



## ACTiCLOUD: ACTivating resource efficiency and large databases in the CLOUD

Project No: 732366

H2020-ICT-2016-1

Technical Report on MonetDB ACTiCLOUD extensions v1.0

18/12/2018

### **Executive summary:**

This technical report is based on ACTiCLOUD's Deliverable 3.3 and presents three extensions developed in the context of ACTiCLOUD to advance the columnar database MonetDB towards an ACTiCLOUD-enabled data center: i) MALCOM, an incremental memory footprint estimation tool for MonetDB; ii) MonetDBLite-Java, an embedded version of MonetDB for JVM; and iii) MonetDB's support for distributed query processing. For each extension, we first describe its initial design and implementation; then we discuss some initial assessments of its functionality/performance; finally we make plans for future work.

**List of authors:**

Author	Affiliation
Pedro Ferreira	MDBS
Pavlos Katsogridakis	MDBS (during his internship)
Panagiotis Koutsourakis	MDBS
Joeri van Ruth	MDBS
Ying Zhang	MDBS

**ACTiCLOUD Consortium:**

Participant No	Participant organisation name	Short name	Country
1 (Coordinator)	Institute of Communication and Computer Systems	ICCS	Greece
2	Numascale AS	NSCALE	Norway
3	Kaleao Limited	KALEAO	UK
4	OnApp Limited	ONAPP	Gibraltar
5	University of Manchester	UNIMAN	UK
6	MonetDB Solutions B.V.	MDBS	Netherlands
7	Neo Technology	NEO	Sweden
8	UMEA University	UMU	Sweden



NUMASCALE

KALEAO

onapp



**Confidentiality:**

This document contains proprietary and confidential material of certain ACTiCLOUD contractors, and may not be reproduced, copied, or disclosed without appropriate permission. The commercial use of any information contained in this document may require a license from the proprietor of that information.

THIS DOCUMENT IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES

(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS DOCUMENT, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

**Table of Contents**

<b>1</b>	<b>Introduction.....</b>	<b>6</b>
1.1	Relevance to ACTiCLOUD’s objectives .....	7
1.2	Relevance to ACTiCLOUD’s business scenarios.....	7
1.3	Relevance to ACTiCLOUD’s use cases .....	8
<b>2</b>	<b>MALCOM: Incremental Memory Footprint Estimation .....</b>	<b>10</b>
2.1	Background.....	10
2.2	Query Execution Profiling in MonetDB .....	10
2.3	Memory Footprint Estimation Model .....	14
2.4	Evaluation and Future Work.....	17
<b>3</b>	<b>MonetDBLite-Java .....</b>	<b>22</b>
3.1	Background.....	22
3.2	MonetDBLite-Java Overview .....	23
3.3	Evaluation .....	24
<b>4</b>	<b>Distributed Query Processing.....</b>	<b>27</b>
4.1	Overview.....	27
4.2	Basic Use Cases .....	28
4.3	Evaluation and Future Work.....	31
<b>5</b>	<b>Conclusion.....</b>	<b>34</b>
	<b>APPENDIX I MonetDBLite-Java Connection APIs .....</b>	<b>35</b>
I.1	Embedded API .....	35
I.2	JDBC API.....	37

## Figures

Figure 1: ACTiCLOUD architecture: position of WP3 marked by the red box. ....	6
Figure 2: MonetDB implementation architecture. ....	11
Figure 3: Estimation error rate of TPC-H queries on SF10 data set. The x-axis shows the number of queries executed. The y-axis shows the error rate in percentage. ....	19
Figure 4: Estimation error rates of all TPC-H queries after 200 iterations (SF10). The x-axis shows the query numbers of the TPC-H queries. The y-axis shows the error rates in percentage. ....	19
Figure 5: Estimation error rates of Air Traffic queries using the complete data set. The x-axis shows the number of queries executed. The y-axis shows the error rate in percentage. ....	21
Figure 6: Performance results for TPC-H scale factor 1 (y-axis uses a logarithmic scale). ....	24
Figure 7: Performance results for TPC-H scale factor 10 (y-axis uses a logarithmic scale). ....	25
Figure 8: Performance results for TPC-H scale factor 20 (y-axis uses a logarithmic scale). ....	25
Figure 9: Querying distributed data in MonetDB: Tables <code>s1</code> , <code>s2</code> , <code>s3</code> , <code>s4</code> are exact copies of each other (shown in the same colour). This fact can be indicated using <code>CREATE REPLICA TABLE</code> , so that when the table <code>repTbl</code> is queried, the query will be run using the nearest replica. Tables <code>t1</code> , <code>t2</code> , <code>t3</code> , <code>t4</code> contain data partitions of a bigger table (shown in different colours). One can merge a number of partitions using <code>CREATE MERGE TABLE</code> , so that when the table <code>mrgTbl</code> is queried, the query will be run on all partition tables included in <code>mrgTbl</code> . ....	27
Figure 10: Air Traffic queries on KMAX: y-axis shows execution times in msec. ....	33
Figure 11: From 1 node to 3 nodes: y-axis shows the scale-out, lower than 1 is better. ....	33
Figure 12: From 3 months to 12 months: y-axis shows the scale-up, lower than 4 is better. ....	33

## Tables

Table 1: Relations between MonetDB extensions and ACTiCLOUD Strategic Objectives. ....	7
Table 2: Relations between MonetDB extensions and ACTiCLOUD Business Scenarios. ....	8
Table 3: Relations between MonetDB extensions and ACTiCLOUD Use Cases. ....	8
Table 4: JSON profiling objects produced for an <code>algebra.projection</code> operation. ....	13

# 1 Introduction

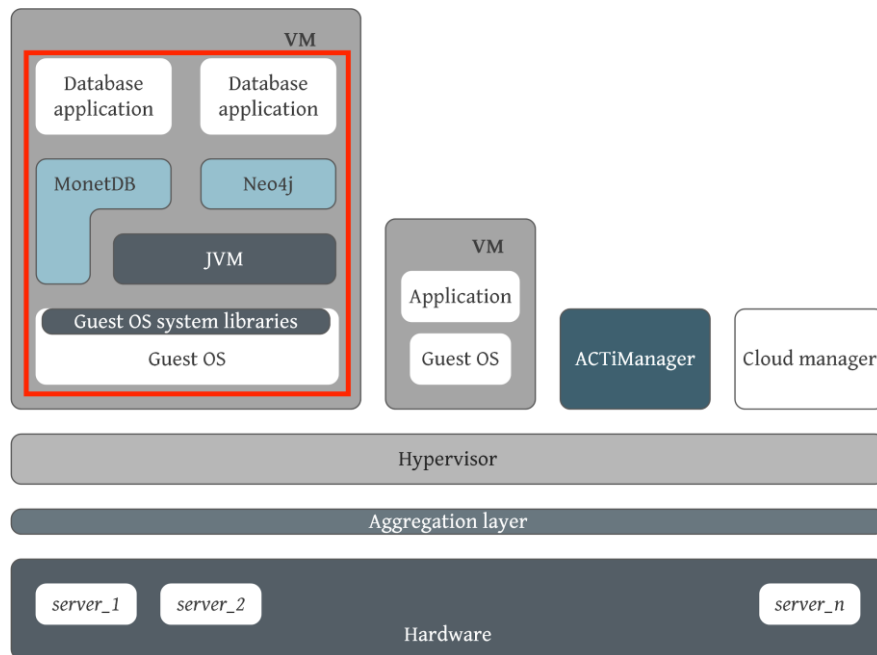


Figure 1: ACTiCLOUD architecture: position of WP3 marked by the red box.

The main goal of ACTiCLOUD is to develop a novel cloud architecture that will break the existing scale-up and share-nothing barriers and enable the holistic management of physical resources both at the local cloud site and the distributed levels, targeting drastically improved utilisation and scalability of resources.

In the ACTiCLOUD architecture (see Figure 1), **WP3 “Evolution of Databases”** is positioned in the top layer. Its main tasks are about extending/improving system libraries, JVM and different types of databases to make full use of the resource capabilities of ACTiCLOUD-enabled systems. In **Task 3.3 “In-memory databases”** we focus on extending the columnar database MonetDB towards an ACTiCLOUD-enabled data center.

In this report, we describe the initial implementation of MonetDB extensions to exploit the functionality enabled by the ACTiCLOUD architecture:

- *MALCOM*: an external tool for incremental memory footprint estimation. It can be used to first estimate the memory footprint of a query using historical statistics, then execute the query, and finally update the memory footprint statistics with information from the actual execution.
- *MonetDBLite-Java*: an embedded version of MonetDB for Java, which provides native mapping of data types between the database and Java, and allows Java application developers to run MonetDB servers inside a JVM process and manage database servers directly in their Java code.
- *Distributed query processing*: a master-worker architecture which transparently executes an SQL query on multiple MonetDB database servers. In ACTiCLOUD, we have enhanced this feature with increased stability of subquery execution on remote MonetDB servers, and improved performance of distributed query execution.

## 1.1 Relevance to ACTiCLOUD's objectives

Table 1: Relations between MonetDB extensions and ACTiCLOUD Strategic Objectives.

ACTiCLOUD Strategic Objectives	ACTiCLOUD Strategic sub-Objectives	MALCOM	MonetDBLite-Java	Distributed query processing
SO1: Effective utilisation of cloud resources.	SO1.1: Resource efficiency	x	x	x
	SO1.2: Performance stability	x		x
SO2: Deployment of resource demanding applications in the cloud	SO2.1: Scalability in resource provisioning	x	x	x
	SO2.2: Elasticity in resource provisioning	x	x	x

Table 1 gives an overview of the ACTiCLOUD Strategic Objectives to which each MonetDB extension relates. In more detail:

- *MALCOM*: contributes to achieving all strategic objectives (SO1.1, SO1.2, SO2.1 and SO2.2), because with the estimation given by this tool, we will be able to adapt the underlying virtual machine (VM) with optimal memory size for a given query. In this way, we can avoid over-/under-provisioning of memory resources, while minimising performance variation.
- *MonetDBLite-Java*: contributes to efficient and flexible resource consumption and provisioning, because an embedded MonetDB server is much more lightweight than a full-fledged MonetDB server and it can easily scale-up/-down with the JVM.
- *Distributed query processing*: contributes to all strategic objectives (SO1.1, SO1.2, SO2.1 and SO2.2) since it allows us to flexibly distribute workloads to satisfy application requirements (e.g. equal distribution of workloads among all instances in a cluster to avoid bottleneck), or adapt to available resources (e.g. distribute less data/workloads to smaller instances in the cluster, but more to larger instances).

## 1.2 Relevance to ACTiCLOUD's business scenarios

Table 2 gives an overview of to which ACTiCLOUD business scenarios each MonetDB extension is related. In more details:

- *MALCOM*: this extension relates to BS1 - 4, because the provided estimation can help determining the optimal resource configuration (BS1, BS3, BS4), determining if and where there are sufficient resources to compute a given query (BS3, BS4), and ensuring stable performance (BS2).
- *MonetDBLite-Java*: this extension mainly relates to cost reduction (BS1), because the embedded database server consumes less resources, does not require dedicated machine and maintenance, and can be more efficient for in-memory processing than a full-fledged

MonetDB server due to the minimal distance between the application and database server.

- *Distributed query processing*: this extension mainly relates to BS2 - 4, because it enables flexible distribution of data and workloads over multiple instances, including instances on remote sites.

Table 2: Relations between MonetDB extensions and ACTiCLOUD Business Scenarios.

ACTiCLOUD Business Scenarios	MALCOM	MonetDBLite-Java	Distributed query processing
BS1. Effective consolidation for increased revenue and reduced TCO	X	X	
BS2. Workload prioritisation	x		x
BS3. Hosting larger workloads	x		x
BS4. Collaboration with sibling cloud sites	x		x
BS5. Enhanced dependability and availability			

### 1.3 Relevance to ACTiCLOUD's use cases

Table 3: Relations between MonetDB extensions and ACTiCLOUD Use Cases.

ACTiCLOUD Use Cases	MALCOM	MonetDBLite-Java	Distributed query processing
UC1. Execution of typical cloud applications			
UC2. Database applications with constant workload and intermittent uptime	x	x	x
UC3. Database applications with constant workload and continuous uptime	x	x	x
UC4. Database applications with predictable workload burst	x	x	x
UC5. Database applications with unpredictable workload burst	x	x	x
UC6. Analysis of social networks with high dynamism			
UC7. Enterprise Operations			



Table 3 gives an overview of to which ACTiCLOUD business scenarios each MonetDB extension is related. All three extensions relate to the database application use cases (UC2 - UC5), because they are all components that can be used in various ways to achieve the level of availability, reliability and performance stability required by different database applications.

Further in this report, we present each of the aforementioned MonetDB extensions in a separate section (sections 2, 3 and 4) with a description of the extension, some preliminary experiment results and our future plan.

## 2 MALCOM: Incremental Memory Footprint Estimation

### 2.1 Background

Estimating the memory footprint of a query is an important feature for a database system. With the information of how much memory an operation will take (e.g. execution of a relational operator), an optimizer can decide whether there is enough memory available for this operation, and how many (different) operations can be run in parallel. In ACTiCLOUD, this information can help us achieve optimal resource usage and reduce performance jitter. Given a query we want to know the following information, without executing it: does one of the existing MonetDB cloud instances (i.e. running in ACTiCLOUD VMs) have sufficient capacity to execute this query? If multiple instances satisfy the requirements, which one is the best match? If a new MonetDB cloud instance needs to be created/launched, how much memory should we allocate?

The approach taken here differs from traditional Cost-Based Optimizers (CBO), which generally sample the state of actual executions of primitive operations, e.g. Oracle CBO<sup>1</sup> and HIVE CBO<sup>2</sup>, because after each query execution we have precise knowledge of the resources consumed. Our memory footprint estimation harvests this information to predict future operations of similar nature. The rationale stems from the common knowledge that many database application environments have a limited number of “business transactions” or “Business Intelligence (BI) templates” where only some parameters are changed with each call. This knowledge has been used in the past to, for instance, drive the development of DBA wizards<sup>3</sup> for index selection and self-tuning optimizers<sup>4</sup> to avoid expensive join paths. In addition, databases are generally long running applications, which allow MALCOM to much information from previous executions to incrementally improve its predictions.

### 2.2 Query Execution Profiling in MonetDB

Figure 2 illustrates the implementation architecture of MonetDB to execute an SQL query. MAL refers to the MonetDB Assembly Language, an internal language into which SQL queries are compiled and executed<sup>5</sup>. The SQL Parser and MAL Optimizer deploy well-known rewriting rules (e.g. parallelisation, and dead code/common expression/constant elimination) to reduce the intermediate sizes and processing time. They do not rely on any cost-model or pre-computed statistics. The middle layer (marked by the dashed box) is a sequence of specialised optimizers that morph the logical plan received from the SQL compiler into a physical execution plan expressed in MAL statements. The bottom layer (under the dashed box) contains the implementation of the relational operators (in the MAL statements). Each operator takes as input the resident intermediates produced by operators executed earlier or the persistent data on disk.

---

<sup>1</sup> [https://docs.oracle.com/cd/B10500\\_01/server.920/a96533/optimops.htm](https://docs.oracle.com/cd/B10500_01/server.920/a96533/optimops.htm)

<sup>2</sup> <https://hortonworks.com/blog/hive-0-14-cost-based-optimizer-cbo-technical-overview/>

<sup>3</sup> <https://www.microsoft.com/en-us/research/project/autoadmin/>

<sup>4</sup> IBM DatArcs Optimizer: <https://www.ibm.com/us-en/marketplace/datarcs-optimizer>

<sup>5</sup> For more information: [www.monetdb.org/Documentation/MonetDB/Introduction](http://www.monetdb.org/Documentation/MonetDB/Introduction).

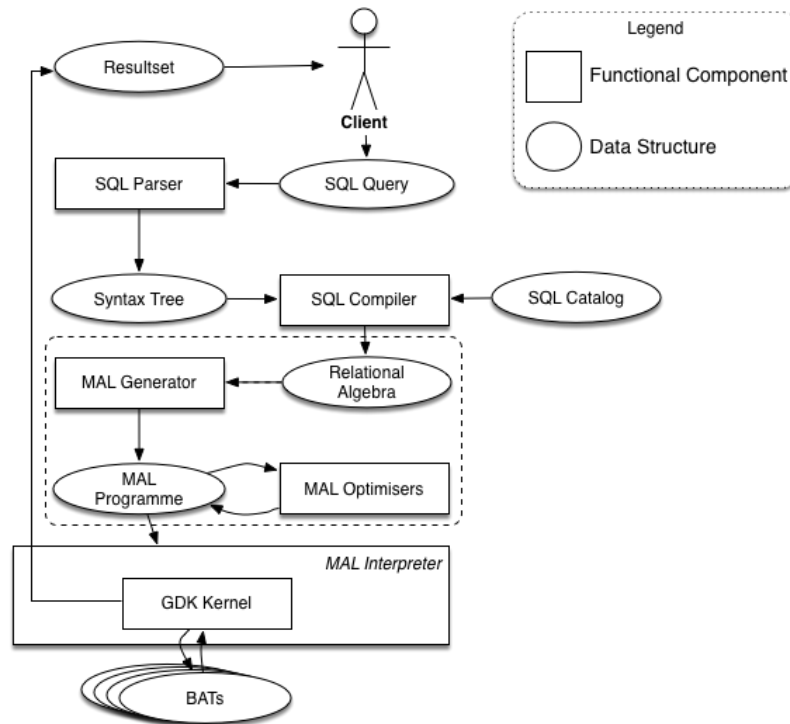


Figure 2: MonetDB implementation architecture.

To illustrate consider the following simple SQL query:

```
SELECT COUNT(*) FROM _tables;
```

That query is eventually translated into the following physical execution plan expressed in MAL statements, which are then executed by the MonetDB kernel:

```

X_1=0@0:void := querylog.define("select count(*) from _tables;":str, ...);
X_4=0:int := sql.mvc();
C_5=<tmp_1524>[92]:bat[:oid] := sql.tid(X_4=0:int, "sys":str, "_tables":str);
(X_13=<sql_empty_oid_bat>[0]:bat[:oid], X_14=<sql_empty_int_bat>[0]:bat[:int]) :=
  sql.bind(X_4=0:int, "sys":str, "_tables":str, "id":str, 2:int);
X_8=<tmp_147>[99]:bat[:int] :=
  sql.bind(X_4=0:int, "sys":str, "_tables":str, "id":str, 0:int);
X_11=<sql_empty_int_bat>[0]:bat[:int] :=
  sql.bind(X_4=0:int, "sys":str, "_tables":str, "id":str, 1:int);
X_16=<tmp_147>[99]:bat[:int] := sql.delta(X_8=<tmp_147>[99]:bat[:int],
  X_13=<sql_empty_oid_bat>[0]:bat[:oid],
  X_14=<sql_empty_int_bat>[0]:bat[:int],
  X_11=<sql_empty_int_bat>[0]:bat[:int]);
X_17=<tmp_1477>[92]:bat[:int] := algebra.projection(C_5=<tmp_1524>[92]:bat[:oid],
  X_16=<tmp_147>[99]:bat[:int]);
X_18=92:lng := aggr.count(X_17=<tmp_1477>[92]:bat[:int]);
barrier X_72=false:bit := language.dataflow();
sql.resultSet("sys.L3":str, "L3":str, "bigint":str, 64:int, 0:int, 7:int, X_18=92:lng);
  
```

The MAL language is purely designed as intermediate language to express the operations. Almost every MAL statement is an assignment in the general format:

```
VAR=<FILENAME>[COUNT]:VAR_TYPE := MOD.FUNC(PARAM1=<FILENAME>[COUNT]:PARAM1_TYPE, ...);
```

Every function belongs to a module. The arguments are either typed scalar values (`:type`) or a reference to a column (`:bat[:type]`). If a variable (`VAR`) refers to a column, which can be memory mapped, it is also tagged with its base `FILENAME` on disk and the number of values in this column (`COUNT`).

The above MAL programme is straightforward. It first reads (`sql.bind`), gathers (`sql.delta`) and projects (`algebra.projection`) the data of one column ("`_tables`".`"id"`) from the disk. Then the data is passed to `aggr.count` to compute the `COUNT`. Finally, `sql.resultSet` emits the query result. MAL statements are important data for MALCOM: before query execution, it gives memory footprint estimation per MAL statement; after query execution, it collects execution statistics per MAL statement.

### Profiling Information

The MonetDB kernel can be instructed to emit<sup>6</sup> profiling events for the execution of MAL statements, e.g. by establishing a connection using MonetDB's profiling tool *stethoscope*<sup>7</sup>. Every MAL function comes with two event records: one taken at the beginning and another upon completion of the operation. The event record contains details on the arguments passed, their type and size. Wherever possible the arguments are linked with the underlying persistent column. Intermediate columns are nameless and we can only rely on their types and cardinalities. Upon completion, we also know the exact size of the result and the elapsed time.

Stethoscope can render the profiling events in two different formats: either as a plain text tuple (one tuple per event) or as JSON objects (one object per event). Table 4 shows the two profiling events produced for the `algebra.projection` operation when executing the MAL programme above. The left column shows the event at the "`start`" and the right column the event when the operation is "`done`". Differences between the two objects are marked in red. Of most interest to our estimation are the properties shown for the arguments ("`arg`") and return variables ("`ret`"). For instance, in a "`ret`" object, the field "`size`" is a good estimation<sup>8</sup> of how much memory the result set of this function consumes; while in an "`arg`" object, the field "`eol`":`1` indicates this argument has reached its end-of-life. This information together with the "`size`" allows us to estimate how much memory is freed after this operation.

---

<sup>6</sup> The profiling events are always constructed, but discarded if there is no connection requesting them. When there is a connection for those events, they will be written to the socket. The cost of the additional emission action is negligible.

<sup>7</sup> <https://www.monetdb.org/Documentation/Manuals/MonetDB/Profiler/Stethoscope>

<sup>8</sup> Yes, it is still an estimation, because most data in MonetDB is memory mapped, so we are never 100% sure which memory mapped pages are in memory. But with auxiliary information, this estimation can be made quite accurate.

Table 4: JSON profiling objects produced for an `algebra.projection` operation.

JSON object at “start” time	JSON object at “done” time
<pre> {"source":"trace",  "clk":1767476753919,  "ctime":1528314717302449,  "thread":8,  "function":"user.s4_2",  "pc":9,  "tag":248,  "module":"algebra",  "instruction":"projection",  "session":"b4a92225-127d-4a1f-b2ef-...",  "state":"start",  "usec":0,  "rss":87,  "size":0,  "nvcs":1,  "stmt":"X_17...:= algebra.projection(...);",  "short":"X_17[0]:= projection(..., ...)",  "prereq":[4,8],  "ret":[{"   "index":"0",   "name":"X 17",   "alias":"sys_tables.id",   "type":"bat[:int]",    "bid":"0",   "count":"0",   "size":0,   "eol":0 }],  "arg":[{"   "index":"1",   "name":"C 5",   "type":"bat[:oid]",   "kind":"transient",   "bid":"863",   "count":"92",   "size":736,   "eol":1 },  {"index":"2",   "name":"X 16",   "alias":"sys_tables.id",   "type":"bat[:int]",   "kind":"persistent",   "bid":"103",   "count":"99",   "size":396,   "eol":1 }] } </pre>	<pre> {"source":"trace",  "clk":1767476754150,  "ctime":1528314717302680,  "thread":8,  "function":"user.s4_2",  "pc":9,  "tag":248,  "module":"algebra",  "instruction":"projection",  "session":"b4a92225-127d-4a1f-b2ef-...",  "state":"done",  "usec":230,  "rss":87,  "size":0,   "stmt":"X_17...:= algebra.projection(...);",  "short":"X_17[0]:= projection(..., ...)",  "prereq":[4,8],  "ret":[{"   "index":"0",   "name":"X 17",   "alias":"sys_tables.id",   "type":"bat[:int]",   "kind":"transient",   "bid":"837",   "count":"92",   "size":368,   "eol":0 }],  "arg":[{"   "index":"1",   "name":"C 5",   "type":"bat[:oid]",   "kind":"transient",   "bid":"863",   "count":"92",   "size":736,   "eol":1 },  {"index":"2",   "name":"X 16",   "alias":"sys_tables.id",   "type":"bat[:int]",   "kind":"persistent",   "bid":"103",   "count":"99",   "size":396,   "eol":1 }] } </pre>

## 2.3 Memory Footprint Estimation Model

The goal of MALCOM is, given a MAL physical execution plan of an SQL query (i.e. a list of MAL statements to be executed in that order), estimate the maximum amount of memory executing this plan will consume, while *only using* information from our *memory footprint estimation model*, which is built using actual execution information of previous queries.

The algorithm used for estimating an upper bound of the memory needed to execute a given MAL plan, is shown in pseudocode below. When a MAL plan is received, MALCOM first annotates each MAL statement with an estimation of how much memory it will consume and release (i.e. the `i.mem_fprint` and `i.free_size` below). Then the algorithm iterates over the MAL plan (i.e. `mal_statements`). At each iteration, it updates `max_mem` with the memory footprint of current MAL statement (`i.mem_fprint`), then it refreshes the current memory consumption (`curr_mem`) by also taking into account the memory that will be freed by this statement (`i.free_size`).

```
max_mem = 0
curr_mem = 0
for i in mal_statements:
    max_mem = max(max_mem, curr_mem + i.mem_fprint)
    curr_mem = curr_mem + i.mem_fprint - i.free_size
```

After the execution of the given MAL plan, we update our memory footprint estimation model with the real execution information. We initialise the estimation model with straightforward heuristics, e.g. the result of an aggregation is always of value “1”. The initialisation can be extended with basic column statistics (min, max, count, etc.) that can be gathered using MonetDB’s `ANALYZE` command<sup>9</sup>.

The estimation model is built by dividing MAL instructions with similar functionality (most of them represent a relational operator each) into several groups and abstracting away their specific signatures. Currently, the model includes 8 groups. We briefly analyse each of them below. Note that we only consider bulk operators here (i.e. taking columns as operands), which are the default ones in MonetDB.

### 2.3.1 Arithmetic Operators

These operators always return the same number of values as their operands (MonetDB requires both operands to have equal size). However, the data type of the output can be a larger-sized data type than both operands to capture possible overflow. Hence, their output size is computed as:

```
arith_ret.size = sizeof(arg1) * sizeof(ret.data_type)
```

### 2.3.2 Aggregate Operators

The number of values returned by these operators equals to the number of groups in which its input data column is divided (by earlier `GROUP BY` statements, or 1 if there is no `GROUP BY`). The most general signature of these operators takes two operands: `arg1` is a column containing the actual values to work on; `arg2` is a column containing the group IDs, one for each value in `arg1`. So, the output size of an aggregate operator is computed by multiplying the number of

<sup>9</sup> <https://www.monetdb.org/Documentation/Cookbooks/SQLrecipes/statistics>

unique values in `arg2` with the size of the return data type (due to automatic type promotion in MonetDB for possible overflow, the size of the return data type can be bigger than that of input data type):

```
aggr_ret.size = COUNT(DISTINCT arg2) * sizeof(ret.data_type)
```

### 2.3.3 Limit Operators

This group includes `firstn` and `sample`<sup>10</sup>. They return at most  $N$  values from its input column `arg` as specified by the limit. Hence, their output size is computed as:

```
limit_ret.size = MIN(COUNT(arg), N) * sizeof(arg.data_type)
```

### 2.3.4 Grouping Operators

MonetDB currently has 24 grouping operators for different situations. For instance, the position of the input data column (`arg1`) in a `GROUP BY` SQL clause determines the use of a `GROUP` operator or a `SUBGROUP` operator in MAL. More variations of `GROUP` or `SUBGROUP` operators are used depending on the availability of auxiliary information (e.g. some statistics of the input column). However, all grouping operators generally return three columns of results: (i) a `groups` column containing the group IDs, one for each value in `arg1`; (ii) an `extents` column containing the `OID` (MonetDB internal type for Object IDentifiers, denoting positions of data values in a column) of a representative of each group; and (iii) a `histo` column containing the number of values in each group corresponding the values in `extents`. The data type of `groups` and `extents` are both `OID`, and the data type of `histo` is `LNG` (MonetDB internal type for LoNG integers). The number of values in `extents` and `histo` is the same, and is estimated using a simple kNN algorithm based on statistics of previous queries or basic statistics of the involved columns. Putting everything together, the total output size of a grouping operator is estimated as:

```
group_ret.size = COUNT(arg1) * sizeof(OID) +
                estimate_nr_groups(arg1) * (sizeof(OID) + sizeof(LNG))
```

### 2.3.5 Set Operators

The current estimations for the set operators use heuristics to compute an upper bound of the result size.

`UNION ALL` simply returns a concatenation of its two input columns `arg1` and `arg2`, so its output size can be precisely computed with:

```
unionall_ret.size = (COUNT(arg1) + COUNT(arg2)) * sizeof(arg1.data_type)
```

`UNION` returns a concatenation of its two input columns `arg1` and `arg2` with duplicate elimination. We use the following formula to compute an upper bound of its result size (since it does not exclude unique values in both `arg1` and `arg2`):

```
union_ret.size = (COUNT(DISTINCT arg1) + COUNT(DISTINCT arg2)) * sizeof(arg1.data_type)
```

<sup>10</sup> <https://www.monetdb.org/Documentation/Cookbooks/SQLrecipes/Sampling>

**INTERSECT** returns values that exist in both of its input columns `arg1` and `arg2` with duplicates eliminated. We use the following formula to compute an upper bound of its result size (since it does not exclude unique values that are only in `arg1` or only in `arg2`):

```
union_ret.size = MIN(COUNT(DISTINCT arg1) + COUNT(DISTINCT arg2)) *
                 sizeof(arg1.data_type)
```

**EXCEPT ALL** returns all values that are in its first input column `arg1` but not in its second input column `arg2`. **EXCEPT** does the same thing as **EXCEPT ALL**, but also eliminates duplicates. We use the following formula to compute an upper bound of their result size (the result again is an upper bound, since the computation does not exclude (unique) values that also exist in `arg2`):

```
excpt_all.size = COUNT(arg1) * sizeof(arg1.data_type)
excpt.size     = COUNT(DISTINCT arg1) * sizeof(arg1.data_type)
```

### 2.3.6 Projection Operators

Projection operators extract a small part of a column. The arguments are a candidate list `cand` containing the OIDs of the to-be-projected values and a reference to the (persistent) column `col`. The number of elements in the output equals to the number of elements of the candidate list. Hence, their exact output size is computed as:

```
proj_ret.size = COUNT(cand) * sizeof(col.data_type)
```

### 2.3.7 Selection Operators

This operator group includes the filter operations `theta-select` and `select`. For these operators, we know that the output is always smaller than or equals to the candidate tuples considered. To estimate the result size, traditional cost-based models assume a uniform distribution of the data and calculate the fraction of the domain, i.e. the selectivity factor. In practice, however, the uniform distribution of the data assumption does not always hold, so these models have limited accuracy. In our model, we keep the results of a series of actual filter operations, so as to use them to find a “historical nearest-neighbor” for any filter operation in a MAL plan whose cost we need to estimate.

MonetDB has numerous select operators to support filtering on different data types, range or point select, with a lower-/upper-bound or both, etc. However, the signatures of all select operators can be abstracted into a single one with three operands `sel(col, range, op)`, where `col` is a reference to the (persistent) column, `range` is the selection range (low, high), and `op` is the comparison operator (<, >, <=, >=, etc). In our model, we keep a dictionary of all the previous selections in the format of this signature, but with an extra value `cnt` to denote the number of values selected.

The estimation for a selection operator `sel(col, range op)` works as follows. First, we find in the dictionary records of all previous selections on the same `col` with the same `op`. Then, we use a *k* nearest neighbour (kNN) procedure to find the 5 nearest records based on the selection range. Next, for each of the 5 records, we extrapolate the number of selected values based on the selectivity and input column size. Finally, we compute the estimated memory footprint of this selection as the average of the 5 extrapolations, multiplied by the data size of the input column. This estimation procedure is shown in the pseudo code below:



```

extrap = 0
for dict in kNN(dictionary, sel, 5)
    extrap += dict.cnt * (COUNT(sel.col) / COUNT(dict)) * (sel.range / dict.range)
sel_ret.size = extrap/5 * sizeof(sel.col.data_type)

```

### 2.3.8 Join Operators

For a cross product of two columns (`col1`, `col2`) we know it will return `COUNT(col1) * COUNT(col2)` number of values. The `crossproduct` operator of MonetDB takes two data columns as its inputs, and returns two columns where each column contains OIDs referring to data values in an input column. The two OID columns together denote how the values from the input columns are aligned in the cross product result. So the exact output size of a cross product is computed as:

```

cp_ret.size = COUNT(col1) * COUNT(col2) * sizeof(OID) * 2

```

The estimation model for the other join operators is similar to that of selection operators. Again, the signatures of all join operators can be abstracted into a single one with three operands `join(col1, col2, op)`, where `col1` and `col2` are the input column, and `op` the join operator (eq, left, outer, etc). We also keep a dictionary of all the previous joins in the format of this signature, annotated with an extra value `cnt` to denote the number of values returned by that particular join operation.

To estimate the result size for a join operator `join(col1, col2, op)`, we first find in the dictionary records of all previous selections on the same columns with the same `op`. Then, we run a `kNN` to find the 5 nearest records based on the sizes of the input columns. Finally, we extrapolate the result count based on the input column sizes, and compute the result size (like cross product, two OID columns are returned). The pseudo code is shown below:

```

extrap = 0
for dict in kNN(dictionary, join, 5)
    extrap += dict.cnt * (COUNT(join.col1) / COUNT(dict.col1)) *
                      (COUNT(join.col2) / COUNT(dict.col2))
sel_ret.size = extrap/5 * sizeof(OID) * 2

```

## 2.4 Evaluation and Future Work

In this section we present some preliminary results of evaluating MALCOM. We are mainly interested in how fast the error rate of our estimation drops with increasing number of executed queries. MALCOM's performance is of no concern yet, since the amount of work for both the database server (i.e. emitting the profiling information) and MALCOM is small.

We have evaluated the precision of our estimator using both the TPC-H benchmark<sup>11</sup> and the Air Traffic benchmark<sup>12</sup>. TPC-H is the industrial standard benchmark. The results we acquired with this benchmark have particularly helped us to understand the estimation results and improve our model. However, the data of TPC-H is too perfectly distributed to really evaluate some definitions in our model, e.g. the extrapolations for selections and joins. Therefore, we have also evaluated our model using the Air Traffic benchmark. This benchmark is based on real-world data containing US national flight arrival/departure information as reported by the airlines at

<sup>11</sup> <http://www.tpc.org/tpch/>

<sup>12</sup> <https://github.com/MonetDBSolutions/airtraffic-benchmark>

the US department of transport. We have downloaded the data from January 1988 to March 2014 (60GB in total). The benchmark queries are based on some popular statistic analyses conducted on the data.

The evaluation was conducted as follows. First, for each benchmark query, we generate a number of “training queries” by assigning random values to the selection ranges/points in the original query. For each TPC-H query we generated 200 training queries. For Air Traffic, the convergence takes longer, so we generated 1000 training queries for each benchmark query. Then, we initialise our model with some basic column statistics, e.g. min, max and count. Next, we execute the training queries one-by-one. After each execution, we add its profiling information into our model and compute an estimation for the corresponding original query. Finally, we compute the error rate of our estimation by comparing it against the actual memory footprint of the original query.

Figure 3 shows the results of several TPC-H queries with typical shapes of how the error rates change with more queries. Several observations can be made:

- Because the TPC-H data is equally distributed, the error rates are generally low, and they converge toward zero fairly quickly (after ~25-50 queries). This is exactly what is expected from our estimation model.
- Although query 06 seems to need many more queries to converge, its error rate is negligible from the beginning.
- The negative error rates in Figure 2.2 (a) and (b) indicate underestimation.

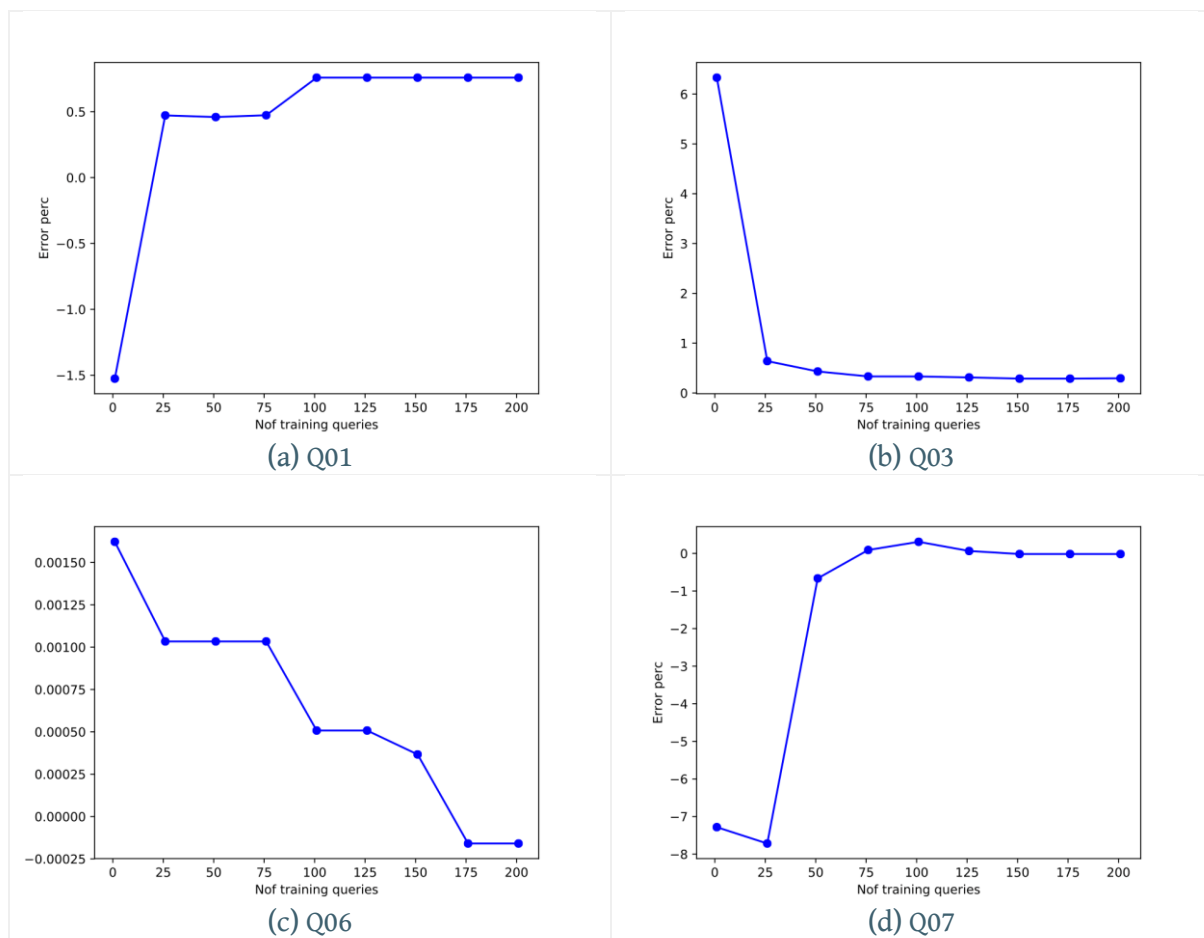


Figure 3: Estimation error rate of TPC-H queries on SF10 data set. The x-axis shows the number of queries executed. The y-axis shows the error rate in percentage.

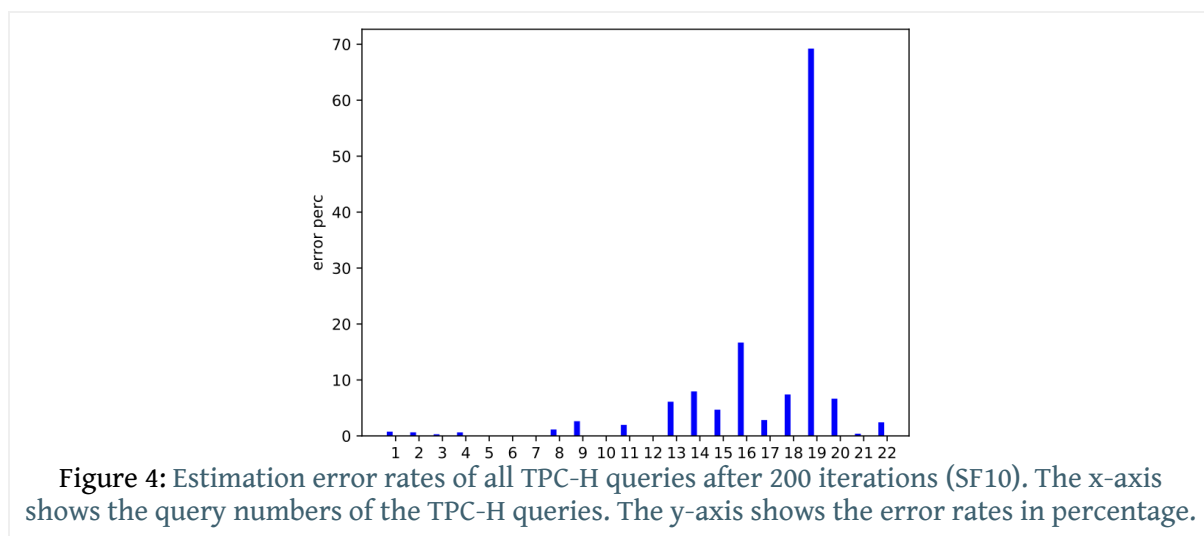


Figure 4: Estimation error rates of all TPC-H queries after 200 iterations (SF10). The x-axis shows the query numbers of the TPC-H queries. The y-axis shows the error rates in percentage.

Figure 4 shows the estimation error rates of all TPC-H queries after having used all 200 variations correspond to each original query. Our estimation model works fairly good for most queries, however:

- Q19 always has a remarkably high error rate during the 200 iterations, because the MAL plan of Q19 contains a lot of merge instructions which merge two intermediate columns into one. Because we currently do not have sufficient information about the input columns, we estimate the output size as the sum of the two inputs. The cascading effect of this overestimation results into a high error rate in the end.
- Q16 also has a high error rate, most of which is caused by the estimation for the join (between `partsupp` and `part`) in this query.

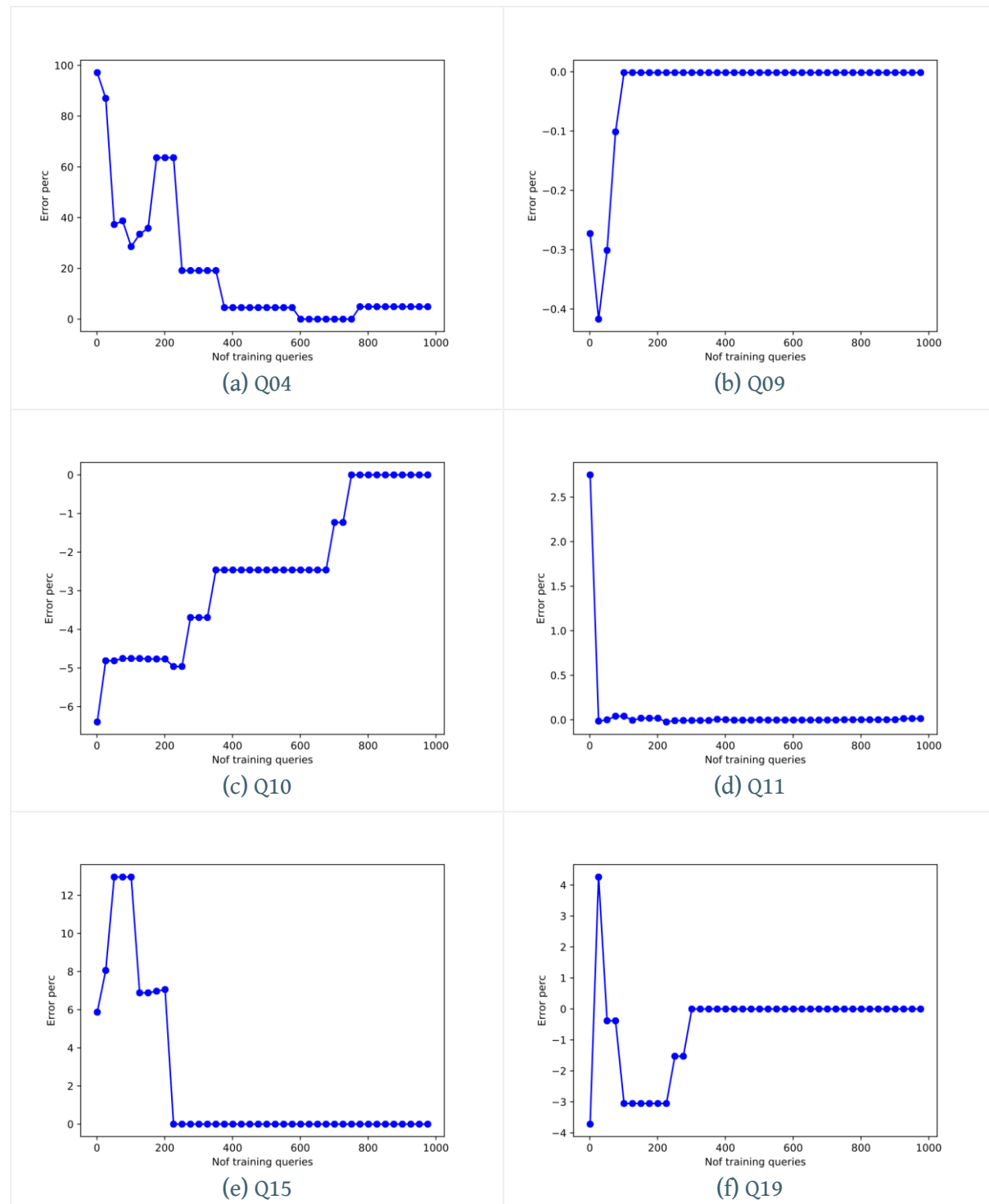
We have been able to evaluate the estimation model on several Air Traffic queries. The main goal is to check that the model developed so far is also applicable for real-world data and queries. The results are shown in Figure 5. Several observations can be made:

- In general, the estimation model needs more queries to converge toward zero. However, the fact that the error rates do always converge indicates the usability of the model for real data.
- The error rates look more realistic (unlike with many TPC-H queries, whose error rates are negligible from the start). We observe that the estimation often starts with a noticeable error rates, but always manages to converge in the end.
- The graphs of Q04, Q15 and Q19 contain some amplitudes, which did not happen with the TPC-H queries. We think this is due to the unequally distributed data and the randomised selection ranges, such that the estimator is sometimes misled by the information it has collected. However, the estimation always manages to recover, as it is supposed to do.

### Next steps

Estimating the memory footprint of a query execution is an important, useful, but challenging task. The current model forms a solid basis for many future work that will be needed to fully exploit the possibilities provided by the ACTiCLOUD architecture:

- Improve the quality of the estimation of some operations, like join and merge.
- Reduce the number of queries needed to converge toward zero.
- Improve the way how new profiling information is merged with existing information (currently we simply append).
- Support different policies: min/avg/max.
- Extend to estimating execution times.
- Extend to a cloud orchestrator, communicate with the ACTiCLOUD manager.



**Figure 5:** Estimation error rates of Air Traffic queries using the complete data set. The x-axis shows the number of queries executed. The y-axis shows the error rate in percentage.

### 3 MonetDBLite-Java

Following the footsteps of MonetDBLite for R<sup>13</sup> and Python<sup>14</sup>, we have developed MonetDBLite-Java in the context of ACTiCLOUD, which deploys MonetDBLite (an embedded version of MonetDB) in a JVM with JDBC support<sup>15</sup>. In the “lite” versions of MonetDB, both client and server run within the same process, saving the cost of eventual inter-process communication, such as through a socket connection.

#### 3.1 Background

With the increasing usage of embedded systems, new challenges arose to supported compacted platforms on these systems. *Embedded databases* are database management systems (DBMSs) that have been developed with embedded systems as a targeted platform. Compared to traditional client-server DBMSs, embedded databases have a number of potential advantages<sup>16</sup>: (i) short latency, (ii) fast data access, (iii) smaller footprint, and (iv) low database maintenance. Because of these potential advantages, embedded databases are often used in applications that are lightweight but must have a quick reaction time, such as real-time applications, IoT applications and mobile applications.

The Wikipedia article on “Embedded database”<sup>17</sup> has a fairly complete list of all embedded databases. Among these systems, the most popular open-source offerings<sup>18</sup> are SQLite<sup>19</sup>, H2<sup>20</sup>, HSQLDB<sup>21</sup>, Apache Derby<sup>22</sup>, LevelDB<sup>23</sup> and RocksDB<sup>24</sup>. SQLite is by far the most widely used embedded DBMS, and its architecture is the closest to MonetDBLite-Java in the sense that it is a relational DBMS written in C and supports among others a JDBC interface for Java applications. H2, HSQL and Apache Derby are the three well-known pure Java embedded DBMSs. They differ mainly in detailed language/interface features and performance for different use cases. LevelDB is a key-value store developed by Google as a lightweight implementation of their Bigtable storage systems. RocksDB is a key-value store developed by FaceBook. It is based on LevelDB, but with several improvements to better fit the needs of FaceBook, e.g. increased write rates and decreased read/write amplifications.

The introduction of MonetDBLite family has broadened the landscape of embedded DBMSs with a full-fledged database engine that is strong in data analytics. Moreover, next to supporting the JDBC standard, MonetDBLite-Java also comes with an embedded communication API, which is especially designed to provide fast data access per column, so as to improve the total performance for analytical workload. To the best of our knowledge, MonetDBLite-Java is the only embedded DBMS that provides both row-oriented *and* column-oriented data access.

---

<sup>13</sup> <https://github.com/hannesmuehleisen/MonetDBLite-R>

<sup>14</sup> <https://github.com/hannesmuehleisen/MonetDBLite-Python>

<sup>15</sup> <https://github.com/hannesmuehleisen/MonetDBLite-Java>

<sup>16</sup> Depending on the actual implementation of a particular system, some of these advantages apply.

<sup>17</sup> [https://en.wikipedia.org/wiki/Embedded\\_database](https://en.wikipedia.org/wiki/Embedded_database)

<sup>18</sup> I.e. within Top 100 of DB-Engines Ranking (<https://db-engines.com/en/ranking>).

<sup>19</sup> <https://www.sqlite.org>

<sup>20</sup> <http://www.h2database.com/>

<sup>21</sup> <http://hsqldb.org>

<sup>22</sup> <http://db.apache.org/derby/>

<sup>23</sup> <https://github.com/google/leveldb>

<sup>24</sup> <https://rocksdb.org>

## 3.2 MonetDBLite-Java Overview

### 3.2.1 Connection APIs

We provide two APIs for MonetDBLite-Java: an *Embedded API* (non-standard) and the *standard JDBC API*. The MonetDB specific embedded API has been introduced for certain scenarios where performance is critical, such as when dealing with large result sets. But, better performance comes at the cost of less portability. The Javadoc of this API can be found in our website<sup>25</sup>.

A MonetDBLite JDBC connection is similar to a regular connection with a MonetDB server. The differences between the two JDBC connections are:

- Only one database is allowed per JVM process. However, multiple connections to the same database is allowed within the same process.
- MonetDBLite is a compact version of MonetDB with some auxiliary features removed in order to reduce the size of the library. The geometry module, merge tables, remote tables, JSON module, and the Data Vaults extension, have all been removed from the "lite" version.
- The authentication scheme is absent from MonetDBLite.
- MonetDBLite-Java native code uses a separate heap from the JVM, which means that connections and result sets descriptions must be explicitly closed to avoid memory leaks.
- No two concurrent MonetDBLite-Java processes can use the same database simultaneously.

More details of both APIs can be found in APPENDIX I.

### 3.2.2 Distribution

The project is hosted and maintained on Github<sup>26</sup>.

Two JAR files are distributed: `monetdb-java-lite` (~5.6Mb) and `monetdb-jdbc-new` (~150Kb). The former depends on the latter and contains MonetDBLite library adapted for the JVM. MonetDBLite-Java is compatible with JVM 8 onwards only. This JAR also provides native libraries for 64-bit Linux, MacOS X and Windows. The latter is a fork of MonetDB's JDBC driver and is used for JDBC connections.

Both JARs are hosted on Maven Central repository<sup>27</sup>. To use it:

- Using Apache Maven:  

```
<dependency>
  <groupId>monetdb</groupId>
  <artifactId>monetdb-java-lite</artifactId>
  <version>2.37</version>
</dependency>
```
- Using Gradle: compile 'monetdb:monetdb-java-lite:2.37'
- Otherwise it is possible to download the JARs from our website<sup>28</sup> and add them to the CLASSPATH.

<sup>25</sup> <https://www.monetdb.org/downloads/Java-Experimental/javadocs/embedded/>

<sup>26</sup> <https://github.com/hannesmuehleisen/MonetDBLite-Java>

<sup>27</sup> <https://search.maven.org/#search%7Cga%7C1%7Cmonetdb>

### 3.3 Evaluation

The experiments were performed on a machine with the following specifications:

- OS: Fedora 26 with 4.14.11-200 Linux kernel
- CPU Intel Core i7-2600K 64-bit with 8 cores and 3.4 GHz maximum clock speed
- RAM 16GB
- Disk 5.5 TB HDD
- JVM version 1.8.0\_161

Versions of the databases benchmarked:

- MonetDBLite-Java 2.36
- SQLite 3.23.1
- H2 1.4.197

For the tests, we use the Java Microbenchmark Harness (JMH)<sup>29</sup> framework. For each query, three measurements were performed. The first two measurements were performed to warm up the JVM, then the third measurement is recorded. The data sets used in the benchmarks are the TPC-H data with Scale Factors SF1, SF10 and SF20 (i.e. 1, 10 and 20 GB data). The TPC-H benchmark is comprised by 22 queries aiming to pressure different components of the database system.

In total, 4 database setups were tested:

1. MonetDBLite-Java with the standard JDBC API,
2. MonetDBLite-Java with the embedded API,
3. SQLite<sup>30</sup> and
4. H2<sup>31</sup> (only on SF1 and SF10 data sets).

MonetDBLite-Java (using both its APIs) and SQLite run in the native heap inside the process, thus communicating with the JVM via JNI while H2 runs completely inside the JVM.

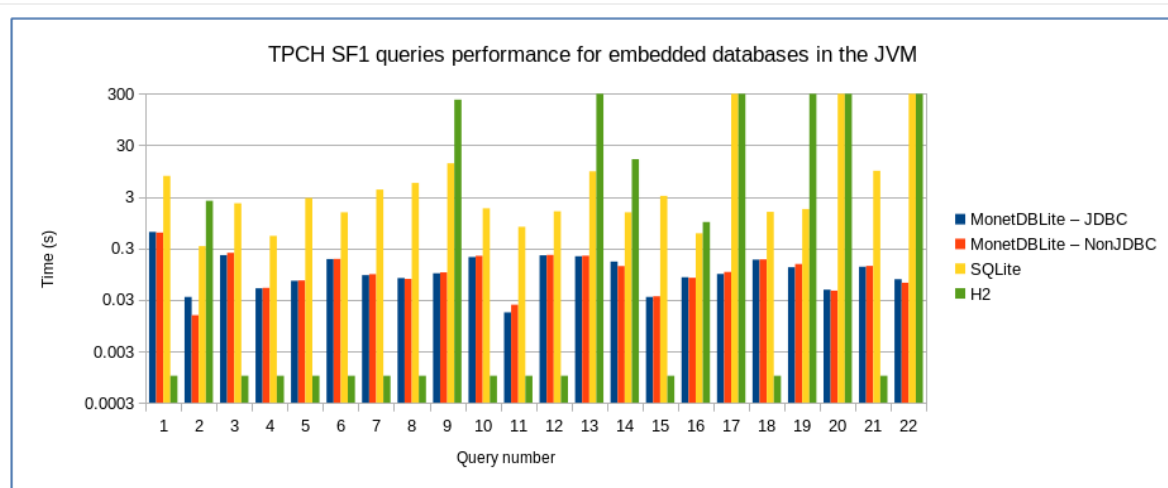


Figure 6: Performance results for TPC-H scale factor 1 (y-axis uses a logarithmic scale).

<sup>28</sup> <https://www.monetdb.org/downloads/Java-Experimental/>

<sup>29</sup> <http://openjdk.java.net/projects/code-tools/jmh/>

<sup>30</sup> <https://www.sqlite.org/index.html>

<sup>31</sup> <https://www.h2database.com/html/main.html>



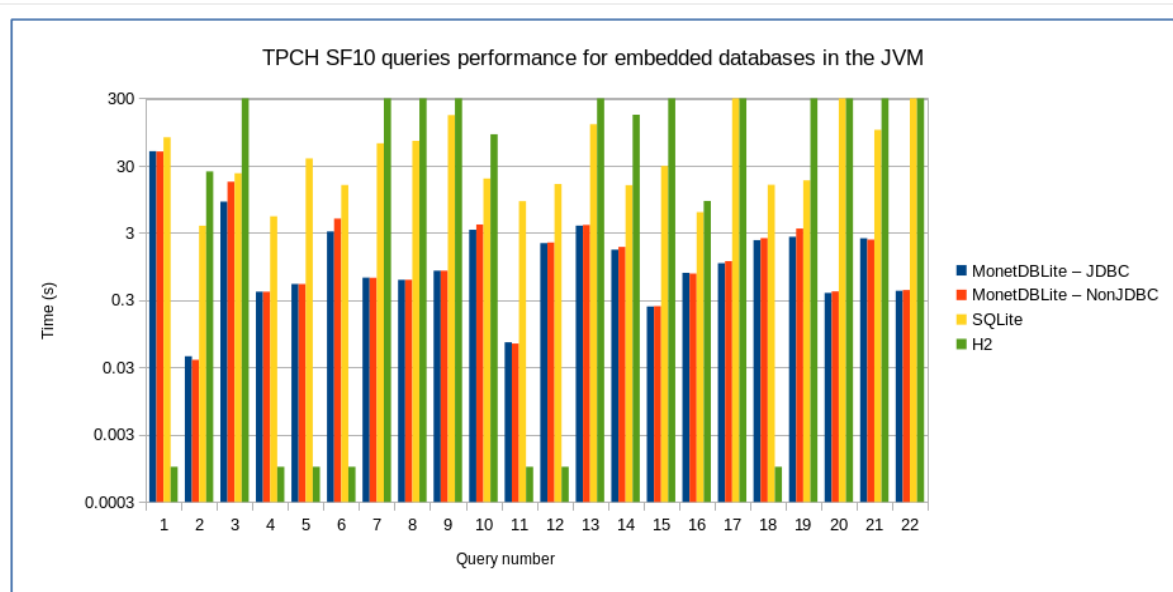


Figure 7: Performance results for TPC-H scale factor 10 (y-axis uses a logarithmic scale).

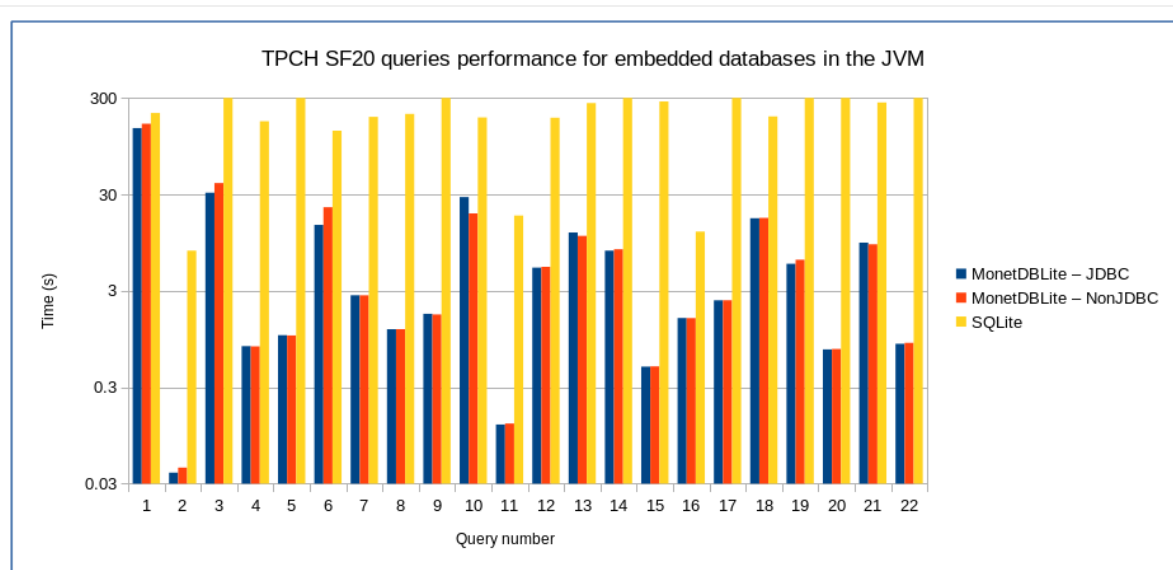


Figure 8: Performance results for TPC-H scale factor 20 (y-axis uses a logarithmic scale).

Figure 6, Figure 7 and Figure 8 show the execution times of the database setups on TPC-H scale factors 1, 10 and 20, respectively. Each query run had a timeout of 5 minutes.

MonetDBLite-Java managed to finish all queries in all scale factors. SQLite did not finish queries 17, 20 and 22 in any of the scale factors, and queries 3, 5, 9, 14 and 19 in SF20. H2 did not finish queries 13, 17, 19, 20 and 22 for SF1, plus queries 3, 7, 8, 9, 15 in SF10. Because of its bad performance already in SF10 tests, H2 was not benchmarked in SF20.

MonetDBLite-Java outperformed SQLite in every single test. One of the most important reasons is the more efficient data storage provided by columnar store.

H2 obtained better results overall compared to MonetDBLite-Java for the queries it finished in scale factors 1 and 10. This is probably due to some query (results) caching in H2, because in the warming up runs, MonetDBLite-Java obtained better results instead. The repeated query performance issue is also confirmed with the client-server version of MonetDB, which obtains better TPC-H results in latter runs of the same query consecutively<sup>32</sup>. This performance degradation in consecutive runs of the same query is being investigated by the MonetDBLite development team.

The overall performance of both MonetDBLite-Java versions was within the same magnitude. The Embedded API connects directly to JNI, thus avoiding JDBC overhead. However, only the query execution time was measured in the tests. Due to the nature of the testing setup, the overall performance difference between these two setups was negligible.

### Next steps

- Resolve the cache issue on consecutively repeated queries on MonetDBLite-Java.
- Test MonetDBLite-Java in different workloads. We have chosen TPC-H for the preliminary evaluation here, because it is an industrial standard. This benchmark is designed for heavy analytics, with read only queries. MonetDBLite-Java was not yet been tested on read/write workloads such as TPC-C.
- Integration with UNIMAN's JVM and evaluation using their JVM's improved garbage collector.
- More extensive scalability evaluation on bigger VMs, especially the VMs developed in ACTiCLOUD, and on the ACTiCLOUD platform.

---

<sup>32</sup> An issue related to the query cache of MonetDBLite-Java can be found in the official GitHub repository: <https://github.com/hannesmuehleisen/MonetDBLite-Java/issues/3>

## 4 Distributed Query Processing

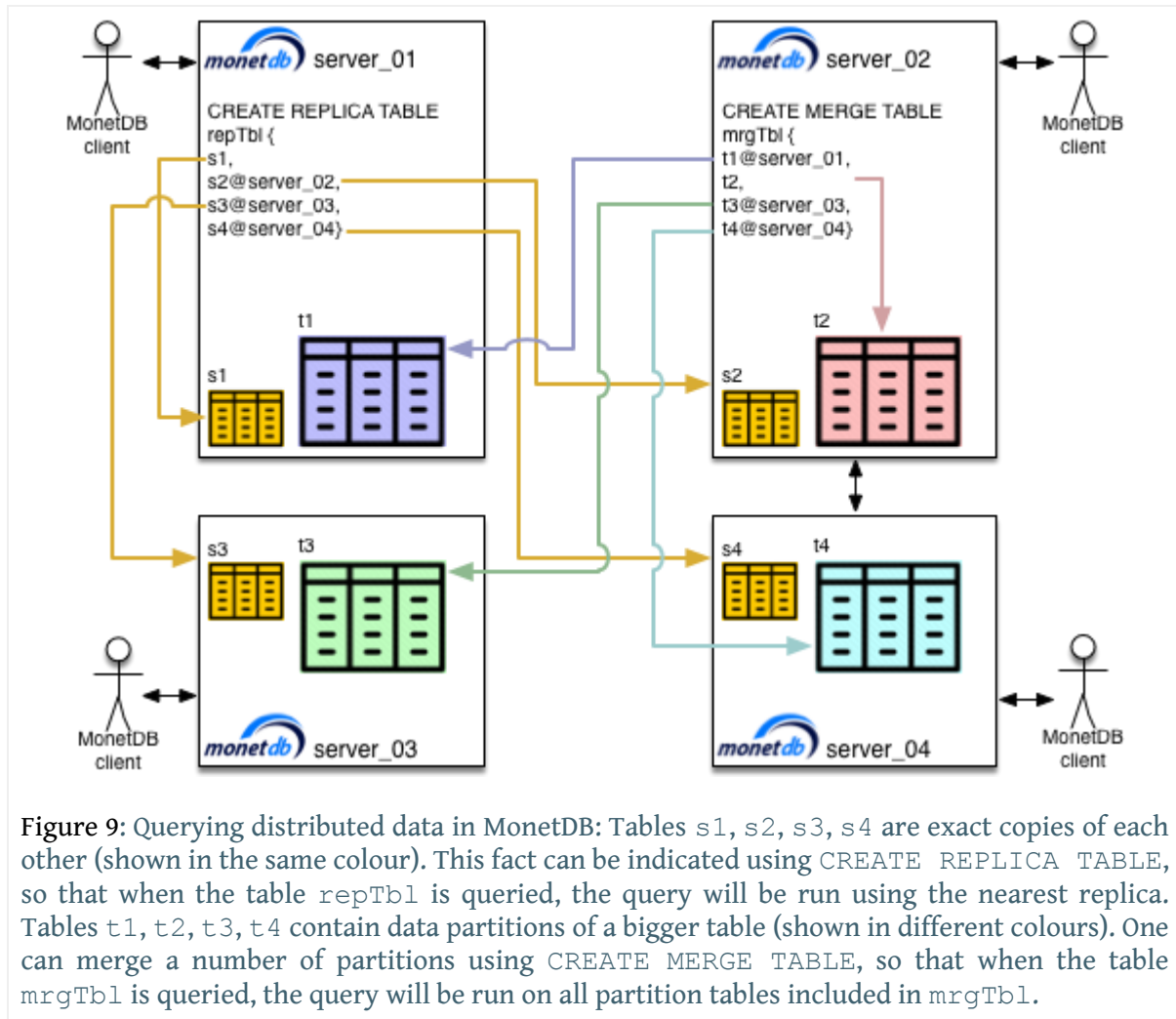


Figure 9: Querying distributed data in MonetDB: Tables `s1`, `s2`, `s3`, `s4` are exact copies of each other (shown in the same colour). This fact can be indicated using `CREATE REPLICATION TABLE`, so that when the table `repTbl` is queried, the query will be run using the nearest replica. Tables `t1`, `t2`, `t3`, `t4` contain data partitions of a bigger table (shown in different colours). One can merge a number of partitions using `CREATE MERGE TABLE`, so that when the table `mrgTbl` is queried, the query will be run on all partition tables included in `mrgTbl`.

### 4.1 Overview

Figure 9 depicts how one can query data that is spread (replicated or sharded) over multiple MonetDB instances in a local cluster. Below we briefly explain how this architecture works, (including concepts defined and SQL features added):

- Every instance in the cluster is a full-fledged MonetDB server, so they have equal functionality. The system can be set up in such a way that a MonetDB client can establish a connection to any one of those instances and run all supported queries.
- Distributed queries are queries which implicitly involve data on remote MonetDB instances. These are handled using a simple master-worker architecture: the server on which the query is started is the *master*; the servers containing remote data addressed by this query are *workers*.
- The master is responsible for parsing the query, generating distributed query plan, sending the subqueries to the workers and merging subquery results into final results. Every server can act as a master and a worker.

- At the SQL language level, three new types of SQL TABLEs have been added as the basic components for distributed queries:
  - *Remote table*: one can create a REMOTE TABLE on the server to refer to a table that physically exists on a remote server by giving the URL of the server and the name of the table on that server Y (see *server\_01* and *server\_02* in Figure 4.1).
  - *Replica table*: one can create a REPLICATION TABLE and add both local tables and remote tables to it. The keyword REPLICATION indicates that all members are an exact replica of each other and the system can use whichever member is most convenient..
  - *Merge table*: one can create a MERGE TABLE and add both local and remote tables to it to form a single big table. The result is a UNION ALL of all partition tables in this MERGE TABLE and the system will potentially have to access all members to answer a query.

With these three extensions, users now have different ways to query data on a remote server: directly on the remote server as a normal table, or through the local server as a REMOTE or REPLICATION/MERGE table.

## 4.2 Basic Use Cases

Below we use a small example to show how the above concepts work. We have started three MonetDB servers, *mdb1*, *mdb2* and *mdb3* running on ports 50001, 50002 and 50003, respectively. Here we use *mdb3* as the master, while *mdb1* and *mdb2* are the workers.

First we connect to *mdb1* and *mdb2* to create two simple tables on each of them. The tables *s1* and *s2* will be used in a replica table, so we insert the same data into them. The tables *t1* and *t2* will be used in a merge table, so we insert different values into them:

```
$ mclient -d mdb1 -u monetdb -p 50001
password:
...
sql>CREATE TABLE s1 (i INT);
operation successful
sql>INSERT INTO s1 VALUES (23), (42);
2 affected rows
sql>SELECT * FROM s1;
+-----+
| i      |
+=====+
| 23     |
| 42     |
+-----+
2 tuples
sql>CREATE TABLE t1 (s VARCHAR(10));
operation successful
sql>INSERT INTO t1 VALUES ('abc'), ('efg');
2 affected rows
sql>SELECT * FROM t1;
+-----+
| s      |
+=====+
| abc    |
| efg    |
+-----+
2 tuples
```

```
$ mclient -d mdb2 -u monetdb -p 50002
password:
...
sql>CREATE TABLE s2 (i INT);
operation successful
sql>INSERT INTO s2 VALUES(23), (42);
2 affected rows
sql>SELECT * FROM s2;
+-----+
| i      |
+=====+
| 23     |
| 42     |
+-----+
2 tuples
sql>CREATE TABLE t2 (s varchar(10));
operation successful
sql>INSERT INTO t2 VALUES ('foo'), ('bar');
2 affected rows
sql>SELECT * FROM t2;
+-----+
| s      |
+=====+
| foo    |
| bar    |
+-----+
2 tuples
```

Then we connect to *mdb3* to create some local data, adding definitions for remote data and run some distributed queries. Here we create another table *s3* to be used in a replica table, so that the replica table will contain both local and remote tables:

```
$ mclient -d mdb3 -u monetdb -p 50003
password:
...
sql>CREATE TABLE s3 (i INT);
operation successful
sql>INSERT INTO s3 VALUES (23), (42);
2 affected rows
```

**Step 1:** we create remote tables to refer to the tables on *mdb1* and *mdb2*:

```
sql>CREATE REMOTE TABLE s1 (i int) on 'mapi:monetdb://localhost:50001/mdb1';
operation successful
sql>CREATE REMOTE TABLE t1 (s varchar(10)) on 'mapi:monetdb://localhost:50001/mdb1';
operation successful
sql>CREATE REMOTE TABLE s2 (i int) on 'mapi:monetdb://localhost:50002/mdb2';
operation successful
sql>CREATE REMOTE TABLE t2 (s VARCHAR(10)) ON 'mapi:monetdb://localhost:50002/mdb2';
operation successful
```

```
sql>SELECT * FROM s1;
```

```
+-----+
| i      |
+=====+
```

```
| 23 |
| 42 |
+-----+
```

```
2 tuples
```

```
sql>SELECT * FROM t1;
```

```
+-----+
| s      |
+=====+
```

```
| abc |
| efg |
+-----+
```

```
2 tuples
```

```
sql>SELECT * FROM s2;
```

```
+-----+
| i      |
+=====+
```

```
| 23 |
| 42 |
+-----+
```

```
2 tuples
```

```
sql>SELECT * FROM t2;
```

```
+-----+
| s      |
+=====+
```

```
| foo |
| bar |
+-----+
```

```
2 tuples
```

**Step 2:** create a replica table *repS* and add *s1*, *s2* and *s3* into it. Selecting from *repS* returns results as if it was a single table. The logical query PLAN shows that only one table is actually queried. However, it simply picks the first replica without looking further for a closer one. This plan will be optimised in the future.

```
sql>CREATE REPLICA TABLE repS (i INT);
operation successful
sql>ALTER TABLE repS ADD TABLE s2;
operation successful
sql>ALTER TABLE repS ADD TABLE s1;
operation successful
sql>ALTER TABLE repS ADD TABLE s3;
operation successful
sql>SELECT * FROM repS;
```

```
+-----+
| i      |
+=====+
```

```
| 23 |
| 42 |
+-----+
```

```
2 tuples
```

```
sql>PLAN SELECT * FROM repS;
```

```
+-----+
| rel |
```

```

+=====+
| project (                               |
| | table                               | |
| | | REMOTE(sys.s2)                     |
| | | [ "s2"."i" as "reps"."i", "s2"."%TID%" NOT NULL as "reps"."%TID%" ] |
| | | REMOTE mapi:monetdb://localhost:50002/mdb2 |
| | | [ "reps"."i", "reps"."%TID%" NOT NULL ] |
| ) [ "reps"."i" ]                       |
+-----+
4 tuples

```

Step 3: create a merge table `mrGT` and add `t1` and `t2` into it. Selecting from `repS` causes the query to be executed on partition tables. The logical query plan shows that the `WHERE` condition is properly pushed down to the remote servers.

```

sql>CREATE MERGE TABLE mrGT (s VARCHAR(10));
operation successful
sql>ALTER TABLE mrGT ADD TABLE t2;
operation successful
sql>ALTER TABLE mrGT ADD TABLE t1;
operation successful
sql>SELECT * FROM mrGT WHERE s <> 'bla';
+-----+
| s      |
+=====+
| foo    |
| bar    |
| abc    |
| efg    |
+-----+
4 tuples
sql>PLAN SELECT * FROM mrGT WHERE s <> 'bla';
+-----+
| rel    |
+=====+
| union (
| | union (
| | | table
| | | | project (
| | | | | select (
| | | | | | REMOTE(sys.t2) [ "t2"."s" as "mrGT"."s" ] COUNT
| | | | | ) [ "mrGT"."s" != varchar(10) "bla" ]
| | | | ) [ "mrGT"."s" ] REMOTE mapi:monetdb://localhost:50002/mdb2 [ "mrGT"."s" ],
| | | table
| | | | project (
| | | | | select (
| | | | | | REMOTE(sys.t1) [ "t1"."s" as "mrGT"."s" ] COUNT
| | | | | ) [ "mrGT"."s" != varchar(10) "bla" ]
| | | | ) [ "mrGT"."s" ] REMOTE mapi:monetdb://localhost:50001/mdb1 [ "mrGT"."s" ]
| | ) [ "mrGT"."s" ],
| | project (
| | | select (
| | | | table(sys.t3) [ "t3"."s" as "mrGT"."s" ] COUNT
| | | ) [ "mrGT"."s" != varchar(10) "bla" ]
| | ) [ "mrGT"."s" ]
| ) [ "mrGT"."s" ]
+-----+
21 tuples

```

Step 4: execute a join between the replica table and remote table. The logical query plan shows that the replica tables `s1` and `s2`, which are local to `t1` and `t2`, respectively, are properly exploited to reduce the communication cost:

```

sql>SELECT * FROM repS, mrGT;
+-----+-----+

```

```

| i      | s      |
+=====+=====+
| 23     | foo    |
| 23     | bar    |
| 42     | foo    |
| 42     | bar    |
| 23     | abc    |
| 23     | efg    |
| 42     | abc    |
| 42     | efg    |
+-----+-----+
8 tuples
sql>PLAN SELECT * FROM repS, mrgT;
+-----+-----+
| rel                                     |
+-----+-----+
| union (                               |
| | table                               | | | | |
| | | project (                         |
| | | | crossproduct (                 |
| | | | | REMOTE(sys.s2)               |
| | | | | [ "s2"."i" as "reps"."i", "s2"."%TID%" NOT NULL as "reps"."%TID%" ], |
| | | | | project (                   |
| | | | | | REMOTE(sys.t2) [ "t2"."s" as "mrgt"."s" ] COUNT                 |
| | | | | ) [ "mrgt"."s" ]              |
| | | | ) [ ]                           |
| | | ) [ "reps"."i", "mrgt"."s" ]       |
| | | REMOTE mapi:monetdb://localhost:50002/mdb2 [ "reps"."i", "mrgt"."s" ], |
| | table                               |
| | | project (                         |
| | | | crossproduct (                 |
| | | | | REMOTE(sys.s1)               |
| | | | | [ "s1"."i" as "reps"."i", "s1"."%TID%" NOT NULL as "reps"."%TID%" ], |
| | | | | project (                   |
| | | | | | REMOTE(sys.t1) [ "t1"."s" as "mrgt"."s" ] COUNT                 |
| | | | | ) [ "mrgt"."s" ]              |
| | | | ) [ ]                           |
| | | ) [ "reps"."i", "mrgt"."s" ]       |
| | | REMOTE mapi:monetdb://localhost:50001/mdb1 [ "reps"."i", "mrgt"."s" ] |
| | ) [ "reps"."i", "mrgt"."s" ]        |
+-----+-----+
20 tuples

```

### 4.3 Evaluation and Future Work

Here we present some functional experimental results we have conducted on the KMAX machine that has been set up for ACTiCLOUD in Manchester. MonetDB has only recently been ported to the ARM64 architecture (with a single node). On KMAX, it is the first time that we can run truly distributed queries on multiple nodes. So the main purposes of these experiments are: (i) conducting a functional verification for both MonetDB and KMAX, and (ii) establishing performance baselines.

For the experiments, we have only used the Air Traffic benchmark, because unlike TPC-H, it is originally designed to evaluate distributed query processing. The Air Traffic data is naturally divided per month. For the experiments, we have used two data sets: one containing 3 months data (~1.5million records), the other 12 months (~5.8 million records). We ran two sets of experiments: on a single KMAX node (all data stored in a single MonetDB server), or on three KMAX nodes (one MonetDB server on each node, hence, one master and two workers; data is more or less equally distributed over the two workers, no actual data on the master).

Each KMAX node has 8 cores and 4GB RAM and runs Ubuntu 16.04. We used MonetDB version Jul2017-SP4. In each run, the queries are executed in a random order. The execution times shown

in Figure 10 are averages of in total 100+ runs, with the maximal value for each query removed. The experiment results give us some useful information:

- Figure 10 shows query execution times in milliseconds, “3mo” and “12mo” indicate the data sets, “1x” and “3x” indicate the number of nodes. We can make the following main observations:
  - Most query execution times are (far) under 200 milliseconds<sup>33</sup>, which is very good for this type of analytical queries.
  - The long running times of q02, q08 and q14 are expected, because they purposefully contain some computations that are difficult to parallelise.
- Figure 11 shows the scale-out from 1 node to 3 nodes (i.e. execution time of 3 nodes divided by that of 1 node). Ideally, the lines of “3M” and “12M” should be under the threshold at 1. The scale-out tests we have measured are all above this threshold for several reasons:
  - The data sets are fairly small, even the 12M data set still fits comfortably in the memory, so distributing them will not give much performance advantages. We expect better scale-out with larger data sets.
  - The shorter running queries scale out relatively worse than long running queries, because they carry higher overhead.
  - Although our experiments have resulted in adding a number of optimisations into MonetDB’s distributed query execution plans (e.g. distribute `ORDER BY`), more (advanced) optimisations are needed. Q02 scales particularly bad, because it contains `QUANTILE` functions that require data of a complete column, hence the computation is not distributed. On a single MonetDB instance, if a `QUANTILE` is applied on a persistent column, it is optimised by parallelising the sorting step and temporarily storing the result for subsequent queries. Implementing similar optimisations like this in a distributed setting can be useful but will require considerable efforts.
- Figure 12 shows the scale-up of data size from 3 months to 12 months (i.e. execution time of 12 months divided by that of 3 months). Since MonetDB is originally designed for scale-up applications, it is not a surprise that both lines are mostly under the threshold (lower is better). However, it is good to have this confirmation on KMAX, since ARM64 is a completely new architecture for MonetDB.

---

<sup>33</sup> A threshold above which latencies become clearly noticeable.



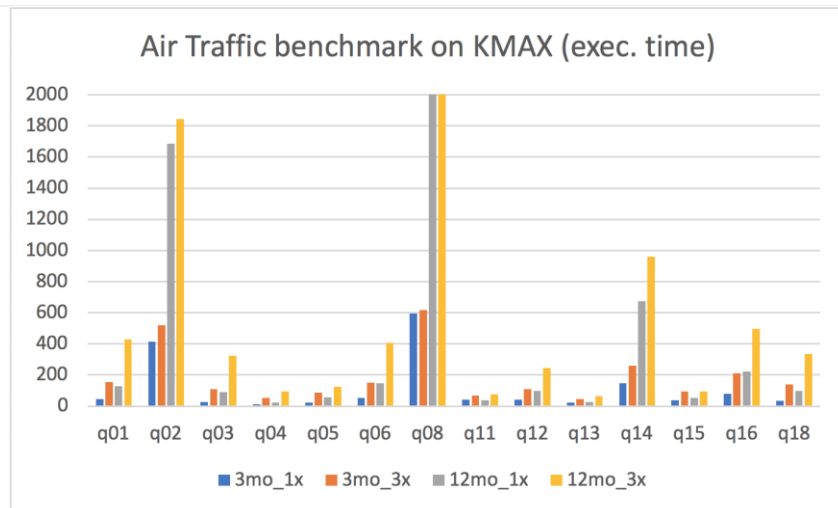


Figure 10: Air Traffic queries on KMAX: y-axis shows execution times in msec.

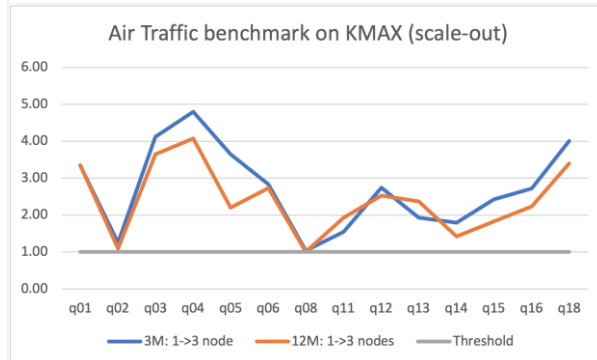


Figure 11: From 1 node to 3 nodes: y-axis shows the scale-out, lower than 1 is better.

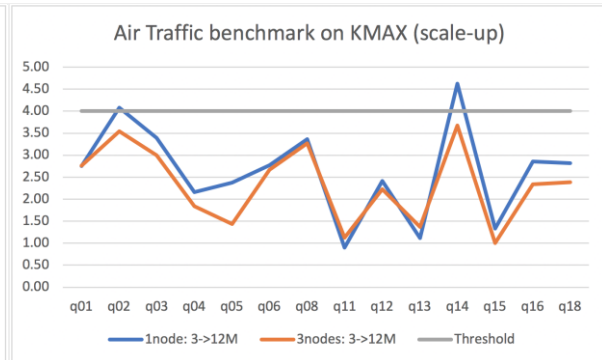


Figure 12: From 3 months to 12 months: y-axis shows the scale-up, lower than 4 is better.

### Next steps:

Supporting distributed queries efficiently is a challenging task, and there is still a long way to go for this new feature in MonetDB:

- We will continue hardening the implementation towards a beta release.
- We will apply further optimisations of distributed query plan.
- We will look into more flexible way of partitioning data based on predicates. Currently, data partitioning is done manually and fixed.
- We will add support for distributed updates.
- We will combine distributed query processing with replication services for higher reliability and availability.

## 5 Conclusions

In this report, we have presented three extensions developed in the context of ACTiCLOUD:

1. MALCOM, an incremental memory footprint estimation tool for MonetDB;
2. MonetDBLite-Java, an embedded version of MonetDB for JVM; and
3. MonetDB's support for distributed query processing.

For each extension, we have described its design, implementation, initial evaluation and future work.

In the remaining of the ACTiCLOUD project, we will continue working on these extensions according to our plan. With these extensions we will be able to advance MonetDB towards ACTiCLOUD-enabled data centres, i.e. improve MonetDB's stability, reliability and scalability to support a broader scope of applications while exploiting the functionality provided by the ACTiCLOUD platform (e.g. more efficient and flexible resource provisioning).

## APPENDIX I MonetDBLite-Java Connection APIs

### I.1 Embedded API

#### I.1.1 Start a database

A database must be started before any connection can be made. When starting the database, one can specify a path to the database directory. If no such path is supplied, an in-memory connection will be automatically established instead. In an in-memory connection, data is not persisted on disk. Instead, transactions are held in-memory; thus, more performance is obtained.

```
Path directoryPath = Files.createTempDirectory("testdb");
MonetDBEmbeddedDatabase.startDatabase(directoryPath.toString());
MonetDBEmbeddedConnection connection = MonetDBEmbeddedDatabase.createConnection();
connection.executeUpdate("CREATE TABLE example (words text)");
//...
connection.close();
MonetDBEmbeddedDatabase.stopDatabase();
```

#### I.1.2 Transactions

After a connection has been established, the user can execute queries against the embedded database and retrieve the results. The connection starts on auto-commit mode by default. The following functions are available to control transactions:

- Transaction management:
  - void startTransaction()
  - void commit()
  - void rollback()
- Handle savepoints in transactions:
  - Savepoint setSavepoint()
  - Savepoint setSavepoint(String name)
  - void releaseSavepoint(Savepoint savepoint)
  - void rollback(Savepoint savepoint)

#### I.1.3 Update queries

For update queries (e.g. INSERT, UPDATE and DELETE), the method `int executeUpdate(String query)` is used to send update queries to the server and get the number of rows affected.

```
connection.startTransaction();
int numberOfInsertions = connection.executeUpdate("INSERT INTO example VALUES ('monetdb'), ('java'), (null)");
connection.commit();
```

#### I.1.4 Queries with result sets

For queries with result sets, one can use the method `QueryResultSet executeQuery(String query)` to send a query to the server, and retrieve the results using a `QueryResultSet` instance.

The result set metadata can be retrieved with the methods `int getNumberOfRows()`, `int getNumberOfColumns()`, `void getColumnNames(String[] input)` and `void getColumnTypes(String[] input)`.

There are several ways to retrieve the results of a query. The family of methods `T get#TYPE#ByColumnIndexAndRow(int column, int row)` and `T get#TYPE#ByColumnNameAndRow(String columnName, int row)` retrieve a single value from the result set. The column and row indexes for these methods (and the other methods in this family) start from 1, same as in JDBC.

A column of values can be retrieved using the family of methods: `void get#TYPE#ColumnByIndex(int column, T[] input, int offset, int length)` and `void get#TYPE#ColumnByName(String name, T[] input, int offset, int length)`. Note that the input array must be initialized beforehand. If there is no desire to provide the offset and length parameters, the methods `void get#Type#ColumnByIndex(int column, T[] input)` and `get#Type#ColumnByName(String columnName, T[] input)` can be used instead.

```
QueryResultSet qrs = connection.executeQuery("SELECT words FROM example");
int numberOfRows = qrs.getNumberOfRows();
int numberOfColumns = qrs.getNumberOfColumns();
String[] columnNames = new String[numberOfColumns];
qrs.getColumnNames(columnNames); //returns ['words']

String singleWord = qrs.getStringByColumnIndexAndRow(1, 1); //gets 'monetdb'
String[] wordsValues = new int[numberOfRows];
qrs.getStringColumnByIndex(1, wordsValues); //returns ['words', 'java', null]
qrs.close();
```

To check if a boolean value is NULL, one can use the method `boolean checkBooleanIsNull(int column, int row)` of the class `QueryResultSet`. For all other data types, one can use the methods `boolean Check#Type#IsNull(T value)` of the class `NullMappings`.

### I.1.5 Append data to a table

To append new data to a table, one can use the method `int appendColumns(Object[] data)` from the class `MonetDBTable`. The data should come as an array of columns, where each column has the same number of rows, and each array class corresponds to the mapping defined above. To insert null values, use the constant `T get#Type#NullConstant()` from the class `NullMappings`.

```
connection.executeUpdate("CREATE TABLE testing (col1 varchar(32), col2 int)");
MonetDBTable interactWithMe = connection.getMonetDBTable("testing");
String[] inserts1 = new String[]{"str1", "str2", "str3", "str4",
NullMappings.getObjectNullConstant<String>());
int[] inserts2 = new int[]{2, 3, NullMappings.getIntNullConstant(), -5, 0};
Object[] appends = new Object[]{inserts1, inserts2};
testing.appendColumns(appends);
```

### I.1.6 Data type mapping

The Java programming language is a strongly typed language, thus the mapping between MonetDB SQL types and Java classes/primitives must be explicit. The usage of Java primitives is favored for the most common MonetDB SQL types, hence requiring fewer object allocations.

However, for the more complex SQL types, such as Strings and Dates, the mapping is made to Java Classes, matching the JDBC specification.

SQL NULL values are mapped into the system's minimum values. In MonetDBLite-Java, this feature persists for primitive types. However for the Java Classes mapping, SQL NULL values are translated into null objects. Note that other more rare data types such as geometry, json, inet, url, uuid and hugeint are missing since the modules they are defined in, are not present in the “lite” version. These types were removed from MonetDBLite to reduce the size of the library.

Other methods provided in this API include Prepared Statements which are detailed in the GitHub documentation and the Javadocs.

## I.2 JDBC API

To start a standard JDBC embedded connection, one must provide a JDBC URL in the format: `jdbc:monetdb:embedded:[<directory>]`, where `directory` is the location of the database. To connect to an in-memory database the directory must be `:memory:` or not present.

When starting a JDBC Embedded connection, it checks if there is a database instance running in the provided directory, otherwise an exception is thrown. While closing, if it's the last connection, the database will shut down automatically.

```
//Connection con = DriverManager.getConnection("jdbc:monetdb:embedded:/home/user/testing");
//POSIX
//Connection con = DriverManager.getConnection("jdbc:monetdb:embedded:C:\\user\\testing");
//Windows
//Connection con = DriverManager.getConnection("jdbc:monetdb:embedded::memory:"); //in-
memory mode
Statement st = con.createStatement();
st.executeUpdate("CREATE TABLE jdbcTest (justAnInteger int, justAString varchar(32))");
st.executeUpdate("INSERT INTO jdbcTest VALUES (1, 'testing'), (2, 'jdbc')");
ResultSet rs = st.executeQuery("SELECT justAnInteger, justAString from test1;");
while (rs.next()) {
    int justAnInteger = rs.getInt(1);
    String justAString = rs.getString(2);
    System.out.println(justAnInteger + " " + justAString);
}
rs.close();
st.close();
con.close();
```