# Database Technology for the Masses: Sub-Operators as First-Class Entities

Maximilian Bandle
Technische Universität München
bandle@in.tum.de

Jana Giceva
Technische Universität München
jana.giceva@in.tum.de

## Abstract

A wealth of technology has evolved around relational databases over decades that has been successfully tried and tested in many settings and use cases. Yet, the majority of it remains overlooked in the pursuit of performance (e.g., NoSQL) or new functionality (e.g., graph data or machine learning). In this paper, we argue that a wide range of techniques readily available in databases are crucial to tackling the challenges the IT industry faces in terms of hardware trends management, growing workloads, and the overall complexity of a rapidly changing application and platform landscape.

However, to be truly useful, these techniques must be freed from the legacy component of database engines: relational operators. Therefore, we argue that to make databases more flexible as platforms and to extend their functionality to new data types and operations requires exposing a lower level of abstraction: instead of working with SQL it would be desirable for database engines to compile, optimize, and run a collection of *sub-operators* for manipulating and managing data, offering them as an external interface. In this paper, we discuss the advantages of this, provide an initial list of such sub-operators, and show how they can be used in practice.

**PVLDB Reference Format:**
Maximilian Bandle and Jana Giceva. Database Technology for the Masses: Sub-Operators as First-Class Entities. PVLDB, 14(11): 2483 - 2490, 2021.
doi:10.14778/3476249.3476296

## 1 Introduction

Databases have been a cornerstone of enterprise computing for decades. As is often pointed out, they offer what very few other systems, if any, provide: a powerful declarative language, a model and algebra to enable reasoning about programs, sophisticated compilation and optimization technologies, and a wealth of fundamental techniques to support very high throughput rates. All this while providing consistency, availability, and strong recoverability guarantees. Nevertheless, more and more users have been turning their backs on databases in the pursuit of flexibility and performance, willingly giving up the enumerated guarantees. For example, building directly upon intermediate formats like Apache Arrow has grown in popularity, offering more flexibility for storing and processing data. While this simplifies things in the short run, it makes management more complicated in the long run, for instance, when synchronizing data. The same holds for big data frameworks like Spark, which demonstrate the expressivity of offering finer granular operations for constructing various dataflows. This supports many use-cases but lacks some advanced features such as an optimizer. We argue that there is no reason why traditional databases cannot support more flexible ways of accessing and working with data.

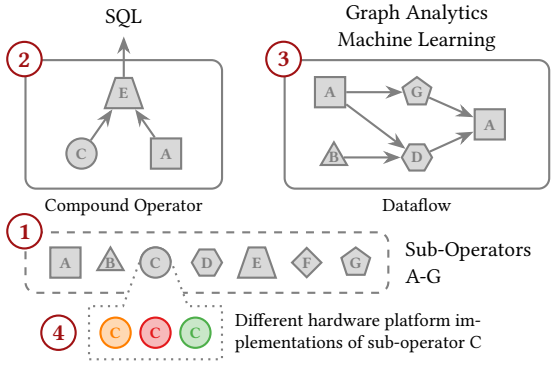


**Figure 1: Sub-operators (1) build more complex data operations (2) or dataflows (3), where each sub-operator can be implemented on multiple hardware platforms (4).**

Currently, both big data processing and hardware advancements are driving the community to develop a variety of techniques. These include domain-specific languages (DSLs) tailored to particular applications [11, 28], cross-compilation techniques to enable execution on different platforms [75, 78], automatic parallelization platforms for running at large scale [30, 84], and connecting different frameworks for cross-optimization [61]. Some of these mirror developments in the database world: new compilation techniques [36, 42], new data types and languages for dealing with them [49], optimizations for multicore [88], designs for GPUs [25, 64] and FPGAs [59].

In this paper, we argue that the most concrete starting points for such innovations are the concepts developed around database engines. Moreover, a great deal of existing technology can be reused, such as operator models [46], compilation techniques [38, 41, 53], composability and orthogonality of operators [19, 39], optimization and scheduling techniques [44], etc. However, the only way to enable more flexibility is to change the explicit abstraction level of the database engine interface. Thus, the system should also expose *sub-operators* and provide them as an intermediate representation to other applications and compilers (Figure 1).

By sub-operators, we mean logical functions that perform fundamental data transformations and management tasks. We call them sub-operators because instead of implementing a full relational operation (e.g., a join), they implement relatively basic functions, for example, hashing, filtering, sorting, scattering, or gathering data. Obviously, some of these are already used within database engines (for optimization or compilation [9, 17, 36, 39, 71]), and


Proceedings of the VLDB Endowment, Vol. 14, No. 11 ISSN 2150-8097.
doi:10.14778/3476249.3476296

the literature is full of new ideas for sub-operators tailored to new hardware (from data exchange operators tailored to RDMA [40, 70] to FPGA-based partitioning [32]). By exposing an *interface* at the level of *sub-operators*, we can transform the database into a *language runtime engine capable of processing much more than just SQL*. This includes different dataflows, like machine learning, graph processing, or easier mapping of the operators onto hardware.

Our proposal produces clear benefits and provides a more elegant and more efficient solution to existing challenges than current ad-hoc proposals. For example, by building complex dataflows using sub-operators, we can reason about the workflow's logic decoupled from the hardware implementation details. Using sub-operators as common building blocks for different dataflows would simplify the maintenance of large codebases, especially when addressing the challenges of a diverse and rapidly evolving hardware landscape, as well as resource disaggregation in the cloud. Cross-compilation of hybrid programs to heterogeneous hardware platforms (CPU, GPU, FPGA) will be made easier by enabling alternative sub-operator implementations. Additionally, small as they are, computationally expensive but frequently used sub-operators can be integrated into the hardware circuit logic, thereby influencing the design of future hardware architectures. This especially pays off in cloud settings, where a holistic approach to hardware/software co-design is of particular interest.

Furthermore, databases can become extensible in a way they currently are not. While UDFs still need to be parts of an SQL-query or table functions have to mimic database functionality to freely customize the query, sub-operators offer more degrees of freedom, while reusing as much of the system as possible. They can be modeled and analyzed more efficiently and thus incorporated into a query optimizer without compromising performance. This also makes them a more natural fit than UDFs, which are out of the optimizer's scope [15]. In such a setting, sub-operators resemble instructions in a processor, available for both the optimizer to rearrange and the compiler to combine as needed. They transform the database engine from a virtual machine for SQL programs to a language runtime executing a complex Instruction Set Architecture (ISA) made up of sub-operators, catering to a heterogeneous range of dataflow workloads.

## 2 Sub-operators as first class entities

Figure 2 presents an overview of the solution we propose. We envision a database engine that exposes a set of *sub-operators as an interface* and combines them into more complex dataflows. Example languages that can run on top are SQL operators and dataflow models used by graph processing and machine learning systems.

Using the sub-operators as an ISA, the database engine becomes more like a language runtime, executing a rich set of alternative sub-operator implementations (as instructions) on heterogeneous hardware platforms. The goal is to support not only relational (column and row) data stores but also graphs, key-value stores, and extensions for complex data types currently stored as blobs.

### 2.1 Sub-operators and interfaces

At present, our choice of sub-operators is based on analyzing prior work that captures basic data access and compute patterns of dataflows that can be mapped to modern hardware. We consider a

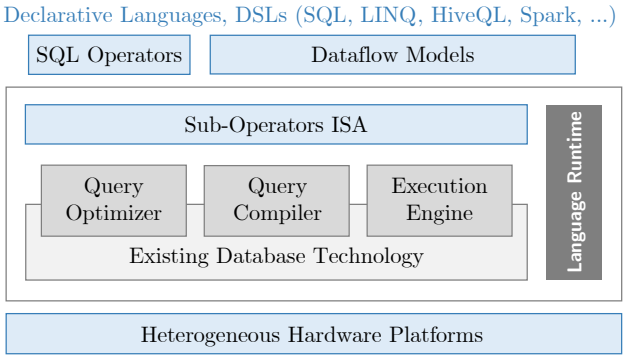


**Figure 2: Overview of a sub-operator-based database engine**

sub-operator to be included in the ISA if it denotes an important management task or if it operates at the right level of data granularity. When an operator is too fine-grained, it becomes difficult to optimize, while if is too coarse (e.g., an SQL join), it limits expressivity and the flexibility to benefit from hardware accelerators.

Table 1 shows and categorizes an example set of sub-operator types, which are sufficient to implement various dataflows. The set can be easily extended with other sub-operator types and concrete instances inspired either by prior work in relational databases (e.g., the exchange operator, range partitioning, string and bit manipulations, fuzzy string matching) or other forms of data processing like machine learning and graph-based analytics (e.g., complex statistical operations over array data).

**Sequential Access** *Scan* and *Materialize* ensure that the system can switch between streamed and buffered execution. It is crucial for working with materialized intermediate states or when distributing the compute pipelines to different hardware targets.

**Random Access** *Scatter* and *Gather* handle memory access to various locations. Combined, they can implement a join. Scatter takes a tuple stream and materializes it based on the scatter function, e.g., the tuple's hash value. Gather fetches tuples from different memory locations based on an input stream and forwards the combined stream for further processing.

**Compute** *Map* and *Fold* provide support for functional programming primitives and typical Map-Reduce workloads. *Map*, for instance, processes a stream of tuples and applies a mapping function $f$ to every element in the stream. It can return a varying amount of return values per tuple, including zero. One example function is predicate evaluation, but several different operations can be implemented, serving as building blocks for more complex dataflows.

**Control Flow** *Loop* is different, as it is not part of a dataflow pipeline. It is necessary for controlling flow and to formulate iterative queries, often typical of incremental and converging workloads.

Depending on the use pattern, there may be compositions of sub-operators that are frequently used by a variety of dataflows and, thus, can be constructed as additional building blocks. For example, an efficient implementation of radix partitioning is often an integral part of relational operators (e.g., radix-join or aggregation).

*2.1.1 Composing sub-operators* We use partitioning to explain how to combine sub-operators into a single, more complex operator. Partitioning consists of several phases, often involving two passes

**Table 1: Example of possible sub-operators**

| Name | Type | Category | Description | Examples |
|---|---|---|---|---|
| **Scan** | ▰ | Sequential Access | Scan materialized data into stream | Tablescan, Buffer scan |
| **Materialize** | ▰\| | Sequential Access | Materialize data into buffer | Output, Print |
| **Scatter** | ▰\| | Random Access | Write to different memory locations | Build hash table |
| **Gather** | ▰ | Random Access | Read from different memory locations | Probe hash table |
| **Map** | ▰ | Compute | Process stream with mapping function | Hash, Compression, UDFs, Filter |
| **Fold** | ▰\| | Compute | Reduce streamed elements by combining | Accumulate, Prefix sum |
| **Loop** | ⇢ | Control Flow | Pass data or state to next iteration | K-means, Gradient descent |

over the input data to enable efficient parallelization in a pipelined system [7]. The left-hand part of Figure 3 illustrates this. Phase 1 performs the first scan and computes the histogram of how the input tuples hash into the partitioning buckets. This phase is constructed using *map* (hash) and *scatter* (build histogram). In Phase 2, each thread calculates the prefix sum to determine each partition's span. To do this, it uses *scan* to read the partitions and *gather* to retrieve the histograms. Then, the *fold* sub-operator computes the prefix sums to determine the offset where each thread needs to write its share of tuples. The final phase performs the second pass over the input data and *scatter*s the tuples to the precomputed locations.

Once constructed, the *new* partition sub-operator can be used as a building block for other relational operators or more complex dataflows. All existing efforts regarding hardware tuning and optimizing the implementation of various relational operators [37, 65], can be immediately applied to the sub-operators, automatically rendering them available to a wider set of operations. For instance, the benefits of software write combining, used to implement the radix-partitioning [83], can now be used for implementing one type of a *scatter* sub-operator. As a result, fast data scattering for many operations is available both in the relational domain and beyond.

Even more, sub-operators provide more intuitive mapping to data processing pipelines and are a natural way to begin automating the introduction of materialization points. We can use them to introduce mini-buffers, as demonstrated with relaxed operator fusion [50]. They also make it very easy to reason about distributing compute functions of a data pipeline to different targets. As shown on the right side of Figure 3, the data flow gradually transforms from reading from storage to compute, which is typical of data processing

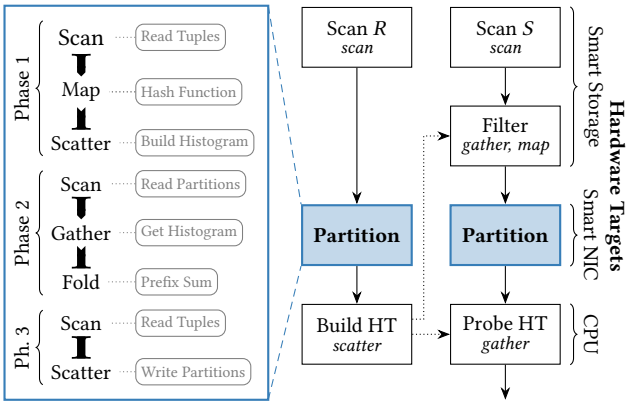


**Figure 3: Composing a partitioner using sub-operators. The new *partition* sub-operator can be used in hash-joins.**

workloads. Close-to-storage workloads can be accelerated by smart storage, like computational storage [57, 74], while partitioning is a good candidate for FPGA acceleration, and the final join logic works best on the CPU. This distributed use case is particularly attractive to every data system in a cloud context.

Also, the hash-join can be further augmented, as shown in Figure 3. For example, a semi-join reducer can be added to avoid materialization overhead in the join [7, 44] by plugging a filter just before partitioning the probe side to drop non-matching tuples early. Such flexibility also allows the database to react gracefully to changes in selectivity and adapt to the workload [19].

Finally, working with sub-operators, also enables us to efficiently compose hybrid relational operators. One example is the *hash teams* operator [34], which merges a hash-join and group-by aggregation to improve performance by performing several hash-based operations without repartitioning the intermediate results.

*2.1.2 Interfacing sub-operators* More complex dataflows can be constructed using sub-operators as graph vertices. The edges of the dataflow represent data dependencies or how data moves from one sub-operator to another. When composing, it needs to be ensured that the input and output of the sub-operators are compatible. The input-output properties can be roughly classified as *buffered* or *streamed*. A streaming sub-operator (▰) directly accepts the input of its predecessor to further process each tuple immediately. A buffered operator (▰|) needs to process all tuples before further advancing, like *fold* and, thus, splits the dataflow at the materialization points into pipelines. The materialized buffers are stateful and encapsulate valuable data properties, such as sortedness, partitioned buffers, min/max statistics for pruning, and data distribution, which can help with additional logical optimizations.

All operators in a pipeline (A ▰ B ▰| C) can be compiled into a single function, which may be offloaded (run) as a compute kernel. This process is referred to as operator fusion and enables performant execution of query pipelines by keeping data in registers or hot caches. To increase performance, the optimizer can introduce mini-buffers, for example to improve locality. Together with the aforementioned data properties, this exploits synergies to choose a faster implementation or avoid extra work, such as sorting data twice.

Common formats like Apache Parquet [4] or Arrow [5] can be used to ensure compatibility with other systems. The *scan* operator, for example, can use Parquet both for reading base tables and for in-memory communication. We can use Apache Arrow as an intermediate format for buffers to store data passed between the pipelines. The *materialize* operator serializes the data stream to allow efficient and convenient processing with existing libraries or user-defined code, similar to user-defined table functions.

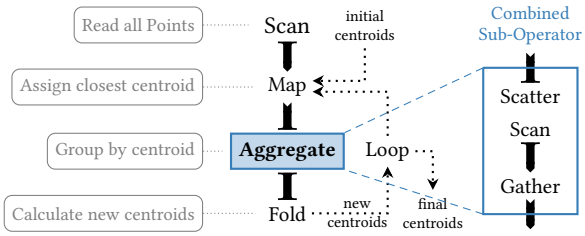


**Figure 4: Schematic overview of one k-means iteration.**

## 2.2 Dataflows beyond standard SQL

Sub-operators offer a more expressive interface for constructing non-conventional operators and generic dataflow systems. One example would be *shared* operators, like a shared scan [80, 89] or a shared join [10], which combine reads of input relations, for example. They are thus effective at performing multiple queries at the same time [47]. It is also possible to combine subsequent relational operators, for example aggregation (group by) and join into a group join [21, 51]. This reuses the hash table for both probing and aggregating if join and aggregation operate on the same predicate.

However, most importantly, we aim to support all other complex dataflows, generated by higher-level declarative or domain-specific languages like HiveQL, LINQ, or Spark. Raven, for example, has previously demonstrated that in-DBMS machine learning can outperform dedicated frameworks [33]. Their system relies on a custom intermediate representation consisting of relational and linear algebra that allows for valuable cross-optimizations. Yet, they still need support for generating code for different hardware platforms and hence explore TVM [12] and Tensorflow [1].

It is along these lines that we propose lowering the data operations for various neighboring data processing domains (e.g., pagerank, k-means, connected components, graph connectivity) onto the sub-operator types: map, reduce, scatter and gather (to name only a few), or adding new ones.To make things more concrete, Figure 4 shows how to compose one iteration of standard k-means using sub-operators. We first *scan* all points and hand them over to the *map* operator, which also accepts the current centroids as parameters. It processes each tuple and determines the closest centroid. The following *aggregation* is composed of sub-operators and materializes the mapped point stream based on whichever centroid is closest. It restarts streaming the data once everything has been mapped and uses *fold* to calculate the new cluster centroids. If the centroids have changed since the previous iteration, *loop* passes the new ones as an argument to *map*, and the dataflow starts again by scanning the points. Otherwise, k-means has converged, and *loop* returns the centroids.

Upon closer examination of this idea, we notice that there are many key optimizations in these dataflow frameworks that can already benefit from existing techniques used in database engines.

Support for desired features such as differential computing has already been addressed by database systems. For example, memoization (caching and materializing intermediate results) within a program and even across concurrent programs can be implemented easily in a database engine, as the required techniques are already in use for queries. Prior work has explored their benefits when implementing memoization for streaming engines [18], or for data lineage and provenance [27]. Similar effort allows efficient algorithm re-computation when the input changes, needed by frameworks like Noria [23] and Naiad [48]. Finally, by allowing alternative sub-operator implementations, we can benefit from existing techniques for processing complex data types, such as images or documents.

## 2.3 Cross-platform compilation and execution

An important feature for modern systems is that they simplify maintenance of complex data processing code-bases for evolving hardware architectures and enabling easy cross-platform portability.

Offering alternative implementations tailored for different architectures allows platform-specific implementations of the sub-operators to be decoupled from the design of higher-level operations and data-processing algorithms. While this allows to abstract from the specifics of hardware implementation, which simplifies reasoning when designing optimal dataflows, it also provides freedom for implementing various flavors of sub-operators, each exploiting the full potential of the underlying architecture [31, 73, 79], deploying it to the cloud [30], or even pushing the implementation down to the hardware circuit logic.

Furthermore, it can simplify reasoning for hybrid platform co-execution. Having multiple implementations of each sub-operator allows us to execute portions of a dataflow computation on different platforms as shown in Figure 3. The actual deployment may depend on specific properties of the underlying resources, the data location, the sub-operator's requirements, and many other factors.

TVM [12] already demonstrates the potential of such an approach. It is an end-to-end ML compiler that picks up ideas from Halide for image processing to separate compute and schedule [69]. Thus, it can optimize the required computations, like tensor operations, and schedule them to run on various hardware targets. Similarly, our sub-operators primarily describe the dataflow, allowing it to be flexible in scheduling the concrete implementation.

This idea extends beyond the compute resources of a single machine. With today's trend for resource disaggregation and compute-capable devices (e.g., smart-NICs, programmable switches, computational storage), certain sub-operators (filtering, projection, partial sorting, partial aggregation) can be offloaded, either down to where the data sits [85], or the data can be processed as it moves (statistics, regular expression evaluation, partial aggregation, partitioning).

## 2.4 Hardware integration

Using sub-operators as common application kernels can also increase the influence that data-processing systems have on hardware design and implementation. Provided that the selected sub-operator instances are simple enough, the majority of them can be integrated into the hardware circuit logic. After all, specialization is one of the most effective ways of increasing performance, as successfully demonstrated by the SIMD instructions present in virtually all CPUs. SIMD instructions offer various primitives, for example, scatter/gather, for vectorized random memory access, which match the proposed sub-operators.

FPGAs have been shown to be an excellent platform for offloading data operations onto hardware logic, as well as for prototyping potential ASICs. Several enterprise systems are already using them accelerate data processing [68, 77] or encrypt it [6].

Following recent trends towards building heterogeneous architectures, more powerful co-processors (e.g., GPUs) are being placed

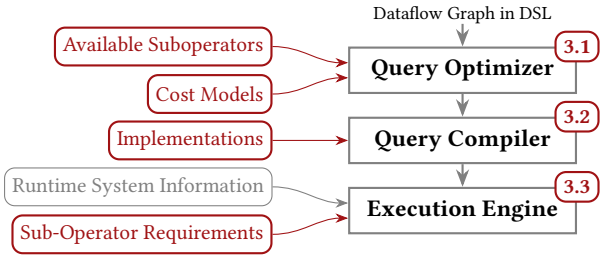


**Figure 5: Overview of the sub-operator system stack.**

the interconnect, to enable more effortless data transfer and co-processing. Intel's research-oriented Xeon+FPGA architecture, in particular, has made the exploration and prototyping of costly data processing primitives on the FPGA even more appealing. For example, the work on FPGA-based histograms [29], while currently used to maintain accurate database statistics, can be repurposed as a sub-operator for data mining algorithms (e.g., for cluster analysis). Similarly, approximated computations and various types of pre-computations (partial sorting or partial aggregation) implemented on an accelerator (FPGA, GPU, or programmable switches) can be used not only in SQL queries, but also for traditional soft computing algorithms, which are by design tolerant of imprecision, like in ML.

Such efforts will complement some of the work done by the architecture community on the design and implementation of various accelerators suitable for data-processing (e.g. Q100 [87], DRAM support for gather-scatter [76], processing-in-memory (PIM) architecture for graph analysis [2], or Oracle's DAX [62]).

Ideally, we can use the generality of the common sub-operators to influence future industry-scale architecture platforms. For instance, an on-chip circuit for automatic (hash and/or range) partitioning and routing of data across parallel entities (hardware threads or machines) can be leveraged by many operators [65]. One example of enterprise chip design, in which such a hardware based partitioner could be integrated, is Oracle's SPARC M8. It refined the DAX (data analytics accelerators) introduced by M7, which are specialized circuits on the memory controller used for the basic data processing operations of *selection*, *scanning*, and *decompression* as data moves between the DRAM and the LLC of the invoking core [3]. Rather than hiding such features behind a faster implementation of SQL, a database engine can encapsulate them as sub-operators and offer them as row primitives.

## 3 Impact on system design

Making sub-operators first-class entities of a data processing system also affects the design and implementation of the system stack. Figure 5 illustrates the components we discuss in this section.

### 3.1 Query Optimizer

One immediate challenge is that query optimization becomes more involved the more choices we have for executing a query. By adding sub-operators, we add a whole new level of customization to each query plan, which is why we propose optimizing statically in layers and dynamically during runtime.

**High-Level Optimization:** The first layer acts on higher-level operations before lowering to sub-operators. For example, by the time the database processes a SQL query, it has already parsed and optimized the query, e.g., decorrelating subqueries [55], and given

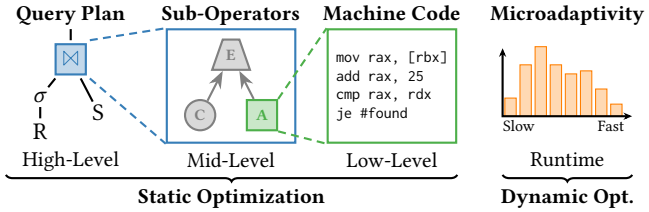


**Figure 6: Optimization levels using sub-operators.**

hints, which physical operator to use [56]. This demonstrates how we still benefit from existing techniques.

**Mid-Level Optimization:** Our optimizer then receives a dataflow graph as input. Each relational operator is deconstructed into sub-operators, as outlined in Section 2.1. This opens up more optimization opportunities at the sub-operator layer, such as reasoning about which physical implementation to choose. This means that we have to decide whether and where to offload computation, and as a direct result, how to split the dataflow between all available hardware resources. Relatively small sub-operators considerably simplify the analysis and derivation of their individual cost models compared to the cost functions for full SQL operators or UDFs.

Since sub-operators typically have a clear data access pattern and mostly consist of a single iteration over the data, the cost models can re-use a significant amount of prior work that has identified how to model the costs of the access patterns for different hardware platforms. For example, for modern multicore machines [46], GPUs [26], or other heterogeneous architectures [82].

**Low-Level Optimizations:** Meta-frameworks like MLIR are used to lower the dataflow to target architectures. [43] They can be applied after the sub-operators are assigned to a hardware target, consider the whole dataflow, and offer their own specific optimizations, such as constant propagation or auto-vectorization.

**Dynamic Optimization:** However, the opportunities of having so many compatible variants of the same sub-operator still poses a lot of challenges, Depending on where the data sits and how busy each of the disaggregated hardware resources is at the moment, offloading might be beneficial or not. Since we cannot foresee all these parameters while preparing the query, we also need runtime adaptivity. The idea is that, each sub-operator implementation alternative can be augmented with auxiliary information, such as the cost of its execution on the particular hardware platform, or its resource footprint (i.e., the resource requirements for efficient execution). The latter is in particular useful for multi-query, parallel executions in a noisy environment (e.g., in the cloud) or microadaptivity, as we discuss in Section 3.3.

### 3.2 Query Compiler

The compiler takes the output from the optimizer and generates a program executable. In doing so, modern compilers already blur the boundaries between relational operators and operate on pipelines of sub-operators until a pipeline breaker is reached [36, 50]. Generating pipelines and pipeline breakers follows on naturally from our discussion on streaming vs. buffering sub-operators in Section 2.1.2.

The query compiler splits relational operators into smaller units. For example, even if an operator consists of multiple stages that logically belong together (build and probe in a join), it is often physically separated into units as parts of different execution pipelines.

As a result, both compiler and execution engine do not operate internally with relational operators, but with their building blocks [44].

To further enhance the flexibility of data processing engines, a compiler could use ready-generated sub-operator implementations, either with different resource footprint(s) [8] or variants that match the desired hardware platform, as suggested by the optimizer. These alternative sub-operator implementations can be either handcrafted using existing optimization techniques for SQL operators or automatically generated. Frameworks like LLVM [42, 43, 75], Voila [24] or Lightweight Modular Staging (LMS) [71] support specialization and are already in use in systems like Tupleware [16], HyPer [35], Umbra [54], and Legobase [38].

Finally, the mere idea of using sub-operators as an IR and explicitly exposing them as an external interface is highly compatible with recent proposals for meta-compiler frameworks like MLIR [43]. Combined with sub-operators as an IR, we consider this to be the only systematic way of developing database systems for heterogeneous hardware. A common intermediate representation enables our community to rely on proposals like MLIR for low-level optimizations or offloading to accelerators like FPGAs [75].

### 3.3 Execution engine

The query executor caches alternative implementations of the same sub-operator, and choose one of them dynamically during query execution, based on the preference(s) suggested by the optimizer.

If integrated within an (operating) system runtime, the query execution engine can also leverage information about current queue lengths and the utilization of various resources on heterogeneous hardware platforms (such as current GPU or FPGA use). Such information is particularly important when executing concurrent workloads, and, hence, multiple dataflows. This, together with auxiliary information about the resource footprint and the requirements attached to the sub-operators can enable the execution engine to react better to runtime noise [22]. Similar to the *microadaptivity* technique, used by Vectorwise [72], this flexibility enables the system to adapt better to changing environments, which can result in both improved performance and better resource utilization.

## 4 Related work

The benefits of working with sub-operators (or fragmenting traditional SQL operators into smaller components) are already known to our community; it is just that we have never properly formalized them or exposed the sub-operators as an interface.

For example, Dittrich et al. [19] propose splitting relational operators, like joins, into smaller fragments that allow finer, more granular performance tuning in the optimizer. Voila [24] uses a custom IR to chart the design space between vectorization and compilation, CVM [52] outlines how to build a common infrastructure, while Voodoo [64] shows that we can use an intermediate IR of database kernels to generate more efficient parallel executables for a variety of hardware platforms. Unsurprisingly, the ideas are also explored in other contexts. For instance, Love et al. [20] identify the most common *shuffle kernels* that can be used as building blocks for various graph algorithms. It is also an attractive approach for engines that support cross-platform execution. He et al. based their design of a hybrid CPU/GPU co-processing system GDB [26] on a set of data-parallel *primitives*, later used to implement common SQL operators. PyWren further demonstrates the elasticity and simplicity of serverless lambda functions as building blocks for maps [30]. Prior work also explored alternative methods of offloading parts of the operator computation onto a co-processor or accelerator [31, 63]. From an optimization perspective, our proposal shares a lot of challenges with workflow management systems that build dataflows from sub-operators that are backed by different variations, even though our focus is much closer to the hardware.

Novel framework proposals from the compiler community, like MLIR [43], LLHD [75] and TVM [12], outline a more generic approach of lowering dataflow systems in a multi-level process of graph transformation and optimization through different granularities of intermediate representations before generating executables for various hardware targets, including accelerators.

Regarding the domain of hardware specialization, we have already referred to the DAX engines introduced in Oracle's SPARC M7 [62] and refined in M8. Other examples are Google's TPU, the Q100 data processing unit [87], the energy-efficient hardware partitioner [86], and Baidu's data-processing accelerators [58, 60]. Reconfigurable hardware, like FPGAs, is an immediate choice for exploring operation offloading down to hardware. However, coarse-grained reconfigurable architectures (CGRA) [67, 81] are even closer to our concept of sub-operators. They are not only faster to reprogram, but work at a coarser granularity with so-called parallel hardware primitives, which are powerful enough to express a variety of dataflows [66]. Templated-based FPGA framework designs customized for data-processing [45] are also worth considering.

We would like to emphasize that we propose building a language runtime using sub-operators as an ISA rather than locally extending the database to run programs, as was the case with stored procedures. An ISA, allowing execution of complex dataflows composed of sub-operators, can be used to augment existing efforts that revisit the interface between applications and databases [13]. In particular, if extended, the QBS [14] optimizer can use sub-operators to execute the application converted code beyond SQL-only queries.

## 5 Conclusion

In this paper, we present the benefits of lowering the explicit level of abstraction on which database engines traditionally operate by making *sub-operators* first-class entities. This enables a database engine to overcome the current limitations of the relational model and SQL and serves as a language runtime that executes an ISA of sub-operators. Such a change makes database engines flexible platforms, which execute various complex dataflows from a range of applications, so that they benefit from existing database technologies and non-functional properties, like consistency or recoverability.

Looking ahead, we believe that the proposed sub-operators are especially attractive in the context of today's trends towards increased hardware specialization and resource disaggregation. In addition to accelerators, pushing compute functions either down to where the data sits (in smart storage) or as the data moves over the network (via smart NICs) is a promising way to address the widening gap of data deluge and the bandwidth capacities of today's hardware. Thinking in terms of sub-operators is an elegant and intuitive way of efficiently approaching the problem of executing dataflow pipelines in such deployment environments.

## References


[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015.

[2] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. A Scalable Processing-in-memory Accelerator for Parallel Graph Processing. In *ISCA '15*, pages 105–117.

[3] K. Aingaran et al. M7: Oracle's next-generation sparc processor. *IEEE Micro*, 35(2):36–45, 2015.

[4] Apache Software Foundation. Parquet — Columnar storage for the people. https://parquet.apache.org, 2013.

[5] Apache Software Foundation. Arrow — A cross-language development platform for in-memory data. https://arrow.apache.org, 2019.

[6] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. Orthogonal Security with Cipherbase. In *CIDR 2013*.

[7] M. Bandle, J. Giceva, and T. Neumann. To partition, or not to partition, that is the join question in a real system. In *SIGMOD '21*.

[8] S. K. Begley, Z. He, and Y. P. Chen. Mcjoin: a memory-constrained join for column-store main-memory databases. In *SIGMOD'12*, 2012.

[9] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the memory wall in monetdb. *Commun. ACM*, 2008.

[10] G. Candea, N. Polyzotis, and R. Vingralek. A scalable, predictable join operator for highly concurrent data warehouses. *PVLDB '09*, 2(1):277–288.

[11] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun. A Domain-specific Approach to Heterogeneous Parallelism. In *PPoPP '11*, pages 35–46.

[12] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Q. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy. TVM: an automated end-to-end optimizing compiler for deep learning. In *USENIX '18*, 2018.

[13] A. Cheung. *Rethinking the Application-Database Interface*. PhD thesis, Massachusetts Institute of Technology, Cambridge, USA, 2015.

[14] A. Cheung, A. Solar-Lezama, and S. Madden. Optimizing Database-backed Applications with Query Synthesis. In *PLDI '13*, pages 3–14.

[15] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, C. Binnig, U. Çetintemel, and S. Zdonik. An architecture for compiling udf-centric workflows. *Proc. VLDB Endow.*, 8(12):1466–1477, 2015.

[16] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, U. Çetintemel, and S. B. Zdonik. Tupleware: "Big" Data, Big Analytics, Small Clusters. In *CIDR 2015*.

[17] C. Diaconu, C. Freedman, E. Ismert, P. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL server's memory-optimized OLTP engine. In *SIGMOD '13*.

[18] Y. Diao, D. Florescu, D. Kossmann, M. J. Carey, and M. J. Franklin. Implementing Memoization in a Streaming XQuery Processor. In *XSym'04*, pages 35–50.

[19] J. Dittrich and J. Nix. The Case for Deep Query Optimisation. In *CIDR 2020*.

[20] Eric Love. Ressort: An Auto-Tuning Framework for Parallel Shuffle Kernels. Master's thesis, University of California, Berkeley, Berkeley, California, USA, 2016.

[21] P. Fent and T. Neumann. A practical approach to groupjoin and nested aggregates. *VLDB '21*.

[22] J. Giceva, G. Alonso, T. Roscoe, and T. Harris. Deployment of Query Plans on Multicores. *PVLDB '14*, 8(3):233–244, 2014.

[23] J. Gjengset, M. Schwarzkopf, J. Behrens, L. T. Araújo, M. Ek, E. Kohler, M. F. Kaashoek, and R. T. Morris. Noria: dynamic, partially-stateful data-flow for high-performance web applications. In *OSDI*, 2018.

[24] T. Gubner and P. Boncz. Charting the design space of query execution using voila. In *VLDB '21*.

[25] P. Harish and P. J. Narayanan. *Accelerating Large Graph Algorithms on the GPU Using CUDA*, pages 197–208. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[26] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational Query Coprocessing on Graphics Processors. *ACM Trans. Database Syst.*, 34(4):21:1–21:39.

[27] T. Heinis and G. Alonso. Efficient lineage tracking for scientific workflows. In *SIGMOD '08*, pages 1007–1018.

[28] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-Marl: A DSL for easy and efficient graph analysis. In *ASPLOS*, pages 349–362, 2012.

[29] Z. Istvan, L. Woods, and G. Alonso. Histograms As a Side Effect of Data Movement for Big Data. In *SIGMOD '14*, pages 1567–1578.

[30] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht. Occupy the cloud: Distributed computing for the 99In *SoCC*, 2017.

[31] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk. GPU Join Processing Revisited. In *DaMoN '12*, pages 55–62.

[32] K. Kara, J. Giceva, and G. Alonso. FPGA-based Data Partitioning. In *SIGMOD '17*.

[33] K. Karanasos, M. Interlandi, F. Psallidas, R. Sen, K. Park, I. Popivanov, D. Xin, S. Nakandala, S. Krishnan, M. Weimer, Y. Yu, R. Ramakrishnan, and C. Curino. Extending relational query processing with ML inference. In *CIDR 2020*.

[34] A. Kemper, D. Kossmann, and C. Wiesner. Generalised Hash Teams for Join and Group-by. In *VLDB '99*, pages 30–41.

[35] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*, 2011.

[36] T. Kersten, V. Leis, and T. Neumann. Tidy tuples and flying start: Fast compilation and fast execution of relational queries in umbra. *Proc. VLDB Endow.*, 2021.

[37] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey. Sort vs. Hash revisited: fast join implementation on modern multi-core CPUs. *PVLDB '09*, 2(2):1378–1389, 2009.

[38] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi. Building Efficient Query Engines in a High-level Language. *PVLDB*, 7(10):853–864.

[39] A. Kohn, V. Leis, and T. Neumann. Building Advanced SQL Analytics From Low-Level Plan Operators. In *SIGMOD'21*.

[40] D. Koutsoukos, I. Müller, R. Marroquín, and G. Alonso. Modularis: Modular data analytics for hardware, software, and platform heterogeneity, 2020.

[41] K. Krikellas, S. D. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *ICDE'10*, pages 613–624.

[42] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO '04*, pages 75–, 2004.

[43] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. MLIR: A Compiler Infrastructure for the End of Moore's Law, 2020.

[44] V. Leis, P. Boncz, A. Kemper, and T. Neumann. Morsel-driven Parallelism: A NUMA-aware Query Evaluation Framework for the Many-core Age. In *SIGMOD'14*, pages 743–754, 2014.

[45] D. Mahajan, J. Park, E. Amaro, H. Sharma, A. Yazdanbakhsh, J. K. Kim, and H. Esmaeilzadeh. TABLA: A unified template-based framework for accelerating statistical machine learning. In *HPCA '16*, 2016.

[46] S. Manegold, P. Boncz, and M. L. Kersten. Generic Database Cost Models for Hierarchical Memory Systems. In *VLDB '02*, pages 191–202.

[47] R. Marroquín, I. Müller, D. Makreshanski, and G. Alonso. Pay one, get hundreds for free: Reducing cloud costs through shared query execution. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 439–450, 2018.

[48] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard. Differential Dataflow. In *CIDR 2013*.

[49] E. Meijer, B. Beckman, and G. Bierman. LINQ: Reconciling Object, Relations and XML in the .NET Framework. In *SIGMOD '06*, pages 706–706.

[50] P. Menon, A. Pavlo, and T. C. Mowry. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *Proc. VLDB Endow.*, 11(1):1–13, 2017.

[51] G. Moerkotte and T. Neumann. Accelerating queries with group-by and join by groupjoin. *Proc. VLDB Endow.*, 4(11):843–851, 2011.

[52] I. Müller, R. Marroquin, D. Koutsoukos, M. Wawrzoniak, S. Akhadov, and G. Alonso. The collection virtual machine: an abstraction for multi-frontend multi-backend data analysis. In *DaMoN*, pages 7:1–7:10. ACM, 2020.

[53] T. Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB '11*, 4(9):539–550.

[54] T. Neumann and M. J. Freitag. Umbra: A disk-based system with in-memory performance. In *CIDR 2020*. www.cidrdb.org.

[55] T. Neumann and A. Kemper. Unnesting arbitrary queries. In *BTW 2015*.

[56] T. Neumann, V. Leis, and A. Kemper. The complete story of joins (in hyper). In *BTW 2017*.

[57] NGD Systems. Newport Platform. https://www.ngdsystems.com/, 2021.

[58] Nicole Hemsoth. An Early Look at Baidu's Custom AI and Analytics Processor. https://www.nextplatform.com/2017/08/22/first-look-baidus-custom-ai-analytics-processor/, 2017.

[59] E. Nurvitadhi, G. Weisz, Y. Wang, S. Hurkat, M. Nguyen, J. C. Hoe, J. F. Martínez, and C. Guestrin. GraphGen: An FPGA Framework for Vertex-Centric Graph Computation. In *FCCM '14*, pages 25–28.

[60] J. Ouyang, W. Qi, Y. Wang, YichenTu, J. Wang, and B. Jia. SDA: software-defined accelerator for general-purpose big data analysis system. In *Hot Chips*, 2016.

[61] S. Palkar, J. J. Thomas, A. Shanbhag, M. Schwarzkopf, S. P. Amarasinghe, and M. Zaharia. A common runtime for high performance data analysis. In *CIDR 2017*.

[62] S. Phillips. M7: Next Generation SPARC. Presented at Hot Chips (HC 26): A symposium on High Performance Chips, August, 2014.

[63] H. Pirk, S. Manegold, and M. Kersten. Waste not... efficient co-processing of relational data. In *IEEE International Conference on Data Engineering*, 2014.

[64] H. Pirk, O. R. Moll, M. Zaharia, and S. Madden. Voodoo - A vector algebra for portable database performance on modern hardware. *VLDB'16*, 2016.

[65] O. Polychroniou and K. A. Ross. A Comprehensive Study of Main-memory Partitioning and Its Application to Large-scale Comparison- and Radix-sort. In *SIGMOD' 14*, pages 755–766, 2014.

[66] R. Prabhakar, D. Koeplinger, K. J. Brown, H. Lee, C. D. Sa, C. Kozyrakis, and K. Olukotun. Generating configurable hardware from parallel patterns. In *ASPLOS'16*, 2016.

[67] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun. Plasticine: A reconfigurable architecture for parallel paterns. In *ISCA'17*, 2017.

[68] A. Putnam et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *ISCA*, pages 13–24, 2014.

[69] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. P. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *PLDI '13*.

[70] W. Rödiger, T. Mühlbauer, A. Kemper, and T. Neumann. High-speed Query Processing over High-speed Networks. *PVLDB '15*, 9(4):228–239.

[71] T. Rompf and M. Odersky. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *GPCE '10*, pages 127–136.

[72] B. Răducanu, P. Boncz, and M. Zukowski. Micro Adaptivity in Vectorwise. In *SIGMOD '13*, pages 1231–1242.

[73] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In *SIGMOD*, pages 351–362, 2010.

[74] ScaleFlux. An Early Look at Baidu's Custom AI and Analytics Processor. https://www.scaleflux.com/, 2021.

[75] F. Schuiki, A. Kurth, T. Grosser, and L. Benini. LLHD: a multi-level intermediate representation for hardware description languages. In *SIGPLAN '20*, 2020.

[76] V. Seshadri, T. Mullins, A. Boroumand, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. Gather-scatter DRAM: In-DRAM Address Translation to Improve the Spatial Locality of Non-unit Strided Accesses. In *MICRO'15*, pages 267–280.

[77] M. Singh and B. Leonhardi. Introduction to the ibm netezza warehouse appliance. In *CASCON'11*.

[78] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *IEEE Des. Test*, 12(3):66–73.

[79] J. Teubner and R. Mueller. How Soccer Players Would Do Stream Joins. In *SIGMOD '11*, pages 625–636.

[80] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann. Predictable performance for unpredictable workloads. *PVLDB '09*, 2(1):706–717.

[81] M. Vilim, A. Rucker, Y. Zhang, S. Liu, and K. Olukotun. Gorgon: Accelerating machine learning from relational data. In *ISCA'20*, 2020.

[82] Z. Wang, B. He, and W. Zhang. A study of data partitioning on OpenCL-based FPGAs. In *FPL*, 2015.

[83] J. Wassenberg and P. Sanders. Engineering a multi-core radix sort. In *Euro-Par'11*, pages 160–169. Springer, 2011.

[84] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2009.

[85] L. Woods, Z. István, and G. Alonso. Ibex: An Intelligent Storage Engine with Support for Advanced SQL Offloading. *PVLDB'14*, 7(11):963–974.

[86] L. Wu, R. J. Barker, M. A. Kim, and K. A. Ross. Navigating Big Data with High-throughput, Energy-efficient Data Partitioning. In *ISCA '13*, pages 249–260.

[87] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross. Q100: The Architecture and Design of a Database Processing Unit. In *ASPLOS '14*, pages 255–268.

[88] C. Zhang and C. Ré. Dimmwitted: A study of main-memory statistical analytics. *PVLDB*, 7(12):1283–1294.

[89] M. Zukowski, S. Héman, N. Nes, and P. Boncz. Cooperative scans: dynamic bandwidth sharing in a DBMS. In *VLDB '07*, pages 723–734.