Waste Not...
Efficient Co-Processing of Relational Data

Holger Pirk
CWI
Amsterdam, The Netherlands
holger@cwi.nl

Stefan Manegold
CWI
Amsterdam, The Netherlands
manegold@cwi.nl

Martin Kersten
CWI
Amsterdam, The Netherlands
mk@cwi.nl

***Abstract*—The variety of memory devices in modern computer systems holds opportunities as well as challenges for data management systems. In particular, the exploitation of Graphics Processing Units (GPUs) and their fast memory has been studied quite intensively. However, current approaches treat GPUs as systems in their own right and fail to provide a generic strategy for efficient CPU/GPU cooperation. We propose such a strategy for relational query processing: calculating an *approximate* result based on lossily compressed, GPU-resident data and *refine* the result using residuals, i.e., the lost data, on the CPU. We developed the required algorithms, implemented the strategy in an existing DBMS and found up to 8 times performance improvement, even for datasets larger than the available GPU memory.**

I. Introduction

Modern computer systems research has yielded a number of different memory technologies with different characteristics. Virtually all of these technologies face a fundamental trade-off between capacity and performance: the higher the storage capacity of a device, the higher its latency and the lower its bandwidth. These tradeoffs are mainly economic (larger, faster memory is simply more expensive to build) but also partially systemic: larger memories are inherently harder to address. Resolving an integer address to a physical location on a chip or disk involves a decoding effort proportional to the length of the address [1]. Figure 1 shows that even devices of the same type (in this case flash memory) face a systemic conflict between storage capacity and performance[1] [2]. To resolve this conflict, most systems combine multiple devices into a memory hierarchy that speeds up localized access. *To achieve maximal performance, all of the available devices have to be used optimally.* Consequently, making efficient use of hierarchical memories for relational data processing is one of the fundamental challenges in data management research. Fields such as cache-conscious algorithms, out-of-memory processing and distributed data management strive to extract maximal performance from the respective memory hierarchy at the expense of an ever-increasing number of techniques, technologies and tuning opportunities.

In this paper, we set out with a different goal:

> *Rather than achieving maximum performance for a specific problem, we want to achieve good performance for a wide range of (relational) operations in the presence of hierarchical memories and large data sets.*

As an exemplary case, we tackle an instance of this problem that has recently received significant attention: the efficient use of fast but small memory of Graphics Processing Units (GPUs) in addition to the large but slow CPU-attached memory[2].

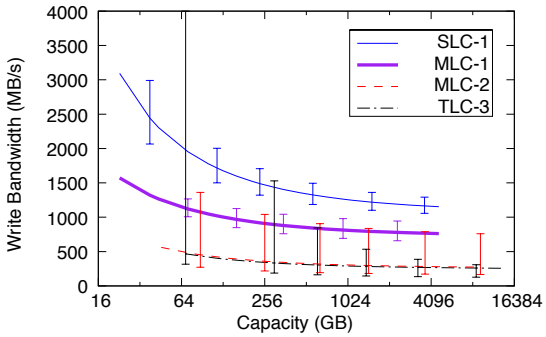


Fig. 1: Flash Memory Capacity/Bandwidth [2] (layout adjusted)

Due to their high bandwidth and computation power, GPUs have been employed to speed up the processing of small, i.e., mostly GPU-resident sets of relational data [3], [4], [5], [6]. Most of this work focuses on the efficient implementation of classic relational operators on GPUs and the selection of GPU or CPU operators at runtime. In general, the GPU operators transfer their inputs through the slow PCI-E bus, process them, transfer the results back and potentially cache data for later reuse. While vendors simplified CPU/GPU transfers through, e.g., "unified virtual addressing", little can be done about the PCI-E bottleneck (which is merely an instance of the well-known "Von Neumann" bottleneck). Therefore, such approaches only achieve good performance if the (hot) dataset fits in the GPUs internal memory. This is generally not the case [7].

In earlier work [8], we proposed *Bitwise Distribution (BWD)*, a simple, generic storage model that helped us to achieve good performance for a prototypical, GPU-supported data management application. While bitwise distributed storage helped improve the performance of a hard-coded prototype, it lacked an appropriate processing model to support the full range of relational queries in a real system. In this paper, we present such a processing model and demonstrate its suitability for generic relational query processing.

To this end, we make the following contributions:

- We propose the novel Approximate & Refine (A&R) processing paradigm for the management of bitwise distributed relational data.

[1] In Big Data terms: there is a conflict between data Volume and Velocity

[2] we discuss other applications for the approach in Section VII-B

- We develop efficient algorithms that form the basis for the A&R operators on bitwise distributed data.
- We present and evaluate our implementation of the paradigm in the relational Database Management System (DBMS) MonetDB [9].

The rest of this paper is structured as follows: in Section II, we introduce the basis of our work: the bitwise decomposed storage model and the bulk processing model. The Sections III, IV and V give a top-down tour through our system: In Section III we present the conceptual framework that is the foundation for our approach to the efficient processing of bitwise distributed data. In Section IV, we describe the unique algorithmic challenges of our processing paradigm and the algorithms that solve them. We present our implementation of the paradigm in MonetDB in Section V and evaluate it in Section VI. In Section VII we compare our approach to similar ideas, explore future work and conclude in Section VIII.

II. Background

Before introducing the A&R query processing paradigm, let us briefly recount the work on the underlying storage and the processing model.

A. Bitwise Distributed Storage

To distribute persistent data across the available devices, we build on the idea of Bitwise Decomposition/Distribution (BWD) [8]. Figure 2 illustrates the bitwise distributed storage model. Data values of a column are vertically partitioned at the granularity of individual bits and (non-redundantly) distributed among the memories of the available devices (leading zeros are removed). The partition with the major bits effectively represents an approximation of the full value. This approximation can be stored and processed in the fast memory, in isolation of the residual (i.e., the minor bits). Since the data size of the approximation scales with its resolution (the number of bits in the approximation), it can be adapted to the storage capacity of the respective device. When necessary, the approximation can be joined with the residual on the tuple id to reconstruct the precise values. Similar to different materialization strategies of attributes in column-stores [10], bitwise decomposition allows different reconstruction strategies (early, late, cost-based).

B. Bulk Processing

In this work, we focus on Memory-Resident DBMSs (MRDBMSs) rather than disk-based systems. Due to the need for CPU efficiency, most MRDBMS are implementations of the bulk processing model (or a variant thereof) [11], [12], [13] on fully decomposed data [14]. In this processing model, operators are simple, tight loops without function calls that physically materialize their results in arrays for the next operator to pick up. This leads to high computational efficiency at the cost of expensive intermediate materialization. While Vectorized [12] and Just-in-Time compiled [13] query processing address the problem of expensive materialization, they follow the same principle: increasing CPU efficiency by eliminating function calls from the critical execution path.

We decided to build upon MonetDB's plain bulk-processing model (we discuss the approach's applicability in other setups in Section VII-B). This model has two main advantages: a) It avoids the problem of explicit buffer management as needed by, e.g., the vectorized model. Such buffer management is non-trivial in a multi-device setup and b) it allows the evaluation of complex queries using a relatively small set of operators. We use the existing MonetDB query compiler [9] to break down complex queries to our new A&R operators (see Section V-B for the query compilation process).

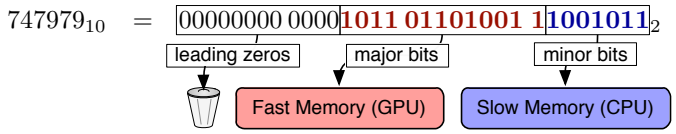


Fig. 2: Bitwise Decomposition & Distribution

III. Approximate & Refine Plans

By their very definition, relational DBMS follow the idea that all data, persistent as well as intermediate, is represented in self-contained relations. Many properties of Codd's relational model [15], e.g., tree shaped plans or the notion of a single implementation for each operator, are relaxed in many systems. The notion of a unified data representation, however, is rarely challenged, because it allows the free mixing and matching of operators in the plan generation phase. Unfortunately, this flexibility at plan generation time comes at a cost in later processing phases, in particular when targeting multiple devices:

1) Operators have to invest effort into converting data from the unified representation into an appropriate internal representation and back. A GPU-operator, e.g., has to ship data to and from the device which might be in vain if the next operator needs the data on the GPU as well.
2) It limits optimization opportunities due to the coarse granularity of operators. Operators can, e.g., not share intermediate/internal data structures like hash-tables.
3) It hides the cross-device parallelization of operations from the execution scheduler, which limits cross-device execution to intra-operator parallelism.

Rather than fight these symptoms individually, we tackle the problem at the root: the relational algebra processing model itself. We propose a processing model that is not based on a unified representation of data: the A&R model. Instead of the classic relational operators, there are two classes of operators. These produce fundamentally different kinds of outputs (see Figure 3): approximation operators (marked in red) that produce a candidate result and refinement operators (blue) that combine such candidate results with additional/residual data to produce a correct result set. Each classic relational operator can be modeled using one approximation and one (or more) subsequent refinement operators. This division of the relational operators has a number of advantages:

1) It simplifies operators implementation in multi-device systems since each operator targets only one device,
2) creates additional opportunities for optimization at plan level (see Subsection III-A),
3) allows the independent scheduling of operations on the different devices at runtime,

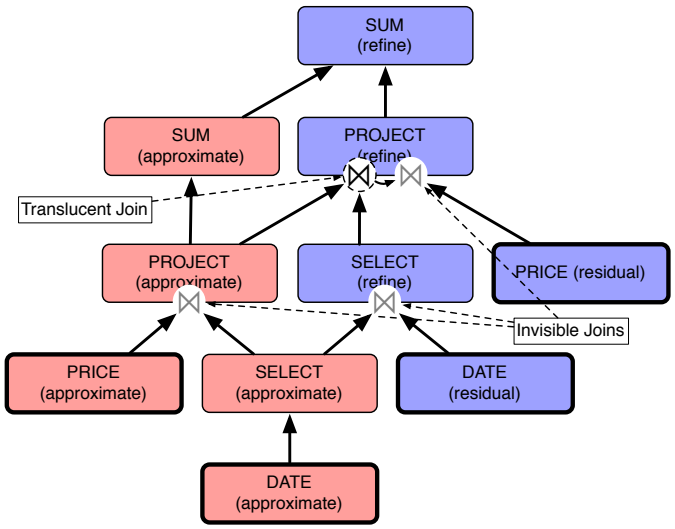


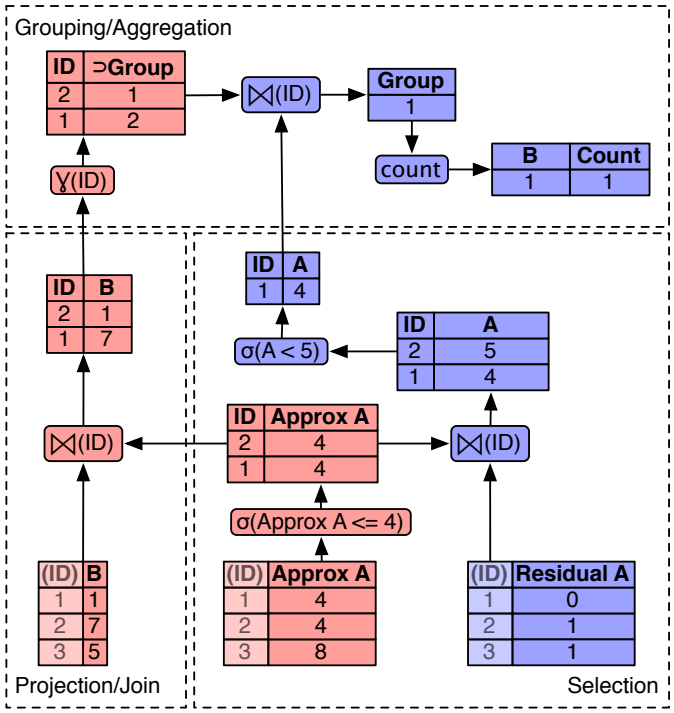
Fig. 3: A Relational Approximate & Refine Query Plan for
`select sum(price) from lineitem where shipdate > $1`

4) allows the fast computation of an approximate query answer without wasting resources by evaluating only the approximation subplan.

To achieve these benefits, however, we need to develop these new classes of operators for each of the classic relational operators. Before describing their implementation in Section IV, we provide a brief overview of the two operator classes.

Approximation: The goal of an approximation operator is to provide a fast approximation of the result of its classic relational counterpart based on approximate inputs. For structural/relational operators like selections and joins, this means that it should provide a superset of the refined/actual output relation. The condition predicates (if any) are relaxed in order to produce an over-approximation of the accurate result. For arithmetic or string operations on tuple values, this means that it yields the expected value and strict error bounds of the result based on the approximate inputs. The implementation of the approximation operators is principally the same as the one of the classic relational operators. The major difference is that the arithmetic operators have to propagate the error bounds as a part of the approximation, so later operators can relax predicate conditions appropriately. Naturally, the operators have to be implemented for the targeted device. Thus, when targeting a GPU, the operators are executed on a massively parallel platform and should, thus, be implemented accordingly.

Refinement: The refinement operators are fundamentally different from their classic relational equivalents. Where a relational operator accepts one or two inputs, a refinement operator accepts an approximation and a refinement input for each operand and an approximation input from the refinement's respective approximation operator. Many (not all) relational operators receive a significant head start on their execution when provided with an approximation of their output. We discuss the respective benefits and the unique challenges of each of the refinement operators in Section IV. Before,



Query: `select count(*) from R where A<5 group by B`
Schema: `create table R(A integer, B integer)`
Storage: `A`: (31 bit GPU, 1 bit CPU), `B`: (32 bit GPU)

Fig. 4: An integrated example for A&R-processing

however, let us outline the new opportunities for optimization that arise from such a multi-step query execution model.

A. (Rule-based) Optimization

While the A&R-processing model allows efficient co-processing of data, it also introduces new degrees of freedom in plan generation & optimization: approximation operators on one attribute can be pushed below refinement operators on another in order to speed up query execution. In general, we assume the approximations to yield a significant reduction of the result-set at (relatively) low costs. It is, therefore, sensible to push down approximate selections as far as possible. This optimization scheme can be extended with an appropriate cost model in the future, but provides a good baseline method for future work. Since the details are largely specific to the DBMS at hand, we briefly cover the implementation of an appropriate MonetDB query plan generator and optimizer in Section V-B.

IV. Approximate & Refine Operators

In this section we present the fundamentals of the individual A&R operators by discussing an example query that involves the relevant operations (Figure 4). Before focusing on the operators themselves, however, let us introduce one of the essential building blocks of our approach: the translucent join.

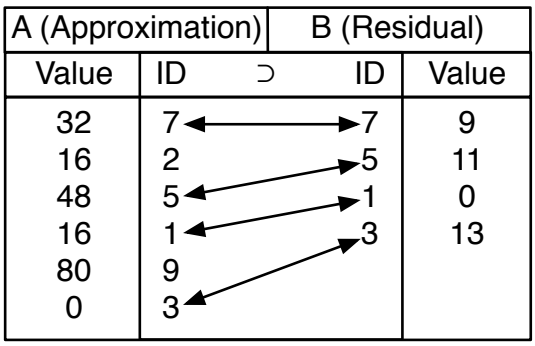


Fig. 5: A case for the translucent join

A. The Translucent Join

Query execution on fully decomposed (column-store) data is known to involve a large number of joins to reconstruct tuples from decomposed columns [10]. As an example, consider the case of a simple selective projection (the blue nodes in Figure 3). In a (late materializing) column-store, the query processor will first perform the selection (in this case on `DATE`). After that, the resulting tuple IDs will be joined with the projected attribute (`PRICE`) and aggregated. While this yields a high number of joins in the plan, these joins are *invisible joins* [10], i.e., mere positional lookups, if the physical location of a value can be easily derived from the tuple id.

Since Bitwise Distribution (BWD) decomposes relations even further than mere decomposition at attribute granularity, the number of joins increases accordingly. Almost every refinement operator involves the join of the (over)approximation with the residual. The full plan in Figure 3, e.g., contains four joins. It is, thus, important to efficiently support this operation. While the joins with persistent residual are cheap, invisible joins, the join of the refined selection and the approximate is not. It is, however, not a generic join either, because some helpful properties of the input relations are known to the operator at runtime:

1) The tuple IDs that are returned by the `SELECT (refine)` are a subset of the tuple IDs that are part of the `PROJECT (approximate)`.
2) The underlying `SELECT (refine)` and `PROJECT (approximate)` operators are order-preserving: `SELECT (refine)` because it is a non-parallelized operation and `PROJECT (approximate)` because a parallel projection writes values at the same positions as the input ids
3) The `SELECT (approximate)`, however, is not order-preserving because a massively parallelized selection can only maintain the input order at additional costs, which we want to avoid.

Due to item 3, we cannot assume that the inputs are ordered. Due to item 2, however, we know that the two inputs have the same permutation and one is a subset of the other (item 1). For this scenario (see Figure 5 for an example), we developed a special kind of merge-join: Following the naming convention of the *invisible join*, we refer to it as the *translucent join* (it is more complex and costly than an invisible join but not as much as a generic equi-join). This algorithm can be applied to perform a natural join of two (enumerated) relations A and B on attribute id under the following conditions[3]:

1) $\{A.id\}$ and $\{B.id\}$ are unique
2) $\{A.id\}$ is a superset of $\{B.id\}$[4] and
3) the elements of $B.id$ that are present in $A.id$ have the same permutation in $A.id$ as in $B.id$, i.e., $(x \in B.id \wedge y \in B.id \wedge i_{A.id}(x) < i_{A.id}(y)) \Rightarrow i_{B.id}(x) < i_{B.id}(y)$ with $i_S(x)$ the number/position of x in set S

```
function TRANSLUCENTLYJOIN(A, R)                    ▷ (on id)
    if SORTED(A.id) ∧ DENSE(A.id) then
        return INVISIBLYJOIN(A, R)
    else
        i_R ← 0, i_A ← 0, C ← ∅
        while i_R < ‖R‖ do
            if R.id[i_R] = A.id[i_R] then
                C[i_R] ← (R[i_R], A[i_A])
                i_R ← i_R + 1
                i_A ← i_A + 1
            else
                i_A ← i_A + 1
            end if
        end while
        return C
    end if
end function
```

Algorithm 1: The Translucent Join

The algorithm, displayed in Algorithm 1, processes data similar to a sort-merge join but does not rely on the sortedness of the values to decide which cursor to advance. Instead, it always advances the cursor on A until a match with the element at the cursor on B is found. In that case, both cursors are advanced. Due to the conditions 2 and 1, all values in $B.id$ find exactly one join partner in $A.id$. It is, therefore, enough to iterate through $B.id$ and find *the* matching value in $A.id$. Due to condition 3, the join partner cannot occur in a position before the current position of the cursor on $A.id$. Consequently, the cursor on $A.id$ only needs to be advanced to where the partner is found. Thus, the algorithm yields correct results in $O(|A.id| + |B.id|)$ memory accesses and $O(|A.id|)$ comparisons under the stated conditions.

Conditions 1 and 2 always hold when an approximation is joined with its residual. Condition 3 has to be established by making sure the tuple order is not changed between the approximation and the refinement. We ensure this by generating and optimizing plans such that no order-changing operator (selections) appear between approximation and refinement.

B. Selections

Selections are the most likely candidate to benefit from cross device processing: Since selections are the most input-bandwidth hungry operators and generally yield a significant

[3] in this example, A contains the approximation and B the residuals

[4] Note that this, together with the uniqueness of $\{A.id\}$ is equivalent to $\{B.id\}$ being a Foreign-Key set to $\{A.id\}$

reduction of the input, they are very well suited for a GPU environment where input bandwidth (device memory) is available in abundance and output bandwidth (PCI-E bus) is scarce.

The approximation of a selection on a single attribute A is straightforward: the conditions on the accurate value are relaxed to match all values that have the same approximation as a matching value and the input data is scanned with the relaxed predicates. To this end, each selection operand x is adapted according to the following function f:

$$f(x) = \begin{cases} appr(x) & \text{if op is '== x'} \\ appr(x) - 1 & \text{if op is '> x'} \\ appr(x) & \text{if op is '>= x'} \\ appr(x) + (1 \ll resbits) + 1 & \text{if op is '< x'} \\ appr(x) + (1 \ll resbits) & \text{if op is '<= x'} \end{cases}$$

With $appr(x)$ the approximation of x according to the decomposition selected for attribute A (bitmasking the value with the bitwise compliment of $(1 \ll resbits)-1$) and $resbits$ the number of bits used for the residuals. This yields a superset of the precise result set of the selection (see Figure 4).

The refinement of a selection is a little bit more involved (see Algorithm 2 or the Selection section in Figure 4 for a conceptual overview). As a first step, the approximation of the result is (translucently) joined with its residual. In the second step, the accurate values are reconstructed by a bitwise concatenation of the approximation and the residuals. Using accurate values, the (precise) condition is reevaluated and false positives are eliminated. In practice, the two operations (the translucent join and the re-evaluation of the condition) can be performed in one loop which eliminates the need for multiple iterations through the data. As illustrated in Figure 4, the refined result of the selection is correct (i.e., the predicate holds for all resulting tuples).

For complex selections as well as other operations that involve, e.g., arithmetic functions, the bulk-processing model advocates breaking down the predicate into multiple primitives that are evaluated using bulk-operators. The result of each of these operators is materialized and used as input for the next primitive operator or the selection operator itself. Using the same technique, we can support arithmetic functions as long as an approximate result (with error bounds) can be derived from approximate operands (with error bounds). This holds for basic arithmetic functions (add, subtract, multiply, divide) as well as some more complex functions (sqrt, power). If a user defined function can fulfill these properties, it also can be supported by our approach.

C. Projections

In late materializing column-stores, projective joins are used to add columns to the result set. As observed in previous work [10], these projections are usually implemented using positional lookups/invisible joins.

The approximation of a projection is implemented as an invisible join/positional lookup of the overapproximated position set and the approximated target values. Its implementation is, thus, straightforward. If all bits of the projected attribute are GPU resident (as they are in the example in Figure 4, Projection/Join section), the resulting relation does not have to be refined. If some bits are CPU-resident, they have to be joined with the approximation to reconstruct the accurate values.

```
function SELECTREFINE(A, R)
    C ← TRANSLUCENTLYJOIN(A, R)
    C_refined ← ∅, i ← 0
    for cand in C do
        if CONDITION(cand.appr +^bw cand.res) then
            C_refined[i] ← o.appr +^bw o.res
            i ← i + 1
        end if
    end for
    return C_refined
end function

+^bw = bitwise concatenation
```

Algorithm 2: Refining a selection

The refinement of a projection is essentially a translucent (potentially invisible) join of the output of the approximation and the residual of the input. This ensures that the residual and the approximation stay aligned. In essence, the refinement step of a projection is equivalent to a selection refinement without a predicate.

D. Joins

Since joins are among the most common yet expensive relational operations, there is much incentive to support them using GPUs. However, they are also among the operators that are most difficult to implement on a GPU. The massively parallel architecture, which is the basis for the superior computational performance of GPUs becomes a curse when processing a non-indexed, generic[5] equijoin: the performance bottleneck in this case is the hash-building phase of the hash-join. The massively parallel construction of a hash-table involves many scattered, conflicting writes into the shared memory which are generally resolved using (partial) locking of the table on insert [16], [17]. While locking is a viable approach if the PCI-E bus is the effective bottleneck [17], it seriously limits performance when reading data from the internal memory.

The efficient approximation of a join on a GPU is particularly difficult: since joins on approximate data expectedly yield many conflicts during hash-build and many hits during probing, the performance of equi-joins on approximate data is unclear.

Theta joins, however, are generally implemented as nested loop joins which a) are generally very bandwidth intensive, b) often subject to computation intensive comparison functions and c) trivial to (massively) parallelize because they do not employ intermediate structures that have to be locked. This makes them a very good candidate for GPU-supported processing.

The refinement of such a join is not trivial either. Since the approximation can only preserve the order of one of the joined attributes, only one of the refinement-joins can be performed using a translucent join. The other column has to

[5] i.e., not invisible or translucent/merge

be joined using, e.g., a hash-join. However the approximation can transform a potential nested loop join into a hash join with the accompanying benefits.

Due to the discussed complexity of the problem of generic GPU join processing, we do not advance the state of the art [5], [18] for this particular problem in this paper. In our implementation, we resort to (pre-)building a hashtable on the CPU in the form of a foreign-key index and leave support for unindexed joins on the GPU for future work. Such foreign-key joins are among the most common joins in analytical applications since they connect fact and dimension tables in multidimensional (star-schema) as well as relational OLAP. With a pre-built hashtable, a foreign-key join is equivalent to a projective join (Projection/Joins section in Figure 4). In our implementation, they share the same code.

E. Grouping

Standalone grouping, i.e., the mere assignment of group IDs to tuples, does not reduce the number of result tuples. Therefore, the benefits of the A&R processing of a grouping are noteworthy but less obvious.

The approximation group operator performs a pre-grouping of the tuples based on the approximate values. In our implementation we use hash-based grouping to assign group IDs to unique values. The output is positionally aligned with the input (see Grouping/Aggregation section in Figure 4).

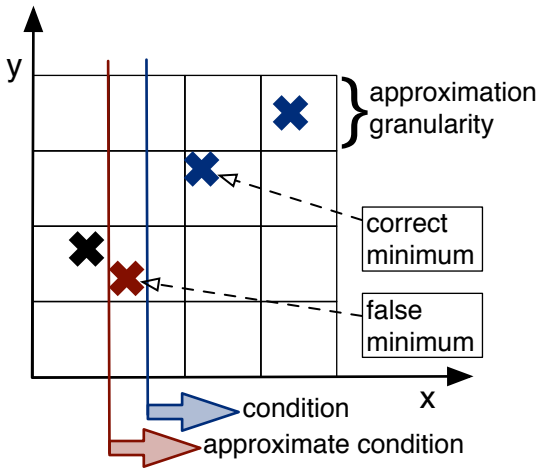
Refinement The benefits of an approximate (pre-)grouping are highly dependent on the physical representation of the grouping result. If, e.g., the tuples are physically grouped, a physical pregrouping in the GPU would localize the memory accesses when refining the grouping in the CPU which can give a significant performance benefit. In MonetDB, however, groupings are physically represented by mappings of implicit tuple IDs (i.e., positions in an array) to group IDs. In this representation, a pregrouping cannot speed up memory accesses and is therefore not used for the refinement.

However, we expect that in practice groupings on attributes with very high cardinality are rare since they yield an equally high number of groups (this holds for, e.g., the TPC-H benchmark). This means that few bits are necessary to represent them which makes it possible to keep these columns GPU resident after compression, which eliminates the necessity for a subgrouping. However, the potential false positives that may still be in the result-set from earlier operators have to be eliminated. This is, again, done using a translucent join (see Grouping/Aggregation section in Figure 4).

F. Aggregation

The handling of (grouped) aggregations in the A&R framework is dependent on the aggregation function: while `count` is trivial, `min` and `max` are slightly more complex. `Sum` and `avg` are victim to a form of *destructive distributivity* (see Section IV-G) and are, therefore, evaluated on the CPU unless all data is GPU-resident.

The Approximation of a `min` or `max` operation is difficult because care has to be taken in order to make sure that the result tuple survives the approximation phase and is considered during the refinement. Since the approximation of two values




```
Precise Query:  select min(Y) from R where x>6
Approx. Query:  select min(Y) from R where x>=4
```

Fig. 6: Calculating Min/Max in the A&R framework

is not enough to decide which one is greater, the approximation of a minimum must be a set of candidates. For a global aggregation without conditions, this set contains all tuples that have the same (minimal) value. If a condition is applied before the aggregation, the case is more complicated. To illustrate this, consider the example in Figure 6. When evaluating the approximate selection on the approximate data, three tuples qualify for the condition on x, one of which is a false positive. The false positive tuple happens to be the one with the single minimal approximate value for y. Thus, it is not enough to return all values with the minimal approximation. The result of the approximation of a minimum has to *assuredly* include the tuple ID of the actual minimum. To guarantee this, the error bounds of the applied selections are propagated to the aggregation.

The refinement of a minimum is comparatively simple: a join of the candidate set with the input residuals and the calculation of the minimum.

G. Destructive Distributivity

While many relational operators can be modeled by an A&R operator pair, there are limitations of the approach. A simple example of these limitations are even basic arithmetic operations. Consider, e.g., the following multiplication of the values a and b represented as the sum of their approximation (x^{ap}) and residual (x^{re}):

$$\begin{aligned}(a^{ap}+a^{re})\cdot(b^{ap}+b^{re}) \quad = \quad & a^{ap}\cdot b^{ap}+a^{ap}\cdot b^{re} \\ & +b^{ap}\cdot a^{re}+a^{re}\cdot b^{re}\end{aligned}$$

The expansion of the product indicates that the result of these multiplications cannot be accurately derived from the product of the approximations of the input and the residuals

of the inputs only[6]. The subterm $a^{ap} \cdot b^{re}$, e.g., can only be calculated when both factors are present on the same device. For this reason, each A&R-operator has access to the approximations of the inputs. While it is usually more expensive to access the approximations of the inputs rather than the approximation of the result during refinement, it is sometimes necessary. To reduce the costs of accessing the approximation, it can be cached on the respective device.

In addition to the architectural implications (a refinement operator has to have access to the inputs of the approximation operators), this also has performance implications: since the approximation cannot be used to speed up the calculation of the exact result, why should it be calculated at all? In addition to the refinement, the approximation could be used in other approximation operators or as the result of the query. If, e.g., a query contains a condition on the product of two attributes, the approximation of the product can be used to approximate the result of the selection.

V. Implementation

In addition to theoretically developing the necessary algorithms, we implemented our A&R query processing paradigm in MonetDB, an existing relational DBMS focused on analytical workloads on memory resident data. In this section, we briefly discuss the implementation.

A. Preparation: Decomposition

Before processing data using A&R operators, it has to be decomposed and distributed to the respective processing devices. From a relational system's perspective, a bitwise decomposed attribute is similar to an index. Consequently, it has to be explicitly defined like an index by the user.

We implemented the bitwise decomposition of attributes in MonetDB as a side-effect of a (dummy) user-defined function with the appropriate parameters. The function call is wrapped in an SQL function. Thus, the query

```
select bwdecompose(A, 24) from R;
```

decomposes the 32-bit integer attribute `A` of relation `R` into 24 GPU-resident and 8 CPU-resident bits and applies a prefix-compression to the approximate data.

B. Physical Plan Generation

MonetDB uses the MonetDB Assembly Language (MAL) to describe the (physical) query plan. An initial MAL-plan is generated from the logical relational algebra plan and repeatedly rewritten by micro-optimizers. To generate a A&R plan, we developed an additional micro-optimizer that replaces classic MAL operators with pairs of A&R operators. This optimizer is added to the end of the optimizer pipeline and, thus, benefits from optimizations that happened in earlier stages. We also added a very simple rule-based optimizer that pushes approximate selections below refined selections. Figure 7 shows the graphical representation of the A&R MAL-plan for a simple select and aggregate query. The paired approximate & refine operators that make up a classic relational MonetDB operator are clearly visible. In addition, it is apparent

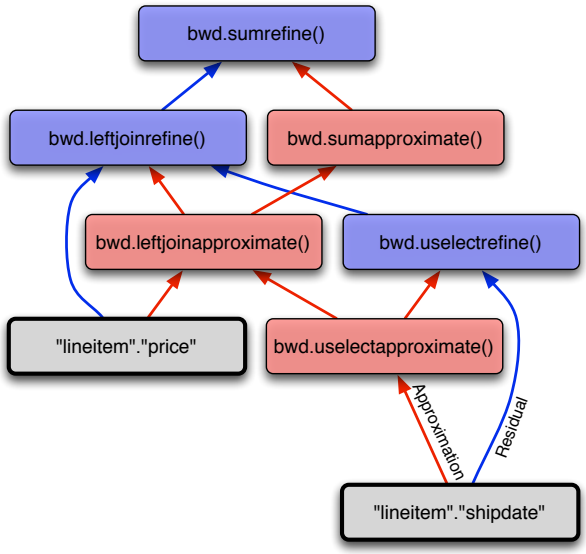


Fig. 7: A Physical MonetDB A&R Query Plan for
`select sum(price) from lineitem where shipdate > $1`

that no approximate operator depends on the result of a refine operator. Thus, the approximation subplan can be evaluated entirely yielding an approximate result before starting the first refinement operator.

C. Processing

Most column-stores use the fully decomposed representation for persistent as well as intermediate data. The individual attributes are stored in contiguous arrays with implicit (positional) or explicit (materialized) tuple IDs. In MonetDB, these arrays come in the form of Binary Association Tables (BATs). These are pairs (hence "*binary*") of arrays that map (hence "*association*") tuple IDs to attribute values. If the tuple IDs are dense (i.e., equi-distant) and sorted (e.g., in persistent attributes), they can be inferred from their position in the array and are, thus, not materialized. We use BATs to represent approximations as well as residuals and use explicit positions to keep them aligned. We extended the BAT data structures to keep pointers to their respective memory regions on the GPU, the CPU resident residuals as well as metadata like the decomposition strategy (i.e., the number and significance of bits that are resident on each device that also serves to encode the error of the approximate data) and the compression strategy (i.e., the base for the prefix compression).
We added the new A&R query processing operators to the MonetDB internal operator set as a supplementary module that has to be enabled at compile-time.

The approximation operators are native (C implemented) wrappers around OpenCL-operators. The C part is only responsible for the data structure management (locating the approximation in the GPU's internal memory, allocating resources, checking types, ...) as well as the error/resolution propagation from the inputs to the produced output. [7]

[6]Even the calculation of error bounds on the result does not allow the correct refinement of the approximation.

[7]Since the error propagation rules are relatively easy to develop and space in this paper is limited, we do not cover them here.

The data intensive part of the code is executed on the GPU. To yield efficient code, the OpenCL operator code is generated and compiled just-in-time. The code is generated using the data type, the decomposition as well as compression-strategy as parameters. We leave it to the OpenCL-compiler to do further optimization. We parallelized the operators over the number of processed tuples which generally yields a very high degree of parallelism. To keep the code portable and maintainable, we did not perform any hardware-specific tuning by, e.g., artificially reducing the parallelism, double buffering or optimization for memory bank conflicts.

The refinement operators are not generated at runtime but implemented, just like the other MonetDB operators, in C using static type expansion. The expansion is implemented using C-preprocessor macros and statically expanded by the C-compiler. This allowed us to generate efficient loops without function calls for the refinement.

We implemented both classes of operators to the best of our abilities but believe that there is potential for optimization that is orthogonal to our approach[8].

VI. Evaluation

To evaluate the approach, we used two benchmarks: the spatial range query benchmark [19] that we used to evaluate the BWD prototype [8] and a representative subset of the TPC-H benchmark. As mentioned before, we believe that with other, orthogonal optimization of individual operators, the performance can be improved further. This evaluation should, therefore, be interpreted as a baseline for the approach.

A. Setup

All experiments were conducted on a server-class system with two eight-core Intel® Xeon® E5-2650 CPUs running at 2.00 GHz. The system was equipped with 256 GB of main memory (16 modules, 16 GB each), which well exceeds the size of all used datasets. Each CPU was connected to eight memory modules through four 1.6 GHz channels. The system was equipped with two GeForce GTX 680 cards (2 GB device memory) using CUDA 4.2.1 and the device driver 304.54.

A&R implementation: We based our implementation on the MonetDB v11.11.5 (July 2012) release and developed the new A&R operators as well as the *'bwd_pipe'* optimizer pipeline which rewrites plans generated by the standard *'minimal_pipe'* optimizer pipeline into A&R-plans. Our A&R query processor implementation currently does not support the use of multiple GPUs for the processing of a single query. Therefore, we only use a single GPU when reporting query times (in Sections VI-C and VI-D) and both of the available cards with replicated datasets when reporting throughput (in Section VI-E).

CPU only implementation: As a CPU implementation, we used standard MonetDB with the *'sequential_pipe'* optimizer pipeline on pre-heated data: We report the third run of each query when showing per-operator breakdowns and averages of 15 runs for benchmarks of single operators.

GPU streaming implementation: To compare against the state of the art approach, i.e., streaming data to the GPU before processing, we would have liked to evaluate an existing system. However, we did not find a GPU supported relational DBMS that is mature enough and of sufficient quality to form a reasonable basis for such a comparison. To give some indication of the performance that can be expected of such a system, however, we report the minimal amount of work any of these systems would have to do assuming that the (hot) data size exceeds the memory capacity of the GPU: copy the input data to the GPU. To assess the costs for this operation, we measured the achievable bandwidth using the `TransferOverlap` tool that is part of AMD's Accelerated Parallel Processing (APP) SDK[9]. We measured an average bandwidth of 3.95 GB/s using DMA-transfer and calculated the transfer time from the size of the input data. In the respective charts, we indicate the time it would (theoretically) take to transfer the input relation through the PCI-E bus with the label *'Stream (Hypothetical)'*.

B. Microbenchmarks

To compare the performance of the individual A&R-operators with their standard MonetDB equivalents, we conducted a set of microbenchmarks. All of them were performed on 100 million unique, randomly shuffled integers (value range 0 to 100 million) and are displayed in Figure 8. All of them display the costs of the approximation phase (*Approximation*) as well as the overall costs (*Approximate+Refine*). Where applicable, we show the costs of the respective *MonetDB* operator.

Figures 8a and 8b shows the performance of our (inequality-)selection operator: our implementation outperforms the standard MonetDB selection unless the data is distributed (24 bit GPU, 8 bit CPU) and the selectivity is above 60%. In this case, the high refinement costs defeat the benefits of the approach.

Figure 8c shows the performance impact of the number of GPU-resident bits on the selection performance for different selectivities (.1%, .5% and 5% qualifying tuples): Naturally, when more tuples satisfy the predicate, fewer bits are needed on the GPU to achieve close to optimal performance.

Figures 8d and 8e show the projection/indexed join performance (recall that MonetDB uses indexed joins for projections). It shows that the A&R projection consistently outperforms the MonetDB projection, though less so at higher selectivities.

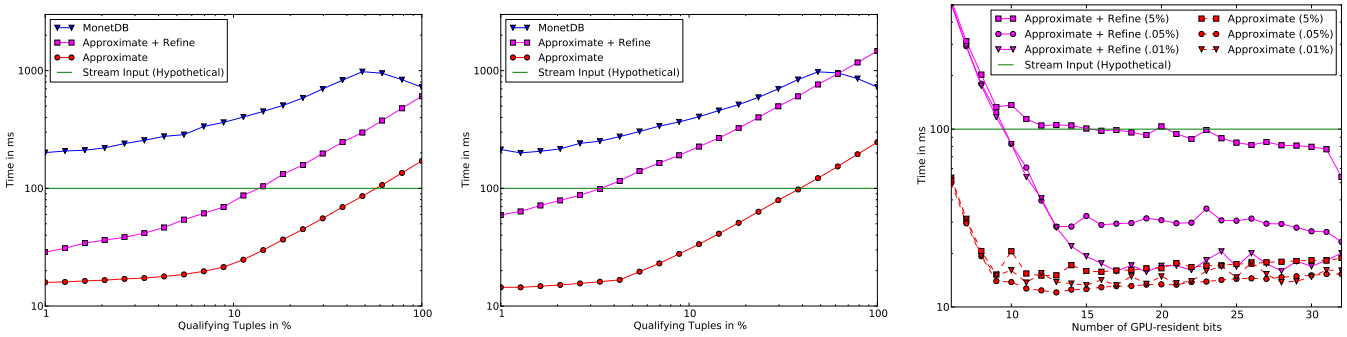
Figure 8f shows that the performance of our grouping implementation is consistently better than the standard MonetDB grouping performance. The performance improves with the number of groups due to fewer write conflicts on the grouping table.

C. Spatial Range Queries

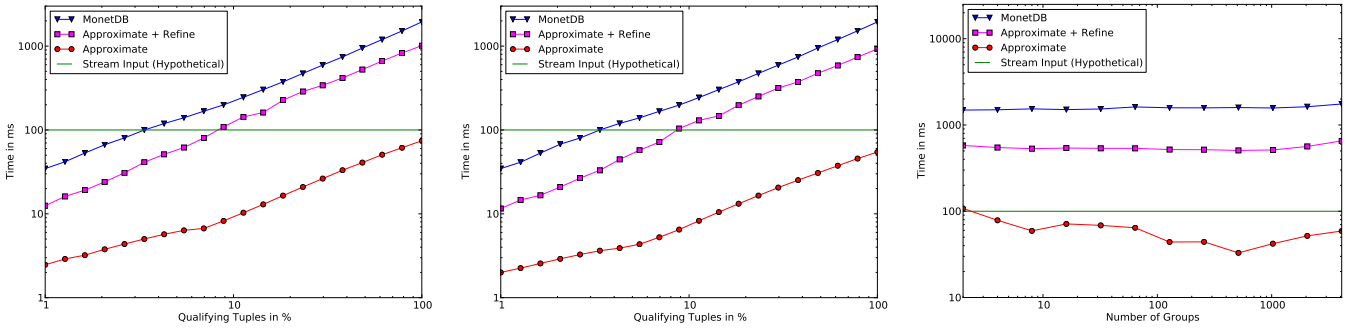
We evaluated the performance of the spatial range query benchmark to compare our generic MonetDB based implementation with the hardcoded, hand-optimized prototype of the original BWD-work [8].

[8]The current version at the time of writing can be accessed using the revision-hash `a46ca0cc4919` in the MonetDB mercurial repository (http://dev.monetdb.org/hg/MonetDB)

[9]http://developer.amd.com/tools/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk/



(a) Selection on GPU Resident Data (b) Selection on Distributed Data (8 bit on CPU) (c) Selection, varying Number of GPU-resident bits



(d) Projection/Join on GPU Resident Data (e) Projection/Join on Distributed Data (8 bit CPU) (f) Grouping on GPU Resident Data

Fig. 8: Microbenchmark Experiments

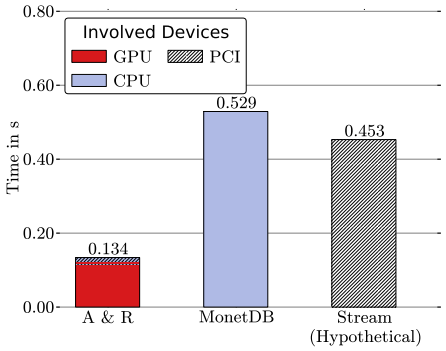


Fig. 9: Performance of the Spatial Range Queries

Schema:	create table trips (tripid int, lon decimal(8,5), lat decimal(7,5), time int);
Decomposition:	select bwdecompose(lon,24), bwdecompose(lat,24) from trips;
Query:	select count(lon) from trips where lon between 2.68288 and 2.70228 and lat between 50.4222 and 50.4485;

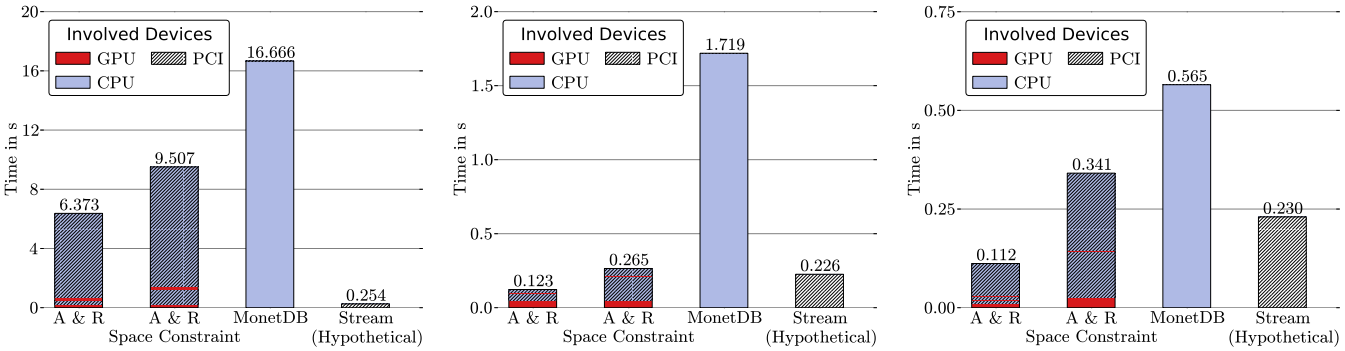
TABLE I: The Spatial Range Query Benchmark

1) Setup: The spatial range query dataset contains around 250 million tuples that represent GPS points (fixes) gathered from users' navigation devices. We stored them using the schema that is presented in Table I and applied the same decomposition that was used in the original work.

2) Data Volume: The points in the spatial dataset span a relatively wide range (27.09371 to 70.13643 and -12.62427 to 29.64975) and respectively use many bits. The opportunities for our (global) prefix compression were, thus, fairly limited: we achieved a 25% reduction of the (cumulative CPU & GPU) data volume by factoring out the highest of the 4 value bytes.

3) Performance: The results of the spatial range query evaluation (Figure 9) give an indication of the impact of the PCI-E bottleneck: Since the total data volume of the coordinate values is around 1.8 GB and some space has to be kept available for data processing, the entire input data for this query does not fit onto the 2 GB available GPU memory. This makes this query the worst case for the streaming approach (assuming a Least Recently Used (LRU) replacement strategy for GPU resident data): multiple runs of the same query cannot benefit from previously loaded data because it has just been evicted. Figure 9 shows that streaming in the input data is almost as expensive as an evaluation of the query on the CPU.

The GPU/CPU A&R implementation outperforms the CPU-only implementation by a factor of around 3.4 and the GPU transfer by around 3.2. Most of the time (almost 80 %) is spent processing data on the GPU. At first sight, this stands in opposition to the original work [8], in which we report performance gains of orders of magnitude. However, the original work was a case-specific program that was a) relying on clustered indices to improve compression as well as access



(a) Query 1 (b) Query 6 (c) Query 14

Fig. 10: Performance of selected TPC-H Queries

locality b) bulking up queries/cooperatively scanning attributes c) manually tuned towards the application. We believe that by incorporating data clustering and cooperative scanning into our approach we can achieve similar performance gains. This is, however, beyond the scope of this (foundations) paper.

D. Relational TPC-H Queries

Since TPC-H is an integrated benchmark that covers many aspects of a data management system, not all of them actually stemming from the relational query processor (but rather arithmetic or string operations processors). Since we focus on the relational aspects of data management, we selected a subset of the standard TPC-H queries that we consider representative for many relational workloads such as relational and multidimensional OLAP (on star- as well as snowflake schemas).

1) Setup: In comparison to the spatial data, the TPC-H dataset turned out more difficult to handle in the A&R paradigm. To illustrate this, consider the `lineitem` attributes that are used in the selections of Query 6: `l_quantity`, `l_discount` and `l_shipdate`. The values of all of these attributes are (almost) uniformly distributed between the extreme values and need only few bits to represent (`l_quantity`: 50 values/6 bits, `l_discount`: 10 values/4 bits and `l_shipdate`: 2526 values/12 bits) – There is simply very little to decompose in `l_quantity` and `l_discount`. Due to the low number of used bits, however, these attributes only occupy little space on the GPU if stored bit-packed. This allowed us to evaluate the performance of TPC-H (SF-10) in an all-GPU case (labeled *A & R*) as well a space constrained case in which we arbitrarily limit the available space and store the data distributed over the available devices (labeled *A & R Space Constraint*). For the space constraint case we decomposed (8-bit CPU, 24 bit GPU) the most important selection column `l_shipdate`.

When conducting the experiments, we also noticed that the costs of TPC-H Query 14 when evaluated by MonetDB are dominated by the evaluation of a string prefix predicate on the `p_type` column of the `part`-table. Since the focus of this paper are relational, rather than string-operations, we replaced this operation by a range-selection on an ordered dictionary of the (125) string values of the column. While other DBMS may support this optimization out of the box, we had to manually implement it for MonetDB.

2) Performance: Since the properties of the data made it possible to keep all necessary data (for the selections) GPU resident, we conducted a GPU-only experiment and CPU & GPU experiment for each query.

Query 1: The costs of Query 1 in MonetDB are split between the selection, the grouping and the aggregation. While the earlier two can benefit from our A&R-approach, the latter involves a multiplication and, consequently, suffers from destructive distributivity (see Section IV-G) which limits the speedup to a factor around 3x (see Figure 10a). However, since almost all tuples qualify in the selection `l_shipdate`, a reduced resolution has limited impact.

Since the necessary input data is comparatively small (around 1080 MB), but used in complex operations (i.e., grouping), the transfer of the data to the GPU is significantly faster than the A&R processing. This indicates that the performance of this query (most importantly the grouping) is bound by the internal memory bandwidth of the GPU. This would, however, also hold for a system that employs streaming of the data.

Query 6: The results (see Figure 10b) generally match our expectations: the GPU-only approach outperforms the CPU-only approach by more than a factor six. By decomposing the `l_shipdate` attribute, the performance naturally decreases by about 35 percent.

Query 14: involves a selection, a foreign key join and subsequent calculations/aggregation. The last, again, suffers from destructive distributivity while the earlier two see a speedup. Since the selection yields a smaller result set than the one in Query 1, a lower data resolution on the GPU has a larger impact (see Figure 10c).

E. GPUs versus Multi-cores versus both

In some of the previous experiments most of the time is spent performing the approximation (especially in the spatial range queries experiment). This indicates that the CPU may be underused. This is consistent with the original work [8] that reports a suboptimal load distribution when evaluating single queries on the GPU (we will discuss a potential parallelization

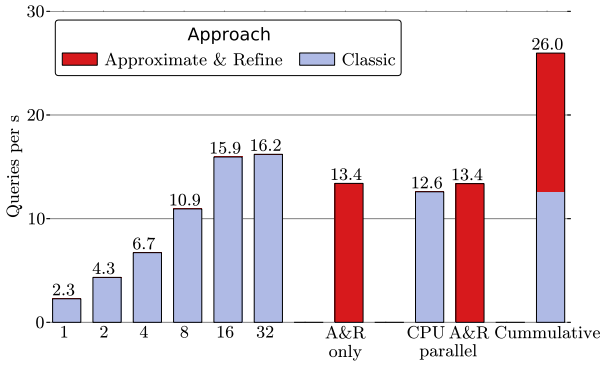


Fig. 11: A Gap in the Memory Wall

method in Section VII-B). While suboptimal load distribution can be a problem, in general, in this case, it is not problematic: freed up CPU resources can simply be used by other queries.

To evaluate this, we conducted another experiment: we ran two parallel query streams on the different devices. The one targeting the GPU runs single threaded while the one targeting the CPU uses all of the 32 CPU hardware threads (16 Cores with Hyperthreading). Figure 11 shows that increasing the number of threads eventually hits a limit due to memory bandwidth saturation. Since the GPU has a separate memory, it is not bound by the same memory bandwidth limitation. The Figure shows that GPU operations (CPU w/ GPU bar) have little impact on the performance of the CPU operation (CPU Parallel bar): these two can be combined to achieve additive performance. Thus, a GPU is a good means to scale out an existing, quite expensive (>$10,000) system using two comparatively cheap (>$500 each) GPU cards.

VII. Future and Related Work

Before concluding, let us put our approach in the context of the data management landscape. To this end, we dedicate this section to the discussion of related and future work.

A. Past (Related) Work

GPU-supported DBMS: In the recent past, GPUs have received significant attention from data management researchers mostly due to their high memory bandwidth [5], [3], [20], [6], [4]. This lead to impressive performance boosts of relational query processing on GPU-resident data. However, these systems usually assumed that data fits into the GPU's internal memory. Since this is generally not the case [7], researchers identified specific operations that can benefit from GPU support even if the data has to be transfered through the PCI-E bottleneck first [8], [18], [21], [17]. To the best of our knowledge, our work is the first approach that holistically accelerates relational query processing on data that is larger than the internal memory of the GPU while minimizing (or eliminating) the impact of the PCI-E bottleneck.

Approximate Query Processors: There is a large body of existing work on providing approximate answers to analytical queries [22], [23]. However, these approaches are usually based on sampling and thus a) rely on assumptions about the distribution of the data and b) do not allow the subsequent refinement of the approximate results. Instead queries have to be re-evaluated on the full dataset. The notion of pre-filtering and refinement to improve processing performance is, however, the idea behind many other techniques. The most prominent ones of these are bitmap indices and Bloom filters.

Bitmap Indices: In many ways, the approximation part of a bitwise decomposed attribute is similar to an underdefined/binned bitmap index. Consequently, techniques of classic bitmap-indexed processing can be applied. However, most of these techniques are used to improve performance of individual operators (largely selective scans). A processing scheme that relies on bitmaps for projections is, to the best of our knowledge, unprecedented. Many DBMSs allow the definition and use of covering indices (B^+-trees or hashes) that allow the projection of values in addition to selection.

Bloom filters: To provide fast set inclusion checking, bloom filters [24] can be added to traditional hashes. The idea is to maintain a data structure that deliberately accepts false positives when checking for inclusion. The checks are, however, very fast due to the small footprint of the underlying data structure which can be kept in memory/cache.

B. Future Work

Hardware Conscious Algorithms: One of the strengths of the A&R paradigm is that it decouples the approximation from the refine phase of the operators. This allows the independent development of efficient algorithms and tuning for the different devices. We hope that, by incorporating sophisticated present-day as well as future algorithms, the performance of relational cross-device processing can be further improved. However, the transfer of existing algorithms might not always be straightforward. The potentially high number of false positives may induce different sweet spots for existing algorithms or require new algorithms altogether. However, the number of false positives also holds opportunities for, e.g., compression of the approximation results that go through the PCI-E bus.

Cooperative Scans: When comparing our results to the original work [8], it is apparent that the performance of our system is not competitive to the original, hand-tuned solution. This can be explained in part by the generality of our system. The original solution uses a technique that is similar to the idea of cooperative scans [25]. While cooperative scans are very challenging to integrate into an existing system, this indicates that they may yield a significant performance boost.

Storage Optimization: While A&R processing enables the efficient use of multiple devices, it also enlarges the, already considerable, database design and tuning problem space. We, therefore, feel that there is a call for automatic decomposition solutions. The problem of optimal physical arrangement of persistent data has been addressed in previous work [26], [27], [28] that can be appropriately extended. Due to the high overhead of re-organization we believe the predictive modeling and optimization approach [26], [27] to be most appropriate.

Query Optimization: In this work, we established a baseline for the expected query processing performance with a very simple, straight-forward heuristic for rule-based query optimization. Since most DBMS involve some form of cost-based query optimization, assessing its impact on our approach would be worthwhile.

Applications: Since we focused on relational processing in this work, we deliberately left other kinds of applications for future work. Applications involving other kinds of operators like string-matching or theta-joins should be studied in depth. In particular string processing on GPUs is still an open problem due to the variable length of string attributes. We believe that our approach can help to solve this problem by approximating variable length strings with a fixed length prefix.

Different Storage and Processing Models: As mentioned in Section I, we consider GPU/CPU combinations an instance of the memory hierarchy problem. Since more instances of this problem exist, it is valuable to evaluate the A&R approach for other instances of this problem. A naturally related instance is the support of CPU-based data management solutions by FPGA-based co-processors. We believe that porting our approach to FPGAs is straight-forward (indeed, some FPGAs already allow OpenCL programming).

Another suitable instance of this problem can be found in the domain of *disk-resident DBMSs*: traditional rotating disks can be accompanied or replaced by Solid State Disks (SSDs). A significant body of research exists that targets such combined setups [29], [30] and could form a good basis for an evaluation of our approach. However, most disk-resident DBMS implement the Volcano-style processing on N-ary (i.e., row) storage, rather than bulk-processing. Consequently, the A&R approach has to be adapted accordingly: Similar to our implementation, each operator has to be divided into approximation and refinement and, consequently, provide two respective iterators. These iterators would operate on data on the different devices (disks, SSDs).

But even in the domain of in-memory data processing, the bulk-processing model is not without alternatives. Vectorized [12] as well as Just-in-Time-compiled (JiT-compiled) [13] query processing provide high cache efficiency for CPU-only setups. Combining these processing models with the A&R-paradigm seems rewarding but challenging: these models use sophisticated processing techniques that might be non-trivial to combine with our approach. While we believe our approach can be adapted to such processing models, it needs a detailed study to prove this.

VIII. Conclusion

To address the problem of wasted resources caused by increasingly heterogeneous hardware, we proposed a generic processing paradigm for bitwise distributed relational data. Based on classic relational algebra, the paradigm is simple, yet powerful enough to a) provide a generic framework for efficient cross device relational data processing, b) allow architecture specific optimizations of individual operations and c) permit the parallel execution of operations on the available devices. This paradigm prescribes the multi-stage (Approximate & Refine) execution of relational operators. This leads to significant performance improvements (up to 8 times faster query evaluation than classic MonetDB) when handling large databases in a CPU/GPU co-processing setup Additionally, it provides the possibility to compute a fast approximation of the query result before it is calculated at no additional costs. To back the paradigm, we introduced efficient algorithms for the arising problems and proved its feasibility by implementing it in an existing relational DBMS.

References

[1] J. Hennessy and D. Patterson, *Computer architecture: a quantitative approach*. Morgan Kaufmann, 2011.

[2] L. Grupp *et al.*, "The bleak future of nand flash memory," in *USENIX FAST*, 2012.

[3] P. Bakkum and K. Skadron, "Accelerating SQL database operations on a GPU with CUDA," in *GPGPU*, 2010.

[4] B. He *et al.*, "Relational joins on graphics processors," in *SIGMOD*, 2008.

[5] R. Fang *et al.*, "GPUQP: query co-processing using graphics processors," in *SIGMOD*, 2007.

[6] B. He *et al.*, "Relational query coprocessing on graphics processors," *ACM TODS*, vol. 34, no. 4, 2009.

[7] C. Gregg and K. Hazelwood, "Where is the data? why you cannot debate cpu vs. gpu performance without the answer," in *ISPASS*, 2011.

[8] H. Pirk *et al.*, "X-device query processing by bitwise distribution," in *DaMoN*, 2012.

[9] P. A. Boncz, "Monet: A Next-Generation Database Kernel For Query-Intensive Applications," Ph.D. dissertation, Universiteit van Amsterdam, May 2002.

[10] D. Abadi *et al.*, "Column-stores vs. row-stores: How different are they really?" in *SIGMOD*, 08.

[11] P. A. Boncz *et al.*, "Breaking the memory wall in monetdb," *Communications of the ACM*, vol. 51, no. 12, 2008.

[12] M. Zukowski *et al.*, "MonetDB/X100-a DBMS in the CPU cache," *Data Engineering Bulleting*, vol. 28, 2005.

[13] T. Neumann, "Efficiently compiling efficient query plans for modern hardware," in *PVLDB*, vol. 4, no. 9, 2011.

[14] G. P. Copeland and S. N. Khoshafian, "A decomposition storage model," in *SIGMOD*, 1985.

[15] E. Codd, "A relational model of data for large shared data banks," *Communications of the ACM*, vol. 13, no. 6, 1970.

[16] J. Cieslewicz and K. Ross, "Adaptive aggregation on chip multiprocessors," in *VLDB*, 2007.

[17] T. Kaldewey *et al.*, "Gpu join processing revisited," in *DaMoN*, 2012.

[18] H. Pirk *et al.*, "Accelerating foreign-key joins using asymmetric memory channels," in *ADMS*, 2011.

[19] K. Bösche *et al.*, "Scalable Generation Of Synthetic GPS Traces With Real-Life Data Characteristics," in *TPCTC*, 2012.

[20] W. Fang *et al.*, "Database compression on graphics processors," *VLDB*, 2010.

[21] K. Wang *et al.*, "Accelerating pathology image data cross-comparison on cpu-gpu hybrid systems," *PVLDB*, vol. 5, no. 11, 2012.

[22] P. Rosch *et al.*, "Designing random sample synopses with outliers," in *ICDE*, 2008.

[23] S. Acharya *et al.*, "The aqua approximate query answering system," *SIGMOD*, 1999.

[24] B. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, 1970.

[25] M. Zukowski *et al.*, "Cooperative scans: dynamic bandwidth sharing in a dbms," in *VLDB*, 2007.

[26] M. Grund *et al.*, "Hyrise: a main memory hybrid storage engine," *PVLDB*, vol. 4, no. 2, 2010.

[27] H. Pirk *et al.*, "Cpu and cache efficient management of memory-resident databases," in *ICDE*, 2013.

[28] A. Ailamaki *et al.*, "Weaving relations for cache performance," *VLDB*, 2001.

[29] N. Zhang *et al.*, "Towards cost-effective storage provisioning for dbmss," *PVLDB*, vol. 5, no. 4, 2012.

[30] T. Luo *et al.*, "hstorage-db: heterogeneity-aware data management to exploit the full capability of hybrid storage systems," *PVLDB*, vol. 5, no. 10, 2012.