Just-in-time Data Distribution for Analytical Query Processing

Milena Ivanova, Martin Kersten, Fabian Groffen

Centrum Wiskunde & Informatica (CWI), Amsterdam, The Netherlands
{Milena.Ivanova, Martin.Kersten, Fabian.Groffen}@cwi.nl

Abstract. Distributed processing commonly requires data spread across machines using a priori static or hash-based data allocation. In this paper, we explore an alternative approach that starts from a master node in control of the complete database, and a variable number of worker nodes for delegated query processing. Data is shipped just-in-time to the worker nodes using a need to know policy, and is being reused, if possible, in subsequent queries. A bidding mechanism among the workers yields a scheduling with the most efficient reuse of previously shipped data, minimizing the data transfer costs.
Just-in-time data shipment allows our system to benefit from locally available idle resources to boost overall performance. The system is maintenance-free and allocation is fully transparent to users. Our experiments show that the proposed adaptive distributed architecture is a viable and flexible alternative for small scale MapReduce-type of settings.

1 Introduction

Data intensive research stresses the need to easily share high-volume data and to accommodate analytical exploration with little technical hurdles to take. Traditional distributed database systems are not well-equipped for these tasks. The data are partitioned by a variety of methods with the common characteristics that the partitions have to be defined and installed before query processing can take place. This leads to a rather static approach: dedicated servers host pre-defined database partitions.

Recent trends in distributed computing established database developments suitable for the Cloud, i.e., the preferred infrastructure to assemble many (virtual) machines for just an hour or days on demand. This trend addresses issues such as database consolidation, live migration, and security [3]. However, the basic mechanism of data distribution used by a cloud-hosted distributed system is not changed: range- or hash-based partitioning needs to be defined, often with the help of database design wizards, and partitions have to be installed before the query processing takes place.

To benefit from flexible use of non-dedicated resources, one often does not need to scale to a thousand-node cloud solution. For many small and middle-size analytical applications it can be sufficient to capitalize upon small-scale clusters already installed in most organizations.

In this work we propose the MonetDB/Octopus architecture providing flexible distributed query processing on non-dedicated resources with full SQL query expressiveness. Installation and deployment in a lab is a matter of minutes. The name *octopus*

was chosen to refer to the symbiosis of servers organized around a master/worker distributed execution paradigm. One node, the *head* of the octopus, is in control of the database and coordinates query processing. It creates distributed execution plans and delegates subquery execution to available worker nodes, referred to as octopus *tentacles*. Data are shipped *just-in-time* (JIT) to the workers and kept in their caches using the recycler mechanism [11]. The run-time scheduler allocates subqueries on tentacles based on up-to-date status information.

The main contributions of the paper are: 1) Just-in-time partial data distribution based on actual workload; 2) Adaptive distributed query processing capable to utilize available non-dedicated servers; 3) Flexible query optimization selecting between central and distributed execution based on the query specifics and available resources; 4) An autonomous and easy scheme for sites to join the octopus symbiosis in support of distributed processing; and 5) Distributed query plan scheduling based on precise status information of the remote servers.

Our experiments with the MonetDB/Octopus system show that the proposed architecture is a viable and low entry approach for improving the query performance of a dedicated database server by using available non-dedicated worker nodes. The evaluation based on the TPC-H benchmark demonstrates benefits comparable to traditional distributed databases. Furthermore, our experiments with the popular Hadoop MapReduce system on a small-scale cluster show that in this setting MonetDB/Octopus is a more efficient and simpler alternative.

The remainder of the paper is organized as follows. Section 2 provides an overall description of the system architecture. The Octopus distributed plan generation is described in Section 3. The run-time scheduling is presented in Section 4, followed by the adaptive distributed execution in Section 5. The prototype implementation is evaluated in Section 6. Section 7 summarizes the related work and Section 8 concludes.

2 Architecture

In this section we present the overall architecture of the MonetDB/Octopus system[1]. It follows the generic master/worker paradigm for parallel and distributed program execution. The master is a MonetDB server that hosts the database and coordinates the query processing. The workers perform subqueries in parallel on MonetDB server instances. The set of workers is dynamic, they can join the symbiosis or leave it depending on their primary purpose and current state.

Figure 1 shows a functional diagram of the system. The master server communicates with the users through the SQL front-end. When a query arrives (1), a set of optimizers transforms the query plan into a parallel execution plan (2). Plan generation uses information from the MonetDB daemon about other MonetDB servers operating in the network. In order to make a good scheduling decision, the master first registers the subqueries at the remote servers and requests them to make their 'bids' for the subtasks (3). The bids (4) reflect the capability of a worker node to perform a subquery efficiently. Based on the collected bids, the scheduler allocates subqueries to the workers (5).

[1] The MonetDB system can be downloaded from http://monetdb.cwi.nl

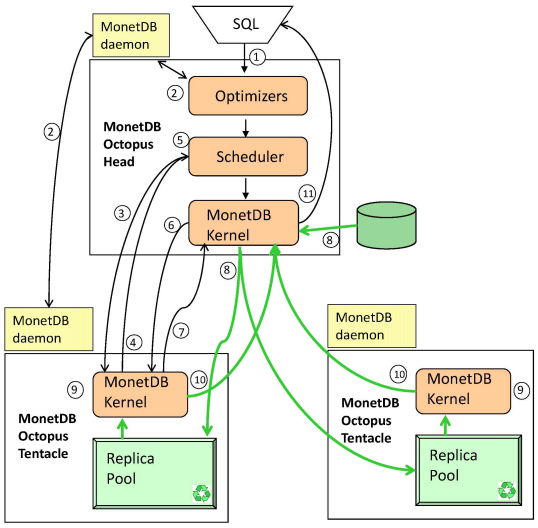


Fig. 1. Octopus architecture

Next, the master starts remote execution of the subqueries in parallel on the workers (6). Each worker requests (7) and obtains (8) from the master just-in-time replicas of data needed by the query. The replicas are kept on the worker using the recycler mechanism.

During the subquery execution (9) the MonetDB workers use only data from their own replica pool and do not need to communicate with each other. The intermediate results of the subqueries are shipped to the master server (10). Finally, it wraps up the query execution and sends the results to the user (11).

Recycler. A crucial component of the Octopus architecture is the MonetDB *Recycler* [11]. It is an extension of MonetDB execution model with capability to store and reuse intermediate results in query loads with overlapping computations. The recycler architecture consists of an optimizer that marks instructions of interest for recycling, and run-time support. At run time each instruction marked for recycling is matched against the current content of the recycle pool of intermediates.

To support distributed execution, the recycler optimizer was extended to recognize and mark remote data transfer operators, and the run-time matching mechanism – to perform correct matching and subsumption of data transfers. In this way, if a part of a table column is already cached at a remote server, a contained part can be subsumed from it instead of being completely transferred again.

Our initial experiments with recycling in distributed settings show that the data transfer times outweigh by far the times of recycled computations. In other words, the absolute effect from reused computations is substantially reduced. Therefore, although there is no principal limitation to recycle both, data transfer operators and computations, in the current system we use a modified version that considers only the data transfer operators over base tables.

Distributed Infrastructure. The distributed infrastructure, that allows master and workers to build a unique symbiosis, is set and maintained with the help of the *MonetDB daemon*. The daemon manages the database servers at a given node and monitors its network vicinity.

Any existing MonetDB database server can become the nucleus of an adaptive distributed system, i.e. the Octopus head. Using the MonetDB daemon, the master discovers the databases in the network willing to participate in shared execution. All workers keep their autonomy, i.e. once added to the potential set of working nodes, they are still free to refuse any work by simply returning a bid of unacceptable cost. Furthermore,

the workers are free to drop any replicated data at any point in time after the query that uses them has finished. There is no limitation to introduce multiple replicated Octopus heads sharing a pool of workers to improve resilience and load balancing.

3 Distributed Plan Generation

The query optimization in a dynamic distributed environment has to deal with two issues different from the traditional one: the system makes a choice between central and distributed execution, and data partitioning is carried out *dynamically*. The master hosts the entire database and can execute the query itself. Alternatively, it can choose for distributed execution and delegate subqueries to the available workers. Thus, the first issue for the optimizer is to decide whether the distributed execution is beneficial in comparison to the centralized one. In case that distributed execution is preferred, the optimizer generates a distributed plan. The crucial issue is to dynamically determine a data and query partitioning scheme that favors efficient parallel execution. The distributed query plans are generated by combined work of three MonetDB optimizers. The *mitosis* optimizer creates logical data partitions. The *mergetable* optimizer propagates partitioning through the plan. The *plan splitter* optimizer breaks the logically partitioned plan into individual subplans to be run in parallel. Each of the optimizers can revert the plan to the centralized one if it discovers a condition that renders the distributed execution as not being efficient. For instance, point queries using fast hash-based access or queries over small tables are already efficient in a centralized setting.

Mitosis. The mitosis optimizer was originally designed to increase parallelism on multi-core systems. Its task is to split the database into fragments and rewrite the query plan so that the result is consolidated as simple union operations over the fragments selected. The mitosis optimizer currently uses size annotations to select the largest table as a target for partitioning. Subsequently, it determines a good partition size based on the amount of main memory and the number of CPU cores. The final step is to horizontally partition the target table over its OID range, which is a zero cost operation in MonetDB, and reflect this in the plan. This approach, oriented to multi-core parallelism, is refined for distributed processing as follows.

The main principle for creation of data fragments is horizontal partitioning of the largest table and replication of the smaller tables in the query. It minimizes the amount of replicated data while avoiding complex algorithms for synchronized partitioning of multiple tables. The efficiency of the algorithm is important, since the partitioning scheme is determined at run time as part of the query optimization. It is also well-suited for analytical queries in data warehouses, where typically a star (or snowflake) schema is used with one large fact table and several, usually much smaller, dimension tables.

Another important task of the mitosis optimizer is to decide on the number of partitions to be created. This decision is based on several heuristics. Since the share of distribution and communication overhead becomes too large when processing small partitions, the optimizer uses a threshold value for the partition size $Size_{min}$. If the largest table is of size smaller than $Size_{min}$, partitioning will be skipped altogether and central execution plan will be produced. Otherwise, the optimizer issues a discovery request to the MonetDB daemon to find out the number of workers available. It uses this

value as an initial number of partitions and checks the size of individual partitions. If the size is too small, the optimizer reduces the number of partitions to get coarse-grained sizes.

Mergetable. The mergetable optimizer takes the fragmentation directives created by mitosis, and propagates them through the plan, effectively unfolding the query plan for each and every fragment identified. The optimizer takes care of handling aggregations, efficient joins, sorting, and grouping over fragmented columns. When the propagation of the fragments is no longer possible, the optimizer adds operators for merging the partitioned results back into a single result.

The final plan is a large parallel query plan, whose parts are handled in a dataflow driven manner by running a MonetDB interpreter in each core. It contains all the ingredients to turn it into a distributed plan as described next.

Plan Splitter. The next step is to break the logically partitioned plan into individual plans, which can be run in parallel on the workers. It is performed by the plan splitter optimizer in several phases. First, it analyzes the query plan specified in the internal MonetDB Assembly Language (MAL) to discover the logical partitions of the plan, called subplans. Here a number of criteria are checked that indicate whether the distributed plan is expected to be more efficient than the central one. If the optimizer decides for distributed execution, subplans are created. Further, the query plan at the master is modified to schedule the subplans, initiate their remote execution, merge their results, and finish up the operators producing the final query result.

Algorithm 1 shows the first phase of plan splitter that analyzes the query plan to determine subplans. The rule of thumb is that each subplan is built around a horizontal partition of the largest query table as determined by the mitosis optimizer. Plan splitting is administered through the result variables of instructions. Each variable can have a set of subplans ($splan[v]$) where its computing instruction belongs to. Initially, all instructions are included in all subplans (lines 3–4). The plan is analyzed sequentially in one pass. If an instruction is a data access to a horizontal partition, it is assigned to the respective single subplan (lines 7–8) as determined by its arguments.

For all other instructions, the assignment depends on the subplan membership of their arguments. If the set intersection of the subplans of the arguments is not empty, meaning that they all belong to at least one common subplan, the instruction is assigned to the same subplan(s) (lines 11–12). Following this general rule, the data access instructions to small query tables are replicated to all subplans.

If the instruction arguments are computed in different subplans at different workers, they need to be collected at the same place in order for the instruction to be computed. Such instructions are assigned to the 'master' subplan 1 (line 14), which has the special function of merging results from workers' subplans. Next, the algorithm ensures that the instruction arguments, or their predecessors, are brought to the master node. They are annotated as results of the subplan computing them (lines 25–26).

To minimize the intermediate transfer between workers and master, we apply several optimizations. Column view instructions in MonetDB provide alternative administrative view over columns without copying data. However, if such instruction is annotated as a subplan result in a distributed setting, it would lead to materialization and transfer along the communication channel to the master. Instead, the view instruction is added to the

Algorithm 1 Plan Splitting

```
1:  Input: MAL plan M after mergetable optimizer
2:  Output: instructions marked with subplans they belong to. Each subplan has a result set.
3:  for all v ∈ symtable(M) do
4:      splan[v] ← ∪_1^max {i}
5:  for all p ∈ M do
6:      r ← res(p)
7:      if horizPartDataAccess(p) then
8:          splan[r] ← {getSubPlan(p)}
9:      else
10:         k ← ∩_{j∈arg(p)} splan[j]
11:         if k ≠ ∅ then                                  ▷ Assign to the same subplan
12:             splan[r] ← k
13:         else                                           ▷ Arguments from different subplans
14:             splan[r] ← {1}                             ▷ Assign to the master subplan
15:             for all j ∈ arg(p) do
16:                 if splan[r] ∩ splan[j] = ∅ then
17:                     q ← source(j)
18:                     while viewIns(q) do                ▷ Search for non-view predecessor
19:                         splan[res(q)] ← splan[res(q)] ∪ {1}
20:                         v ← arg_1(q)
21:                         q ← source(v)
22:                     if dataAccess(q) then
23:                         splan[res(q)] ← splan[res(q)] ∪ {1}
24:                     else
25:                         c ← max(splan[res(q)])
26:                         addResult(c, res(q))           ▷ Set subplan result
```

master subplan (lines 18–21) and its predecessor, a non-view instruction, is annotated as a subplan result.

Similarly, projection joins and their data access operators are added to the master subplan (lines 22–23). Note, that in the above cases we keep the previous subplan assignments of the instructions, effectively replicating computations, since other parts of the subplans may also need their results. Such multi-plan assignment of instructions leads, however, to very limited replication in reality.

Given the subplan annotations, the creation of subplans is straightforward: all instructions annotated with a subplan number are copied to the respective subplan, generated as a MAL function. Few key modifications and additions are needed. The plan starts with establishing a connection to the master node; all data access instructions are replaced with a remote version that ships data needed from the master; and the subplan returns the intermediate results according to the result annotations.

Finally, the plan splitter optimizer modifies the query plan to be executed at the master. It inserts a number of instructions administering the query coordination with the workers. In particular, a call to MonetDB daemon to discover MonetDB servers currently available, instructions for registering of subplans at the remote servers, a call requesting query bids from workers, scheduling of subplans using the bids, and remote

execution of subplans. All instructions assigned to the master subplan 1 remain in the plan to perform merging of subplan results and finishing the query processing.

4 Scheduling

The Octopus scheduler grounds its decisions on precise status information exchanged with the workers. The *query bidding* mechanism is a generic way to capture the status of a remote server with respect to the individual subplans at hand. We assume that participating workers are cooperating and honest about their actual status. Bidding proceeds with one phase of exchange of information among the master and the worker nodes. For small networks we broadcast the list of subplans Q_i to all nodes with the request to make a bid for their execution.

The bidding algorithm takes as input parameters the subplan Q_i and the bid type and produces a bid using a cost model. Parameterizing the bid type enables flexibility of the system to aim at different optimization goals. Following our observations that the data transfer costs are substantial, we implemented a *data transfer* type of bid. Such bid request means that the worker node should estimate the amount of saved transfer should the subplan be scheduled at this node. The server advises the actual state of its replica cache considering the sizes of replicas that can be reused.

The result of the bidding phase is a matrix with bids from all workers for all subplans. The limited number of subplans enables deployment of an optimal scheduling algorithm, which finds a schedule that maximizes the amount of *data transfer savings*. In other words it exploits maximum the replicas already available at the workers.

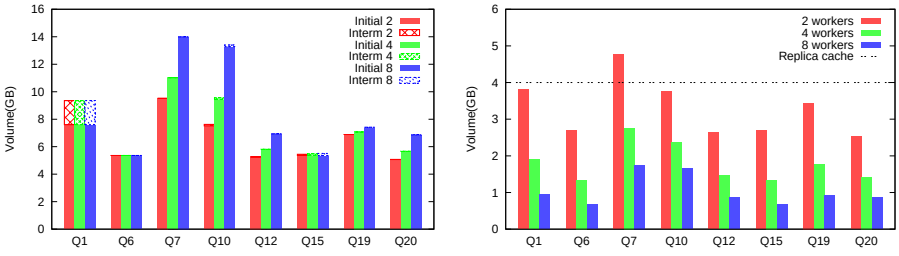
5 Distributed Execution

Distributed execution is realized by the means of parallel remote calls to subplans already registered at the workers during the bidding phase. Each worker obtains all data needed for its assigned subplan by *just-in-time data shipping* integrated completely in the query processing. As explained in Section 3, the data fragments are either horizontal partitions of columns of the largest query table, or entire columns from the smaller query tables. The actual data transfer is instrumented by the plan splitter which injects instructions to establish a connection to the master and access the remote data.

Obviously, JIT data shipping takes time and resources that may delay the response of the initial queries. However, this overhead is limited by the size of the hot data set that is actually replicated. In fact, only the columns used in the queries are distributed. Furthermore, in a workload with a limited number of query patterns, the overhead is quickly amortized, and subsequent queries demonstrate advantage over centralized execution.

To avoid slowing down of initial queries due to the JIT shipping, we provide a warming up mode of operation. During it both central and distributed plans are created, the central one producing query results for the user, and the distributed one warming up the caches of the workers.

Merging. When the workers finish execution of the subplans, the results are collected and merged at the master and processed further with aggregations, joins, etc. to



(a) Total volume (b) Per worker

Fig. 2. Transferred volumes of individual TPC-H queries

produce the final result. This phase can be considered analogical to the 'gather' phase in traditional parallel processing, or to the 'reduce' phase in the MapReduce frameworks. When the query completes, each worker can decide autonomously to clear out the replica cache, or, when the owner of the workstation has returned to deploy it for more pressing task, leave the Octopus group altogether.

6 Evaluation

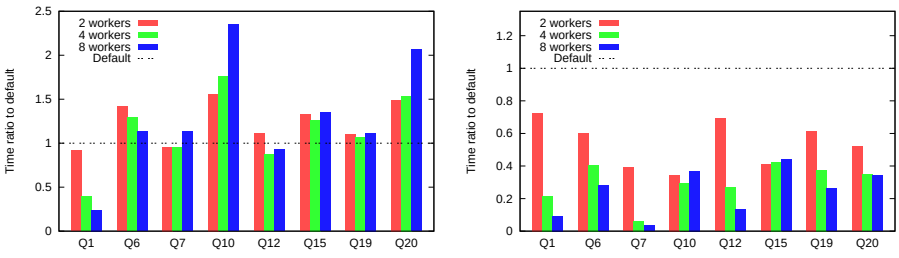
In this section we report on our evaluation of MonetDB/Octopus using the TPC-H benchmark. In addition, we also ran TPC-H on the popular Hadoop MapReduce system.

All MonetDB/Octopus experiments are run on Dual Core AMD Opteron 2GHz processors with 8 GB RAM. The master server hosting the database has 1 TB of disk space. The master and workers are all connected by a 1 Gbit/s ethernet network.

To align with the hardware resources we chose as a test database in all our experiments TPC-H SF-40. It does not fit in the main memory of a single server and response times for a number of queries are not interactive (longer than a minute). We used 8 queries of the benchmark, namely 1,6,7,10,12,15,19, and 20, which passed the criteria of Octopus optimizers for distributed execution.

Individual Queries. The first experimental set studies the effect of the dynamic distributed processing over individual queries. We evaluate the overhead incurred by the JIT data shipping and the potential benefits for the performance. Figure 2 shows the data volumes transferred by the TPC-H queries upon cold execution on two, four, and eight worker nodes. In Figure 2a the total volume exchanged between the master and workers is presented. It includes the initialization of worker caches with just-in-time replicas, as well as the volume of the intermediates returned to the master.

Our first observation is that the query scalability depends on the size of replicated data. Recall that the smaller tables in the queries are replicated among workers. Queries over a single large partitioned table, such as Q1 and Q6, or with very small size of replicated tables, such as Q15, transfer the same total amount of data and have the potential to scale well out. Queries with more substantial sizes of replicated tables, such as Q7 and Q10, have increased total volume transferred with the increase of the number of workers, which is a potential limitation for their scalability.



(a) Cold (b) Hot

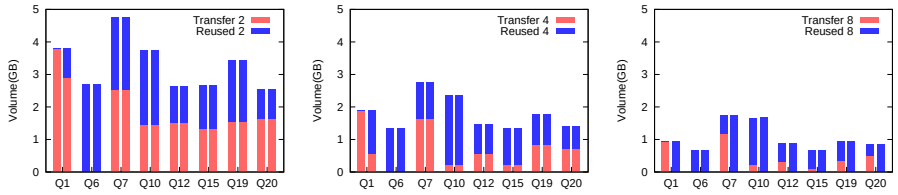
Fig. 3. Octopus performance for individual TPC-H queries

Furthermore, some queries have substantial sizes of intermediates returned to the master (Q1). The overhead for transferring those intermediates may also put a limit on the achievable improvement. Third, the total volume is substantial for some of the queries. For example, Q7 on 2 workers needs 10 GB hot data set, which means that 25% of the total database size has to be transferred.

Figure 2b shows the data volume transferred per worker. It illustrates how well the just-in-time replicas fit into the workers caches, set in our experiments to 4 GB. Since the large data set needed for query Q7 does not fit in the cache of 2 workers, even an immediate hot execution would still replicate part of the data and limit the performance benefits.

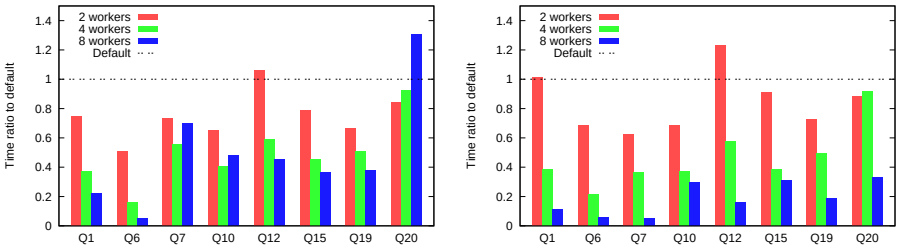
Figure 3 shows the performance of individual queries upon cold (a) and hot (b) distributed execution with respect to the central one. The majority of queries upon cold execution show as expected an initial overhead which outweighs the benefits of the parallel execution. Figure 3b illustrates the best benefits achievable by the hot execution. The majority of the queries scale well with the increased number of workers. Queries Q15 and Q20 do not improve further from the time achieved on 4 workers. They have lower computational complexity and relatively efficient central execution. Hence, the improvement in a parallel setting is limited and might be smaller than the communication overhead. Finally, as pointed before, the scalability of query Q10 is limited by the large volume of the replicated tables, almost 1 GB per worker. The experimental results comply with the general problem faced in distributed query processing. Effective use depends to a large extent on the query patterns.

Query Batch. In the next set of experiments we run queries in a batch, where each query but the first has a chance to reuse some of the replicas obtained by the previous queries. Figure 4 shows the data volumes reused during the query batch execution with increased number of workers. The batch is run twice. The results of the two runs of each query are presented next to each other for better comparison. Note, that all queries, but the first execution of Q1, benefit from the previous queries, due to the overlap among the tables and columns they process. In fact, Q6 does not transfer any data, since all replicas needed have already been cached by Q1. However, the total volume of the hot data set for the eight queries does not fit in the worker cache for two and four workers. Hence, the 'hot' run needs to transfer as much data as the cold one with an exception of



(a) 2 workers (b) 4 workers (c) 8 workers

Fig. 4. Transferred and reused volumes in TPC-H query batch



(a) Cold (b) Hot

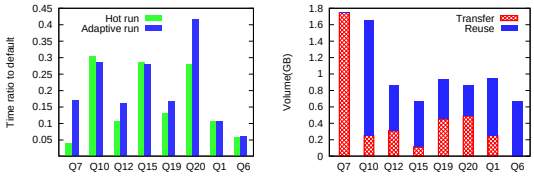
Fig. 5. Octopus performance for TPC-H batch

Q1, that reuses some of the replicas from the previous queries. This situation changes for execution on eight workers, where the hot data set fits in the cache and during the hot run all the queries reuse replicas (second bars are entirely blue). Therefore, the performance of a dynamic distributed system depends also on the size of the hot data set for the entire query workload, and its ratio to the available cache sizes.

Figure 5 shows the performance of the query batch during the cold (a) and hot (b) execution. The cold execution is improved with respect to the individual query runs, since queries in the batch utilize existing replicas. The hot execution is less efficient than the individual 'best case' due to the eviction of some replicas by queries competing for the common replica cache.

Adaptive Behavior. In this experiment we study the performance of MonetDB/Octopus upon leaving of a worker node and replacing it with another one. After several runs of the batch on 8 workers, we forced one of the workers to leave the symbiosis just before query Q7, and replaced it with another server. Figure 6 shows the performance (a) and the volumes transferred or reused (b) during the adaptive run.

The new worker needs to acquire all data for query Q7, but the impact is smaller for the subsequent queries that reuse some of the replicas. We observe limited performance degradation for queries 7,12,19, and 20, the last being the worst case running for 40% of the central execution time.



(a) Performance (b) Data volumes

Fig. 6. Octopus adaptation to change of servers

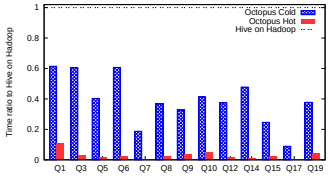


Fig. 7. Octopus query times relative to Hadoop with Hive

To summarize, the factors that determine how suitable is the query workload for dynamic distributed execution are: the size of the database hot set, the volume of replicated tables, computational complexity, and the size of the intermediate results.

Hadoop with Hive. Due to the popularity of Hadoop [9], in this experiment we compare it to MonetDB/Octopus, although both target a different audience. We used the TPC-H queries expressed in HiveQL[2] and translated by Hive [20] into a series of map and reduce jobs to be executed on the Hadoop cluster.

It is reported on Hadoop's website, each tutorial, and reference books, that Hadoop starts to pay off starting in a cluster with 1000+ machines. The 9 machines we used in our experiments with MonetDB/Octopus are not even near the amount required for Hadoop to shine. With this in mind, we ran TPC-H SF-40 using Hadoop 0.20.205.0 and Hive 0.7.1 on 9 Intel Core i7 8x3.4GHz machines with 16 GB of memory and a 2TB single disk running Fedora 15.

One machine was assigned NameNode/JobTracker capabilities, the remaining 8 machines were DataNode/TaskTrackers. We used a replication count of 1, to have the data spread across the nodes from the NameNode to the DataNodes in a resource conservative way, like MonetDB/Octopus. This way, we treated the NameNode as equivalent of the Octopus master. We had to load the TPC-H data in the HDFS filesystem a priori.

In both systems, we omit loading times of the initial data into the system. For MonetDB/Octopus this means loading the data in tables, for Hadoop we put the files in HDFS, where Hive picks it up again for every query. Because Hadoop distributes the data over all DataNodes, data is already spread across the network, while MonetDB/Octopus starts from the master that needs to ship relevant data to the workers first. For this reason, we compare the running times of the Hive queries against a cold and hot run of MonetDB/Octopus, where hot refers to the data being already shipped to the worker.

Figure 7 shows the MonetDB/Octopus running time for a subset of the TPC-H queries relative to the time Hadoop with Hive took to process the query. The Hadoop setup used 8 Mappers and 8 Reducers on each DataNode/TaskTracker, a setting which we found to be most efficient after some trial runs. We confirmed that the 8 worker machines were using their full capacity during the Map-phase of the queries.

We have run Hadoop and Hive more or less out of the box, without much effort to tune its performance. Preliminary experiments have shown that when using more data

[2] `https://issues.apache.org/jira/browse/HIVE-600`

(higher scalefactor) the running times do not increase as much, indicating that Hadoop is much better suited for much larger data sets, on much larger cluster settings. Tuning software for Hadoop, such as Starfish [10] could probably improve the performance considerably, bringing the performance of Hive closer to that of MonetDB/Octopus.

7 Related work

Distributed database technology has been a key area of research for over thirty years. The main challenge and contribution of our work is the *dynamic* mechanism for scaling out a read-optimized database system. Our approach is close to the Data-In-The-Network (DITN) proposal for parallel querying with non-dedicated computers [18]. DITN utilizes inter-fragment parallelism and splits queries into independent work units to avoid shipping of tuples between operators. This approach provides for better flexibility in situations with variable loads, failures and heterogeneity. Similarly, MonetDB/Octopus splits a plan into independent subplans avoiding communication between workers, but in addition exploits caching and query overlaps, and chooses between central and parallel execution based on the query and data specifics.

Caching has been traditionally used to improve performance in distributed systems [7, 13]. Similarly to hybrid-shipping query processing in [7] Octopus allocates at run time subplans to workers based on the current cache content. However, JIT data shipping is not a processing by-product, but an intentional act performed for the purpose of parallel processing on non-dedicated machines.

Replication is widely used technique for improving system availability and throughput [17, 19]. Such full replication-based systems do not address intra-query parallel execution for complex analytical queries in read-optimized distributed databases. The just-in-time data shipping creates partial replicas that enable parallel processing on currently available resources. The replicas do not assume dedicated servers and are just invalidated upon updates in the master database.

The most recent development in distributed environments is the Cloud which offers a cost-efficient provision of potentially unlimited computational resources on demand. We share the idea to exploit non-dedicated resources, but focus on small-to-middle size locally available resources, also known as private clouds, an area not deeply explored to date.

Adapting distributed database techniques for the Cloud takes considerable changes to the software to fit well in the highly volatile environment [3]. Among the active areas of research on databases in the Cloud is the database live migration [5]. It shares some issues with our JIT data shipping, such as pulling data on demand from the owner database, but differs in purpose and level of abstraction.

MapReduce [4] and its open-source implementation Hadoop [9] have become a popular tool for large-scale data analysis. It is recognized for its elastic scalability and fine-grained fault tolerance. Its performance, shown to be sub-optimal in the database context [16], has been recently boosted by adding features and developing optimization frameworks. Often, solutions are found in well known techniques from database world, such as indexing [12] and column-oriented storage [6].

The MonetDB/Octopus starting point is different, a column-store database system, known for its efficiency for analytical workloads. Our goal was to augment the system with ability to scale out into a distributed execution platform utilizing non-dedicated machines. The extensions implemented can be considered as an implementation of coarse-grained MapReduce-style of processing inside the database.

Several projects propose higher-level abstractions [2] or languages [14, 20] facilitating parallel processing specification, which is translated into MapReduce jobs for execution. Our optimization framework shares some ideas with the optimizers employed in those systems, such as supporting choice between central and parallel execution [2], and caching data fragments [15].

Many vendors of parallel DBMSs also embrace ideas from MapReduce paradigm. The HadoopDB [1] attempts to bring together the best features of both worlds. Our work differs in using *dynamic* partitioning that provides for elasticity: non-dedicated nodes can easily join or leave the system. We tackle similar problems of how to split the work among the system components. However, MonetDB/Octopus carries out SQL query processing entirely in the database.

8 Summary and Conclusions

The potential benefits from just-in-time commissioning of system resources, (e.g. Cloud), has become a major driving force to innovated database processing. In this paper we provide a solution geared at harvesting non-dedicated idle local resources using an adaptive distributed database platform. Such resources are readily available in many labs and organizations.

Any group of systems can participate in distributed query processing without a priori need for data partitioning and distribution. All that is needed is installing the MonetDB software stack, starting the MonetDB daemon, and announcing which (distributed) database each node is allowed to support. The result is a system that provides a rich declarative MapReduce functionality, which does not require programmer interference, but supports fully-fledged SQL queries.

MonetDB/Octopus provides the performance advantages of distribution for long-running analytical queries combined with adaptive and flexible behavior. It dynamically distributes data driven by the needs of the current query load. The initial investment in data transfer is amortized by the subsequent queries by the means of database caching (recycling). Scheduling of distributed subplans is based on actual information about the status of the remote servers exchanged through a bidding mechanism.

Compared to the de-facto MapReduce implementation Hadoop, MonetDB/Octopus shows beneficial for the setting it is aimed at. The ease in which systems can be added and retracted from the pool, allows for a great adaptivity in smaller settings where flexibility is desired, but analytical full-fledged SQL queries are the norm.

Ongoing and future research investigates design questions, such as the optimal number of workers in relation to database size and query complexity, and alternative data transport mechanisms, such as RDMA calls over InfiniBand as in the companion Data-Cyclotron project [8]. While the current version assumes workers with similar capacity and prepares equally-sized partitions, another future direction is to generate distribution

plans suitable for heterogeneous environments. The Octopus prototype code is available as part of the MonetDB release, which opens a road for others to join in the exploration of these opportunities. The adaptive distribution scheme does not require a large pre-installed hardware base, a few spare workstations is all that is needed to exploit the potential parallelism.

Acknowledgments. This work was partially supported by the Dutch research programme COMMIT and the European project TELEIOS.

References

1. K. Bajda-Pawlikowski, D. J. Abadi, et al. Efficient Processing of Data Warehousing Queries in a Split Execution Environment. In *SIGMOD*, pages 1165–1176, 2011.
2. C. Chambers, A. Raniwala, et al. FlumeJava: easy, efficient data-parallel pipelines. In *PLDI*, pages 363–375, 2010.
3. C. Curino, E. P. C. Jones, et al. Relational Cloud: a Database Service for the Cloud. In *CIDR*, pages 235–240, 2011.
4. J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of OSDI*, pages 137–150, 2004.
5. A. J. Elmore, S. Das, D. Agrawal, and A. E. Abbadi. Zephyr: live migration in shared nothing databases for elastic cloud platforms. In *SIGMOD Conference*, pages 301–312, 2011.
6. A. Floratou, J. M. Patel, E. J. Shekita, and S. Tata. Column-Oriented Storage Techniques for MapReduce. *VLDB*, pages 419–429, 2011.
7. M. J. Franklin, B. T. Jónsson, and D. Kossmann. Performance tradeoffs for client-server query processing. In *SIGMOD Conference*, pages 149–160, 1996.
8. R. Goncalves and M. L. Kersten. The data cyclotron query processing scheme. In *EDBT*, pages 75–86, 2010.
9. Hadoop. `http://hadoop.apache.org/`, 2012.
10. H. Herodotou, H. Lim, et al. Starfish: A self-tuning system for big data analytics. In *CIDR*, 2011.
11. M. Ivanova, M. L. Kersten, N. J. Nes, and R. Goncalves. An architecture for recycling intermediates in a column-store. *ACM Trans. Database Syst.*, 35(4):24, 2010.
12. D. Jiang, B. C. Ooi, L. Shi, and S. Wu. The Performance of MapReduce: An In-depth Study. *PVLDB*, 3(1):472–483, 2010.
13. D. Kossmann, M. J. Franklin, and G. Drasch. Cache investment: integrating query optimization and distributed data placement. *ACM Trans. Database Syst.*, 25(4):517–558, 2000.
14. C. Olston, B. Reed, et al. Pig latin: a not-so-foreign language for data processing. In *SIGMOD Conference*, pages 1099–1110, 2008.
15. C. Olston, B. Reed, A. Silberstein, and U. Srivastava. Automatic optimization of parallel dataflow programs. In *USENIX Annual Technical Conference*, pages 267–273), 2008.
16. A. Pavlo, E. Paulson, et al. A Comparison of Approaches to Large-scale Data Analysis. In *SIGMOD Conference*, pages 165–178, 2009.
17. C. Plattner, G. Alonso, and M. T. Özsu. Extending DBMSs with Satellite Databases. *VLDB J.*, 17(4):657–682, 2008.
18. V. Raman, W. Han, and I. Narang. Parallel querying with non-dedicated computers. In *VLDB*, pages 61–72, 2005.
19. U. Röhm, K. Böhm, and H.-J. Schek. Cache-Aware Query Routing in a Cluster of Databases. In *ICDE*, pages 641–650, 2001.
20. A. Thusoo, J. S. Sarma, et al. Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2:1626–1629, August 2009.