Integrating analytics with relational databases

Raasveldt, M.

CHAPTER 8

Conclusion

1 Big Picture

In this thesis, we have investigated each of the methods in which analytical tools can be combined with relational database management systems. For each of the methods, we have provided improvements in both the efficiency and the usability departments.

Each of the three methods that we have considered has its place. As for our original goal of making the RDBMS as easy to use as flat file storage; the embedded database systems definitely shine. They are easy to install, and once installed can be used directly from within the analytical tools with very little setup overhead.

However, they share one of the same drawbacks as working with flat files in that they are not designed for collaborating with multiple people over multiple machines. In these scenarios, setting up a separate database server is the preferred solution.

The client-server connection is effective when the user wants to export the data and run an analysis pipeline only once. In this scenario, our proposed client-server protocol can significantly accelerate the speed of data export and assist the user in running their analysis pipeline efficiently.

When the user wants to run their analysis pipelines several times, for example as part of reporting software that periodically generates new graphs based on new data, MonetDB/Python UDFs shine as the stand-alone server architecture can be combined with the fast data transfer between the RDBMS and the analytical tool.

2 Future Research

In this section, we will present potential future research directions in the area of combining analytical tools and RDBMSs. We split up this section by each of the different methods of combining analytical tools with RDBMSs and discuss future research directions for each of the different methods.

2.1 Client-Server Connections

Adaptive Compression

In our current protocol, we use a simple heuristic to determine which compression method to use. An optimization that can be made to our protocol is therefore to use the network speed as a heuristic for which compression method to use. Using compression methods that offer degrees of compression, the cost of the compression can be fine tuned and dynamically adapted to changing network conditions.

Parallel Serialization

Further performance could be gained over our proposed protocol by serializing the result set in parallel. This can be advantageous for parallelizable queries. In these scenarios, the threads can immediately start serializing the result set to thread-local buffers as they compute the result without having to wait for the entire query to finish. However, since writing the data to the socket still has to happen in a serialized fashion we only expect performance gains in a limited set of scenarios. For example, when result set serialization is expensive due to heavy compression or when the query is fully parallelizable.

2.2 In-Database Processing

Polymorphism

Currently, MonetDB/Python functions are only partially polymorphic. The user can specify that the function accepts an arbitrary number of arguments, however, the return types are still fixed and must be specified when the function is created. Allowing the user to create complete polymorphic functions would increase the flexibility of MonetDB/Python functions.

The problem with polymorphic return types is that the return types of the function must be known while constructing the query plan in the current execution engine. Thus we cannot execute the function and look at the returned values to determine the column types. The solution proposed by Friedman et al. [30] is to allow the user to create a function that specifies the output columns of the function based on the types of input columns. This function is then called while constructing the query plan to determine the output types of the function.

This allows the user to create functions whose output columns depend on the number of input columns and the types of those columns. However, it does not allow the user to vary the output columns based on the actual data within the input columns. Consider, for example, a function that takes as input a set of JSON encoded objects, and converts these objects to a set of database columns. The amount of output columns depends on the actual data within the JSON encoded objects, and not on the amount or type of the input columns, thus these types of polymorphic user-defined functions are not possible using the proposed solution.

The ideal solution would be to determine the amount of columns during query execution, however, this provides several challenges as the query plan must be adapted to the amount of columns returned by the function, and must thus be dynamically modified during execution.

Data Partitioning

MonetDB/Python supports parallel execution of user-defined functions. It does so by partitioning the input columns and executing the function on each of the partitions. Currently, the partitioning simply splits the input columns into n equally sized pieces. This is the most efficient way of splitting the columns, but it limits the parallelizability of user-defined functions. Functions that operate only on the individual rows, such as word count, can be parallelized using this partitioning.

However, as noted by Jaedicke et al. [44], certain functions cannot be efficiently executed in parallel on arbitrary partitions, but can be efficiently computed in parallel if there are certain restrictions on the partitioning scheme. Allowing the user to specify a specific partitioning scheme would increase the flexibility of the parallelization.

There are performance implications in arbitrary partitioning in a column-store. Normally, the identifiers of every row are not explicitly stored, as shown in Figure 2-1a. The current partitioning scheme does not rearrange the values in the columns, which allows these identifiers to remain virtual. However, if we rearrange the values in the columns to match a user-defined partitioning scheme, we would need to explicitly store the row identifiers, resulting in significant additional overhead. This is avoided by the special partitioning used for computing parallel aggregates, because we do not need to know the individual tuple identifiers of each of the values as we are accumulating the actual values, thus we only need to know the group that the value belongs to.

Still, parallelization could lead to big improvements in execution time of CPU-bound functions. It would be interesting to see how big the set of functions is that cannot be parallelized over arbitrary partitions, but can be parallelized over restricted partitions. It would also be interesting to see if it would be worth the performance hit of creating these restricted partitions over the data so we can compute these functions in parallel.

Distributed Execution

Currently, MonetDB/Python can only be parallelized over the cores of a single machine. While this is suitable for a lot of use cases, certain data sets cannot fit on a single node and must be scaled to a cluster of machines. It would be interesting to scale MonetDB/Python functions to work across a cluster of machines, and examine the performance challenges in a parallel database environment.

Query Flow Optimization

Currently, we treat MonetDB/Python functions as black boxes in query execution. However, queries involving MonetDB/Python functions could be optimized if we knew more about the computational complexity of the function. Determining this automatically is an extension of the halting problem, as if we could compute the exact run-time of a function, we would also know if the function would terminate. However, estimations could be made.

It has been suggested by Hellerstein et al. [38] and Chaudhuri et al. [13] to make the user specify the complexity of their function by making them fill in the cost per tuple. However, this can be very difficult to determine for the user and places the burden of optimization on them.

There has been some work by Crotty et al. [19] on automatically trying to estimate this information by looking at the actual compiled code. Alternatively, we could look at run-time statistics to try and determine the complexity of the functions, although this does require the user to use the function in an unoptimized data flow first.

Script Optimization

In this thesis we have focused mainly on optimizing the dataflow around user-defined functions. We have seen in Figure 4-5 that this dramatically speeds up functions for which transportation of data is the main bottleneck. However, when the computation time dominates the transportation time this optimization will not provide a significant speedup. We have provided the ability to execute functions in parallel, which can still

provide significant speedups to these functions. However, we still treat the user-defined functions as black boxes. Additional speedups could be achieved by looking into the user-defined functions and optimizing the code within the functions.

Cardinality Estimation

MonetDB uses heuristics based on table size when creating the query plan to determine how the columns should be partitioned for parallelization, as partitioning small tables significantly degrades performance. However, when the table is generated by a table-producing UDF, this table could potentially have any size. An interesting research direction could be estimating the cardinality of these table-producing functions.

Code Translation

When creating MonetDB/Python, we have tried to make it as easy as possible for data scientists to make and use user-defined functions. However, they still have to write user-defined functions and use SQL queries to use them if they want to execute their code in the database. They would prefer to just write simple Python or R scripts and not have to deal with database interaction.

An interesting research direction could be analyzing these scripts, and automatically shipping parts of the script to be executed on the database as user-defined functions. This way, data scientists do not have to interact with the database at all, while still getting the benefits of user-defined functions.

2.3 Embedded Databases

Embeddability

In MonetDBLite, there are still several open issues that result from the nature of how MonetDBLite was created. Because the database that is based on, MonetDB, operates as a stand-alone server several limitations are present in the code that introduce problems when it is used as an embedded database.

MonetDB traditionally only allows a single database process to read the same database. There are is no fine grained locking between several database processes. Instead, a global lock is used on the entire database. If the user attempts to start a database server with a database that is currently occupied by another server an error will be thrown ("database locked") and the process will exit. This makes sense in the stand-alone server scenario, as running multiple database servers on the same database does not make much sense. However, it is a problem in the embedded database scenario because multiple processes might want to access the same database.

Another limitation is that the MonetDB server can only run on a single database at a time because of the large amount of global variables present in the codebase of MonetDB. This is no problem in the stand-alone server case, because another server can be started in a different database directory. However, for the embedded case, this is a limitation because only a single database can be opened in the same process.

As MonetDB is designed to run on a large machine that is dedicated to running the database server and has enough memory to handle the working set, MonetDB also does not perform gracious handling of out-of-memory situations. In many places in the codebase, `malloc` returning a `NULL` is not handled and will lead to the database server crashing. The processing model of MonetDB also does not lend itself well to low memory devices, as large intermediates are materialized entirely in memory. In the case of smaller devices, this may lead to the system frequently swapping or even running out of disk space and crashing the server.

These issues are so ingrained into the MonetDB codebase that they are very difficult to address. In fact, fixing these issues will require almost a complete rewrite of the entire codebase. Instead, we have decided to write DuckDB from scratch in order to fix these issues.

Hardware Distrust

Hardware distrust is another unexplored area of database research that is very relevant to embedded database systems. As the embedded database runs on low quality consumer hardware instead of high quality server hardware it is very possible that the

system is ran on broken hardware. In the case of broken hardware, it is important that the database system prevents (or at the very least limits) the corruption of the data stored by the database system.

Any component used by the database system can be broken and can be broken in different ways. The hard disk can report successful writes even when writes have not occured, or it can flip bits within the file. Random bits can be flipped in the memory, or entire memory regions can be corrupt. Even the CPU can return incorrect results when broken or overclocked. Detecting and attempting to limit the damage caused by broken hardware components without significantly impacting performance is an area that we are actively working on in DuckDB.

Resource Contention

Embedded database systems always run alongside their host application, and the resources used by the host application can vary wildly. The host application can be either a simple shell that performs almost no additional work or a full-fledged analytical application that consumes large amounts of memory and CPU resources.

Currently DuckDB employs the standard solution of letting the user configure the resource consumption of the database system. While this works, it adds additional knobs for the user and does not allow for adaptive resource balancing of the database system. For example, the database system could switch to using less memory as the host system requires more memory, or switch to using more CPU resources when these resources become available.