Pipeline Group Optimization on Disaggregated Systems

Andreas Geyer, Alexander Krause, Dirk Habich, Wolfgang Lehner
Database Systems Group
Dresden, Germany
first.last@tu-dresden.de

ABSTRACT

While hardware disaggregation is considered the "next big thing" providing unique opportunities for database systems, the pipeline-based execution model is state-of-the-art in modern query engines on monolithic systems. Within this paper, we propose a lightweight way of adapting this pipeline-based model to disaggregated memory systems to soften the inherent overhead induced by arbitrary memory accesses. Instead of executing pipelines in strict isolation including a pipeline-local data transfer, we group pipelines with similar data access characteristics of concurrently running queries into *pipeline groups*. Each such pipeline group is then executed separately, but shared data across pipelines within each group is only transferred once from memory resources to compute resources and potentially re-used multiple times. This method dramatically reduces redundant data transfers and – in combination with a suitable caching strategy as well as a fast communication layer – increases the performance significantly in comparison to traditional pipeline-based execution of multiple queries.

1 INTRODUCTION

The pipeline-based query execution model is state-of-the-art in modern query engines [3, 5, 8, 9]. Within this model, pipelines are logical units of execution and each pipeline consists of multiple *pipeline-friendly* operators with a *pipeline-breaking* (sub-)operator at the end. While pipeline-friendly operators are able to produce an output before the end of an input data stream is reached, pipeline-breaking operators have to create an intermediate state and are only able to produce the first output after all elements of an input stream have been seen. Moreover, every pipeline consumes a set of data (either a base table, a column or an intermediate) and produces a set of data (usually another intermediate or – at the end – the result) in a format optimal for subsequent pipelines. Thus, every query execution plan consists of a set of pipelines and a list of pipeline dependencies (usually reflected as precedence relationships [8]).

In this context, the query optimizer generates the (estimated) cardinalities of individual pipeline fragments to come up with an optimal plan (considering join orders and table access primitives) – also including estimates for the final fragments of a pipeline determining the required size of the necessary intermediate. Taking both inputs, the pipeline dependencies as well as the estimated size of the output of a pipeline, into consideration, different pipeline execution schedules are possible. The scheduling and the management of intermediates specifically (when, where, for how long, ...) has been recently discussed in [8]. As they have shown, a *good* ordering of execution pipelines may lead to a memory-efficient query execution plan. Moreover, [9] refined the pipeline-based execution model to a chunk-based approach for multi-socket scale-up systems offering a high degree of parallelism with *non-uniform memory access (NUMA)* behavior. In this approach, the input data of a pipeline is partitioned into chunks, so that the chunks can be processed in parallel (intra-pipeline parallelism) and the scheduling ensures that worker threads work on NUMA-local data [9]. Additionally, [9] demonstrated that a single pipeline-at-time processing fully utilizes the whole system yielding the best query execution performance.

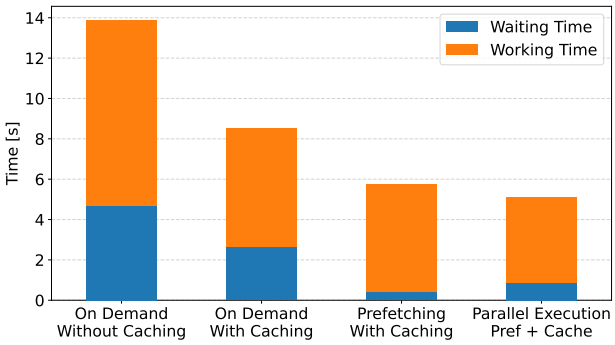


Figure 1: Optimization potential compared to state-of-the-art naïve pipeline processing. More details in Section 3.

Contribution: In this paper, we are taking the next step and investigate the pipeline-based execution model in the context of disaggregated memory systems. In general, hardware disaggregation is considered the "next big thing" [13] by separating potentially heterogeneous compute and memory components (usually in the form of network attached DRAM) in large data centers. This allows an independent scaling of compute and memory resources, but always requires the explicit transfer of data from memory to the compute resources. However, data transfer has been already identified as one of the biggest challenges even in the presence of high-speed interconnects and thus requires primary attention especially when running data-crunching database systems.

We mainly focus on two main contributions within this paper: First, to optimize the execution of concurrently running analytical queries, we introduce a lightweight optimization of grouping pipelines with similar data access characteristics, hence the name *pipeline groups*. In more detail, we continuously batch a set of queries, compile their pipelines with a state-of-the-art optimizer, and finally orchestrate the resulting pipelines based on their data access. Pipelines with shared data are executed together to avoid the redundant data transfer. The potential performance benefits of our approach are illustrated in Figure 1 where we execute just two

 13th Annual Conference on Innovative Data Systems Research (CIDR '23). January 8-11, 2023, Amsterdam, The Netherlands.

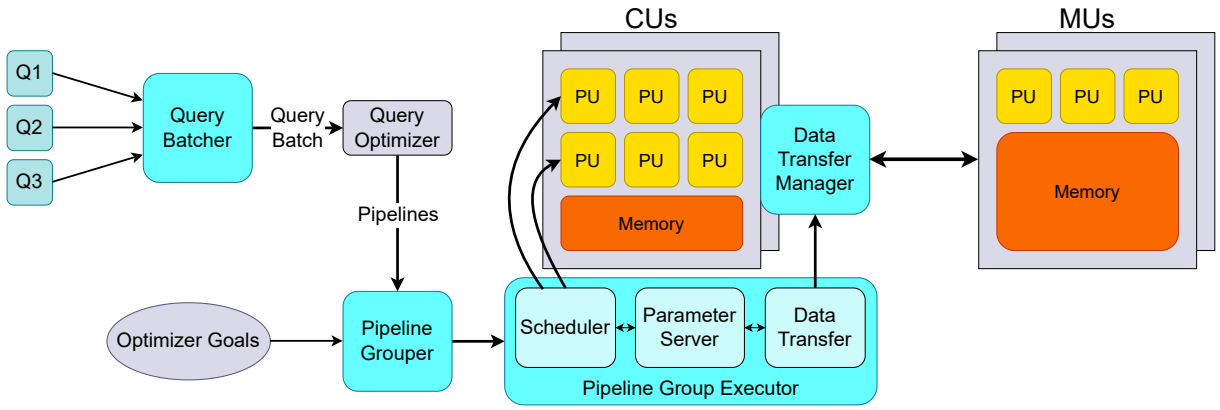


Figure 2: Envisioned system design based on a disaggregated hardware environment consisting of clearly separated compute units (CUs) and memory units (MUs). The MUs are responsible to store the primary business data, while the CUs provide the necessary compute capabilities for query processing. Due to the separation of CUs and MUs, the required data has to be transferred from the MUs to CUs for processing in this hardware environment. To optimize the execution of concurrently running analytical queries, our *pipeline group* optimizes the necessary data transfer and pipeline execution in a holistic way.

simple pipelines accessing the same data: the leftmost bar shows the time spent, when both pipelines are executed in sequential fashion and both fetch their data on demand (no data reuse). In all cases, the disaggregated memory systems are realized using two severs with RDMA. While one server acts as compute resource executing the pipelines, the second server only provides memory resources for base data. Second to left, we display naïve caching, where previously fetched data is reused for the second pipeline. Our novel approach and second contribution is shown with bars three and four, where we issue data chunk prefetching for the whole pipeline group. Obviously, reducing the amount of transferred data (caching) yields a higher benefit, because network bandwidth is always lower than the local memory bus. However, this effect can be enhanced further by applying our pipeline grouping technique (first contribution). Clearly, executing both pipelines simultaneously (fourth bar) leads to a slight increase in waiting time, since both have to stall for the arrival of data. However, this allows a better interleaving of data transfer and pipeline processing leading to the best overall runtime.

Outline: The remainder of the paper is organized as follows: In Section 2, we introduce our new *pipeline group* concept in more detail. Then, we introduce our proof-of-concept implementation including selected evaluation results highlighting the optimization potential of our *pipeline group* concept in Section 3. Finally, we discuss related work in Section 4 and close the paper in Section 5.

2 PIPELINE GROUPS

Resource disaggregation and a corresponding system design may adhere to a multitude of parameters. Thus, we start with a description of our anticipated system design. Figure 2 depicts a related sketch. To explicitly reflect the separation of compute and memory, we assume the existence of dedicated compute units (CUs) as well as memory units (MUs). While MUs feature high memory capacities (DRAM, NVRAM, ...) with a limited amount of compute resources, CUs provide a high amount of compute resources with a limited

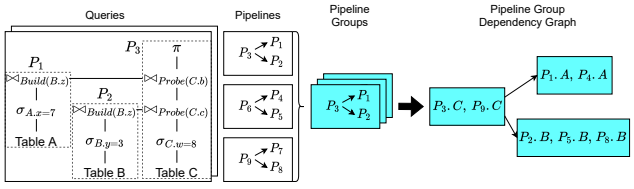


Figure 3: Example of a pipeline group schema.

main memory capacity. Thus, CUs are merely responsible for executing queries, managing the lifecycle of intermediates and feeding results to the clients. Base data has to be fetched from MUs.

In order to optimize the execution of concurrently running queries on such disaggregated memory hardware settings, our system design features a `Query Batcher` as central starting point (cf. Figure2). The task of this `Query Batcher` is to collect a set of queries and to trigger the joint execution. For this joint execution, every query within a query batch is turned into a separate pipeline execution graph (Figure 3) by a state-of-the-art optimizer. Generally, pipelines may be divided in two categories: (1) starting pipelines, without any dependency on intermediate data of other pipelines (cf. P_1, P_2) and (2) intermediate pipelines, which have to wait until some others have finished (cf. P_3, it depends on both P_1 and P_2). Obviously, the optimizer generates the data requirements for each pipeline, i.e. which tables and attributes have to be loaded for subsequent execution. The optimizer further annotates pipelines with their expected output cardinality allowing to estimate the overall memory consumption of individual pipelines.

Pipelines without dependencies or whose input requirements are met can be executed concurrently, as their processing scope is isolated by definition. Obviously, some pipelines may access the same attributes and blindly executing them agnostic to the context of concurrently existing pipelines within the same or within a query batch may result in redundant data access (which is OK for local

memory but cruel for remote memory locations). To overcome that, one of the optimizations is then to cluster pipelines with congruent or somehow overlapping data requirements into groups. Similar to the approach of group commits as proposed in [6], several data requests of pipelines with similar data characteristics are grouped together and executed like a group read. That means, intelligently forming and subsequently executing *pipeline groups* allows us to batch the access to the same data from MUs and thus reduce memory transfers (*transfer sharing*).

As illustrated in Figure 3, a global *pipeline group* dependency graph over all batched queries is constructed, which is subsequently used to guide the join execution by processing one *pipeline group* after another. To execute a *pipeline group*, the `Scheduler` has (i) to assign pipelines to compute resources and (ii) to trigger the *Data Transfer Manager* in order to fetch the required data from MUs. Depending on the concrete implementation, the `Data Transfer Manager` may need parameter for an optimal transfer. These parameters are provided by a `Parameter Server`. Nevertheless, handling *pipeline groups* imposes a set of challenges:

Building. Identifying which pipelines across batched queries should be clustered together in a group is not a straightforward task. The most crucial criterion may be the amount of overlapping data requirements following the main memory constraints of CUs. Memory consumption of a pipeline is not solely influenced by its output cardinality, but also intermediates, even if they are not materialized to the MUs. Thus, balancing the required data from MUs, expected peak memory consumption and the amount of pipelines in a group is just as important.

Scheduling. Allocating enough compute resources for all pipelines in a group has to consider several aspects. The first is accounting for intra- and inter-pipeline parallelism. Depending on the actual operator code, some pipelines may not exhibit internal data dependencies and thus allow for concurrent loop processing, e.g. via OpenMP. The second aspect is data availability. If the first data requirement of some pipelines differs, some pipelines may be started later than others to prevent network congestion and idle time of compute resources.

Caching. Eliminating redundant data accesses can improve both waiting and hence query processing time, as illustrated in Figure 1. Considering our system design, caching can be performed on a varying granularity: per pipeline-group to avoid redundant concurrent data fetching, per query batch (i.e. for all pipeline-groups of a batch) or even leverage historic information by identifying frequently used data over a certain amount of previous query batches.

Resource Adaptivity. Naturally, deferring the execution of pipelines to reduce network traffic as described above raises the question if resources should be suspended, e.g. to save energy or reduce the overall TCO. Hence, resource adaptivity implies the existence of a cost model, that incorporates the expected data parallelism for concurrent pipeline-groups, the latency of disabling and enabling certain resources as well as the possibility to cache base data and intermediates from finished pipeline groups.

Further, computation can be pushed to the storage or network layer by leveraging intelligent network cards, i.e. Smart NICs, FPGAs or through our notion of an MU, which already comes with an attached CPU. The identification of pushable code, like common predicates of pipelines is certainly a costly and time intensive task, which can be considered a form of Multi Query Optimization. In this paper, we focus on the building of pipeline groups and leave general operator pushdown for future work.

3 PROOF OF CONCEPT EVALUATION

To show the efficiency and applicability of our system design including the *pipeline group* optimization, we implemented a prototype from scratch in C++[1]. Our experimental evaluation for this paper was conducted on an emulated disaggregated memory system consisting of two servers, where Server $S1$ plays the role of a CU and Server $S2$ plays the role of an MU. While Server $S1$ – the CU – is equipped with four Intel Xeon Gold 6130 with up to 3.7 GHz, Server $S2$ – the MU – is equipped with four Intel Xeon Gold 5130 with up to 3.2 GHz. Both servers feature a Mellanox ConnectX-4 card with up to 100 GBit/s and are connected via InfiniBand. These cards (or RNICs) are directly connected to NUMA socket 0 of either server and thus, to cope with unwanted NUMA-effects, we use `numactl` to force thread and memory assignment to socket 0. Every experiment was repeated 5 times and we report the median runtimes.

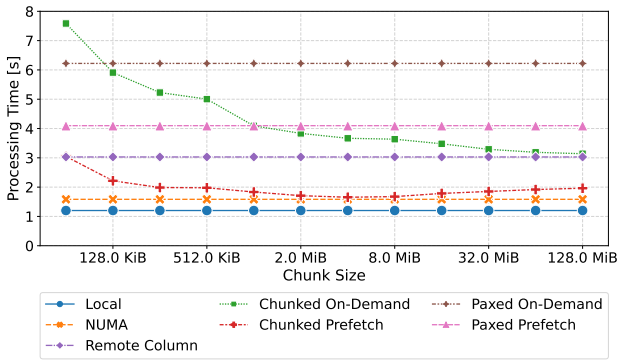
As communication layer, we leverage single-sided RDMA verbs (`RDMA_WRITE`) via Infiniband and multiple send (SB) and receive buffers (RB) on each server to perform well-known double- or multi-buffered data exchange. Whenever the requested data does not fit into a single send buffer, it is split into multiple smaller packages, that are enriched with meta data and then sent to the requesting server. We observed that a single sender/receiver thread is unable to saturate the available bandwidth and thus we employ one thread per buffer. To handle the disaggregation-simulation through RDMA, we need to copy received data packages from the RDMA buffer to another location, where it can then be consumed by the CU. Preliminary experiments revealed, that 512 KiB sized SBs and RBs yield the overall best performance for transmitting both smaller and larger data chunks, hence we use this size for our experiments, if not stated otherwise.

We experimented with columns containing 2 M, 20 M, 200 M and 400 M randomly generated `uint64_t` values each. However, while the size of the data set influenced the absolute numbers, the overall proportions did not vary greatly. Thus, we present the results for 200 M values or 1.5 GiB per column.

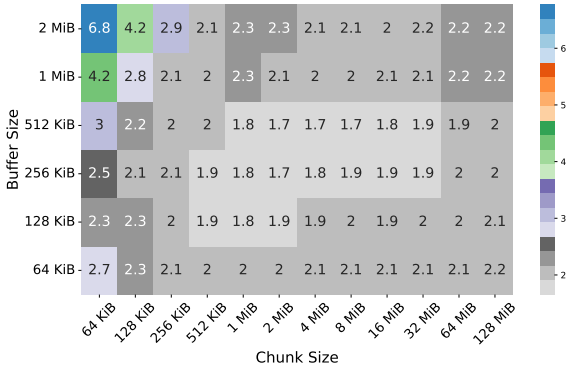
3.1 Single Pipeline-at-a-Time

The baseline of our experiments is the local execution of each pipeline. This means the data is completely held in the main memory attached to the socket of the working thread on the CU and therefore, this is the fastest way of accessing the data. Realistically, this is the fastest approach, since data cannot be streamed without any delay from a remote source, e.g. an MU, into the local cache of the CU. We then repeated this experiment in a NUMA setting, by pinning the processing threads to NUMA socket 0 and placing the data set on NUMA socket 1, i.e. forcing a NUMA hop during the processing. This setting gives us a good indication of the target performance for the RDMA measurements. Since both RNICs are connected to socket 0 of either system and the processing/transferring threads are also forced on socket 0 via numactl, we

[1] https://github.com/alexKrauseTUD/memoRDMA and https://github.com/alexKrauseTUD/dataProvider



(a) All approaches



(b) Heatmap of the pipeline runtime for the chunked approach with prefetching

Figure 4: Single-pipeline experiment with different chunk sizes, with pipeline selectivity of 50%, 512 KiB **buffer size, 2 RB, 2 SB and 1 Thread each.**

can interpret an RDMA transfer as a single NUMA hop but with higher latency.

As for the data transfer granularity, value- or tuple-at-a-time is undoubtedly not performant for accessing remote data as this would incur more overhead then the actually sent data. Thus, we use the naïve approach of requesting whole columns from the MU to serve as a baseline for subsequent experiments with more fine grained data transfer. After the request is sent, the CU waits until the whole column is transmitted and starts the pipeline processing right after the data is completely available in local memory. It is necessary to wait for the complete column, as there is no guarantee in which order the RBs are processed and thus we might encounter gaps in the transferred data during processing. Naturally, this implies a lot of waiting time for large columns.

For a more performant approach with less mandatory waiting time, the column is requested as a set of independent column *chunks*. These chunks can be of arbitrary size and hold the corresponding amount of data of one column. Therefore, the approach of transmitting the whole column at once is a special case of transmitting only one chunk with the size of the whole column. To find the best performing chunk size, we thoroughly test several sizes in the corresponding experiments. Chunking a column yields the benefit, that the pipeline processing can start earlier, since less data has to be transmitted. Further, this enables us to prefetch the next chunk, while the pipeline is busy processing the already received one.

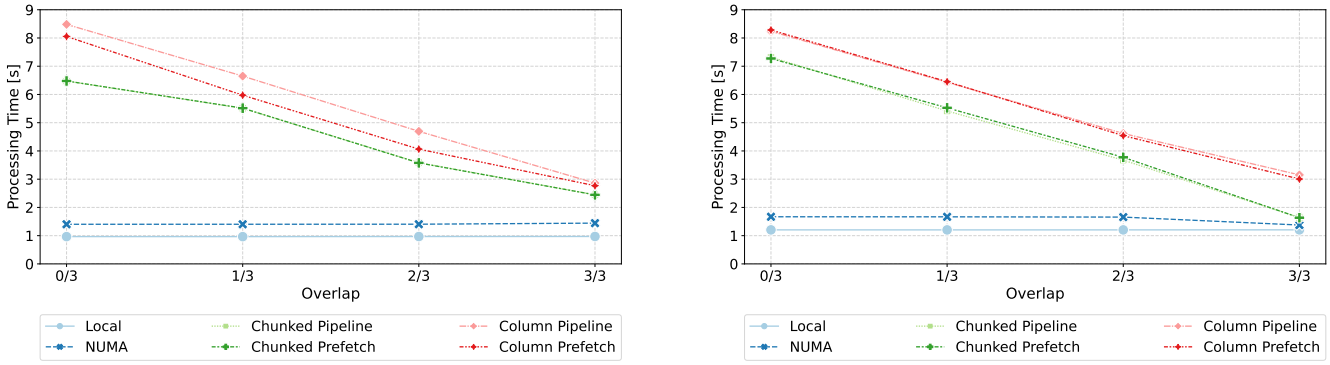
The actual prefetching is signaled every time, when the processing of a new chunk starts. A dedicated core of the CU is then activated to communicate with the MU and fetches the corresponding data into the local memory. We rebuild a mirrored image of the remote column and thus, the chunk is placed in the corresponding consecutive memory region, according to its offset. In an optimal setting, the processing of the local data takes longer than the copy procedure. In that case, no memory stalls would happen, since the new chunk is copied into memory, that is adjacent to the already locally residing data. If the local processing overtakes the fetching of the remote data, the processing core is sent to sleep via a condition variable and will be awaken, as soon as the chunk has been transferred completely.

After receiving any processable data, our pipeline iterates over it in a blocked scheme to achieve cache-friendliness. We identified that iterating over batches of 64 KiB (or 8192 elements, if the data type is `uint64_t`) yields the best pipeline performance. As a consequence, regardless of the availability of more data, it is split up into blocks of 64 KiB for faster processing. However, even though this is applied to the local (no data transfer) as well as the remote (data has to be fetched) experiments, it does not affect the transmission of the data, only the processing behavior of the pipeline. Consequently, if chunked data transfer is used with a chunk size smaller than 64 KiB, this also forces the block size to be equal to the chunk size.

Following the principle of *PAX (Partition Attribute Across)* as introduced by [2], we also investigate a PAX-like approach. In this transmission mode, we allow the CU to request multiple columns at once and the MU will store as much data of each column into a single SB as possible and all columns will have the same share in such a package. As an example: given a pipeline, that requires three different columns, an SB with the size of 1024 KiB and some meta data of 4 KiB per package leaves 1020 KiB for all columns to use. That means, every column can occupy 340 KiB per package. For simplicity, we will label this approach as paxed from hereon.

All these approaches allow for two different data fetching modes. The first is *On-Demand*, which means that the data is requested at the moment it is needed and thus reflecting the traditional approach when working on local memory where the data is immediately available. The second is *Prefetch*, where the first data request is submitted at the start of the pipeline, i.e. before the execution starts, and the next piece of data is already requested while working on the previously received one. The reason for this is to hide as much of the latency introduced by the data transfer as possible. For these experiments at most one chunk/paxed package was allowed to be prefetched.

Both On-Demand and Prefetch were tested on different pipeline types and we observe different results, depending on the complexity



(a) 4 pipelines executed sequentially with 4 threads for each pipeline

(b) 4 pipelines executed parallel with 1 thread for each pipeline

Figure 5: Two different pipeline group execution strategies with different amount of data overlap.

and selectivity of the respective pipelines. The experiments in this paper are executed for pipelines, that are built from the following, fairly simple query:

```
SELECT SUM(col2 * col3)
FROM data
WHERE col1 < n
```

In general, the results have shown, that more complex pipelines tend to hide more of the latency than simpler ones, due to the longer compute time. Nonetheless, we show our results for this fairly simple pipeline type to demonstrate the possibilities even on suboptimal workloads.

Figure 4a displays the influence of varying the chunk size on the overall pipeline runtime. The local, NUMA, remote column and paxed experiments are obviously not influenced by the chunk size and thus straight lines, since the data can be directly accessed. As expected, the local approach is unmatchable the fastest one and the on-demand data requests are much slower as their corresponding equivalent with prefetching. We derive a chunk size of 4 MiB as the local optimum from this experiment. That the chunked approach is outperforming paxed might seem surprising at first. The reason for this is that the most performant configuration of our RDMA implementation is a buffer size of 512 KiB, while larger buffer sizes perform significantly worse. As the optimal chunk size of 4 MiB implies, it is necessary to prefetch these 4 MiB of data to hide as much latency as possible. With the paxed approach we can only prefetch at most 512 KiB divided by the number of columns for every column, which is simply not enough data to keep the CU busy until the next paxed message arrives. A larger buffer size might solve this problem for paxed, but at the same time worsens the performance of the underlying RDMA layer. We conclude that the paxed approach does not work well with our transport layer and is thus excluded from further experiments.

Figure 4b shows the pipeline runtime as a heatmap of the chunked approach with prefetching, depending on a combination of the chunk size and buffer size. The result is that the buffer sizes of 256 KiB and 512 KiB perform best. For the buffer size of 512 KiB the chunk sizes of 2 MiB, 4 MiB and 8 MiB perform similar with 4 MiB having the absolute best performance. Following these results, the buffer size of 512 KiB is fixed for the next experiment and a chunk size between 1 MiB and 16 MiB is further evaluated.

3.2 Pipeline Group

We now leverage our insights from the single pipeline experiments and evaluate the behavior for pipeline-groups, as introduced in Section 2, executed concurrently with multiple threads. Due to space constraints, we will explain all setups but only discuss experiments ② and ④ in detail and only briefly summarize ① and ③. The first experiment ① executes n pipelines sequentially, i.e. with only one thread working on the pipelines. This trivial execution is expectedly also the slowest one. Experiment ② executes n pipelines sequentially but with n threads working on a single pipeline concurrently. This follows the approach from [9] where there is no pipeline parallelism, but each pipeline is executed with the available resources. The third experiment ③ executes n pipelines with $n/2$ pipelines in parallel and two threads working on each pipeline. This is a hybrid approach between the second and the fourth experiment. The fourth experiment ④ executes n pipelines in parallel with one thread per pipeline. This approach is orthogonal to [9] with full inter-pipeline parallelism, but no intra-pipeline parallelism.

The target is to group pipelines with similar or equal data needs together to minimize data transfer and optimize data requests. For such a pipeline group it is possible to request all necessary data upfront and therefore, request columns only once, even if they are needed for several pipelines. Additionally, it enables to prefetch the data before even entering the pipeline execution. Hence, we need to slightly adapt our labeling for Figure 5. For a single pipeline execution (cf. Section 3.1) "prefetched" meant that the pipeline decides whether and when to request data. For a pipeline group "prefetched" means that the group optimizer decides when data requests are executed and "pipeline" means the pipeline decides. As outlined in Section 2, we now cache the requested data during the processing of the whole pipeline-group.

Figure 5a depicts experiment ② for $n = 4$. If the pipelines in a group exhibit no overlap (0/3), the execution time is significantly longer compared to local and NUMA execution. No overlap implies the transmission of 12 individual columns, each with 1.5 GiB of size.

This result also represents the traditional, isolated execution of 4 pipelines. In this case the same three columns would be transmitted for each pipeline execution again. With increasing overlap, both lines for local and NUMA stay on the same constant level. This is due to the fact, that for these approaches the data is already locally available and the local memory controller is not saturated with the data access. For the remote approaches on the other hand, the amount of overlapping columns has a big impact on the performance. The overlap of 1/3 means that each of the 4 pipelines share 1 column request. This allows to transmit only 9 individual columns and therefore, save a total traffic of 4.5 GiB. For an overlap of 2 and 3 columns the performance increases even further. With a complete overlap of 3 columns for the 4 pipelines the data traffic is down to a total of 3 columns and therefore, 4.5 GiB to be transferred. Nonetheless, we can still not reach the performance of the experiment with one NUMA hop. This is due to the fact that the data is much faster processed with 4 threads than it is transmitted.

In Figure 5b the corresponding graph for experiment ④ is shown. Again, we observe increasing performance with an increasing overlap of shared columns. For this experiment the lines for local and NUMA drop slightly for full overlap. We argue that this effect stems from a better exploitation of the hardware prefetcher, since all threads work on the same data and thus fewer cache thrashing occurs. It is worth mentioning that the absolute times for local as well as for NUMA are slightly worse than for experiment ②. This follows the results of [9] where it is shown that executing pipelines in parallel is less beneficial than executing the pipelines sequentially but with all available resources. For the remote approaches it is evident that for full overlap the chunked approach gets very close to the desired NUMA performance. As an interesting note, remote chunked prefetch with full overlap is slightly faster than NUMA with less than full overlap.

Both Figures 5a and 5b show that the implemented prefetching is not beneficial for the chunked approach. Experiments ① and ③ are not displayed as both performed worse than ② and ④. For experiment ① this is expected but for experiment ③ a small surprise. We found that the design of experiment ③ amplifies the observed negative effects of experiments ② and ④ and thus is not a favorable execution mode. All experiments were also executed with more threads and more data, however we observed the same behavior as for the already presented experiments and thus omit these numbers.

We could show that our approach of grouping similar pipelines together is very beneficial if there is some overlap in the needed columns. One of our configurations with full data overlap could even approximate the performance of NUMA with one hop, but with partial overlap. We anticipate that with more optimizations, e.g. in the RDMA layer, our chunked approach is able to compete with one-hop NUMA. On the other hand, it is also shown that no grouping of any kind results in a serious performance degrade.

4 RELATED WORK

Disaggregated systems revolutionize the design and architecture of modern database systems and thus database researchers have just started to investigate the potential implications for such a novel hardware model. For example, [13] discusses the general impact and infer a new architecture as well as database primitives. We fully agree that disaggregation leads to an alteration of traditional query handling and thus we focus on optimizing an already well known processing model to cope with the new disruptive hardware trend.

This goes hand-in-hand with Teleport [14]. The authors observe that the high network latency of 'remote' accesses is impacting data intensive systems and thus opt for compute or operator push-down. We see this as a potential future but orthogonal optimization that can be incorporated into our *pipeline group* concept. Moreover, there already exist system prototypes like LegoOS [11], PolarDB [4], Farview [7] and more emerge. LegoOS tackles the operating system side for steering and controlling the actual hardware components, which is an extremely interesting feature for elasticity, but orthogonal to our proposed concept of pipeline groups and prefetching. PolarDB – very similar to our architectural blueprint – plans with separate compute nodes but attributes the remainder of the resources to individual pools. We argue that having a dedicated MU with individual compute resources yields its benefits. Traditional NUMA systems followed the trend of moving the computation to the data, similar to the Near Memory Processing paradigm. With our dedicated MUs, we preserve the opportunity of optimizations as, e.g. operator push down. Farview's on-demand provisioning of compute nodes paired with the FPGA-controlled storage serves as a general inspiration for our work. However, Farview considers the execution of individual pipelines, which is contrary to our *pipeline group* approach which tries to co-execute pipelines with overlapping data access requirements and can thus exploit data caching and prefetching for a given query batch. Hence, our *pipeline group* approach follows the general idea of cooperative scans [10], but it is adopted to fit to the architecture of disaggregated systems.

On the one hand, recent work also has just shown the viability of CXL-attached main memory [1]. Our prototype implementation is currently based on one-sided RDMA verbs, but our memory access layer is already prepared to also work with memory via CXL as soon as we have access to corresponding hardware. On the other hand, DFI [12] is a framework to efficiently exploit high-speed networks, such as Infiniband. They show that adding an abstraction layer on top of RDMA verbs does not impose a significant performance degrade. However, their experiments are tailored towards tuple-based data processing, whereas we focus on column- or batch-oriented data transfer.

5 CONCLUSION

In this paper, we introduced our vision of a DBMS-friendly system architecture for disaggregated hardware with the separation into CUs and MUs. Based on this architecture, we see the biggest challenge in optimizing the data transfer over the network layer between MUs and CUs. For this, we investigated the state-of-the-art pipeline-based execution model of modern query engines. With a selection of experiments, we substantiated our claim that the state-of-the-art pipeline model is not working that well on disaggregated hardware systems. To overcome that, we have shown that there is a lot of optimization potential when grouping pipelines with similar data access characteristics together. This optimization can significantly reduce the redundant data transfer via network and therefore, increase the overall performance of the system. With simple optimizations for the pipeline group execution, we already

came close to the performance of a standard scale-up system with NUMA distance between the data and the processing threads.

Future Research Topics

In the future, we will fine-tune our concept for different optimization goals and we expect to increase the performance further. Within our shown system architecture in Figure 2, we have already shown several components that are necessary for our *pipeline group* approach. Nonetheless, we are currently only at the beginning of our research for several of these components. For the `Query Batcher`, the `Pipeline Grouper`, as well as the components of the *Pipeline Group Executor*, we used rather basic approaches and they will be extended as follows:

Query Batcher. At the moment, we imply a strategy of batching all queries arriving in a defined time. However, this is only the most simple approach and there is a multitude of other strategies possible. It might even be necessary to find a way to adapt the batching strategy depending on the workload.

Pipeline Grouper. Similar to the `Query Batcher`, we also used a rather simple approach for this component and grouped the pipelines by their highest overlap in data need. With growing workloads and more queries being processed, this simple strategy might not be applicable anymore or needs at least to be refined. However, also other strategies for grouping the pipelines not only or not at all on their data need might be relevant for example for better load balancing.

Pipeline Group Executor. This component has already some work done with our research on different data transfer techniques presented in this paper. Although, there is still room for improvement as already mentioned throughout this paper. Additionally, topics like scheduling or work as well as data placement are not researched by us at all at the moment and therefore, leave a lot of possibilities open.

These three components will form the core of our next research steps. However, there are also a lot of other possible improvements and steps. Therefore, feel free to reach out to us for a discussion of proposed pipeline group topic.

REFERENCES

[1] M. Ahn et al. Enabling CXL memory expansion for in-memory database management systems. In *DaMoN*, pages 8:1–8:5, 2022.
[2] A. Ailamaki et al. Weaving relations for cache performance. In *VLDB*, pages 169–180, 2001.
[3] P. A. Boncz et al. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, 2005.
[4] W. Cao et al. Polardb serverless: A cloud native database for disaggregated data centers. In *SIGMOD*, pages 2477–2489, 2021.
[5] P. Damme et al. DAPHNE: an open and extensible system infrastructure for integrated data analysis pipelines. In *CIDR*, 2022.
[6] R. Hagmann. Reimplementing the cedar file system using logging and group commit. In *SOSP*, page 155–162, 1987.
[7] D. Korolija et al. Farview: Disaggregated memory with operator off-loading for database engines. In *CIDR*, 2022.
[8] L. Landgraf et al. Memory efficient scheduling of query pipeline execution. In *CIDR*, 2022.
[9] V. Leis et al. Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age. In *SIGMOD*, pages 743–754, 2014.
[10] L. Qiao et al. Main-memory scan sharing for multi-core cpus. *PVLDB*, 1(1):610–621, 2008.
[11] Y. Shan et al. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *USENIX ATC*, 2019.
[12] L. Thostrup et al. DFI: the data flow interface for high-speed networks. *SIGMOD Rec.*, 51(1):15–22, 2022.
[13] Q. Zhang et al. Rethinking data management systems for disaggregated data centers. In *CIDR*, 2020.
[14] Q. Zhang et al. Optimizing data-intensive systems in disaggregated data centers with TELEPORT. In *SIGMOD*, pages 1345–1359, 2022.