Integrating analytics with relational databases

Raasveldt, M.

CHAPTER 2

Background

As our goal is to improve the coupling of relational database management systems and analytical tools, it is clear that the existing techniques for combining external programs and RDBMS servers must be investigated.

In this chapter, we will describe existing techniques for combining external programs and RDBMS servers, and also provide the necessary background for understanding these techniques. In Section 1, we give a brief description of the history of RDBMS engines. In Section 2 we briefly describe the different types of RDBMS engines and the different physical storage models and database processing models they utilize. In Section 3 we discuss the different methods in which an external analytical program work in combination with a RDBMS. In Section 4 we describe the internal design of MonetDB, a popular open-source RDBMS that we have used as a test-bed for implementing a lot of the work in this thesis. In Section 5, we briefly describe the internal design of the CPython interpreter and the NumPy library, as we rely on these for the implementation of the vectorized user-defined functions described in Chapter 4.

1 Relational Database Management Systems

Database management systems have been around in some form or another for almost as long as computers themselves have been around. They solve the fundamental problem of manipulating and persistently storing data for future usage. This is a problem that is encountered by almost any application. Whether it be a bank manager, an online store or even a video game, they all require a method of persistently storing state, updating that state and reading back that state.

The most popular form of database management systems are relational database management systems. These types of systems allow users to interact with the data they store using languages based on relational algebra, pioneered by E.F. Codd [16] in 1970. In the relational model, data is organized in $n - ary\ relations$ where every row in the relation consists of n different values. Data stored in this model can be stored into multiple relations, and combined at query time using the join operator ($\bowtie$).

The relational model offers two crucial advantages: (1) data can be stored in a normalized way, avoiding data duplication and improving data integrity, and (2) the way data is accessed is separated entirely from the physical way in which the data is organized, allowing for the engineers that create the database management system to have complete freedom in the way the data is represented on disk and the way in which it is accessed. This has allowed relational algebra to stay relevant even while storage methods, indexing algorithms and query execution models have changed.

After Codd's paper several languages popped up that were based on relational algebra. The clear winner, and the language used almost universally by relational database management systems today, is SQL [12] (Structured Query Language). SQL was initially developed in 1973 at IBM for use in System R [6] and was afterwards used in DB2 [91]. In the late 1970s it was adopted by Oracle [71], and it was standardized by the International Organization for Standardization (ISO) in 1987. Currently, SQL is the database language of choice for relational systems. It is supported by every major database vendor, and even many non-relational systems implement (limited) dialects of SQL as users are so familiar with it.

2 RDBMS Design

As relational algebra grants RDBMSs immense freedom in their underlying physical implementation, there have been many different proposed designs for RDBMSs. Each of the designs are catered towards different use cases, and have different advantages and disadvantages. In this section, we will discuss the most common trade-offs that are made in RDBMS design.

2.1 Workload Types

Before we discuss the types of RDBMS systems, we will describe the types of workloads that these systems are typically optimized for: OLTP workloads, OLAP workloads and hybrid workloads.

On-Line Transactional Processing (OLTP) workloads are focused on managing operational data for businesses. As an example of operational data, consider managing the in-stock items of a retailer, or updating account balances of a bank.

In OLTP workloads, there are many queries fired at the database concurrently. Individual queries are very simple and touch very few rows. In general, queries consist of either selecting, inserting, updating or deleting a single row. Queries that need to access data from a large subset of the database are (almost) never performed.

On-Line Analytical Processing (OLAP) workloads are focused on analyzing and summarizing the data stored inside a data warehouse. As an example of these analytical queries, consider for example generating business reports containing the sales of certain products over time, or the popularity of items in certain regions.

In OLAP workloads, there are relatively few queries fired at the database. However, the individual queries are very complex, and often touch the entire database. In these workloads, changes to the data in the form of inserts, updates or deletes happen in bulk (or might not even occur at all).

Hybrid Workloads consist of a mix of transactional statements and analytical statements. Typically, there is a high amount of small transactional statements fired at the system, mixed with the occasional reporting query.

2.2 System Types

Disk-Based Systems. When database systems were first created, computer systems were not equipped with much high-speed memory. When DB2 was originally released in 1987, the price of RAM was around 200USD per MB [55]. As such, these systems could not rely on a significant portion of the database fitting inside main memory. Instead, these systems were primarily designed for the database to reside on disk, with only a small portion of the data (that is currently being processed) residing in RAM. As the slow reading and writing speed of the hard disk was the primary bottleneck for these systems, they primarily considered how to optimize for minimizing disk access. These systems were primarily designed for OLTP workloads.

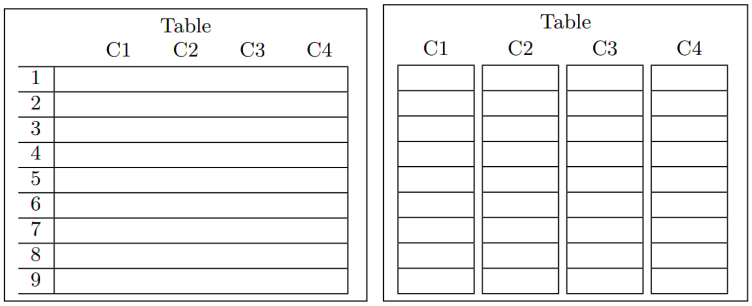
Main-Memory Resident Systems. When the prices of main memory fell and memory sizes grew, it became possible for the entire database (or at least the working set) to reside entirely in memory. As a result of the increasing memory sizes, it became possible to create systems optimized for main-memory resident data sets.

In systems optimized for main-memory, the disk no longer needs to be accessed at all for read-only queries, and data only needs to be written to disk for persistence purposes. As a result, these systems can achieve much faster speeds than the earlier systems that were bottlenecked by disk latency, but only if the system has sufficient memory to hold the working set. These systems have been designed for both OLTP, OLAP and hybrid workloads.

2.3 Physical Database Storage

The physical layout of the database influences the way in which the database can load and process data, and can significantly influence the performance of the database depending on the access pattern that is required by the query. The main decision in physical database layout is whether to horizontally fragment the data or to vertically fragment the data. These different physical layouts are visualized in Figure 2-1.

Row Storage Databases fragment tables horizontally. In this storage model, the data of a single tuple is tightly packed. The main advantage of this approach is



(a) Row-Store.

(b) Column-Store.

Figure 2-1: Physical layout of row-store and column-store databases.

that operations on individual tuples are very efficient, as the data for a single tuple is tightly packed at a single location. The main drawback of this approach is that columns cannot be loaded individually from disk, as the values of a single column are surrounded by the values of the other columns. Because of this, unused columns in the table definition will affect query performance. When a query only operates on a subset of the columns of a table, the entire table must be loaded from disk regardless. This is especially relevant for OLAP-style queries that only touch a handful of columns in large tables with hundreds or even thousands of columns.

Column Storage Databases fragment tables vertically. In this storage model, the data of the individual columns is tightly packed. The advantages of this approach are two-fold: (1) the columns can be loaded and used individually, which means we do not need to load in any unused columns from disk, and (2) packing data of individual columns together leads to significantly better compression. The trade-off, however, is that reconstructing tuples is costly as the values of individual tuples are spread out over different memory locations. As a result, operations on individual tuples are expensive. As these types of operations are typically performed in OLTP workloads, column storage lends itself towards OLAP workloads.

2.4 Database Processing Models

The processing model of the database heavily influences the design and performance of the user-defined functions, as the processing model defines how the data is transferred between the database and the user-defined function. The processing model is closely related to the physical storage of the database.

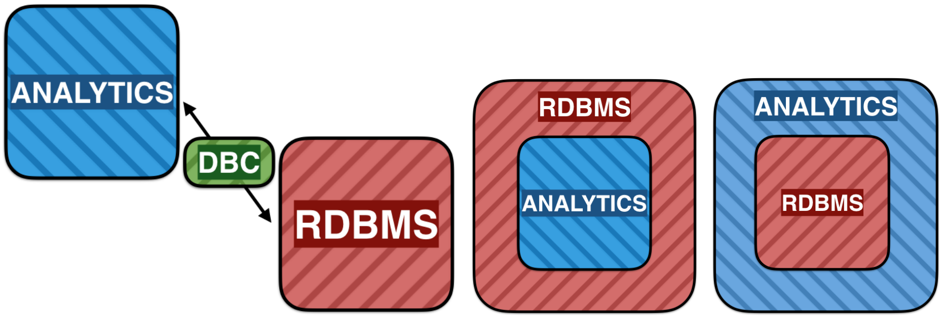
Tuple-at-a-Time Processing is the standard processing model used by most disk-based systems. In this processing model, the individual rows of the database are processed one by one from the start of the query to the end of the query.

The primary advantage of this processing model is that the system does not need to keep large intermediates in memory. In extremely low memory situations, processing queries in this fashion is often the only possibility. However, in situations where many rows are processed the tuple–at–a–time processing model suffers heavily from high interpretation overhead. This approach is used by PostgreSQL, MySQL and SQLite.

Operator–at–a–Time Processing is an alternative query processing model. Instead of processing the individual tuples one by one, the individual operators of the query are executed on the entire columns in order. As the operators process entire columns at a time, the function call overhead of this processing model is minimal.

The main drawback of this processing model is the materialization cost of the intermediates of the operators. In the tuple-at-a-time processing model, a single tuple is processed from start to finish before the query processor moves on to the next tuple. By contrast, in the operator–at–a–time processing model, the operator processes the entire column at once before moving on to the next operator. Because of this, the intermediate result of every operator has to be materialized into memory so the result can be used by the next operator. As these intermediate results are the result of an entire column being processed they can take up a significant amount of memory. This approach is used by MonetDB.

Vectorized Processing is a hybrid processing model that sits between the *tuple-at-a-time* and the *operator–at–a–time* models. It avoids high materialization costs by operating on smaller chunks of rows at a time, while also avoiding overhead from a significant amount of function calls. This approach is used by Vectorwise [8].



(a) Client-Server connection.

(b) In-database processing.

(c) Embedded database.

Figure 2-2: Different ways of connecting external programs with a database management system.

3 Database Connectivity

While RDBMSs are very powerful, SQL is not a general purpose language. As such, it is necessary for clients to write their actual application code in a different programming language and communicate with the RDBMS in order to exchange data between the application and the database management system.

As the focus of this work is on combining a RDBMS with analytical tools, we focus especially on users wanting to use analytical tools (e.g. Python or R programs) for the purpose of performing analysis on large amounts of data that reside in the RDBMS. Figure 2-2 shows the three main methods in which a relational database can be combined with an analytical tool. In this section, we will describe each of these methods and discuss how they operate from both a usability and a performance perspective.

3.1 Client-Server Connection

The standard method of combining a standalone program with a RDBMS is through a client-server connection. This is visualized in Figure 2-2a. The database server is completely separate from the analytical tool. It runs as either a separate process on the same machine or on a different machine entirely. The analytical tool communicates with the database server through a socket connection through an application programming interface (API). After an initial authentication phase, the client can issue a query to the database server. The server will then execute the query. Afterwards, the result of the query will be serialized and written to the client over the socket. Finally, the client will deserialize the result.

The main advantage to this approach is that it is mostly database agnostic, as the standardized ODBC [32] or JDBC [24] connectors can be used to connect to almost every database. In addition, it is relatively easy to integrate into existing pipelines as loading from flat files can be replaced by loading from a database without having to modify the rest of the pipeline.

However, this approach is problematic when the client wants to run their analysis pipelines on a a large amount of data. The time spent on serializing large result sets and transferring them from the server to the client can be a significant bottleneck. In addition, this approach requires the full dataset to fit inside the clients' (often limited) memory.

3.2 In-Database Processing

In order to avoid the cost of exporting the data from the database, the analysis can be performed inside the database server. This method, known as in-database processing, is shown in Figure 2-2b.

In-database processing can be performed in a database-agnostic way by rewriting the analysis pipeline in a set of standard-compliant SQL queries. However, most data analysis, data mining and classification operators are difficult and inefficient to express in SQL. The SQL standard describes a number of built-in scalar functions and aggregates, such as *AVG* and *SUM* [43]. However, this small number of functions and aggregates is not sufficient to perform complex data analysis tasks [86].

Instead of writing the analysis pipelines in SQL, user-defined functions or user-defined aggregates in procedural languages such as C/C++ can be used to implement classification and machine learning algorithms. This is the approach taken by Hellerstein et al. [36]. However, these functions still require significant rewrites of existing analytical pipelines written in vectorized scripting languages. In addition, writing user-defined functions in these languages require in-depth knowledge of the database internals and the execution model used by the database [14].

3.3 Embedded Databases

Both the client-server model and in-database processing require the user to maintain a running database server. This requires significant manual effort from the user, as the database server must be installed, tuned and continuously maintained. For small-scale data analysis, the effort spent on maintaining the database server often negates the benefits of using one.

Embedding the database system inside the client program, as shown in Figure 2-2c, is more applicable for these use cases. As the database can be installed and run from within the client program, maintaining and setting up the database is much simpler than with standalone database servers. As the database resides directly inside the client process, the cost of transferring data between the client and the database server is negated. The primary disadvantage of this solution is that only a single client can have access to the data stored inside the database server.

4 MonetDB

MonetDB is an open source column-store RDBMS that is designed primarily for data warehouse applications. In these scenarios, there are frequent analytical queries on the database, often involving only a subset of the columns of the tables, and unlike typical transactional workloads, insertions and updates to the database are infrequent and in bulk or do not occur at all. The core design of MonetDB is described in Idreos et al. [41]. However, since this publication a number of core features have been added to MonetDB. In this section, we give a brief summary of the internal design of MonetDB and describe the features that have been added to MonetDB since.

Data Storage. MonetDB stores relational tables in a columnar fashion. Every column is stored either in-memory or on-disk as a tightly packed array. Row-numbers for each value are never explicitly stored. Instead, they are implicitly derived from their position in the tightly packed array. Missing values are stored as "special" values within the domain of the type, i.e. a missing value in an `INTEGER` column is stored internally as the value -2^{31}.

Columns that store variable-length fields, such as CLOBs or BLOBs, are stored using a variable-sized heap. The actual values are inserted into the heap. The main column is a tightly packed array of offsets into that heap. These heaps also perform duplicate elimination if the amount of distinct values is below a threshold; if two fields share the same value it will only appear once in the heap. The offset array will then point to the same heap entry for the rows that share the same value.

Memory Management. MonetDB does not use a traditional buffer pool to manage which data is kept in memory and which data is kept on disk. Instead, it relies on the operating system to take care of this by using memory-mapped files to store columns persistently on disk. The operating system then loads pages into memory as they are used and evicts pages from memory when they are no longer being actively used. This model allows it to keep hot columns loaded in memory, while columns that are not frequently touched are off-loaded to disk.

Concurrency Control. MonetDB uses an optimistic concurrency control model. Individual transactions operate on a snapshot of the database. When attempting to commit a transaction, it will either commit successfully or abort when potential write conflicts are detected.

Query Plan Execution. SQL is first parsed into a relational algebra tree and then translated into an intermediate language called MAL (Monet Assembly Language). MAL instructions process the data in a column-at-a-time model. Each MAL operator processes the full column before moving on to the next operator. The intermediate values generated by the operators are kept around in-memory if not too large, and passed on to the next operator in the pipeline.

Optimizations happen at three levels. High level optimizations, such as filter push down, are performed on the relational tree. Afterwards, the MAL code is generated and further optimizations are performed, such as common sub-expression elimination. Finally, during execution tactical decisions are made about how specific operations

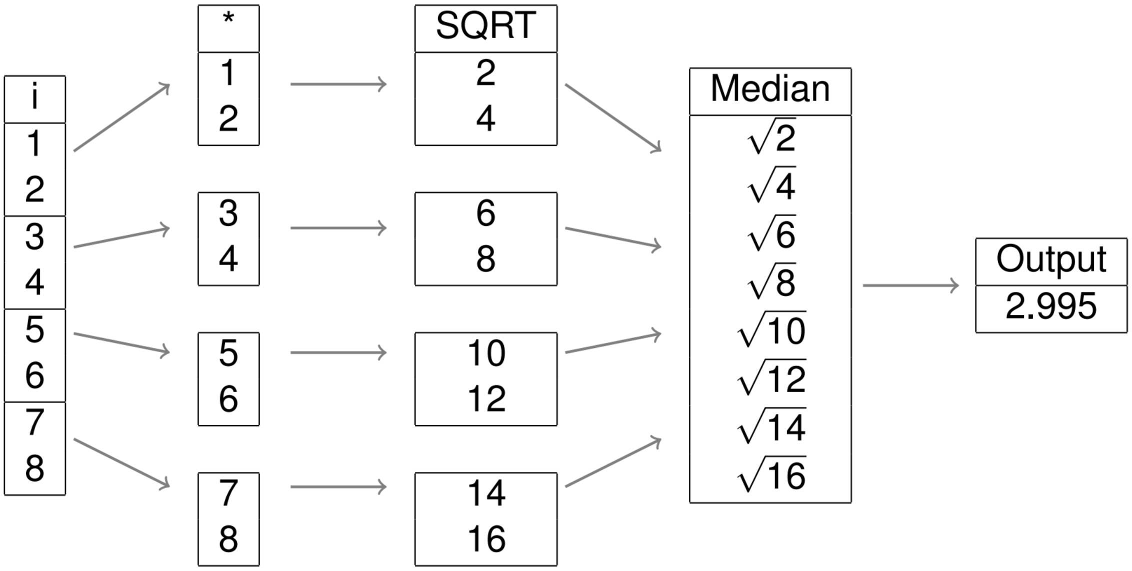


Figure 2-3: Parallel execution in MonetDB.

should be executed, such as which join implementation to use.

Parallel Execution. Initially, a sequential execution plan is generated. Parallelization is then added in the second optimization phase. The individual MAL operators are marked as either "blocking" or "parallelizable". The optimizers will alter the plan by splitting up the columns of the largest table into separate chunks, then executing the "parallelizable" operators once on each of the chunks, and finally merging the results of these operators together into a single column before executing the "blocking" operators. This is visualized in Figure 2-3 for the query `SELECT MEDIAN(SQRT(i * 2)) FROM tbl`.

The amount of chunks that are generated is decided by a set of heuristics based on base table size, the amount of cores and the amount of available memory. The database will attempt to generate chunks that fit inside main memory to avoid swapping, and will attempt to maximize CPU utilization. In addition, the optimizer will not split up small columns as the added overhead of parallel execution will not pay off in this case.

Automatic Indexing. In addition to allowing the user to manually build indices through the `CREATE INDEX` commands, MonetDB will automatically create indices during query execution.

Imprints [75] are a bitmap index that are used to assist in efficiently computing point and range queries. The bitmap index holds, for each cache line, a bitmap that contains information about the range of values in that cache line. They are automatically generated for persistent columns when a range query is issued on a specific column. They are then persisted on disk and used for subsequent queries on that column. Imprints are destroyed when a column is modified.

Hash tables are also automatically created for persistent columns when they are used in groupings or as join keys in equi-joins. These are also persisted on disk. Hash tables are destroyed on updates or deletions to the column. Unlike imprints, however, they are updated on appends to the tables.

Order Index. In addition to imprints and hash tables, MonetDB supports creation of a sorted index that is not created automatically. It must be created using the `CREATE ORDER INDEX` statement. Internally, the order index is an array of row numbers in the sort order specified by the user. The order index is used to speed up point and range queries, as well as equi-joins and range-joins. Point and range queries are answered by using a binary search on the order index. For joins, the order index is used for a merge join.

5 Python

Python is a popular interpreted language, that is widely used by data scientists. It is easily extensible through the use of modules. There are a wide variety of modules available for common data science tasks, such as `numpy`, `tensorflow`, `scipy`, `sympy`, `sklearn`, `pandas` and `matplotlib`. These modules offer functions for high performance data analytics, data mining, classification, machine learning and plotting.

While there are various Python interpreters, the most commonly used interpreter is the CPython interpreter. This interpreter is written in the language *C*, and provides bindings that allow users to extend Python with modules written in *C*.

Internally, CPython stores every variable as a `PyObject`. In addition to the value this object holds, such as an integer or a string, this object holds type information and a reference count. As every `PyObject` can be individually deleted by the garbage collector, every Python object has to be individually allocated on the heap.

The internal design of CPython has several performance implications that make it unsuitable for working with large amounts of data. As every `PyObject` holds a reference count (64-bit integer) and type information (pointer), every object has 16 bytes of overhead on 64-bit systems. This means that a single 4-byte integer requires 20 bytes of storage. In addition, as every `PyObject` has to be individually allocated on the heap, constructing a large amount of individual Python objects is very expensive.

Instead of storing every individual value as a Python object, packages intended for processing large amounts of data work with NumPy arrays instead. Rather than storing a single value as a `PyObject`, a NumPy array is a single `PyObject` that stores an array of values. This makes this overhead less significant, as the overhead is only incurred once for every array rather than once for every value.

This solves the storage issue, but standard Python functions can only operate on `PyObjects`. Thus if we want to actually operate on the individual values in Python, we would still have to convert each individual value to a `PyObject`.

The solution employed in Python (and other vector-based languages) is to have *vectorized functions* that directly operate on all the data in an array. By using these functions, the individual values are never loaded into Python. Instead, these vectorized operations are written in *C* and operates directly on the underlying array. As these functions operate on large chunks of data at the same time they also make liberal use of *SIMD* instructions, allowing these vectorized functions to be as fast as optimized *C* implementations.