Distributed Processing and Transaction Replication in MonetDB - Towards a Scalable Analytical Database System in the Cloud

Ying Zhang[1], Dimitar Nedev[2]*, Panagiotis Koutsourakis[1], and Martin Kersten[1,3]

[1] MonetDB Solutions, Amsterdam, The Netherlands
{firstname.lastname}@monetdbsolutions.com
[2] Picnic, Amsterdam, The Netherlands
dimitar.nedev@teampicnic.com
[3] Centrum Wiskunde & Informatica, Amsterdam, The Netherlands
martin.kersten@cwi.nl

Abstract. Thanks to its flexibility (i.e. new computing jobs can be set up in minutes, without having to wait for hardware procurement) and elasticity (i.e. more or less resources can be allocated to instantly match the current workload), cloud computing has rapidly gained much interests from both academic and commercial users. Increasingly moving into the cloud is a clear trend in the software developments. To provide its users a fast in-memory optimised analytical database system with all the conveniences of the cloud environment, we embarked upon extending the open-source column store database MonetDB with new features to make it cloud-ready. In the paper, we elaborate the new distributed and replicated transaction features in MonetDB. The distributed query processing feature allows MonetDB to horizontally scale-out to multiple machines; while the transaction replication schemes increase the availability of the MonetDB database servers.

1 Introduction

Cloud computing has become one of the leading utilities for data processing. Cloud services help both the academic researcher and commercial organisations in getting more insights out data with fewer resources. One of the main advantages of cloud computing is that in just a few minutes one can launch a high-performance virtual instance running in the cloud and shut it down once its now longer needed ([**?**], [**?**]). This significantly shortens the time it takes to bring an idea from inception to implementation.

While previously one often have to wait for authorisation or even procurement of computing resources, nowadays accessing high-performance machines can be just a click away [**?**]. Such convenience was previously only available to those with constant access to the hardware and software needed for their job. In the case of large-scale data analytics, more powerful machines would have been required, thus more resources have to be spent on their procurement. In commercial organisations, buying hardware generally falls under capital expenses, which are often planned a year in advance. Hence, the decision to buy new hardware required for a project must be made well on time. In

* Work conducted while working at MonetDB Solutions.

addition, with fluctuating prices and new hardware models and software versions frequently being introduced, it is virtually impossible to make accurate estimations so far in advance. Consequently, a project with such a fixed plan often has to live with either an underestimation with possible negative impact on performance, or an overestimation which potentially wastes budget and energy.

In comparison, costs of cloud instances generally fall under operational expenses, which can be allocated in much shorter periods. Moreover, cloud-computing resources are only used when needed and can be immediately shut down afterwards [**?**]. Given this, one can consider cloud resources one of the common utilities of the 21st century. In this respect, cloud service providers are nowadays comparable to gas, electricity and telephone companies [**?**].

With the advent of large, easily accessible compute clusters, analysis of primarily key-value file stores has taken on a wave. Map-Reduce [**?**] is the underlying programming technique to scale out. The rigid database scheme is replaced by a schema oblivious solution, which works well for single scan data aggregation. As the need for persistence increased, updates and application interaction became more pressing. The NoSQL cloud data stores are expanding their reach into the traditional structured database arena, albeit mostly focused on re-engineering a subset of the SQL'92 functionality. Conversely, SQL-based systems, as always, are incorporating the essential ingredients of the NoSQL world to provide the functionality favoured in NoSQL systems. Consequently, an ever-growing collection of vendors provides Cloud-based SQL-like database offerings (for short: *cloud database*) today[4]. Cloud databases typically have the following features:

- Web-based consoles, which can be used by the end users to provision and configure database instances.
- A database manager component, which controls the underlying database instances using a service API.
- Making the underlying software stack transparent to the user - the stack typically includes the operating system, the database and third-party software used by the database.
- Database services take care of scalability and high availability of the database.

The MonetDB open-source columnar database management system (DBMS) has a long track-record in business analytics, and it is often used as the back end database engine by various scientific applications[5] to manage large volumes of data. A cloud deployment of MonetDB will give its users the fast query responses of in-memory optimised database, while maintaining the convenience of immediate access to computing resources. Therefore, in the European project CoherentPaaS[6], we embarked upon introducing features into MonetDB to increase its usability as a cloud database.

In our previous paper [**?**], we described the deployment automation process we have built to deploy MonetDB in the cloud environments. The cloud deployment of MonetDB gives a wider range of users easy access to a high-performance database. Another

[4] `https://en.wikipedia.org/wiki/Cloud_database`
[5] `https://monetdb.org/Home/ProjectGallery`
[6] `http://coherentpaas.eu/`

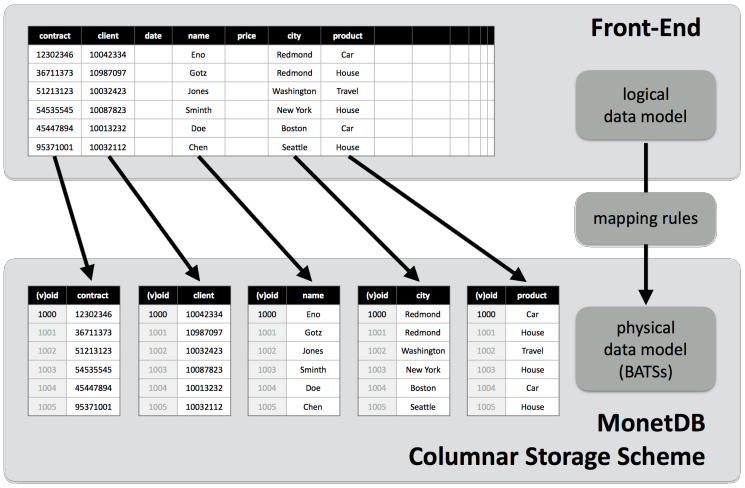

Fig. 1. MonetDB column-oriented storage model.

step we have taken to make MonetDB cloud read is to improve its horizontal scalability (i.e. scale-out) and availability. Scale-out scalability is enabled by introducing distributed query processing features; while higher availability comes from replicated databases. We describe both in details in this paper.

Deployment automation and distributed and replicated transactions together make MonetDB more appealing to a larger set of users. Next to the ease of deployment, scalability and high availability are critical requirements for both researcher and commercial database users. With the work we have done, we aim to bring MonetDB in the Cloud age, giving data analysts a powerful tool that is only a single-click away.

This paper is further organised as follows. First, we give a brief overview of the MonetDB software suite in Section 2. Then, we zoom in on the MonetDB features that make MonetDB more suitable for a cloud environment. This includes *i)* how transactions are managed by a MonetDB server (Section 3); *ii)* how one can register data partitioned over a cluster of MonetDB instances to allow running distributed queries on the data (Section 4); and *iii)* how transactions can be replicated on multiple MonetDB instances for higher availability and reliability (Section 5). Finally, we conclude in section 6.

2 MonetDB Overview

MonetDB is an open-source DBMS for high-performance applications in data analysis, business intelligence, online analytical processing (OLAP), geographic information system (GIS) and data warehousing. These applications are characterised by very large databases, which are mostly queried to provide business intelligence or decision support. Similar applications also appear frequently in the area of e-science, where results from experiments are stored in a scalable system for subsequent scientific analysis.

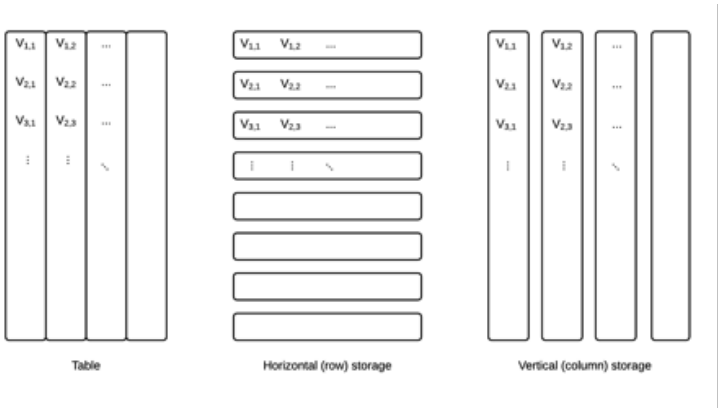

Fig. 2. Row vs. column oriented storage model.

The design of MonetDB is built around the concept of *bulk processing*, i.e. simple operations applied to large volumes of data, which enables efficient use of the modern hardware for large-scale data processing. This focus on bulk processing is reflected at all levels of the architecture and the functionality offered to the user. MonetDB achieves its goal by innovations at all layers of a DBMS, e.g. a storage model based on vertical fragmentation (column store), modern CPU-optimised query execution architecture, automatic and self-tuning indexes, and run-time query optimisation [**?**].

One of the main differences between MonetDB and the traditional database systems, such as PostgreSQL, MS SQL server and DB2, is the storage model of MonetDB, which is shown in Figure 1. Unlike the traditional database systems that store data in rows, MonetDB stores data in columnar format and the columns are virtually stitched together to form tables. Figure 2 shows the main difference the row-based and the column-based storage model. The vertical storage gives column-stores some advantage in read-intensive workloads, such as data analytics. At the same time MonetDB is a fully functional relational database. It provides an SQL:2003 compatible query layer and connectors for many commonly used programming languages. This means that the common tools and applications designed for working with SQL systems can easily adopt MonetDB. In addition, MonetDB makes use of a number of novel techniques for efficient support of a priori unknown or rapidly changing workloads [**?**]. *Recycling* [**?**] is a fine-grained flexible intermediate result caching technique. *Database cracking* [**?**] is an adaptive incremental indexing technique.

To further improve data analysis capabilities, MonetDB has been integrated with the R statistical analysis software [**?**]. This works both ways: *i)* a connection to a MonetDB database can be established from inside R to run SQL queries, and *ii)* user defined SQL functions can be implemented in R and evaluated directly in MonetDB. Statistical packages are optimised for advanced algorithms, while database systems provide fast access to large volumes of data. Their combination creates a powerful platform to speedup data discovery. The MonetDB.R connector decides which portions of the data analysis should be performed by either the statically software or the database. In this way, each

systems is used to its best performance. The users are freed from the tedious task of shuffling data around. This also significantly reduces the overhead of transferring data between different systems, which can significantly reduce data processing times [**?**].

3 MonetDB Transaction Management Scheme

MonetDB is primarily designed as an analytical (i.e. OLAP) database, where most operations are `SELECT-FROM-WHERE-GROUPBY` queries. Therefore, MonetDB has consciously chosen for a lightweight optimistic transaction management scheme that is tuned for reading large chunks of data, rather than writing small chunks of data at high speed concurrently. Nevertheless, MonetDB fully supports the Atomicity, Consistency, Isolation, and Durability (ACID) properties of transactions of the SQL standard.

3.1 Optimistic Concurrency Control and Snapshots Isolation

The MonetDB transaction management scheme is an Optimistic Concurrency Control (OCC) scheme[7] realised by implementing Snapshot Isolation[8] in its multiversion concurrency control (MVCC)[9]. MVCC is a common way to increase concurrency and performance; and Snapshot Isolation allows better performance than *serialisability*, while avoiding most (although not all) of the concurrency anomalies that serialisability avoids.

In MonetDB, a snapshot of the database in its latest stable state is created for each transaction. Subsequently, the whole transaction is executed on its private snapshot, thus avoid the need to acquire any locks on, e.g. the database. With concurrent transactions, multiple versions of the database can coexist in the system. Conflict is only detected, when a transaction tries to commit. In that case, the transaction will be aborted and its snapshot (which includes all changes made by this transaction) is discarded. In MonetDB, there are two main types of conflicts: *i)* concurrent transactions updating the same table, and *ii)* a schema-altering transaction and any update transactions.

OCC is particularly useful for read-heavy applications, with the design that transaction management overhead should only be paid when necessary. While providing each transaction with a consistent view on the database, updates are collected in an addendum, which is only processed on transaction commit. For each relational table, a so-called *delta column* with deleted positions is kept. Deltas are designed to delay updates to the main columns, and allow a relatively cheap Snapshot Isolation mechanism. However, a downside of OCC is its potentially negatively impact long running transactions, as these can be affected by concurrent updates on their underlying tables. The same holds for applications that try to perform concurrent updates to the same tables from multiple application threads or to create/modify the database schema concurrently. Although a transaction aborted due to concurrency can be safely rerun until it succeeds, it often causes (unnecessary) resource contention. A better alternative is to let the application serialise the transactions.

[7] `https://en.wikipedia.org/wiki/Optimistic_concurrency_control`
[8] `https://en.wikipedia.org/wiki/Snapshot_isolation`
[9] `https://en.wikipedia.org/wiki/Multiversion_concurrency_control`

3.2 Transaction WAL log

The atomicity and durability of transactions are guaranteed in MonetDB by using Write-Ahead Logs (WAL)[10]. When a transaction is being executed, instead of applying its changes directly on the base tables (in its private snapshot), all changes are both recorded in two auxiliary tables[11], as well as written in the associated WAL log file. The WAL log contains the state of the transaction and all other information needed to recover a transaction. Committing a transaction in MonetDB means persisting the associated WAL log file on the hard disk. However, to make the changes visible to all following transactions, the auxiliary `ins` and `delta` tables are merged with the primary database in a logical view.

All information of one transaction is recorded in one log file, while a log file can contain multiple transactions. Multiple log files can coexist. However, usually, there is only one transaction file in the WAL directory - the current, not yet committed one. The WAL directory is by default under `<database directory>/sql_logs`. Inside it, there is a directory `sql` for the SQL module transactions. Different modules or top-level interfaces can have different WAL directories. The inner directory contains the following files:

- `log`: the WAL catalog file.
  - The first line of this file is the internal MonetDB kernel WAL version.
  - The next line is empty.
  - The third line is the latest transaction id, stored as a 64bit integer.
- `log.<transaction id>`: each of these files stores a single MonetDB transaction
  - Transaction data, partially binary

3.3 Garbage Collection and Transaction Recovery

In MonetDB, most data are garbage collected automatically, for instance, intermediate data created during a query execution are automatically cleaned up (e.g. memories are freed, temporary files are deleted); snapshots of aborted transactions are discarded and automatically cleaned. Only the transaction Write Ahead Log (WAL) files need to be explicitly garbage collected. A WAL file can contain information of both committed and aborted transactions, because they are append-only. If a transaction eventually aborts, only an ABORT record for this transaction will be added to the log file. None of the existing data in the log file is altered.

During a server session, the MonetDB server will clean up the log files at a regular time interval (currently, 30 seconds) and if there are sufficient number of changes (currently, 1 million). The garbage collection process works as the following:

- Apply the changes made by all committed transactions on the persistent database.
  - Merge the `ins` table with the base table.
  - Persist the `delta` table on disk.

[10] `https://en.wikipedia.org/wiki/Write-ahead_logging`

[11] One `ins` table for all inserted tuples. One `delta` table with the positions of the deleted tuples. Updates are handled as an atomic delete+insert.

– Discard all log records belonging to a aborted transaction.

During a database startup, all transaction still left in the WAL are read and any changes of committed transactions that are not yet applied are written into the persistent BAT storage. Changes of uncommitted and aborted transactions are discarded. Hence, in case of failure, all transactions that were reported as committed shall be present, once the database is started up and made available to the users.

After the process of applying the log files on the persistent database, database files that are no longer needed (e.g. corresponding files of a dropped table) are automatically deleted. Finally, once a log file is completely cleaned up (i.e., all transactions in the file are either applied or discarded), the file is removed from the persistent storage.

4 MonetDB as a Distributed Database

As of early 2014, as part of the CoherentPaaS project, we have been working on introducing data partitioning and distributed query processing features into MonetDB. The new features were released in 2015. Together, they enable transparently running queries at a central MonetDB server over data partitioned over a cluster of MonetDB instances. In this section, we describe how each of them works.

4.1 Data Partitioning

The prime method to enable finer control of locality of data access during query evaluation is to horizontally partition the tables in a database. This is made possibly by introducing the concept of *merge table*, which allows a table to be defined as the union of its partitions. Since the Jul2015 release, this partitioning logic has been moved up the MonetDB software stack to the SQL layer, where tables can be easily defined as the union of its partitions, similar to the merge table approaches found in other database management systems, such as in MySQL[12].

To create a merge table in MonetDB, one simply uses a `CREATE MERGE TABLE` statement. The SQL statements below show how to create a `MERGE TABLE mt` and populate it with two partitions `t1` and `t2`. Note that the partitions must contain the same column signature (i.e. same number of columns of the same data type declared in the same order). Then the partition tables can be queried both individually and jointly through the merge table.

```
-- Create a MERGE TABLE
CREATE MERGE TABLE mt1 (t int);

-- Partition tables mush have identical column types. Column names are insignificant.
CREATE TABLE t1 (i int);
CREATE TABLE t2 (j int);
INSERT INTO t1 VALUES (11), (13);
INSERT INTO t2 VALUES (23), (27);

-- Add the partition tables into the MERGE TABLE
ALTER TABLE mt1 ADD TABLE t1;
```

[12] `https://dev.mysql.com/doc/refman/5.7/en/merge-table-advantages.html`

```
ALTER TABLE mt1 ADD TABLE t2;

sql> -- sanity check
sql> SELECT count(*) FROM mt1;
+------+
| L1   |
+======+
|    4 |
+------+
```

Two `MERGE TABLE`s can contain overlapping partition tables. A `MERGE TABLE` can also contain other `MERGE TABLE`s:

```
CREATE TABLE t3 (k int);
INSERT INTO t3 VALUES (31), (37);

CREATE TABLE t4 (l int);
INSERT INTO t4 VALUES (41), (47);

-- An overlapping MERGE TABLE (with mt1)
CREATE MERGE TABLE mt2 (t int);
ALTER TABLE mt2 ADD TABLE t1;
ALTER TABLE mt2 ADD TABLE t3;

-- A MERGE TABLE of MERGE TABLE
CREATE MERGE TABLE mt3 (t int);
ALTER TABLE mt3 ADD TABLE mt1;
ALTER TABLE mt3 ADD TABLE t4;

sql> -- sanity check
sql> SELECT count(*) FROM mt2;
+------+
| L1   |
+======+
|    4 |
+------+
sql> SELECT count(*) FROM mt3;
+------+
| L1   |
+======+
|    6 |
+------+
```

One can remove a partition table from a `MERGE TABLE` using an `ALTER TABLE ... DROP TABLE ...` statement, as shown below. Together with `ALTER TABLE ... ADD TABLE ...` statements, one can dynamically change the contents of a `MERGE TABLE` without affecting the underlying partition tables. However, when dropping a partition table, all MERGE TABLE-s that depend on it must be dropped first.

```
-- Remove a partition table from a MERGE TABLE
ALTER TABLE mt1 DROP TABLE t1;

sql> -- sanity check
sql> SELECT count(*) FROM mt1;
+------+
| L1   |
+======+
|    2 |
+------+
sql> -- Drop table must happen in the correct order of dependency
sql> DROP TABLE t2;
DROP TABLE: unable to drop table t2 (there are database objects which depend on it)
```

```
sql> DROP TABLE mt3;
operation successful (3.048ms)
sql> DROP TABLE mt1;
operation successful (1.948ms)
sql> DROP TABLE t2;
```

4.2 Distributed Query Processing

Distributed query processing has been introduced since the Jul2015 release of MonetDB through the support for *remote tables*. The remote table technique complements merge table by allowing the partitions of a merge table to reside on different databases. Queries involving remote tables are automatically split into subqueries by the master database server and executed on remote databases. The combination of merge table and remote table enables fine control of distributing data and query workloads to maximally benefit from the available CPU and RAM resources.

Remote table adopts a straightforward *master-worker architecture*: one can place the partition tables in different databases, each served by a worker MonetDB server, and then glue everything together in a `MERGE TABLE` in the master database served by the master MonetDB server. Each MonetDB server can act as both a worker and a master, depending on the roles of the tables it serves. The shell commands and SQL queries below show how to place the partition tables `t1` and `t2` on two worker databases, and glue them together on a master database.

The format of the address of a `REMOTE TABLE` is:

```
mapi:monetdb://<host>:<port>/<dbname>
```

where all three parameters are compulsory. The MonetDB internal communication protocol "mapi" is currently used to exchange subquery plans and results between the master and the workers. A worker database can reside on a local cluster node or a remote machine. Note that, the declaration of a `REMOTE TABLE` must match exactly the signature of its counterpart in the remote database, i.e. the same table name, the same columns names and the same column data types. Also note that, currently, at the creation time of a remote table, the remote database server is not contacted to verify the existence of the table. Hence, when a `CREATE REMOTE TABLE` reports "operation successful", it merely means that the information about the new `REMOTE TABLE` has been added to the local SQL catalogue. The check at the remote database server is delayed until the first actual query on the remote table.

```
-- Start a server and client for worker-database1, and create partition table1
$ mserver5 --dbpath=<path-to>/rt1 --set mapi_port=50001
$ mclient -d rt1 -p 50001
sql> CREATE TABLE t1 (i int);
sql> INSERT INTO t1 VALUES (11), (13);

-- Start a server and client for worker-database2, and create partition table2
$ mserver5 --dbpath=<path-to>/rt2 --set mapi_port=50002
$ mclient -d rt2 -p 50002
sql> CREATE TABLE t2 (j int);
sql> INSERT INTO t2 VALUES (23), (27);

-- Start a server and client for the master-database,
--   and create a MERGE TABLE containing two REMOTE TABLEs
```

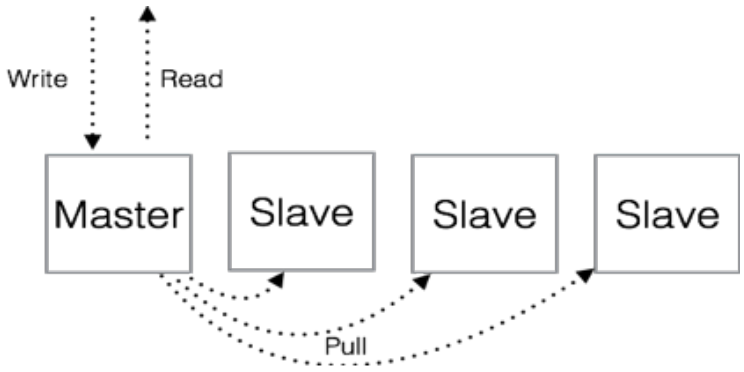

Fig. 3. MonetDB transaction replication with warm standby configuration.

```
$ mserver5 --dbpath=<path-to>/mst
$ mclient -d mst
sql> CREATE MERGE TABLE mt1 (t int);
sql> -- Identify t1, t2 as REMOTE TABLEs with their locations
sql> CREATE REMOTE TABLE t1 (i int) on 'mapi:monetdb://localhost:50001/rt1';
sql> CREATE REMOTE TABLE t2 (j int) on 'mapi:monetdb://localhost:50002/rt2';
sql> -- Add the remote tables into the MERGE TABLE
sql> ALTER TABLE mt1 ADD TABLE t1;
sql> ALTER TABLE mt1 ADD TABLE t2;
sql> -- Sanity check:
sql>SELECT count(*) from t1;
+------+
| L1   |
+======+
|    2 |
+------+
sql>SELECT count(*) from t2;
+------+
| L1   |
+======+
|    2 |
+------+
sql>SELECT count(*) from mt1;
+------+
| L1   |
+======+
|    4 |
+------+
```

5 Replication Services in MonetDB

High-availability is critical for production environments. Downtime in an academic research organisation can cause one to miss important deadline or delays in research results. Availability of mission critical commercial systems is of even higher importance. If the data warehouse goes down, neither business reporting nor planning can be done. High-availability is often achieved via redundancy of critical components. For database management systems this is achieved using multiple instances of the database running in parallel. The data is continuously synchronised between the instances. This ensures that in case a single server is lost, the other replicas can still serve queries.

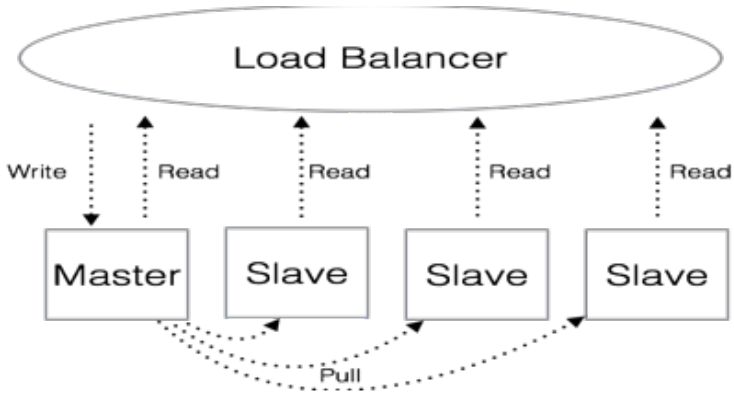

Fig. 4. MonetDB transaction replication with active-active configuration with read-only slaves.

To increase the availability of MonetDB servers, we have extended the system with support for transaction replication. This is achieved via log shipping (of the transaction logs) of a master instance to a number of slave instances. It is generally described as a pull model [**?**], where each slave pulls transactions independently and asynchronously. There is no master-slave information exchange (aside from transactions). Formally this is considered Lazy Centralised replication with Limited Transparency [**?**].

By default the MonetDB kernel stores transactions in Write-Ahead Log (WAL) files, before the transaction are persisted in the primary persistent storage. During the database start up, the transaction log files are read and any data changes non-persisted in the primary storage are applied. A MonetDB slave instance can be configured to read the WAL files of a master instance, load the transactions and persist the data in its own persistent storage.

On the master instance, the MonetDB should be configured to keep all transaction log files, even those for transactions already persisted in the primary storage. By default, the database cleans- up persisted transaction log files. The transaction log files on the master have to be shipped to the slave instance(s), which can be done using a highly available shared file system or alternative means.

On a slave instance, the location of the master transaction log files must be configured. In addition, the transaction drift threshold between the slave and the master must be set. The drift is the difference between the transactions processed by the master and the slave. If a slave detects that it has passed a pre-set threshold, it will not process any additional client read queries until it catches up with the master. A slave must also be set to run in read-only mode.

The master is the only instance that can apply changes to the data (e.g. create, update, delete) to avoid any data inconsistencies. As such all slave instances must run in read-only mode, where data changes will be propagated only through the transaction-replication mechanism.

There are two possible cluster configurations, each with its pros and cons:

– Warm standby slaves (see Figure 3)

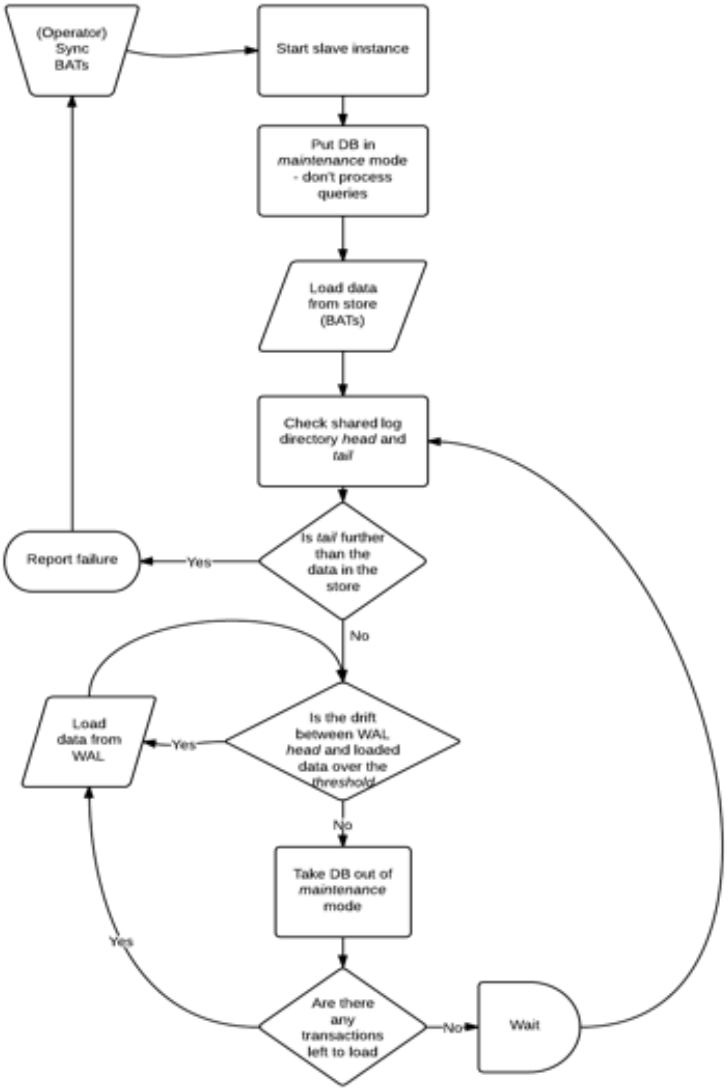

Fig. 5. MonetDB transaction processing on a slave instance.

- Single master instance, which can do read and write operations.
- One or more slave instances processing no queries, only replicate the master transactions asynchronously.
- Upon master failure, a slave instance can be restarted in non-read-only mode, to take the role of a master.

The warm standby configuration has the advantage that it can provide increased fault tolerance and is relatively simple to setup. Since the instance loading the data can

be queried only, the queries operate on an always up-to-data store. The most significant disadvantage is the somewhat wasteful use of resources, as the slaves will to do any query processing.

- Active-active with read-only slaves (see Figure 4)
  - Single master instance, which can do read and write operations
  - One or more slave instances that can do read operations, next to replicating the master transactions asynchronously
    - These read-only instances can improve the read query load
    - Load-balancing must be provided on the client side
  - Upon master failure, a slave instance can be restarted in non-read-only mode, to take the role of a master.

The active-active configuration provides improved query capacity, compared to the single read instance warm standby. At the same time, queries sent to the slave instances can be executed on not-up-to-date data, since the transaction replication is asynchronous. This setup also comes at the price of increased complexity at the client side, due to the load balancing required for query distribution.

MonetDB is primarily designed as an analytical database. As such, data is best loaded in large bulk transactions. This will also guarantee that only single large files are shipped to the slaves for replication, minimising the transaction drift.

To set-up MonetDB transaction replication, first and foremost, the shipping of the transaction log files between the master and the slave instance(s) must be configured. The most straightforward way is to use a shared file system that can also provide high-availability for the master transaction log files. It is advisable to choose a shared file system that is also fault tolerant, such that loss of the master MonetDB instance will not lead to loss of the master transaction logs as well. This way the shared file system provides both log shipping, as well as log backup. Since the master instance preserves all transaction log files, there will be a complete copy of the database (in form of the transactions).

Good examples of such file system include Ceph and GlusterFS [**?**] (Write once, read everywhere, 2014). For public cloud deployment, the native storage of the cloud provider can be used. For example, in AWS one can setup WAL file replication to Amazon S3. S3 has high durability and availability [**?**], making it ideal for both log shipping and backup. If the MonetDB transaction log is written on S3, slave instances can read the files directly from the remote store over HTTP or mounted locally in user space. At the same time the high durability guarantees constant backup of the data. The main drawback is that S3 is object based [**?**], and on file update, the complete file must be uploaded. In the case of MonetDB, as the older transaction log files are not modified, the effect will be minimal. In addition, if the data is loaded in large bulk transactions, the impact is further minimised, since there will not be many files.

For the implementation of the transaction replication support we decided to take the path with least risks. First we evaluated the existing transactions logging scheme. On a slave instance, if the correct flags are set, the database will set up a second, read-only logger. Every few seconds the logger will scan its own (shared) directory and if new transactions are detected, it will load them and persist them in the local storage.

If new tables or schemas were created in that transaction, these changes will not be visible immediately. To fix it, we also party reload the SQL store, forcing it to update the schemas and tables.

The mode of operation of a slave instance is shown on Figure 5. The instance will start up in maintenance mode, not processing any queries. Any data in the slave's own WAL will be persisted first. Next the WAL synced from the master will be examined. The salve will verify if it can replicate all data. In case the tail of master data is further than the head of the transactions on the slave, the instance will report failure. An operator/DBA must manually sync the instance coping the persisted data from the master to the slave. If tail of the master transaction is behind the head of the salve, the slave will then verify if the drift between the two instances is past the threshold. If that is the case, the slave will load all needed transaction from the synced master WAL. Once it's passed the threshold, it will unlock the store and begin to process read queries. At regular time interval the synced master WAL will be re-examined and the replication process begins anew.

On the master instance, as little as possible changes were done, in order not to compromise the performance of the database. The only major change is that more than one transaction is now preserved. This is needed since there is no communication between the master and the slave instance and all transactions that have to be replicated at the slaves need to be available.

6 Conclusion and Future Work

Database systems running in the cloud have significantly reduced the time to process large volumes of data. To support this process, many database management systems have also adapted to the paradigm of the cloud age. As part of the cloud integration of MonetDB, we focused on the activities that serve the needs of data analysis best. The lightweight optimistic transaction management scheme of MonetDB favours bulk updates that are typical in OLAP applications. With the introduction of merge table and remote table for distributed data processing, MonetDB starts expanding its scalability from mainly vertical scale-up to horizontal scale-out, which is of particular importance in cloud-like environments. The transaction replication support, in addition, enables high-availability of running the MonetDB instances. The log shipping based approach reuses as much as possible the existing system. This way the risk and time to market of this solution is minimised.

With the features introduced mainly in the context of the CoherentPaaS project, MonetDB is accelerating towards a scalable analytical cloud database system to help its users gaining insights from their data in lightening speed. However, there is still much work to be done. We will continue hardening and extending MonetDB's support for distributed and replicated transactions, which will mainly be conducted in the context of the ExaNeSt project. For instance, currently, a `MERGE TABLE` cannot be directly updated. All updates must be done on the individual partition tables. We plan to also allow updating queries on `MERGE TABLE`s, which effectively support distributed updates on the `REMOTE TABLE`s through a `MERGE TABLE` on the master node. To implement this

feature, the 2Phase-like transaction management scheme we have implemented in CoherentPaaS can be nicely adopted.

Acknowledgements

This works was partially supported by the EU FP7-ICT-2013-10 project CoherentPaaS (`http://coherentpaas.eu/`). This work was partially carried out within the ExaNeSt project (`www.exanest.eu`), funded by the European Union's Horizon 2020 research and innovation programme under grant agreement No 671553.