Parallel Query Processing for ViDa

Julien Ribon
EPFL
julien.ribon@epfl.ch

January 28, 2016

Abstract

As the size and heterogeneity of data increase, traditional database systems can no longer scale and new innovative techniques are required to provide higher scalability. For database systems to support querying at a larger scale, they must exploit modern hardware and enable highly optimized, parallelized query execution. The ViDa system offers optimized query processing capabilities by using adaptive, just-in-time operators. ViDa uses an efficient compilation strategy to translate each incoming query into compact and efficient machine code, while aiming at good code and data locality, as well as predictable branch layout. Yet, ViDa does not leverage parallelization opportunities that lie in the query plan. In this work, we design and implement a simple, yet powerful parallelization infrastructure in order to introduce intra-query parallelism in ViDa. We also carry out experiments in order to validate our approach, which show clear performance improvements compared to sequential query processing.

1 Introduction

The ongoing data explosion is leading to an increasing volume and heterogeneity of the underlying data that needs to be analyzed. In this context, traditional database systems become an obstacle to data analysis, as integrating and loading data into these databases is quickly becoming a bottleneck in face of massive data volumes and increasing number of data sources. Moreover, queries in this systems are often ad-hoc and use pre-cooked operators that are not adaptive enough to optimize data access.

ViDa is a dynamic, fully adaptive system that reads data in tis raw format and processes queries using adaptive, just-in-time operators. The key idea behing ViDa is the use of *data virtualization*, i.e. abstracting data out of its form and manipulating the data regardless of the way it is stored or structured. Even though ViDa has shown promising results in handling raw data files [1], however, ViDa does not yet leverage any parallization techniques to process the query plan. Nearly all database systems exploit multi-core architectures for inter-query parallelism, but as the number of core available on modern CPUs continue to increase, intra-query parallelism will become even more important.

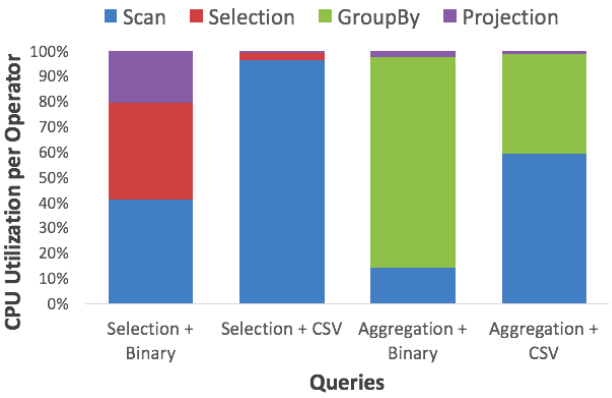


Figure 1: *Profiling Results*

In this work, we exploit intra-query parallization opportunities by investigating, at the query plan level, how to orchestrate operators in parallel. While some operators like projection and selection are straightforward to implement, higher optimization potential lies in other operators, such as joining and aggregation. On one hand, we extend the query processing engine of ViDa in order to enable parallel execution of queries by using state-of-the-art techniques. Data-intensive queries can benefits from light-weight horizontal partitioning by distributing queries across different portions of the data, while computational-intensive queries could be parallelized by partitioning operators or sub-parts of the query plan across multi-core CPUs. In particular, we explore techniques used to improve vectorized query execution [6, 7], where entire query pipelines are executed in parallel rather than individual operators. In order to investigate how parallel query execution can be performed in the ViDa system, we use code profiling, as an analysis mechanism, to identify optimization bottlenecks of generated code and detect potential enhancements for the query processing engine. Profiling results are shown in Figure 1. These results shows the portion of CPU utilization (in percentage) per query operator for four different queries. The goal of these use-case queries is, first, to point out the key challenges to run operators of the query plan in parallel and, second, to provide a set of queries to be used in experiments in section 5. As we can see, the scan operator always account

for a significant portion of the CPU time. In case of CSV data file, the scan operator even account for more than half the CPU time and, as a result, group-by and filter operators have more impact on CPU utilization when queries are run on binary data. For both types of query, we expect a linear speed-up in their execution time as most of their operators can fully benefit from intra-query parallelism. Finally, it is important to note that these results were obtained by varying the number of operator in each individual query. Hence, we rewrite the same query several times (i.e. removing and adding operators), such that the time spent in each operator can be estimated by looking at the difference in execution times.

After identifying key challenges and optimization opportunities, we develop a framework that allows operators to be executed in parallel and queries to be distributed across different portions of raw data. This framework supports parallel execution of selection queries (i.e. containing predicate filters) and, to some extent, aggregation queries. Our experiments validate our approach and show clear improvements compared to ViDa's original, sequential execution of query plans.

The rest of this report is organized as follow. In chapter 2, we present earlier work done in parallel databases as well as modern query parallelization techniques. This chapter also provides the reader with some basic concepts and background knowledge. In chapter 3, we briefly describe the query processing engine of ViDa. Chapter 4 describes the structure and implementation of our parallel infrastructure that allows queries to be run in parallel. Chapter 5 presents experimental results based on use-case queries. Finally, we suggest possible further enhancements for our parallel infrastructure in chapter 6 and end this report with some concluding remarks in chapter 7.

2 Related Work

Query plans are ideally suited for parallel execution because they are composed of uniform operations applied to streams of data [9]. The traditional way to execute these algebraic plans is the iterator model, sometimes also called Volcano-style processing [10]: Every physical algebraic operator conceptually produces a tuple stream from its input, and allows for iterating over this tuple stream by repeatedly calling the next function of the operator. As a consequence, multiple operators can be combined into a highly parallel dataflow graph, such that one operator consumes the output produced by another.

By chaining operators together and considering the dataflow as a vector of tuples, pipeline parallelism can be used to allow operators to work in series [4, 8]. Each pipeline then performs a group of operators, where vectors of tuples are consumed as a whole as soon as they have been produced. Furthermore, by partitioning the input data among multiple processors, an entire pipeline can be split into many independent subpipelines, each working on a certain portion of the data. This technique, called partitioned parallelism, distribute tuples among several processes such that entire pipelines can be executed independently from each other. In the design of our parallel infrastructure, we exploit this kind of parallelism as Figure 4 illustrates.

Different levels of parallelism can therefore be applied to query processing:

- *inter-query parallelism*, where several independent queries are executed concurrently at different sites;
- *intra-query, inter-pipeline parallelism*, where independent groups of operators belonging to the same query are run concurrently;
- *intra-query, intra-pipeline parallelism*, where multiple processors work together to compute the same group of operators.

In this work, we focus essentially on the later.

Much works have been done in the field of parallel databases, covering subjects such as data partitioning, load balancing, parallel operators, or parallel query optimization. Important research efforts were devoted to parallel algorithms for various operations such as sorting and join. Parallel sorting is widely used to facilitate other operations such as merge join, duplicate elimination, and aggregation functions and, thus, has received particular attention [12, 13, 14]. Join is another frequently used operation. Hence, a comparable effort has be expended in the development of efficient join algorithms [15, 16, 17], especially in the case where two very large relations must be processed. Among many algorithms, the well-know Grace hash join [18] exploits parallelism by first partitioning the data into buckets using a hash function and then applying smaller join operations in parallel on independent buckets. We can also find significant works that tackle other issues from parallel database systems, such as data skewness [19, 20] or parallel query optimization in relation database systems [11, 21]. Load balancing and data-skew avoidance play a central role in parallel architectures, as data must be evenly partitioned among processors in order to avoid bottlenecks and to benefit from the highest degree of parallelism.

More recently, modern parallelization techniques have emerged during the last decades. Morsel-driven query processing [4] allows the degree of parallelism to be elastically changed during query execution, while memory scan sharing [5] partitioning queries into batches such that the memory footprint for each batch can fit into a cache. In [8], the authors developed an in-memory column-store databases, where the query

is considered as a whole and translated into a single function. This allows highly efficient CPU utilization, minimal materialization, and (parallel) execution in a single pass over the data for most queries. Similarily, the HyPer main-memory database management system [3] uses a novel compilation strategy that translates a query into a compact and efficient machine code using the LLVM compiler framework, which already supports intra-query parallelism by partitioning the input of operators, processing each partition independently, and then merging the results from all partitions. This is equivalent to horizontal partitioning in relational database systems. We use this approach to introduce intra-query parallelism in ViDa.

3 The ViDa System

Data management must become a lightweight, flexible service, instead of a monolithic software centering around the status quo of static operators. ViDa is a novel data management paradigm offering just-in-time data virtualization capabilities over raw data sources [1, 2]. The key idea of ViDa is that data is left in its raw form and is treated as first-class citizen in the system. ViDa envisions transforming databases into "virtual" instances of the raw data that can be access, manipulate and exchange data independently of their physical location, representation or resources used. Hence, data analysts build databases by launching queries, instead of building databases to launch queries. To efficiently support diverse data models like tables, hierarchies and arrays, ViDa employs code generation to adapt its engine's operators and internal data layout to the data models, as well as to operate over raw datasets and avoid data loading.

Figure 2 illustrates a high-level description of ViDa. Incoming queries are translated into an internal "wrapping" query language or expressed directly in it to enable accesses across data models, using monoid comprehension calculus [23]. The entire query execution phase is monitored by ViDas optimizer, which takes runtime decisions related to raw data accesses. The optimizer is responsible for performing the query rewriting and the conversion from logical to physical query plan. In the physical query plan, the various abstraction layers of the database engine collapse into highly efficient machine code. Dataset descriptions are used to adapt the generated code to the underlying formats and schemas. In this work, we extended the JIT query executor with our infrastructure described in section 4 while inducing as less modification to the core engine as possible.

To execute queries on raw datasets, ViDa requires an elementary description of each data format, called *plug-in*. Data files follow a variety of formats. Typical examples are CSV, XML, JSON, but there is also a variety of ad-hoc and domain-specific formats. In the context of ViDa, we identify widely-used data formats and define a minimal grammar that is sufficiently rich to describe the raw data structure for upper layers of the query engine. As described in the next section, we modify the plug-in of certain data types to distribute and parallelize the scan operator in a simple, yet effective way.

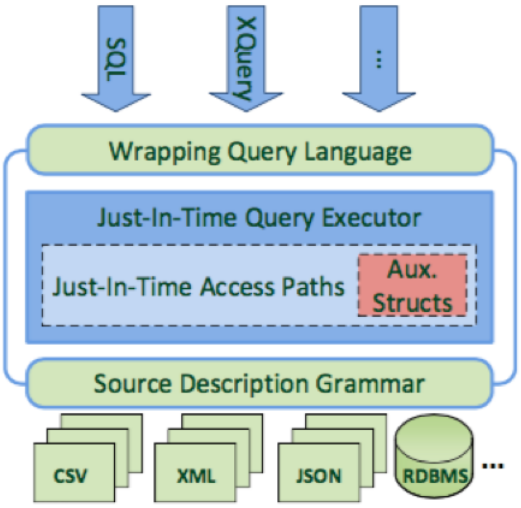


Figure 2: *ViDa's Architectural Components*

The operators in ViDa attempt to pipeline data when possible. For instance, access paths of ViDa (i.e. the scan operators) do not materialize any intermediate data structures at early processing stage, and no "database page" or "data column" has to be built to answer a query. Instead, data bindings retrieved from each tuple of a raw file are placed in CPU registers and are kept there for most of the query processing. The idea is to apply as much operation as possible on a vector of tuples before discarding that data from CPU registers and last level caches (LLC). If an operator has to explicitly materialize part of its output, an output plug-in is called to generate the code materializing the data in a specific layout. While designing our parallel infrastructure, we keep this concept of vectorized pipeline execution in mind and parallelize not individual operators but, instead, entire pipelines bounded by plug-ins and materialization phases.

4 Parallel Infrastructure

This section presents the structure, goals and boundaries of our parallel infrastructure. The design of the system architecture, as described here, was inspired from related work done in [22].

In our parallel infrastructure, queries are instanti-

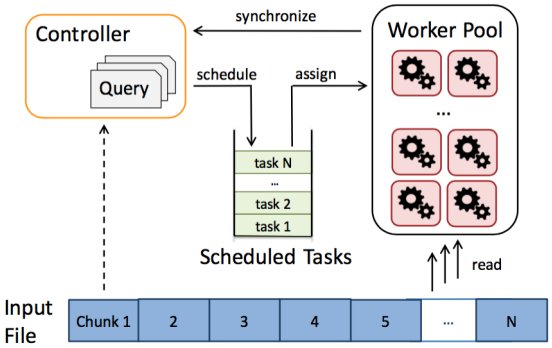


Figure 3: *Parallel Infrastructure*

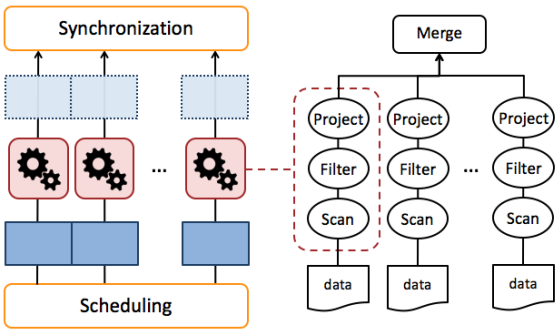


Figure 4: *Parallel Pipeline Execution*

ated individually by a *controller* on top of a farm of processing entities, called *workers*. These workers are organized by the controller to form a dependency graph between *tasks*, holding the user query, and *data chunks*, representing in-memory pieces of the input file.

The system is composed of the following components:

- *Controller* inspects the user query, splits the input file into data chunks, schedules independent tasks (one per data chunk) and inserts them into a queue. The controller is also responsible to create and synchronize workers.
- *Scheduled tasks* contain a function representing the query and additional metadata about data chunk and query result. Tasks are used as a communication mechanism between the controller and workers.
- *Workers* are identical threads that handle tasks. Each worker runs one sub-query at a time on a given input chunk and marks the task with a timestamp once completed.
- *Data chunks* represents portions of the input dataset and, in case of materialization, intermediate results produced by workers.

Figure 3 illustrates these different system components and shows how they interact with each other. In the remainder of this chapter, we explain in more details how the workflow between components takes place and what are their specific roles.

4.1 Controller

The controller is the main program of our system. Its purpose is to abstract the details and complexity of the back-end infrastructure, and to make easy for users or developers to execute a query in parallel. The controller component is handling one query at a time. This means that it is not yet possible to instantiate multiple queries at the same time and take advantage of inter-query parallelism. This can however be easily extended by including additional information inside the task objects, such as a query or program identifier. Once the query and input data are provided, the controller schedules a set of tasks inside a queue. This allows the controller to individually keep track of each part of the parallel query execution. Next, the controller initiates a group of workers that will handle the tasks. For the sake of simplicity, we define the number of workers as being equal to the number of tasks and data chunks provided by the controller. Hence, a worker is responsible for exactly one task and run (part of) the query on top of a single data chunk. Once the parallel execution is finished, the controller gathers all intermediate results, merge them together and returns the final result back to the user. Finally, the controller delete completed tasks from the queue and prepare the system to receive the next query.

4.2 Scheduled Task

As explained in the previous section, tasks are initially created and scheduled in a queue by the controller itself and are then assigned to and completed by a group of workers.

A task represents a query (or part of a query) that must be executed on a fraction of the input data. Therefore, the mandatory information that must be contained inside a scheduled task is the query itself together with the data chunk on which that query must be run. Still, we think that a task object must at least contain the following information: a unique identifier to distinguish individual tasks; a reference to an input chunk containing data on which the query must be executed; a function name and body (or reference pointing to a code location) containing the query; the task status, e.g. "pending", "in-process", or "completed"; and (eventually) a reference to a materialized output chunk containing intermediate results returned by the sub-query. In our case, the task status is implemented using two timestamps, i.e. *start time* and *end time*: the starting timestamp is set as soon as the task is assigned to a specific worker, while the ending timestamp is set once the worker has completed the task. If both times-

tamps are empty, this means that the task has been scheduled but not yet undertaken by a worker.

```
struct task {
  int id;
  int chunk_size;
  time_t start_time;
  time_t end_time;
  Function *query;
  ExecutionEngine *engine;
  RawOperator *last_operator;
};
```

The above code shows how the task object is implemented in our parallel system. In addition to the above fields, our task object contains a reference to the LLVM execution engine. The need for this field is explained in more details in the following section. Also, note that the reference to the output chunk is replaced by a reference to the last operator of the query pipeline.

4.3 Worker Pool

A possible analogy is to compare the worker pool as a reservoir full of processing entities, where the controller endorses the role of a tap that regulates the flux of entities allocated to the current query.

Hence, a farm of identical entities, known as workers, is present at the back end of the system. Workers are independent threads that are organized in a shared-memory architecture. Because all workers are both independent and identical, they can be arranged in any order and follow any structural pattern. Figure 5 illustrates the system architecture from the point of view of a single worker. The worker context is always limited to a unique task that corresponds to an input data chunk, a function, a reference to the execution engine, and eventually a materialized output data chunk. This elegant design allows us to completely decouple the implementation of our parallel infrastructure from the JIT query execution performed by ViDa: workers have a common API that is completely independent of the original query plan.

The execution workflow of a worker is as follow. First, a worker gets a task from the queue. Using the function reference, the worker retrieves the function from the LLVM context, which represents the query to be executed in parallel. Second, the worker runs the code over the tuples contained in the input chunk by invoking the function with the execution engine. Third, the worker either returns projected tuples directly to the controller, or materialize results in memory for later merging, as shown in Figure 6. Finally, each worker marks its task as completed and synchronize with the controller.

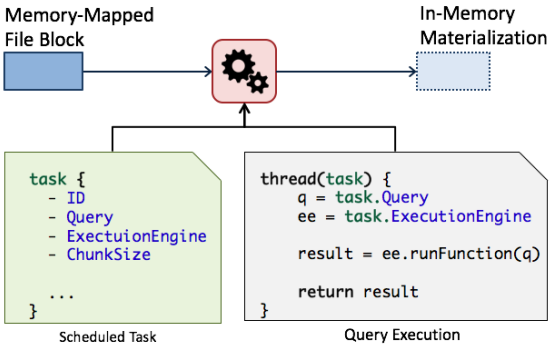


Figure 5: *View of a single Worker Thread*

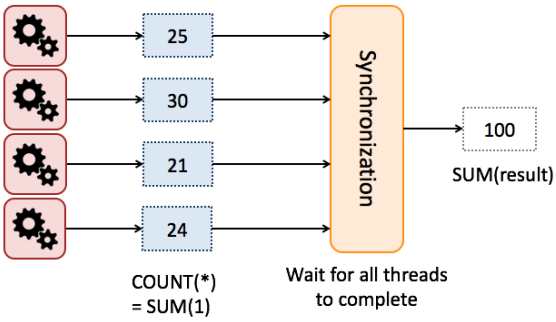


Figure 6: *Example of Merging Projections*

Note that two different workers always work on independent input chunks and write intermediate results into different memory blocks. Another approach would be to share the same in-memory output block among several workers. However, this approach would require locking mechanism that could induce a significant synchronization overhead. For this reason, we prefer to let the controller take part in the query execution and merge intermediate results stored in multiple memory blocks.

4.4 Data Chunk

As explained above, input data chunks must be prepared prior to parallel execution. This is done by the controller that splits tuples of a potentially large input file among multiple chunks. The main reason behind file chunking is to obtain a high degree of parallelism and allow a maximum number of threads to work concurrently and independently on the same query. In this manner, each worker handles a small portion of data, avoiding single point of contention when scanning.

Scanning the input file in parallel is a crucial step that accounts for a significant portion of the query execution time [5], especially when dealing with raw data files. Ideally, when scanning, input tuples must be grouped according to group-by and/or join attributes,

such that items having the same hash value end up in the same data chunk. However, in our case, we prefer to differ this grouping phase at a later stage (when merging) in order to avoid unnecessary operations. Moreover, not every query might benefit from earlier grouping phases. Therefore, we keep the chunking operation as simple as possible to induce the smallest possible overhead. The idea is to map the input file in memory, based on a fixed chunk size, and to allow each thread to work on a single memory-mapped file block. Mapping file chunks from disk to memory has almost no additional overhead. However, finding the right chunk size requires extra care, as the chunk size must be a multiple of the virtual page size, the data item size, and the number of workers.

While input data chunks contains parts of the input file, we also use in-memory data chunk to hold intermediate results that are produced, for instance, by aggregation queries. These data chunks belong to the memory footprint of the last query operator and hold the accumulated results. Once all intermediate results are available, the controller access them though the last query operator and returns back the final merged result to the query issuer. In other words, output data chunks are conceptually used as communication mechanism between the controller and workers. In a more general approach, however, workers could also read each other's results without returning to the controller. This is useful when a single query is composed of several pipelines. In such scenario, pipeline boundaries represent blocking operations where intermediate results needs to be materialized. Hence, materialized output chunks of one pipeline become input chunks of the next pipeline, and so on. Our architecture could easily be adapted to enable such generalized scenario.

Notice also that we do not make copy of tuples, but rather use references pointing to the memory regions holding intermediate tuples. Finally, it is worth mentioning that chunking a small input file can result in poor performance because query parallelism comes with an overhead to set up the infrastructure.

5 Experiments

In this section, we describe both the hardware and software environment used to deploy our infrastructure and to run experiments.

5.1 Environment

During the experiments, we deployed our parallel infrastructure on a single server machine running Red Hat Enterprise Linux 7.1 as operating system. The server has Xeon Haswell E5-2650L v3 (1.8 GHz) CPU with two sockets and a total of 12 cores per socket. Every core enables hyper-threading, maintaining two hardware contexts in parallel, which allows up to 48 threads to run in parallel. The server machine is a 64-bit platform that provides 64KB L1 Cache and 256KB L2 Cache per core, as well as a shared L3 Cache of 30MB, 256 GB of main memory RAM and 2TB of disk storage (7200 RPM SATA-3). All experimental runs were performed using this machine.

We deployed ViDa/RAW on this server. ViDa is using LLVM 3.4 together with GLOG 0.3.0 and GTest 1.7.0. For our experiments, we use two kinds of queries based on the TPC-H benchmark as a case study for our parallel infrastructure. These queries are described in more details in the next section. We use a scale factor of 10x (SF10) to generate the TPC-H "Lineitem" dataset, giving 8 GB of data for a total of $6*10^7$ data items. The dataset was generated in two different formats: binary column and CSV. The former represents the ideal case when columns are stored, formatted, and compressed in binary files (one per column) independently from each other, while the CSV format stores the entire dataset as a single text document, where tuples are represented as a comma-separated lines in the data file.

5.2 Use-Case Queries

In order to validate our approach and evaluate our novel parallel infrastructure, we use two types of queries as described in [2]. We describe here two concrete examples.

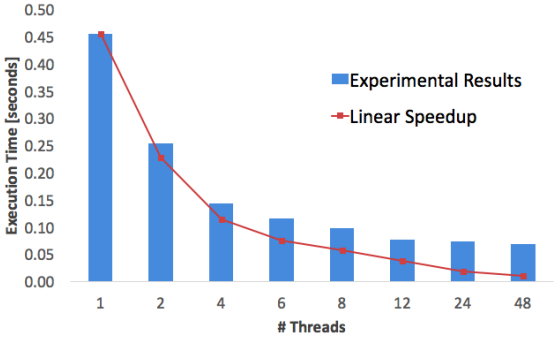
Selection Query consists of a non-blocking pipeline that contains at least one predicate expression:

```
SELECT COUNT(*)
FROM lineitem
WHERE l_quantity <= 40
  AND l_extendedprice > 50000
  AND l_discount = 0
```

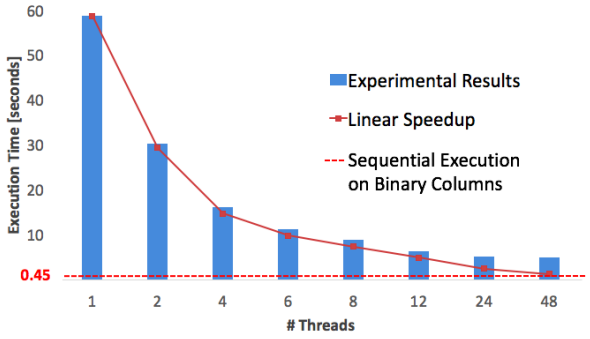
Aggregation Query consists of at least one group-by clauses together with one or more aggregation functions. This kind of query is blocking, as it requires data to be materialized in memory to accumulate the results. Furthermore, aggregation query needs an additional pass at the end of the pipeline to merge intermediate results together.

```
SELECT l_linenumber,
    SUM(l_quantity),
    MAX(l_extendedprice),
    MAX(l_discount)
FROM lineitem
GROUP BY l_linenumber
```

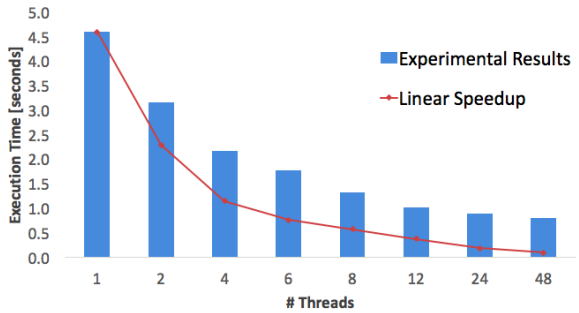
The above selection query has a selectivity of about 1.5%, the while aggregation query is performed on all tuples that are grouped by "linenumber". Both queries



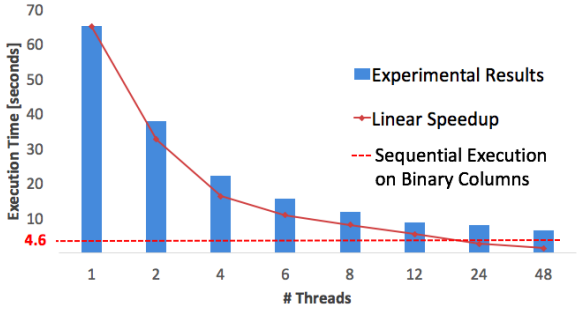
(a) Selection Query on Binary-Column Data Format



(b) Selection Query on CSV Data Format



(c) Aggregation Query on Binary-Column Data Format



(d) Aggregation Query on CSV Data Format

Figure 7: *Experimental Results*

were run on two different data format, namely binary column and CSV. The binary column format represents the ideal case, where data items are formatted in a well-known, proprietary format that is easily accessible by the query processing sub-system. On the other hand, CSV is a type of formatted text file where arguments are separated by a specific delimiter (e.g. comma-separated or semi-colon-separated files). We do not consider concrete use cases for other operators (such as join and sort) and other data formats. These are out of scope of this report and are left as future work.

5.3 Results

Figure 7 shows the experiment results for the selection and aggregation queries run on both binary-column and CSV files. For every experiment, we vary the number of threads from one to 48 threads (i.e. 1, 2, 4, 8, 12, 24 and 48, respectively), where the one-thread experiment represents ViDa's original, sequential query execution. For each variation of the number of threads, we run the experiment 10 times and measure the query response time in seconds.

As we can see, the parallel execution on both queries outperforms their respective serial execution by an order of magnitude when using more than 8 threads. When using more than 12 threads, however, the speed up becomes less effective because of increasing pressure on the memory bandwidth and the overhead caused by setting up the parallel infrastructure.

For the selection queries (Figures 7(a) and 7(b)), we achieve a near-linear speed-up in execution times. In Figure 7(b), we also compare the parallel executions on CSV data with the sequential execution on binary data. As we can see, even if execution times show near-linear speed-up, the sequential execution on binary columns still outperforms any parallel execution on CSV data by an order of magnitude. This implies that, even using intra-query parallelism on CSV data, it is not possible to achieve the same execution time as with binary data. The same trends can be observed for aggregation queries, as shown in Figures 7(c) and 7(d). However, in the case of aggregation query on binary format, we observe that the gain in performance is not as effective compared to selection queries. This can be explained by the fact that the group-by operator accounts for the major part of the query execution, as explained in section 1. Group-by operators have larger memory footprints compared to other operators such as projections and, as opposed to selection and scan, require in-memory results to be merged when worker threads are synchronized. It is important to note that the ad-

ditional overhead due to merging is negligible here, as we do not merge aggregation results directly. Instead, we use an additional projection operator (similar to selection queries) that simply counts intermediate results produced by the different pipelines. This is useful as this approach allows the parallel infrastructure to consider only a single API that is exposed by the last operator of the query plan, namely the *reduce* operator. Merging intermediate results from group-by operators is left as future work. Finally, in Figure 7(d), we compare again sequential execution on binary columns with parallel execution on CSV data. Again, no parallel execution on CSV can cope with the query response time when using binary files. This time, however, the response time using more than 8 threads in only 2x slower than the serial response time on binary columns.

All together, these experiments have confirmed our expectations by demonstrating clear performance improvements from our parallel infrastructure. Experiments have shown that our framework outperforms serial processing by an order of magnitude when enough workers are set up. These results therefore validate our approach and show that our system can be used to parallelize query on a potentially large dataset. More experiments are naturally required to confirm that our approach can also cope with other data formats and queries. However, based on the current results, we are convinced that the parallelization infrastructure can already be used to instantiate a large variety of queries.

6 Future Work

The next step in this line of work is to support other types of query operators, such as "join", which requires their own parallel algorithm. Such extension requires two key enhancements of our infrastructure. First, since the context of a worker thread is currently limited to one input data chunk, we need to extend workers to be able to produce tuples out of two or more data chunks. Allowing a worker to accept several input chunks is useful not only to join tuples but also to sort or merge tuples in parallel. Second, we need to introduce the notion of dependency between tasks. As multiple pipelines can compose a query, workers should know which pipeline to execute first. To introduce extra planning, a solution would be to add a priority field into task objects that indicates to workers the order in which pipelines have to be executed.

Another line of work that we plan to follow is to introduce elasticity in our parallel infrastructure. So far, the number of workers and tasks is known at compile time. Therefore, this approach cannot dynamically distribute the work among threads, and CPU resources cannot be reassigned on-the-fly to different pipelines of the query.

7 Conclusion

In this work, we successfully design and implement a simple, yet powerful infrastructure that allows queries to be run in parallel using ViDa. Our system takes advantage of independent pipeline execution to orchestrate parallel query execution over raw data files. We also demonstrate that selection and (to some extent) aggregation queries benefit from our novel infrastructure. The experiments validate our approach, as parallel query execution clearly outperforms serial processing and, in case of selection queries, show a near-linear speed-up in execution time. This first set of results are therefore convincing and a true motivation to undertake further work in this direction. The next development step is now to enable parallel execution of more complex queries that include join and sort operators. Although further investigations are still needed, we are convinced that this is a very promising approach that opens new perspectives for querying large and heterogeneous datasets.

Acknowledgement

I owe my deepest gratitude to Anastasia Ailamaki for allowing me to perform this work in the DIAS laboratory. I am also very thankful to Danica Porobic and Manos Karpathiotakis for their time and support, and for providing constructive and very much appreciated remarks during the realization of this work.

References

[1] M. Karpathiotakis, I. Alagiannis, T. Heinis, M. Branco, A. Ailamaki, "Just-In-Time Data Virtualization: Lightweight Data Management with ViDa". CIDR 2015.

[2] M. Karpathiotakis, A. Ailamaki, "Proteus: A shape-Shifting Query Engine". 2016.

[3] T. Neumann, "Efficiently Compiling Efficient Query Plans for Modern Hardware". VLDB 2011.

[4] V. Leis, P. Boncz, A. Kemper, T. Neumann, Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age. SIGMOD 2014.

[5] L. Qiao, V. Raman, F. Reiss, Peter J. Haas, Guy M. Lohman, Main-Memory Scan Sharing For Multi-Core CPUs. VLDB 2008.

[6] K. Anikiej, H. Bal, P. Boncz, M. Zukowski, "Multi-core parallelization of vectorized query execution". University of Warsaw and VU University Amsterdam, Master thesis, July 2010.

[7] J. Sompolski, H. Bal, P. Boncz, M. Zukowski, "Just-in-time Compilation in Vectorized Query Execution". University of Warsaw and VU University Amsterdam, Master thesis, August 2011.

[8] J. Dees, P. Sanders, "Efficient Many-Core Query Execution in Main Memory Column-Stores". IEEE 2013.

[9] D.J. DeWitt and J. Gray, "Parallel Database Systems: The Future of High Performance Database Systems", *Comm. ACM*, New York, USA, 1992.

[10] G. Graefe and W. J. McKenna. "The Volcano optimizer generator: Extensibility and efficient search". IEEE 1993.

[11] H. Pirahesh, C. Mohan, J. Cheng, T.S. Liu, and P. Selinger, "Parallelism in Relational Data Base Systems: Architectural Issues and Design Approaches", *Proc. Second Symposium Databases in Parallel and Distributed Systems*, New York, USA, 1990.

[12] J. Menon, "A Study of Sort Algorithms for Multiprocessor Database Machines", *Proc. 12th VLDB Conf.*, San Francisco, USA, 1986.

[13] D.J. DeWitt, J.F. Naughton, and D.A. Schneider, "Parallel Sorting on a Shared-Nothing Architecture using Probabilistic Splitting", *Proc. First Int. Conf. Parallel and Distributed Information System*, Madison, USA, 1991.

[14] R.A. Lorie, H. C. Young, "A Low Communication Sort Algorithm for a Parallel Database Machine", *Proc. 15th VLDB Conf.*, San Francisco, USA, 1989.

[15] P. Valduriez and G. Gardarin, "Join and Semi-Join Algorithms for a Multiprocessor Database Machine", *ACM. Trans. Database Systems*, 1984.

[16] M.S. Lakshmi and P.S. Yu, "Effectiveness of Parallel Joins", *IEEE Trans. Knowledge and Data Eng.*, Yorktown Heights, USA, 1990.

[17] J.P. Richardson, H. Lu, K. Mikkilineni, "Design and Evaluation of Parallel Pipelined Join Algorithms", *Proc. ACM SIGMOD*, New York, USA, 1987.

[18] M. Kitsuregawa, H. Tanaka, and T. Moto-oka, "Architecture and Performance of Relational Algebra Machine GRACE", *Proc. Int. Conf. Parallel Processing*, Chicago, USA, 1984.

[19] K.A. Hua and C. Lee, "Handling Data Skew in Multiprocessor Database Computers Using Partition Tuning", *Proc. 17th VLDB Conf.*, San Francisco, USA, 1991.

[20] H. Lu and K.-L. Tan, "Dynamic and Load-Balanced Task-Oriented Database Query Processing in Parallel Systems", *Proc. 3rd EDBT Conf.*, London, UK, 1992.

[21] G. von Bltzingsloewen, "Optimizing SQL Queries for Parallel Execution", *ACM SIGMOD Record*, New York, USA, 1989.

[22] J. Ribon, G. Fourny, M. Brantner, T. Westmann, and D. Kossmann, "Big Data Query Parallelization", Master thesis, March 2013.

[23] L. Fegaras and D. Maier. Optimizing object queries using an effective calculus. ACM Trans. Database Syst., 25(4):457516, 2000.