Overtaking CPU DBMSes with a GPU in Whole-Query Analytic Processing
with Parallelism-Friendly Execution Plan Optimization

Adnan Agbaria[1], David Minor[2],
Natan Peterfruend[3], Eyal Rozenberg[4], Ofer Rosenberg[5], and Roman Talyansky[6] *

[1] Intel | adnan.agbaria@intel.com
[2] GE Global Research | david.minor1@ge.com
[3] Huawei Research | natan.peterfruend@huawei.com
[4] CWI Amsterdam | E.Rozenberg@cwi.nl
[5] oferrose73@gmail.com
[6] Huawei Research | roman.talyansky@huawei.com

Abstract. Existing work on accelerating analytic DB query processing with (discrete) GPUs fails to fully realize their potential for speedup through parallelism: Published results do not achieve significant speedup over more performant CPU-only DBMSes when processing complete queries.
This paper presents a successful effort to better meet this challenge, in the form of a proof-of-concept query processing framework. The framework constitutes a graft onto an existing DBMS, altering some parts of it and replacing its execution engine entirely. It intensively refactors query execution plans, making them better-parallelizable, before executing them on either a CPU or on GPU. This results in a significant speedup even on a CPU, and a further speedup when using a GPU, over the chosen host DBMS (MonetDB) — which itself already bests most published results utilizing a GPU for query processing.
Finally, we outline some concrete future improvements on our results which can cut processing time by half and possibly much more.

1 Introduction

Database Management Systems (DBMSes) in wide use today were designed for execution on a 'serial' processing unit. Even when multi-thread and multi-core capabilities are taken into account in the design, massive parallelism is typically not a significant consideration: The execution strategy, the fundamental internal operations used in executing queries, the representation of data in memory — these are all incarnations of original designs with serial execution in mind, even if these days. Even when multiple threads are used, they mostly behaving like so many single-thread DBMSes, each processing a large chunk of the data, independently.

As the use of GPUs for computation more general than graphics processing is spreading, industry and academic have begun exploring its potential use in

* Work carried out by all authors as members of the Heterogeneous Computing Group at Huawei Research, Israel. Authors appear in alphabetical order.

processing relational database queries. Initially, contributions such as [5] focused on efficient implementation of primitive query-processing-related computational operations: These relatively self-contained pieces of code are what the CPU actually spends time on; and replacing them with carefully-optimized kernels running on the GPU does accelerate them. However, this does not immediately translate to impressive acceleration in processing *entire* queries.

This fact has motivated two avenues of research. One of them, not explored here, focuses on integrated CPU-GPU processors, such as AMD's APUs. These remove the bandwidth limitations of the PCIe bus, a key reason for the underwhelming performance of GPU DBMSs; [14] is a recent contribution in this vein, with references to additional work. A second approach is processing *queries as a whole* rather than only their constituent operations. Most experimental work in this avenue (ours included) involves *grafts* onto an existing *host DBMS*. A graft overrides parts of the normal compilation process and modifies existing code to create interfaces and hooks for new and replacement functionality *of complete sub-sections* of the query plan. Instead of merely replacing the code for execution of *individual* query plan operations, they alter and substitute *large sections* of the entire plan. Thus the generated plans are significantly different, and so is the execution mechanism, which is sometimes replaced altogether. Prominent recent examples of such frameworks include Red Fox [21] and GPU-DB[23] (also cf. [4]). However, despite the progress made so far, these efforts have not produced systems with query processing speed on par with the more performant free-software DBMSes, such as MonetDB [11, 10] (which are themselves bested by some closed-source DBMSes, such as Actian Vector[24] and HyPerDB [7]).

We perceived previous work as being overly attached to existing DBMS' massive-parallelism-unfriendly execution planning — in other words, it seems that most often *they are still having a GPU "do a CPU's job".*

To gain a performance benefit from using a GPU, we decided that instead of optimizing its execution of the tasks it is given by the traditional SQL optimizer, we should instead focus our effort on creating new GPU-freindly tasks and feeding them conveniently-represented data on which they could shine. Very roughly, such computational work is characterized by:

- Less code path divergence;
- More work by related threads on small, localized data;
- Well-coalesced memory accesses;
- Avoidance and circumvention of data dependencies, or at least the 'flattening out' of dependency relation into a shallow forest;
- a focus on throughput rather than on latency;

Some constituent operations in CPU-targeted execution plans cannot be implemented as-is in this fashion; but often their semantics can be tweaked, or their input or output formats altered, so that they admit a GPU-parallelism-friendly implementation (as is well-evidenced by the recent study of approaches for optimizing `LIKE` pattern matching on string columns [16]). Although many operations do not allow for such an adaptation, or do not benefit from it as significantly as others — we need to remind ourselves of our objective: It is the *plan* whose execution we wish to speed up, not the individual operations that are just a means to that end.

Often including a less-then-optimal operation in an optimal GPU query plan will still lead to an overall improvement for the entire plan. In most cases, however, we can, in fact, avoid computational operations which the GPU does not favor, choosing alternate (sub)plans for that part of the query's execution. This approach underpins the query processing framework we developed as a proof-of-concept, and as the rest of this article demonstrates, it provides a significant improvement over other state-of-the-art in processing systems for *full* TPC-H queries.

2 The processing framework

With numerous query processing frameworks utilizing GPUs already in existence, Breß, Heimel, et alia devised a classification scheme for these in [4, §4.3]. Before describing our framework, here is how it fits into this scheme:

Storage: Location	In-memory only
Storage: Model	Column store
JIT compilation	None (but with IR transforms)
Processing: [x] at-a-time	Operator (not tuple or block)
Device support	Single-device and multi-device
Transactions	Not supported (read-only)
Hardware portability	CPUs & (CUDA) GPUs

Implemented Breß-et-alia-listed optimizations: GPU-aware query optimizer; Efficient data placement strategy; Overlap of data transfer and processing (partial); Pinned host memory.

Table 1. Breß-et-al.-style classification

2.1 From query to execution run

The processing framework adopts the common approach of grafting onto an existing DBMS; our choice was the analytics-oriented column store MonetDB. Figure 1 summarizes which components of MonetDB are replaced or modified and which additions the graft introduces. It is helpful to the diagram keep in mind as the processing of incoming queries is described further below and in Section 3.

As a new (analytic) query arrives, the host DBMS parses it, considers its relational algebra, and generates an initial execution plan using its internal representation — for MonetDB, the single-assignment language MAL [19]. We interrupt the usual sequence of optimizers which MonetDB applies, replacing some of the final optimizers with a mock optimizer, whose task is to convert the sequence of MAL instructions into an alternative intermediate representation. Specifically, MonetDB's data-parallelism-inducing transformations are not applied; our execution engine will later transform the plan to utilize multiple CPU cores and/or GPU devices instead. Finally, instead of invoking MonetDB's execution engine / MAL program interpreter (named GDK), our own execution engine is invoked, ignoring the MAL sequence itself.

Our execution engine takes the following input: A (directed acyclic) graph, the 'execution' of which should obtain the query results; access to a library of GPU

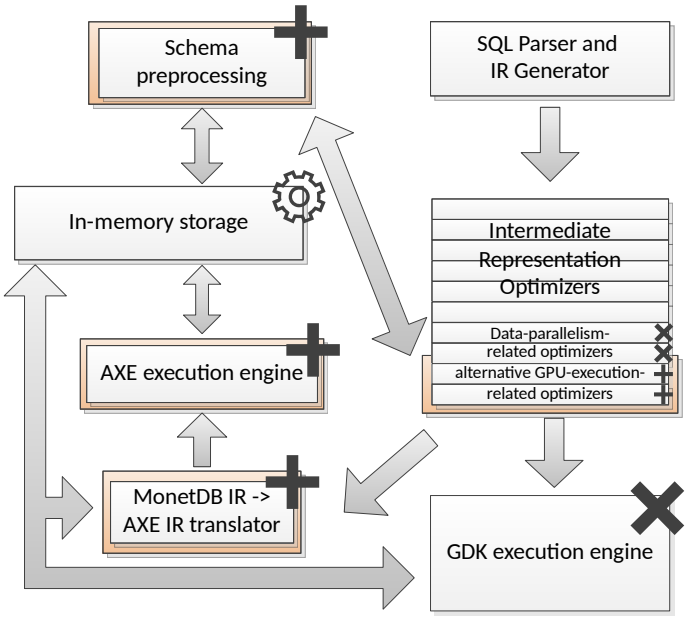


Fig. 1. The processing framework as a graft onto MonetDB

optimized computational primitive implementations (as these are not inherent to the engine); access to a library of similar implementations for CPUs; locations of buffers in the system's main memory (for schema columns and auxiliary data; see Subsection 2.3); and a set of analytic-query-related transformation rules it may apply to its execution graph. When the execution engine completes its work, results are present in main memory and are passed back to MonetDB as though produced by its own native execution of the query.

2.2 The execution engine

Our query processing framework had at its core an execution engine called *AXE* (Adaptive Execution Engine) developed by the Heterogeneous Computing group of Huawei's Shannon Lab; this engine was designed independently of its specific use in this work — as the group's areas of interest are not limited to analytic DB query processing. It was designed to accommodate multiple types of computational devices and applications. It is built on the abstraction of a DAG of computational *operations*. Operations are drawn from a pool of hand-optimized domain-specific libraries. A run of the execution engine involves the concurrent and sequential execution of multiple operations. Some of these operations are massively parallel (e.g. binary vector operations) and others are run in task parallel fashion (e.g host to device data transfers). AXE operations operate on *buffers* — an abstraction of regions of memory, which may be instantiated on the memory space of different devices, copied between memory spaces, pre-allocated, resized and set as inputs or outputs for operations both

on and off the device. The AXE engine's IR (intermediate representation) is a DAG-like execution plan, describing the dependencies, operations and buffers needed for execution, along with additional information to help guide transformations by the engine. An internal queuing and scheduling mechanism allows for asynchronous execution of operations, dependency enforcement, synchronization and task level parallelism. Fine-grained, regularized (often synchronous) parallelism — typical of GPU code — is encapsulated into the implementations of the *operations* themselves, so that the engine is not GPU-specific. AXE also supports data parallelism, by cloning subgraphs at the IR level, splitting inputs among the subgraphs, and finally joining the results computed by each of these partition subgraphs . In order to accommodate the varigated of SQL semantics, a variety of partitioning and joining schemes are used (e.g. duplicate all, bit-or). Which scheme to use is providing by annotations over the inputs/outputs of individual operations, or reverts to a standard default.

Execution plan transformations occur at two distinct stages in the compilation/execution process. To see why this is so, consider the following: The *higher strata* of transformations, those within the DBMS interpreter itself, are *oblivious of the hardware* on which the plan will eventually execute, e.g.. which computational devices, device capabilities, communication buses, memory space sizes, etc. The *lower stratum* of transformations, those taking place within our execution engine, is oblivious to the original application which provided the engine with the plan. It holds no information regarding databases, queries, relational tables, foreign-key relations and so on. This separation of concerns between the domain-specific (higher-strata, within DBMS) and hardware-specific (lower-strata, within AXE) is a useful technique, allowing for effective execution optimization catering to different applications (some more on this in Section 3). Of course, the separation is somewhat artificial, as hardware-related choices impact the benefit of domain-specific choices w.r.t. the plan; we therefore compensate with hints, statistics and suggested partitioning and transformation options, passed down to the runtime engine in addition to the actual plan, compensate address this fact partially. This aspect of the design merits a separate discussion which is beyond the scope of this paper.

2.3 Schema preprocessing

When the modified host DBMS passes the execution engine a plan to execute, this execution will not be applied merely to the DB columns themselves, as-is. Instead, the execution engine receives references to the results of some offline preprocessing of the schema. In a row-oriented DBMS such preprocessed data might be multiple indices into different tables; and MonetDB has its "imprints" structure [15]. Preprocessing can theoretically be quite extensive (and time-consuming); in some work on GPU acceleration, authors go as far as pre-joining tables or materializing full denormalizations. Also, the more auxiliary information one maintains, the less one can scale a DBMS while remaining entirely in-memory; but such economy of resources is beyond the scope of this work (especialy since we do not use compression; see Section 6).

For our work with the TPC-H benchmark, our preprocessing adhered strictly to its rules and restrictions, i.e. we limited it to single columns of data, never involving information regarding multiple columns. Of course, the different choices

of preprocessed data made available to a query processor muddy the waters to some extent when measuring and comparing performance, and this is especially true for comparisons with processing frameworks not bound by TPC-H restrictions (such as GPU-DB [23] or the recent results in [14]).

The data derived from each column and used in this work falls into one the following categories:

scalars: Data of the column's own type (e.g. minimum, maximum, median, mode), integral statistics (e.g. support size) and binary predicates (e.g. sorted/unsorted)
same-dimension auxiliary columns: such as a sorted copy of a column, or a breakdown of date/time columns into their constituent subfields
support-dimension auxiliary columns: Essentially a small auxiliary table with one row per distinct value in the original column, containing the histogram, as well as minimum and maximum positions of incidence, for each value (i.e. a reverse-index for the column).

The host DBMS, MonetDB, is not made aware of this preprocessed data — nor does our framework use MonetDB's Imprints or any other such auxiliary data.

3 Making execution plans more amenable to (GPU) parallelism

Setting aside the specifics of our framework design, and how it differs from the host DBMS's, this paper's title begs the following question: Why is query plan optimization *particularly critical* for GPU execution performance?

The general importance of query optimization to processing performance is well-recognized [17, Chapter 7] and widely studied; Join order, nested query reformulation, intelligent estimation of intermediary result cardinality and so on. Our framework does not actually brave this important task: It does not try to second-guess most of the host DBMSes decisions; while this would probably be useful as well, the focus of this work is more lower-level. Namely, our optimizing transformations regard

Implementation special-casing for generally-challenging computations using statistics & predicates obtained by preprocessing the schema (or in some rare cases at query runtime).
Representation format change including mostly two aspects of how data is represented: dense vs. sparse representations of subsets/subsequences (see Subsection 3.1), and sortedness constraints (whether plan operations are required to produce sorted output, and whether they require their inputs to be sorted).
Missing implementation circumvention Replacement of operations without GPU implementations by multi-node subgraphs with equivalent output.
Fusion of certain particularly-suitable consecutive operations. This is not the comprehensive fusion of multiple operations using compilation infrastructure used in HyPer [12] or Spark 2.0 Catalyst [1]; instead, we apply more complex fusion, implemented a-priori in CUDA code, which a compiler could *not* automatically derive.

Fission Some plan operations have inherently multi-staged implementations (at least on a GPU); others can be semantically decomposed (e.g. in a reversal of the fusion described above). This can be reflected in the plan, allowing constituent parts or phases to be involved with other operations in the further application of transformation rules.
Cleanup when duplicate/inverse operations, remain in the plan after other transformations, or when an operation's outputs are unused, etc.

These transformations are applied greedily — that is, no change is made to the plan unless it is certain to be positively beneficial (to the plan as a whole, individual operations may not be optimal). Having examined the various kinds of query plans that MonetDB generates, we formulated a number of transformation rules — limited to small subgraphs — which are likely to speed up execution. For each of these we formulated constraints on columns, intermediary buffers and operations involved, under which this likelihood of benefit becomes a certainty. These constraints are expressible in terms of the statistics and predicates we obtain regarding the data as part of the schema preprocessing described in Subsection 2.3. Of course, we strove to formulate rules with the weakest possible applicability constraints, to maximize variety and usefulness over multiple queries. Rules are applied repeatedly until a fixed point is reached, with the exception of an initial analytic phase. Thus our optimization of an execution plan is *rule-based*, and mostly *heuristic*.

It should also be noted our current set of rules is not very extensive. Even for the queries for which we present results, one could well conceive of additional rules applicable as-is (see the end of Subsection 3.3 below), or additional auxiliary data (Bloom filters, Imprints, etc.) and new rules able to utilize it. Although we probably missed many opportunities for further optimization, this paucity of rules prevented us from facing the problem of inopportune choice of rule application order leading our framework away from better optimization routes.

The rest of this section is an elaboration on two of the aspects mentioned above, followed by a detailed example of how all aspects combine in the optimization for a single specific query.

3.1 Optimization aspect: Subset/sequence representation

Consider the result of some predicate applied to a DB column. In MonetDB (v11.15.11) this result is sparse — a column of matching record indices — rather than a dense bit vector. Computing the former, a (serial) CPU core repeatedly appends matching indices to the output; but this does not parallelize very well, since the final location to write to for any individual element satisfying the predicate requires information regarding previous matches. *Some* parallelization is still possible here: For example, one may compute a prefix-sum of the number of elements passing the filter — and well-optimized prefix sum implementations on GPUs are available [9, 18] — but this is still much slower than element-wise bit-setting.

From a complexity-theoretical perspective, a sparse representation is certainly the appropriate choice: Further computation is linear, based on the length of the result, not of the original data. But in practice, we count bits: A record index is likely 4 or 8 bytes; and memory is typically read in units of a cache line (64B on Intel

Haswell, 32B/128B on nVIDIA Kepler & Maxwell). So only for *very* selective predicates does the benefit-in-principle actually manifest. A sequential-CPU-oriented DB might prefer the sparse representation earlier: It has fewer elements to perform writes for, i.e. less sequential work (ignoring caching at least); and presumably this is the case for MonetDB. A DBMS oriented for massive regular parallelism will opt for the dense representation in most cases.

Most existing GPU acceleration frameworks seem to resign themselves to respecting the DBMS' data-structure choices, and will compute this sparse result as best they can; after all, the plan uses this array of indices later on. But you can ask the question — does the plan really have to use it? As will be illustrated in Subsection 3.3 below, this use is itself conditioned on this choice, which can be undone or overruled. When doing so, one often ends up avoiding some reordering of data, a costly effort in itself, and expanding the opportunities for using more parallel-efficient computational primitives. Last but not least, dense subset representation often lends itself to avoiding the need to produce sorted intermediary results (again, see below).

3.2 Optimization aspect: Join special-casing

A general-case single-column inner Join in a (MonetDB-like) column-store takes as inputs two columns (the LHS and the RHS; assume they hold integral values). The result of the Join are two columns, $\mathrm{LHS}_{\mathrm{out}}$ and $\mathrm{RHS}_{\mathrm{out}}$, whose length is the number of matching pairs; the tuple ($\mathrm{LHS}_{\mathrm{out}}[i]$,$\mathrm{RHS}_{\mathrm{out}}[i]$) is the i^{th} match found by the Join, so i may range from 0 to $|\mathrm{LHS}\times\mathrm{RHS}|-1$ theoretically (and the output is sorted lexicographically).

Our framework observes the following noteworthy features for each column with respect to a Join operation (phrased in terms of the LHS below):

- Is this a column coming directly from the schema, or is it an intermediate result following other operations?
- Are the column values sorted? If so, do they appear consecutively with no gaps (i.e. $\mathrm{LHS}[i+1]=\mathrm{LHS}[i]+1$)?
- What are the minimum and maximum column values?
- What are the minimum and maximum multiplicities of individual values within the column?
- Is the other input column known to contain all values in this one?
- Is every value in the column known to have at least one match on the other side?
- Is the Join output only used to filter the input column?

After applying our preprocessing, whenever a query comes in we have most of this information readily available, without computing anything — using scalar values and predicates regarding individual columns, and the schema structure. Some of these statistics may not be available — minimum and maximum values, multiplicities for non-schema columns — and in some cases (see below) we may take the time to compute them. Now, here are several cases of Joins for which we had special Join implementations (each corresponding to combinations of the above criteria):

FK to dense PK: LHS: All values match. RHS: Dense.

Self-join of filtered schema column: LHS: All values match. RHS: Subset of a schema column, sorted, known max. multiplicity.
FK to small-support PK: LHS: All values match. RHS: max. multiplicity 1; LHS values are in the range $[v_{\min},v_{\min}]$; there's sufficient memory for a bit vector of length $v_{\max}-v_{\min}+1$.
RHS-Unique Join: LHS: No assumptions. RHS: max multiplicity 1.
For each of these special cases we have a corresponding transformation rule; and these rules are applied, when applicable, in the above order of priority, to all Join operations encountered in the execution plan. Some of these rules are replacements of the single Join operation DAG node with the appropriate special-case-Join node — for which we've written hand-optimized special-case implementations. In other rules, the Join is replaced with a small subgraph of non-Join computations (e.g. using original values instead of hash keys).

3.3 A query example: Optimizing TPC-H Q4 execution

Typically, no single transformation rule mentioned above is sufficient to fundamentally change how a query is processed; it's rather a combination of rules which allows for more fundamental changes. We illustrate how the rules combine using the sequence of transformations our framework applies for TPC-H Q4 `\ref{Q4}`. We consider the main part of the original plan, but for brevity, dropping the part retrieving a string column at the end. [7]

Figure 2 represents our initial derivation of a plan from the one obtained from MonetDB. Without going into detail, this involves removing some redundant/irrelevant MAL statements from MonetDB's own plan, and more importantly: Splitting up these operations into constituent parts, to the extent we have kernels for them — particularly when sparse/dense format changes are involved. Now, this can be considered a non-greedy transformation — as fusing these operations back causes a slowdown — which our subsequent repeated transformation process would not apply. However, this slowdown is usually marginal: It's a write an intermediary buffer to global memory by the first constituent operation, and a read of that buffer by the second. This opens up the possibility of some proper optimization (see below); and if any such non-trivial optimization can be applied, it will most likely compensate for the extra I/O. The diagram is a dataflow DAG, with source nodes being schema columns or constant values, and all other nodes being computational operations. The full plan for TPC-H Q4 has two sink nodes — for the order counts and priority string columns— but the part of the plan generating the latter has been removed to focus the example on the former. Also removed — for brevity and legibility — are the details of which edge targets which parameter of its destination operation. Table 2 describes the semantics of the operations used on the plan and its transformed versions.

Considering the operations appearing in the initial plan, in Figure 2, one notices they produce mostly sorted sparse-representations (sorted indices into columns) intermediary columns; and that many of the operations require inputs of this kind.

[7] Q4 was chosen for this example for being a query with a short plan with few operations, but involving more than one table.

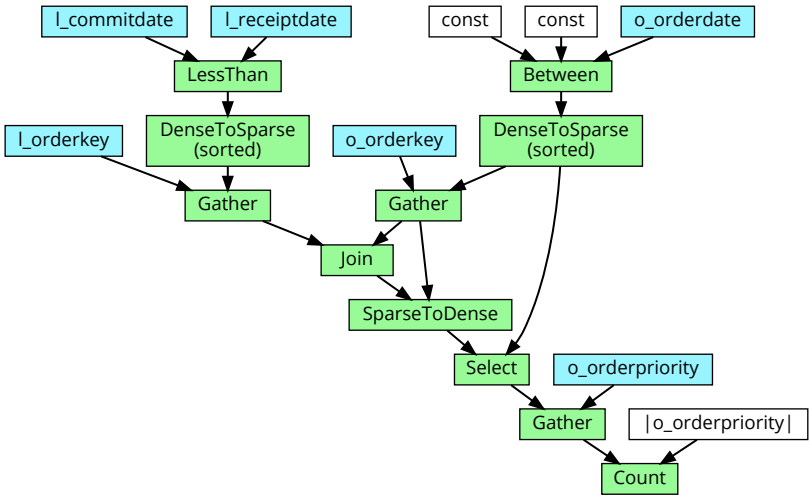


Fig. 2. The initial execution plan for Q4 (string output column clipped)

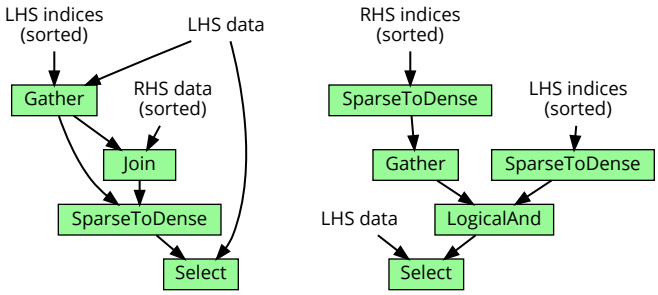
Operation	Column inputs	Output description
Select	data D bit vector F	Values in D whose corresponding bit in F is set
Gather	data D indices I	ouput$[i] = D[I[i]]$
Scatter **Disjunction**	data D indices I zero-initialized T	$T[I[i]] = D[i]$
DenseToSparse	bit vector D	The indices of all bits set in D
DenseToSparse	indices I	A bit vector with bit i set iff $i \in I$
LessThan, **BitwiseAnd**	L, R (of same type)	Elementwise binary operations
Between	input X	A bit vector with bit i set iff $c_1 \leq X[i] < c_2$
Join	L, R (of same type)	all pairs (i,j) such that $L[i] = R[j]$, in the form of columns of corresponding i's and j's
Count	indices I	A histogram of I, where the bins are $0...m$ for a known maximum value m.

Table 2. TPC-H Q4 execution plan operation semantics

Our optimizer performs the following transformations; note that for some of these transformations there are requirements not represented in the diagrams, most frequently "no other outgoing edges" when removing operations. :

1. **Subsequence semi-join special-casing**: Initially it seems we cannot get rid of the DenseToSparse operations. Consider, however, the use of the `ORDERS` table in the plan: The table is 'first' used is to Gather data for the Foreign Key – Primary Key Join. The combination of DenseToSparse and Gather admits an

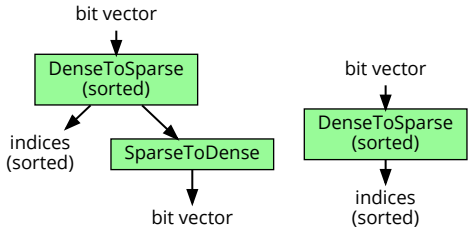
optimization in itself (see below); but, there is a far more beneficial transformation possible here: The supposedly-general-case Join is actually made in the context of a Semi-join, a filtering of the `ORDERS` table. This can be inferred locally (seeing how the `o_orderdate` DenseToSparse output is used both in the Join and immediately with the Join's output); Now, instead of Join'ing, we can apply the filter on a dense representation in the value space of {`l`|`o`}`_orderkey` as a BitwiseAnd (Figure 3).



(a) before (b) after

Fig. 3. TPC-H Q4 Transformation 1

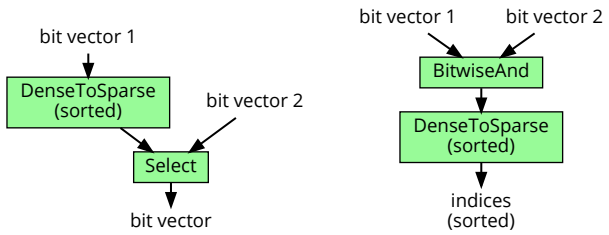
2. **Cleanup I**: The previous transformation leaves us with a dense-to-sparse-to-dense conversion sequence; we can't eliminate it entirely, since the intermediate sparse result is still in use, but we may bypass it and thus discard one of its constituent operations (Figure 4).



(a) before (b) after

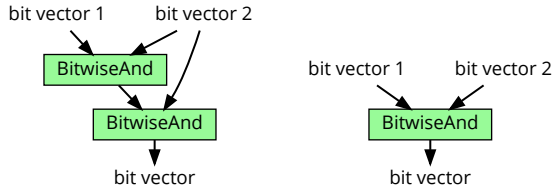
Fig. 4. TPC-H Q4 Transformation 2

3. **Pushing DenseToSparse down**: The remaining use of the sparse `ORDERS` filter results is in Selecting from the semi-join results, which themselves are in the form of a dense subsequence representation due to transformation 1. The DenseToSparse can therefore be "pushed down" past the Select, which becomes a bitwise AND to preserve its semantics (Figure 5).
4. **Cleanup II**: We now have an artifact due to previous transformations: redundant BitwiseAnd operations; we remove one of them (Figure 6).
5. **Fusing DenseToSparse and Gather**: We note that the action of Select can be described as converting a bit vector input into a sparse representation,



(a) before (b) after

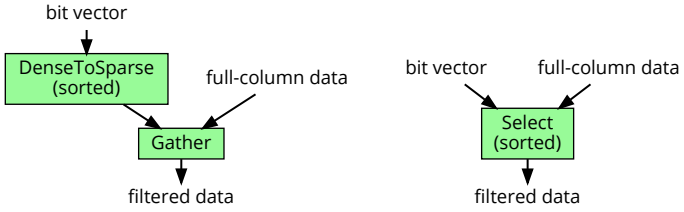
Fig. 5. TPC-H Q4 Transformation 3



(a) before (b) after

Fig. 6. TPC-H Q4 Transformation 4

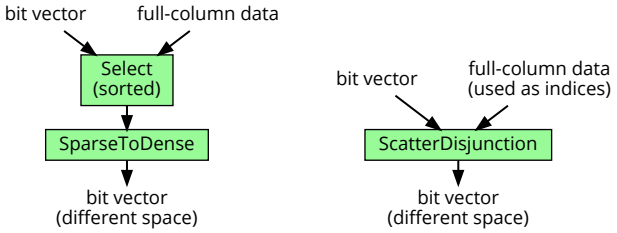
then replacing the indices with actual data using a Gather; and that if the DenseToSparse has sorted output, so will the Select. In the other direction, these two operations can be fused together into a Select (Figure 7); we now do so for both our DenseToSparse's.



(a) before (b) after

Fig. 7. TPC-H Q4 Transformation 5 (applied twice)

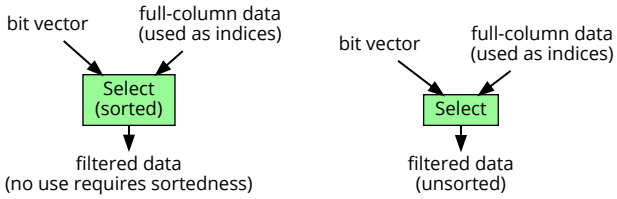
6. **Fusing Select and SparseToDense**: These two operations may be fused into a ScatterDisjunction (Figure 8), similarly to transformation 5. This transformation has an attractive side-effect: We are now rid of the sortedness constraints for one of the filters, as its sorted (sparse) results are no longer used anywhere.
7. **Dropping last sortedness constraint**: The output of the remaining Select operation (of the created earlier by fusions), is only used by a Count operation. While Count might benefit from its input being sorted — it certainly doesn't require sortedness (and in this specific case the benefit would be marginal).



(a) before (b) after

Fig. 8. TPC-H Q4 Transformation 6

Thus we have pushed the sortedness requirement further enough down the plan DAG to a point where it can be simply discarded (Figure 9).



(a) before (b) after

Fig. 9. TPC-H Q4 Transformation 7

The end result is the pleasing, relatively parallelism-friendly execution plan in Figure 10; the kernels corresponding to each of the green nodes would now gets scheduled to execute on the GPU. However, on closer inspection we note that even further optimization of the plan is possible: ScatterDisjunction can be avoided in favor of Gathering after applying a fixed offset to the indices, and Select can be avoided altogether in favor of a predicated Count; this would make the plan embarassingly parallel except for the final aggregation — so much so that it might theoretically be compiled into a single GPU kernel. However, these additional optimizations were not supported by our framework when the experimental results (Section 4) were obtained.

4 Experimental Results

4.1 Test platform, protocol and procedures

Results were all obtained using a 2-socket machine, with 2 Intel Xeon E5-2690 CPUs (8-core each) clocked 2.9 GHz. Each socket had its won independent PCIe 3 bus, through which it was connected to a GeForce GTX 780 Ti card (MSI TwinFrozer, clocked at 875 MHz). The machine ran Kubuntu GNU/Linux 14.04, CUDA 7.0 RC and nVIDIA driver v346.29. The reference DBMS was an unaltered version of MonetDB [11] v11.15.11 (by now not the latest version), using 32 threads.

We tested using queries from the TPC-H benchmark [20]: Q1, Q4, Q9 and Q21 (ranging from simple to complex). This limitation is the result of a constrained

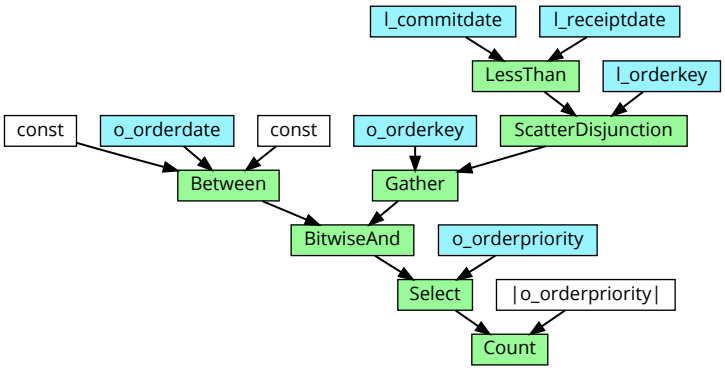


Fig. 10. Final execution plan for Q4 (string output column clipped)

amount of time and effort to put into this proof-of-concept — which is not a full-fledged query processor. We did not send random queries to the host DBMS repeatedly over a prolonged period of time (as in the actual TPC-H procuedure); rather, we tested individual queries separately on the cold DBMS, immediately after it was loaded. Timing figures are the mean over 3 runs, in milliseconds. The database Scale Factor (SF) is 1 unless otherwise stated.

4.2 TPC-H query processing time comparison

A 'bottom line' of our results appears in Table 3: Execution time for the final query plan for all of our benchmarked queries. A result for CPU execution of Q21 is missing as it requires a yet-unimplemented feature of our subgraph partitioning feature; and without it, performance is dismal (as our multi-threaded execution depends on subgraph partition).

TPC-H Query	**Q1**	**Q4**	**Q9**	**Q21**
MonetDB	159.4 ms	54.0 ms	125.9 ms	217.5 ms
MonetDB/AXE CPU	41.9 ms	24.5 ms	31.1 ms	
MonetDB/AXE GPU	25.8 ms	18.4 ms	21.5 ms	44.0 ms

Table 3. Final plan execution times (SF 1 including I/O)

The speedup over MonetDB execution ranges from 2.9 to 6.8; in a more apples-to-apples comparison — the same modified plan on a CPU rather than a GPU — the speedup factor ranges from 1.3 to 1.6. This too should be taken with a large grain of salt, since the comparison is between two CPUs "against" just one GPU. The more important figure is the GPU plan execution time itself.

The results charted in Table 3 do not include the time spent by MonetDB or our framework on parsing the query and preparing the plan; Figure 11 adds this information, as part of a breakdown of the overall query processing time into (mostly-consecutive) phases.

One obvious problem is the large amounts of time spent before query execution even begins. This is particularly bad in our CPU-only configuration, which is un-

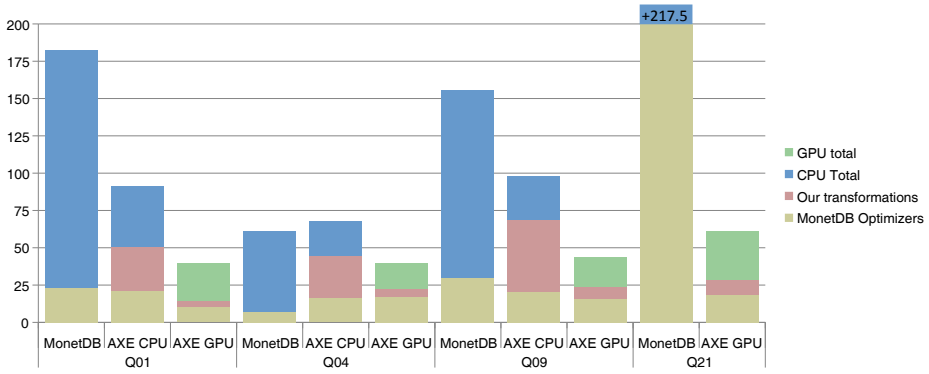


Fig. 11. Processing time breakdown (SF 1); clipped at 250ms

fortunately inefficient in transforming the plan (it performs a 32-way partitioning of the execution graph, in an unoptimized fashion; and this takes more time than all other transformations combined). MonetDB also seems to suffer from a similar phenomenon when adapting a complex execution plan such as Q21 to accommodate many threads. Such deficiencies can be mostly be done away with by straightforward optimization of our code (as opposed to optimizing the execution plan); we simply lacked the time to do so before our work needed to be wrapped up for publication.

Another issue noticeable in the chart is the 'GPU idle overhead', comprising an initial period before the GPU receives any data, and a final period after it has sent back all of its results. Some of this time is taken up by subgraph partition splitters and joiners; some is due to implementation artifacts which can probably be optimized away; and some of if it are some final operations on a tiny amount of data, which are not scheduled to run on the GPU (but possibly could have).

4.3 GPU activity breakdown

Let us dig into the GPUs' activity with Figure 12, which breaks the GPU time down into the activities of I/O (over PCI/e) and Compute.

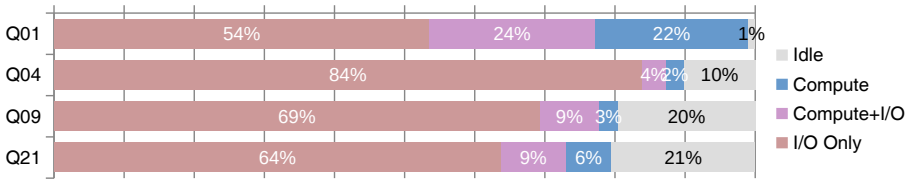


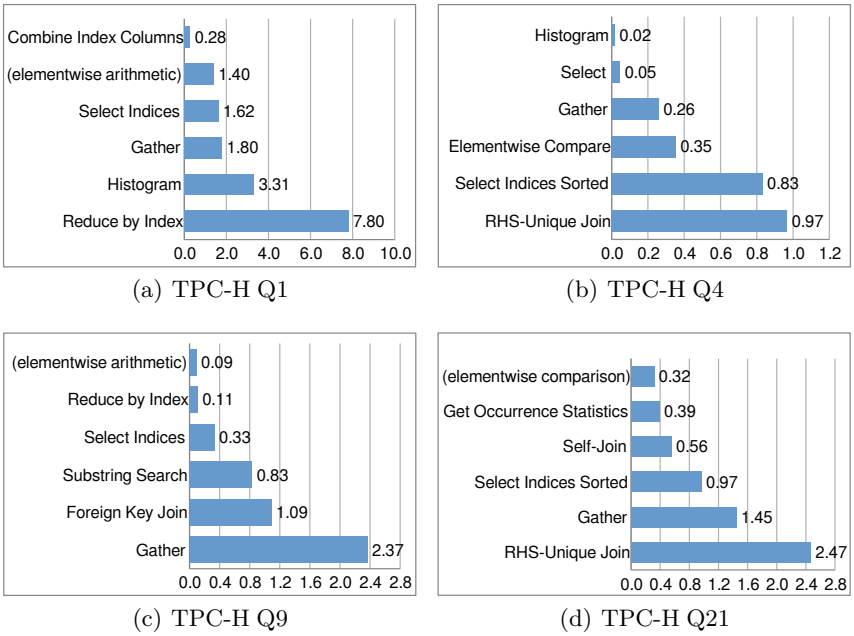
Fig. 12. GPU time breakdown (single-GPU, SF 1)

This illustrates very clearly what is the "bane" of discrete GPU; Most of the time is spent on nothing but I/O over the PCIe bus. This point is discussed in Section 6 below. An interesting question is how much would this disappear in a multi-query situation, where contention over the GPU's and PCIe bus would be high.

Note that Figure 12 shows some of the GPU time as entirely idle; this is an artifact of our implementation, due to over-conservative stream synchronization,

and can be reduced to a negligible level with some programming effort, but no degradation of performance elsewhere

Going another level deeper, let us consider the breakdown the GPU's Compute activity, presented in Figure 13. A query execution run involves a few dozens of technically distinct kernels, but for clarity of presentation we place them in several groups (e.g. elementwise arithmetic of all data types), and limit ourselves to the top six time consumers for each query. These take up between 92% and 98.8% of kernel execution time, making 6 a reasonable cutoff point.



(a) TPC-H Q1 (b) TPC-H Q4

(c) TPC-H Q9 (d) TPC-H Q21

Fig. 13. Top time-consuming operations (msec)

Space constraints preclude a discussion relating these time consumption distributions to the relevant queries. It is the authors' belief that, in general, such query-specific breakdowns of time by computational operation are lacking in many papers on DBMS performance enhancements, especially those involving GPUs — while they are an important guide for the researcher or engineer regarding what merits further optimization (or rather, circumvention).

4.4 Effects of increasing database size

We tested our frameworks with scale factors 1–8 (1 GB – 8 GB total size); these are not so high by today's standards, but our framework lacks a GPU memory management mechanism, and our GPU's available memory was just 3 GB. Fortunately, the sample points of SF 1,2,4,8 already allow some visualization of trends as the DB scales, in Figure 14.

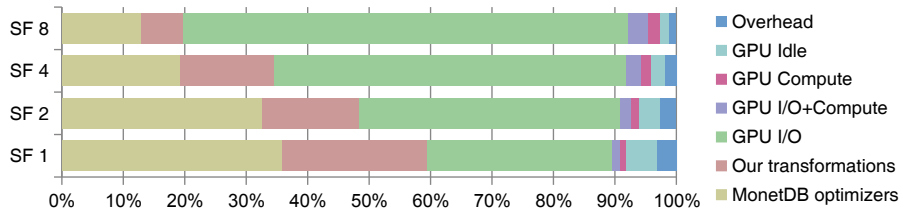


Fig. 14. TPC-H Q4 processing time breakdown with increasing DB size, GPU execution

Besides the obvious decrease in weight of the initial work on the plan relative execution proper, we note a further increase in the fraction of time spent on I/O only, i.e. the "bane of the discrete GPU" from Figure 12 becomes even more pronounced. On the other hand, we note an improvement in Compute-I/O overlap, as deeper computation nodes in the execution DAG tend to scale sub-linearly in their duration with DB size. Note we have taken TPC-H Q4 as the example, but the trends are similar in the other three queries.

5 Comparison with other work

Unfortunately, most frameworks using GPUs for query processing presented so far are not pairwise-comparable in performance: For some, no benchmark results are presented; others focus on transactions rather than analytics; some use hardware for which comparison is difficult; and some alter the TPC-H schema significantly (e.g. by denormalization). Most work surveyed in [4] falls into one of those categories (and there's *GPU-DB* [23], which uses a different benchmark — SSB rather than TPC-H). The *design* of these various frameworks is interesting to compare with, however, as some of them exhibit desirable features missing in this work (and vice-versa); unfortunately, space constraints preclude this.

A GPU-featuring processing framework which is close-to comparable is the MonetDB-based *Ocelot* [6]: It was benchmarked with a "hot cache", i.e. much of the data already in GPU memory [6, §5.3]; with an older GPU (GeForce GTX 460); and with a focus on portable implementation rather than maximum performance. Still, the Ocelot on-GPU time of TPC-H queries 1, 4 and 21 is 200, 30 and 300 ms respectively (for SF1), compared to our 16.8, 4.7 and 9.6 ms. We believe this factor of 7-30x in execution speed is mostly the result of Ocelot adhering closely to MonetDB's CPU-oriented execution plan on the GPU as well.

The few remaining frameworks, which *are* possible to compare against, were executed on hosts DBMSs that were rather slow to begin with. The interesting example in this category is the *Red Fox* framework [21]: Its Breß-et-al classification is similar to ours; it has a similar IR–Compilation–Second-IR chain design, and its motivation also goes beyond the execution of SQL query plans (see [21, §4.3]; again, we skip a more in-depth comparison of design features). Red Fox is grafted onto LogicBlox[8] as its host DBMS; comparing [21, Table 3] with Table 3, we note that MonetDB is 5.3-14.7x faster than the unmodified LogicBlox[8]. This seems to

[8] LogicBlox figures normalized by 0.85 to account for HW differences.

be a foil for Red Fox: Despite its solid speedup over its reference (7x average), it still only gets close to MonetDB speed, and is 5-12x slower than our framework on the four queries. One can also get a rough notion of how the execution plans differ by comparing the breakdown of execution time by computational operation: [21, Figure 10] compared to Figure 13 above.

Another example is Galactica [22]: Based on PostgreSQL, it also does not speed up execution to Monet-level speed, and is an order-of-magnitude slower than our framework (compare [22, §3.1.3] with Table 3 above).

Finally, a GPU-utilizing query processor named GPL (for "GPU Pipelining") has been described in the very-recently published [13] by Johns et alia. They take up the challenge of execution in chunks (a.k.a. tiles, or tablets), a concept first explored with Viriginian [2]. This reduces the size of materialized intermediary results and the overhead of communicating them via global memory. GPL also utilizes pipelining support in OpenCL 2.x, a theoretically promising approach. The paper does not report results for any TPC-H query with which we had tested, except Q9, and it does not make absolute results available for execution on a discrete GPU — making a proper, explicit comparison difficult. Still, the performance comparison it makes vis-a-vis Ocelot [13, Figs. 21, 22] shows a speedup of up to 2.5x, and typically under 1.5x. It thus seems to be the case that this new query processor is still significantly slower than the one presented in this work (and typically slower on an nVIDIA K40 than MonetDB on a typical dual-socket Xeon system).

6 Discussion and further performance enhancement

Our framework does not process queries quickly; and it is certainly not a good measure of the potential for processing performance with a GPU. This much is evident merely from observing how most of our execution time is spent idly waiting for I/O over PCIe. Thus, instead of resolving the shortcoming of discrete GPUs our work has merely masked it with performance improvements elsewhere. This is an unintended and somewhat ironic outcome: During the initial phases of our work, the picture was the exact opposite: 80%-90% of the time was being spent on GPU Compute. As we were laboring on providing the GPU with a better-parallelizable plan and data that is more easily accessible in parallel (as well as improving some of naive kernel implementations), total Compute time decreased further and further, eventually losing its dominance — so that squeezing it even more no would no longer yield much benefit. This phenomenon was discussed in [23], with the metaphor of a changing "balance of Yin and Yang".

Having put much effort into suppressing the "Yang" (improving GPU Compute time), three actions now come to mind for curbing the effects of the "Yin" (PCIe transfer):

I/O-Compute overlap via mapped memory: GPUs offer the feature of host-device-mapped memory, which triggers PCIe transactions on memory reads. Using these can allow computation on the GPU to begin immediately, with data transferred on on-demand; this approach is taken in [23] (although it does not present an I/O-Compute breakdown). It does have several drawbacks, how-

ever (less cache-friendly; PCIe transaction overhead; potential underuse of the bus).

I/O-Compute overlap via 'chunk-at-a-time' execution: Several DBMSes process data at the resolution of a column/table chunk rather than an operator-at-a-time on entire columns. In the CPU world this is a key feature of Actian Vector [24] (although more for the reason of fitting data in the CPU's cache). With regard to GPU-utilizing query processing frameworks, Virginian [4, 3] has employed it, but a more in-depth exploration of its merit and a case for its significance is the recent [13]. Considering our own results, even rough chunks of size, say, 1MB-4MB should already cut most of the initial idle period of the GPU waiting for data to arrive with nothing to work on. Chunk-at-a-time execution would also enable the use of chunk-level meta-data, potentially allowing a query processor to filter-out entire chunks rather than sending them to the GPU.
A refined variation of this approach is GPU pipelining, as in [13] (see Section 5). While not yet available in CUDA, it could theoretically allow for avoiding not only the initial idle time, but also much of the overhead inherent in manipulating chunks.

Data compression: In most real-world scenarios column data has many regularities and correlations, lower effective domain etc. — making it very amenable to compression; and this is true even for the somewhat artifical example of the TPC-H. Also, it just so happens we have a mostly-idle ALU-rich computational device to use for decompression on the receiving end. While the other two methods are limited in benefit by the amount of Compute Time (100% overlap), compression is limited the information inherent in the data and the GPU's ability to decompress effectively. Of course, such computation will itself contend with actual query operation application, so a different balance will need to be struck.

A fourth option we could have listed is using *bit weaving*, transferring columns one bit at a time; but we are skeptical of the utility in this approach, among other reasons because most of its potential benefit is subsumed by compressing the data.

Another important challenge in evaluating GPU-accelerated query processors is the scaling of bencmark schema. With only 1 GB of data overall, some queries take as little as 3 ms or less of actual computation time — with results potentially skewed by some minor inefficiency here or there. A larger scale also forces the more realistic setting of inability to hold all data in GPU memory, being limited on discrete GPUs compared to main system memory. For our framework, implementation of either *memory management* would obviously allow scaling beyond the equivalent of scale factor 10 for TPC-H queries, with no significant cost — even with 'operator-at-a-time' execution. The 'chunk-at-a-time' approach, mentiond above, automatically enables scaling to handle much larger benchmark data.

References

[1] Armbrust, M., Xin, R.S., Lian, C., Huai, Y., Liu, D., Bradley, J.K., Meng, X., Kaftan, T., Franklin, M.J., Ghodsi, A., Zaharia, M.: Spark SQL: Relational data processing in spark. In: Proc. SIGMOD. pp. 1383–1394. SIGMOD '15, ACM (2015)

[2] Bakkum, P., Chakradhar, S.: Efficient data management for GPU databases. NEC Laboratories America, Princeton, NJ, Tech. Rep (2012)

[3] Bakkum, P., Chakradhar, S.: Efficient data management for GPU databases. In: NEC Laboratories America, Princeton, NJ, Tech. Rep [2]

[4] Breß, S., Heimel, M., Siegmund, N., Bellatreche, L., Saake, G.: GPU-accelerated database systems: Survey and open challenges. In: Proc. BigDataScience. ACM/IEEE (2014)

[5] He, B., Lu, M., Yang, K., Fang, R., Govindaraju, N.K., Luo, Q., Sander, P.V.: Relational query coprocessing on graphics processors. Trans. DB Sys. 34(4), 21:1–21:39 (Dec 2009)

[6] Heimel, M., Saecker, M., Pirk, H., Manegold, S., Markl, V.: Hardware-oblivious parallelism for in-memory column-stores. In: Proc. VLDB. vol. 9, pp. 709–720 (2013)

[7] Kemper, A., Neumann, T., Garching, D.: HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In: Proc. ICDE (2011)

[8] http://www.logicblox.com/

[9] Luitjens, J.: Faster parallel reductions on Kepler (2014), http://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler/

[10] Manegold, S., Kersten, M., Boncz, P.: Database architecture evolution: Mammals flourished long before dinosaurs became extinct. Proc. VLDB 2(2), 1648–1653 (2009)

[11] MonetDB webpage, http://www.monetdb.org

[12] Neumann, T.: Efficiently compiling efficient query plans for modern hardware. Proc. VLDB 4(9), 539–550 (June 2011)

[13] Paul, J., He, J., He, B.: GPL: A GPU-based pipelined query processing engine. In: Proc. SIGMOD. ACM (2016)

[14] Power, J., Li, Y., Hill, M.D., Patel, J.M., Wood, D.A.: Toward GPUs being mainstream in analytic processing: An initial argument using simple scan-aggregate queries. In: Proc. DaMoN. p. 11. ACM (2015)

[15] Sidirourgos, L., Kersten, M.: Column imprints: a secondary index structure. In: Proc. SIGMOD. pp. 893–904. ACM (2013)

[16] Sitaridi, E.A., Ross, K.A.: GPU-accelerated string matching for database applications. J. VLDB pp. 1–22 (2015)

[17] Stonebraker, M., Hellerstein, J., Bailis, P.: Readings in Database Systems (the Red Book). 5th edn. (2015), http://www.redbook.io/

[18] The CUB library, http://nvlabs.github.io/cub/

[19] https://www.monetdb.org/Documentation/Manuals/MonetDB/MALreference

[20] The TPC Council: TPC Benchmark H (rev 2.17.1) (2014), http://www.tpc.org/tpch

[21] Wu, H., Diamos, G., Sheard, T., Aref, M., Baxter, S., Garland, M., Yalamanchili, S.: Red Fox: An execution environment for relational query processing on GPUs. In: Proc. CGO. p. 44. ACM (2014)

[22] Yong, K.K., Karuppiah, E.K., See, S.: Galactica : A GPU parallelized database accelerator. In: Proc. BigDataScience. ACM/IEEE (2014)

[23] Yuan, Y., Lee, R., Zhang, X.: The Yin and Yang of processing data warehousing queries on GPU devices. Proc. VLDB 6(10), 817–828 (2013)

[24] Zukowski, M., Boncz, P.: Vectorwise: Beyond column stores. IEEE Data Engineering Bulletin 35(1), 21–27 (2012)