, where a scan is used to access tables, the scan primitive in this context produces for each value of a vector an aggregate of all predecessors of that value. This primitive is useful in determining the index for storing values in a parallel processing environment. The scan is also useful in searching regular expressions, solving recurrences etc. [3]. Other than these, the scan primitive is also used in sorting a vector by splitting input into arbitrary segments.

3.2.1 Naive Implementation

A scan primitive takes a single vector as input and performs over all the values between initial and current index the given binary operation $\oplus$ (c.f. Equation 1). Some binary functions commonly applied are sum, min, max and multiply.

$$scan([v_0, v_1, ..., v_n]) \mapsto [v_0, v_0 \oplus v_1, v_0 \oplus v_1 \oplus v_2, ...] \quad (1)$$

The parallel implementation presented by Blelloch has the input and output data found in the leaf nodes of a balanced binary tree and computes results in two phases. The phases are up-sweep and down-sweep.

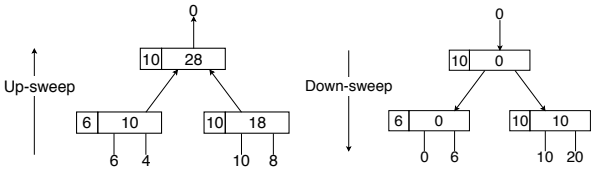

Fig. 2: Example of prefix-sum phases

Up-sweep: During up-sweep, the binary operation is carried out on the child node, and the resultant values are stored in their corresponding parent nodes. Along with the result, the value of the left node is also stored in the parent.

Down-sweep: In down-sweep phase, the binary operation is applied to the left node stored in memory and the corresponding parent node value and the result is stored in the right child. The value of the parent node is stored in the left child of the current node. Note, the initial value for the root node is given as zero for further propagation.

In Figure 2, we show an example of the prefix-sum calculation using the two phases. The result from the root to the leaf is carried down and the binary operation is performed again. The result is stored in the right child thereby computing the result or output vector.

This algorithm has complexity of $O(n \log(n))$. The final results are present in the leaf nodes.

3.2.2 Segmented Scan

As mentioned previously, the simple scan operation is extended to perform a segmented scan. These segments are marked using flags. The common functions used with segmented scan are arithmetic and comparison. These two variants of scan are used as the primitive function for many composed operations such as enumerate, distribute, etc. Their implementations are detailed by Blelloch et al. [4]. The prefix-scan composed database operations are discussed in Section 4.

3.2.3 Code Optimization on Scan

Since prefix sum is a multi-fold operation (where the result is calculated in multiple stages), data parallelism

and task parallelism provides additional benefits by reducing latency. In CPU implementations, this is realized using threading and pipelining of instructions. Whereas, a GPU with its massively parallel processing capabilities is used as an ideal candidate for computing prefix-scan results. One of the common implementation of prefix scan in GPU is given by Horn et al. [18]. Whereas, Senguptha et al., provide a GPU implementation of segmented scan [34].

3.3 Reduce / Aggregate

Reduce computes the aggregate of values in a given vector. A reduce evaluates multiple input values and returns a single output. Hence, the result is not forwarded until all the corresponding input values are evaluated. This operation is a pipeline-breaker as the result is not forwarded until all the corresponding values are processed [21].

3.3.1 Naive Implementation

The implementation of a reduce is similar to that of a map. However, unlike the map, the intermediate result from a single value is reused by the given function f for computing the next result. Algorithm 2 depicts the execution of a reduce.

```
for Value in Input_vector do
 |  Result = f(Value, Result)
end
return result
```

Algorithm 2: Reduce

Apart from using a UDF, all aggregate operations carried out by SQL are implemented using reduce (e.g., min, max, sum).

3.3.2 Code Optimization on Reduce

Similar to other primitives, parallel execution of a reduce improves its throughput. Horn et al., provide a two-phase algorithm for computing reduce [18]. Since the final result depends on the intermediate results of applying the function f, a parallel reduce needs at least two passes to compute the final result. In the first pass, a local aggregate is calculated within each of the thread group with an intermediate output size equal to the available work-group size. In the second pass, the intermediate results are aggregated to compute the final result. Since reduce is similar to map, all the optimization criteria of the map can be applied to reduce such as SIMD acceleration (data parallelism) and instruction pipelining for faster processing. Rosenfeld et al. have given an extensive analysis report on the impact of these optimizations for reduce computation [30].

3.4 Scatter & Gather

Scatter and gather are data movement primitives that performs a indexed read/write on given data. These primitives are used to rearrange the given dataset. These two different primitives are discussed below.

- **Scatter:** Writes data into a vector from the input on the index given.
- **Gather:** Read data from the input vector at the given index.

These are the atomic primitives required to compose the coarse granular primitives or a complete DBMS operation. In the next section, we detail the composed primitives developed by combing atomic primitives.

4 Composed Primitives

The next layer of DBMS primitives from Figure 1 are composed primitives. Although composed primitives can be implemented by pipelining atomic primitives, they can also be implemented from scratch by fusing the code of atomic primitives.

4.1 Filter

Filter is used for selecting vales from a given input vector that satisfy a certain condition. This primitive is used for selection and returns a subset of the given input vector.

4.1.1 Composing Filter

A primitive-based filter implementation is given by He et al. [15]. The pipeline is depicted in Figure 3.

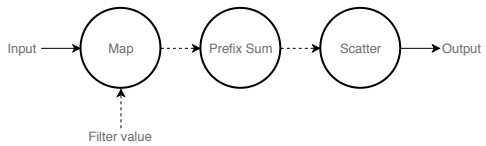

Fig. 3: Composing Filter

Since the result size is not known in advance, memory space may not be allocated for it or else have multiple empty spaces unused. Hence, the filter pipeline

requires a map to determine the results followed by a prefix scan to allocate the required space and finally scatter to store the results.

4.1.2 Code Optimization of Filter

A filter can also be implemented from scratch using code optimization strategies for efficient execution. There are three major improvements available for a selection [10]

- **Branched logical:** The execution breaks as soon as a condition fails. It has the disadvantage of costly multiple branches (if the data distribution is highly selectivity).
- **Branched bitwise:** It evaluates all the conditions in a bit-wise manner and a final branching statement will check the result.
- **Predicated:** The predicated version converts control dependencies into data dependencies thereby omitting branches.

Along with different implementations, the conjunction order of selections also provides additional throughput. Ross et al. provide a comprehensive analysis of filtering using conjunctive selections [31], while Broneske et al. [8] and Zeuch et al. [40] show the impact of hardware sensitivity on selections. A GPU implementation of select conditions is given by Sitaridi et al. [36]. Finally, selection over compressed vectors provides additional advantages and many works also investigate the impact of the compression in filtering [37,38,26]. Selections over compressed columns require an additional step of generating the column values from the intermediate result. This additional step is called materialization, which we detail in the next section.

4.2 Materialize

Materialize is mainly used in synthesizing column values from encoded results. Abadi et al. have done an extensive analysis in materialization [1]. Materialization is commonly used in heterogeneous computing environments, since the data transfer is the bottleneck. The selected results are encoded into bitmap or position list values and transferred, thereby reducing the transfer bottleneck. A materialize is then used to retrieve the column values from the generated intermediate results.

4.2.1 Composing Materialize

Materialize is executed using a map and a gather primitive based on the encoded data.

Bitmap: For filter-materialize pipelines with a bitmap, a map compares the input and generates a series of 1s or 0s as a result (either bit-packed or byte-packed). In this case, materialize uses a method similar to filtering, where a prefix-scan is used to identify the location and a map to store the results (refer Figure 3).

Position list based: In case of performing materialize with a position list, a map function compares the input vectors followed by a gather primitive to collect the values. The composing primitives are constrained only on the order of execution and there can be other primitives executed between their execution. We detail the execution of filter-materialize with position list in Figure 4.

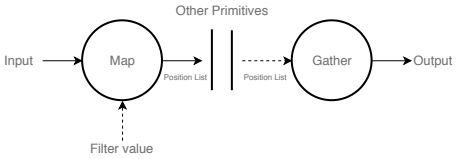

Fig. 4: Composing Materialize

4.2.2 Code Optimization of Materialize

Since, materialize is a special functionality of map and gather, their optimizations can be applied to it. Abadi et al. provide a cost model for placement of the gather primitive early or late in the pipeline [1], calling it early materialization and late materialization, respectively.

4.3 Hash Build

Hash build takes a value and based on the underlying hash function and technique stores the value in its respective bucket. A hash table is built using a map along with a tailor-made hash put component for the underlying hashing technique. Note: we represent these tailor-made components using double circles.

4.3.1 Composing Hash Build

To implement a hash build, a map is used with a hash function (e.g., multiplicative hashing [20]) to provide the index of a bucket to store the value in. Since the underlying techniques are prone to collision, the hashing technique defines a collision-resolution mechanism as hash put. The primitive composition of the hash build is shown in Figure 5

There are many hashing techniques available and based on the technique, a different collision resolution mechanism is used. An extensive analysis on the hash functions and the available hashing techniques is given by Richter et al. [29].

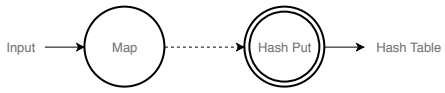

Fig. 5: Composing Hash Build

4.3.2 Code Optimization of Hash Build

Hashing from scratch combines hash build with a probe phase. The code optimization on the hash build is applied on the hash probe as well. Hardware sensitive optimization such as SIMD optimized cuckoo hashing is given by Ross et al. [32]. Polychroniou et al. have given an abstract overview of using SIMD on hashing techniques [24].

4.4 Hash Probe

Hash probing is used to retrieve values from a given hash table. This along with hash build is used to evaluate join and aggregation queries. Probing performs a search of a value from a computed index in the hash table. Since hashing has collisions, the given search value may have to filtered from possible candidates.

4.5 Composing Hash Probe

Probing is highly dependent on the hashing technique used to insert a value into the hash table. Hence, the hash put component is modified to provide the possible indexes to probe. We name this as hash get. Values in the indexes given by hash get are fetched using a gather and finally, a map is used to compare the value at the determined indexes. The flow of hash probe is given in Figure 6.

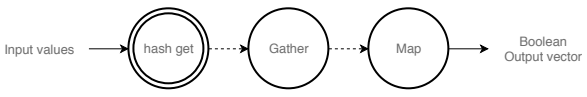

Fig. 6: Composing Hash Probe

4.6 Split

A split or partition divides the given array of values into multiple arrays. Split gets in an input vector along with a flag vector and based on the flag, the given input is split into multiple chunks.

4.6.1 Composing Split

He et al. have devised a way to compile split using their data parallel primitives [15]. Based on their interpretation, split is compiled using map, prefix-sum, gather and scatter primitives. Their interpretation is lock-free and uses a histogram to compute the required space. This is given in Figure 7.

Fig. 7: Composing Split

Split is one of the complex operations and requires five steps. At first, a map is used to prepare the histogram of values per bucket in each thread. Then these histogram values are scattered into an array and a prefix sum is performed. The results from the prefix sum are scattered back to the threads marking the write locations for the threads. Finally, the values are scattered based on the indexes.

4.6.2 Code Optimization of Split

The impact of code optimizations in radix partitioning is extensively researched in various work. Polychroniou et al. give a complete overview of different partitioning variants based on cache efficiency and provide details on the impact of hardware to these algorithms [25]. Schuhknecht et al. provide a guided approach for optimizing partitioning [33].

4.7 Sort

Finally, similar to split a sort is also implemented using the atomic primitives. Specifically, He et al,. have presented a way to perform quick sort using atomic primitives [15].

4.7.1 Composing Sort

To perform a quick sort, He et al. divide the given input by arbitrarily chosen pivots until a partition fits local memory. Here, the split primitive is used to divide the vector into chunks. Sorting a local memory chunk is done using bitonic sort networks.

4.7.2 Code Optimization of Sort

Similar to split, the sort can also be implemented from scratch with code optimization strategies. As sorting

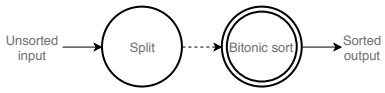

Fig. 8: Composing Sort

requires high availability of data in memory, cache-efficiency improves the processing speed. To this end, Govindaraju et al. provide an implementation of a cache-efficient sorting algorithm for GPUs [12]. To further improve the availability of data, Albutiu et al. proposed a NUMA aware algorithm for sorting [2].

A DBMS operation is executed by combining the above atomic and composed primitives. Other than code optimization strategies mentioned for each of the primitives, considering certain external factors also improves the performance of the primitives. We discuss some of these factors in the next section.

5 Other Impact Factors

The most common influencing factors are the chosen access pattern, parallelism, as well as data structures.

5.1 Access Pattern

In a heterogeneous system, a device-specific access pattern (e.g., sequential or coalesced memory access) is used for additional efficiency. Rosenfeld et al. analyzed the impact of the access patterns on various devices and show that the right access pattern improves efficiency on the underlying hardware [30].

5.2 Parallelism Mode

One major advantage of splitting DBMS operations into granular primitive is to enable concurrent execution. Based on the parallelism type – data or instruction parallelism – two different functionalities are required. For instruction parallel execution, the result of concurrent primitives must be materialized before being used in the next primitive. This is processed using a custom materialize function based on the used primitives. In the data parallel approach, the given data is divided into multiple chunks and each of the chunks is executed in parallel. As a final step, these locally processed results are combined by performing the same operation over the final results. We call this secondary combine as *defer*. Similar to the materialize primitive, defer is also based on the underlying function. Thus, based on the parallelism method an additional hidden function is used. This also influences evaluating a complete query.

5.3 Data Structure

Though sequential access provides additional advantages of prefetching and data locality, different data structures are used for efficient processing of selective operations [28]. Powerful indexes such as CSB-tree [27], FAST [19], k-ary search trees [41], or ELF [9] can be used for efficient selection.

6 Primitive-Based Execution in a Query Engine

Since the DBMS operations are divided into primitives, the query engine is also modified to execute them efficiently. The first step is to split the given relational operation into granular primitives. Since multiple levels of primitives are available, a DBMS operation is either a set of atomic primitives or a combination of the predefined composed primitives and the atomic primitives. We depict the necessary primitives for composing a complete operation in Figure 9.

For example, grouped-aggregation operation can be executed using two techniques: sort-based or hash-based. To perform a hash-based grouped-aggregation, first the given input is grouped using hash build followed by a reduce primitive to perform the aggregation. Similarly, based on the selected operation, a different sequence of primitives can be selected for execution.

These primitives are executed either as stand-alone functions or as a pipeline of primitives. Once a query plan with different granularities of primitives is determined, fusing the primitives together improves the performance. There are common patterns available in fusing these primitives available and we discuss them in the subsequent section.

6.1 Pipeline Patterns

The flow of execution among the primitives is one of the factors affecting overall processing efficiency. The traditional iterator model suffers from the overhead of multiple function calls [13]. One of the alternative models is the compiled-query execution model where the given query is compiled from granular functions [21]. This model reduces the working size as the processing data resides in the device register rather than in memory. Using this execution model, multiple primitives are combined into a single coarse-granular function. There are various pipeline patterns proposed for compiling the primitives [11, 39, 28]. They include *project*, *product*, *select*, *set operation*, and *join*. In Figure 10, we map the different pipeline patterns possible. Notably, although

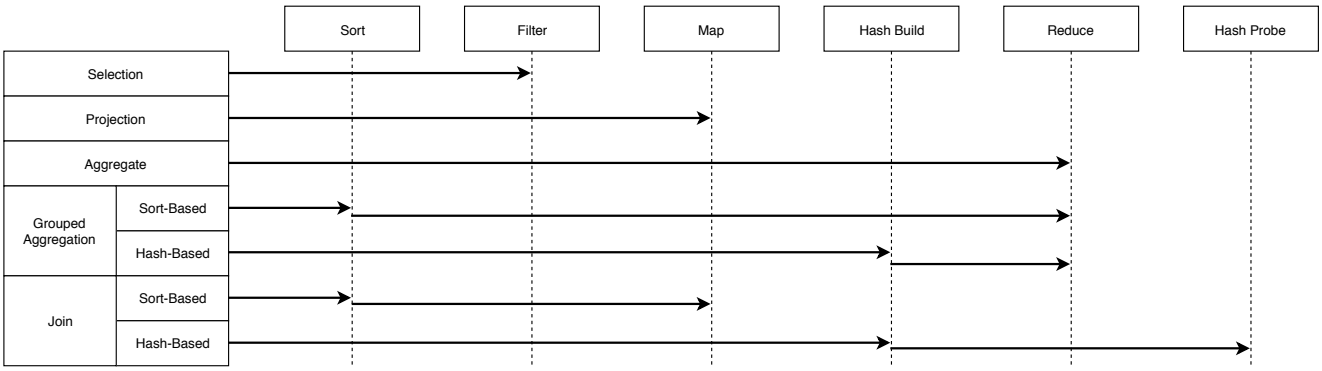

Fig. 9: Composing DBMS operators from primitives

some of the pipelines in the figure can be composed together into single pipeline, we kept them separate for better illustration of the pipelines.

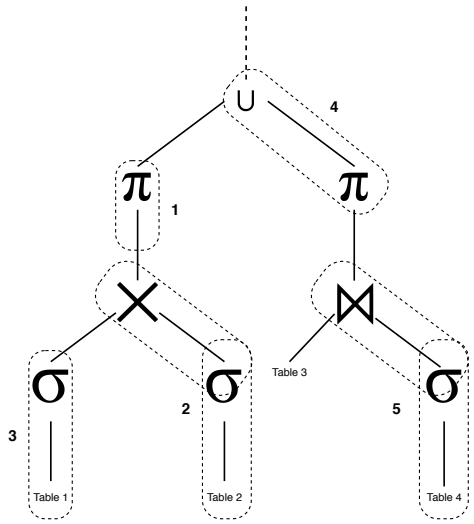

Fig. 10: Pipeline patterns

6.1.1 Project

The project pattern is the simplest pattern. As shown in Figure 10(1), it is used for pipelines that only project an input on a set of attributes. Since there is no blocking operation, all primitive functions can be combined in a single tight loop.

6.1.2 Product

In this model, the result is the product of the number of entries in the input relations. This is single-pass pipeline as not extra step for computing the result size is required (c.f. Figure 10(2)).

6.1.3 Select

The selection pipeline consists of two passes. The passes are similar to the filter passes with the first pass computing the local results and their histograms. Then, prefix-sum is computed over the histogram to determine the total size and index positions. In the second pass, the data are stored in their respective indexes. The selection pipeline is depicted in Figure 10(3).

6.1.4 Set Operation

This pipeline is specifically designed for set operations. We show the this pipeline in Figure 10(4). It is comprised of three major stages. First, the data is partitioned into smaller chunks. Second, these chunks are processed with the given set operation. Finally, the local resultant values are merged providing the complete result. Additionally for efficient processing, Wu et al. argue the use of combining set operations into a single operation for additional throughput. They call this fusing of operations as kernel fusion stage [39].

6.1.5 Join

As the name suggests, this involves a join operation in the pipeline (c.f. Figure 10(5)). Join pipelines are evaluated using a custom join operations [11].

6.2 Pipeline Operators

Apart from using the pipeline skeletons to assemble an operator chain, Breß et al. list a set of operators used for developing a compiled execution pipeline [7]. These additional operators can be combined with the required pipelines discussed above. Further, Rauhe et al. have given a two-phase model based on thread-level execution [28]. In this model, the first phase is called *compute phase*, splits the input horizontally into logical

partitions. After compute phase, the threads are synchronized the so-called *accumulate phase* by merging them to produce final results.

7 Conclusion and Future Work

A DBMS using primitives to implement operations has multiple advantages. Granular functions are combined to form a complete operation and tuning one of the available primitives for a device provides efficiency in multiple operators. We show in this work different levels of granularity available among these primitives and discuss hardware-based tuning for the finest-granular level of primitives. Finally, we discuss the impact of these primitives in the design of a query engine.

We have shown three levels of primitives available for compiling a DBMS operation. The atomic-level primitives are present at the lowest granular level. These are stand-alone functions that are coupled into all the primitives in upper layers. The atomic-level primitives also have multiple implementation variants based on the underlying hardware used. In the next layer, an abstract level of primitives is presented known as composed primitives. Composed primitives are coarse-grained and need other components for executing a given DBMS operation. Finally, the implementation of a database operator itself is present in the top level.

Also, we list possible ways of compiling primitives into pipelines for efficient processing. We provide a brief overview of the different pipeline patterns available for executing the primitives and the operators needed for implementing the pipeline. Finally, we provide a discussion on the general factors affecting the execution of primitives.

Thus, our main contributions in this work are

- DBMS operation primitives in different granularities,
- variants of the available primitives,
- composing these primitives into complete operation,
- factors affecting the efficiency of query evaluation.

The current primitive list is comprehensive yet not complete. There are different data structure defined primitives available to be explored. Also, we restricted this survey to relational DBMSs leading to the need to explore primitives (especially composed primitives) for graph query engines. Furthermore, how to assemble primitives to suit graph query operators is open for future work.

8 Acknowledgments

This work was partially funded by the DFG (grant no.: SA 465/51-1 and PI 447/9)

References

1. Abadi, D.J., Myers, D.S., DeWitt, D.J., Madden, S.: Materialization strategies in a column-oriented DBMS. In: Proceedings of the International Conference on Data Engineering (ICDE), pp. 466–475 (2007)
2. Albutiu, M.C., Kemper, A., Neumann, T.: Massively parallel sort-merge joins in main memory multi-core database systems. Proceedings of the VLDB Endowment **5**(10), 1064–1075 (2012)
3. Blelloch, G.E.: Prefix sums and their applications (1990)
4. Blelloch, G.E.: Vector Models for Data-parallel Computing. MIT Press (1990)
5. Boncz, P.A., Kersten, M.L.: MIL primitives for querying a fragmented world. The VLDB Journal **8**(2), 101–119 (1999)
6. Breß, S.: The design and implementation of CoGaDB: A column-oriented GPU-accelerated DBMS. Datenbank-Spektrum **14**(3), 199–209 (2014)
7. Breß, S., Köcher, B., Funke, H., Rabl, T., Markl, V.: Generating custom code for efficient query execution on heterogeneous processors. Computing Research Repository (CoRR) **abs/1709.00700** (2017)
8. Broneske, D., Breß, S., Heimel, M., Saake, G.: Toward hardware-sensitive database operations. Proeedings of the International Conference on Extending Database Technology (EDBT) pp. 1–6 (2014)
9. Broneske, D., Köppen, V., Saake, G., Schäler, M.: Accelerating multi-column selection predicates in main-memory - the Elf approach. In: Proceedings of the International Conference on Data Engineering (ICDE), pp. 647–658 (2017)
10. Broneske, D., Meister, A., Saake, G.: Hardware-sensitive scan operator variants for compiled selection pipelines. In: Datenbanksysteme für Business, Technologie und Web (BTW) (2017)
11. Diamos, G., Wu, H., Lele, A., Wang, J., et al.: Efficient relational algebra algorithms and data structures for GPU. Tech. rep., Georgia Institute of Technology (2012)
12. Govindaraju, N., Raghuvanshi, N., Henson, M., Tuft, D., Manocha, D.: A cache-efficient sorting algorithm for database and data mining computations using graphics processors. Tech. rep., University of North Carolina (2005)
13. Graefe, G.: Query evaluation techniques for large databases. ACM Computing Surveys (CSUR) **25**(2), 73–170 (1993)
14. He, B., Lu, M., Yang, K., Fang, R., Govindaraju, N.K., Luo, Q., Sander, P.V.: Relational query coprocessing on graphics processors. ACM Transactions on Database Systems (TODS) **34**(4), 21:1—-21:39 (2009)
15. He, B., Yang, K., Fang, R., Lu, M., Govindaraju, N., Luo, Q., Sander, P.: Relational joins on graphics processors. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 511–524. ACM (2008)
16. Heimel, M., Saecker, M., Pirk, H., Manegold, S., Markl, V.: Hardware-oblivious parallelism for in-memory column-stores. Proceedings of the VLDB Endowment **6**(9), 709–720 (2013)

17. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach, 5th edn. Morgan Kaufmann Publishers Inc. (2011)
18. Horn, D.: Stream reduction operations for GPGPU applications. In: M. Pharr (ed.) GPU Gems, vol. 2, pp. 573–589. Addison-Wesley (2005)
19. Kim, C., et al.: FAST: Fast architecture sensitive tree search on modern CPUs and GPUs. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 339–350. ACM (2010)
20. Knuth, D.E.: The Art of Computer Programming: Fundamental Algorithms, vol. 1, 3rd edn. Addison Wesley Longman Publishing Co., Inc. (1997)
21. Neumann, T.: Efficiently compiling efficient query plans for modern hardware. Proceedings of the VLDB Endowment **4**(9), 539–550 (2011)
22. Pantela, S., Idreos, S.: One loop does not fit all. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 2073–2074. ACM (2015)
23. Pirk, H., Moll, O., Zaharia, M., Madden, S.: Voodoo - A vector algebra for portable database performance on modern hardware. Proceedings of the VLDB Endowment pp. 1707–1718 (2016)
24. Polychroniou, O., Raghavan, A., Ross, K.A.: Rethinking simd vectorization for in-memory databases. Proceedings of the International Conference on Management of Data (SIGMOD) pp. 1493–1508 (2015)
25. Polychroniou, O., Ross, K.A.: A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort. Proceedings of the International Conference on Management of Data (SIGMOD) pp. 755–766 (2014)
26. Polychroniou, O., Ross, K.A.: Efficient lightweight compression alongside fast scans. Proceedings of the International Workshop on Data Management on New Hardware (DaMoN) (2015)
27. Rao, J., Ross, K.: Making B^+-Trees cache conscious in main memory. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 475–486. ACM (2000)
28. Rauhe, H., Dees, J., Sattler, K.U., Faerber, F.: Multi-level parallel query execution framework for CPU and GPU. In: Proceedings of the European Conference on Advances in Databases and Information Systems (ADBIS), pp. 330–343. Springer (2013)
29. Richter, S., Alvarez, V., Dittrich, J.: A seven-dimensional analysis of hashing methods and its implications on query processing. Proceedings of the VLDB Endowment **9**(3), 96–107 (2015)
30. Rosenfeld, V., Heimel, M., Viebig, C., Markl, V.: The operator variant selection problem on heterogeneous hardware. Proceedings of the International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS) (2015)
31. Ross, K.A.: Selection conditions in main memory. ACM Transactions on Database Systems **29**(1), 132–161 (2004)
32. Ross, K.A.: Efficient hash probes on modern processors. In: Proceedings of the International Conference on Data Engineering (ICDE), pp. 1297–1301 (2007)
33. Schuhknecht, F.M., Khanchandani, P., Dittrich, J.: On the surprising difficulty of simple things. Proceedings of the VLDB Endowment **8**(9), 934–937 (2015)
34. Sengupta, S., Harris, M., Zhang, Y., Owens, J.D.: Scan primitives for GPU computing. In: Proceedings of the ACM Symposium on Graphics Hardware (SIGGRAPH), pp. 97–106. Eurographics Association (2007)
35. Sidler, D., Owaida, M., Istvn, Z., Kara, K., Alonso, G.: doppiodb: A hardware accelerated database. In: International Conference on Field Programmable Logic and Applications (FPL), pp. 1–1 (2017)
36. Sitaridi, E.A., Ross, K.A.: Optimizing select conditions on GPUs. Proceedings of the International Workshop on Data Management on New Hardware (DaMoN) pp. 1–8 (2013)
37. Willhalm, T., Boshmaf, Y., Plattner, H., Popovici, N., Zeier, A., Schaffner, J.: SIMD-Scan: Ultra fast in-memory table scan using on-chip vector processing units. PVLDB **2**(1), 385–394 (2009)
38. Willhalm, T., Oukid, I., Müller, I., Faerber, F.: Vectorizing database column scans with complex predicates. In: Proceedings of the International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS), pp. 1–12 (2013)
39. Wu, H., Diamos, G., Cadambi, S., Yalamanchili, S.: Kernel weaver: Automatically fusing database primitives for efficient GPU computation. In: Proceedings of the International Symposium on Microarchitecture (MICRO), pp. 107–118. IEEE (2012)
40. Zeuch, S., Freytag, J.c.: Selection on modern CPUs. Proceedings of the International Workshop on In-Memory Data Mangement and Analytics (IMDM) pp. 1–8 (2015)
41. Zeuch, S., Freytag, J.C., Huber, F.: Adapting tree structures for processing with SIMD instructions. In: Proceedings of the International Conference on Extending Database Technology (EDBT), pp. 97–108 (2014)