Armada: an evolving database system
Groffen, F.E.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
8 MonetDB MAL Approach

8.1 Introduction

In previous chapters, the Armada model and a mapping of the model to SQL statements have been described. As shown there, SQL itself is not powerful enough to allow an implementation of Armada without having to make modifications to existing systems. Instead of focusing on the SQL layer of a database system, in this chapter we explore an implementation of the Armada model on a deeper level towards the core. For this exploration we use MonetDB and its kernel language MAL. Instead of implementing Armada on a user visible level, such as in SQL tables, in this approach we make numerous Armadas on an abstract level, hidden from the user. Being on a deeper level, allows for a better grip on the actions taken in the Armada model, with possible better performance as result. As entry point we assume the use of SQL, but tables no longer are Armadas. Instead the underlying representation of the database uses the Armada model to distribute the components it uses.

Typically, each SQL query is translated into MAL statements within MonetDB for execution. The units that MonetDB works with, called BATs, are target for distribution. Each SQL table consists of multiple of such BATs. Figure 8.1 depicts the architecture that we use in this chapter. Clearly, a user interacts with the database using plain SQL. Internally, this is converted to MAL code, and the Armada model is implemented via MAL programs that reference remote sites. The actual work is performed by the site that the user connected to. It hence can be considered to be the agent of the system.

SQL catalog In the modular architecture of MonetDB, the SQL compiler references BATs via the catalog. To make the SQL compiler unaware of any Armada activities, catalog entries can be adjusted such that they point to BATs which represent Armadas.
This approach keeps the SQL compiler unmodified, and hence is a cheap solution, focussing on the MAL layer. However, this approach limits the possibilities to communicate with the user about decisions that can be made in the Armada execution. For user feedback, partial execution is necessary and some support to have a user decide on how to continue execution. The SQL standard does not have any means for achieving this, and hence no handles for this are available. Instead, we focus on full execution without decisions such that we are compatible with the (unmodified) SQL compiler.

8.2 MonetDB

The MonetDB database system has been an open source product since 2000, hosted on SourceForge.net. Even though it is open sourced, a clear research focus is the drive behind the database. This becomes apparent when considering its very non-traditional design, aimed at non-traditional workloads. Core of MonetDB is its main-memory processing of data. As this puts restrictions on when data can be processed efficiently, a full vertically fragmented storage model is in place that allows to only process certain conventional columns, instead of full rows. The quest for performance on large workloads with MonetDB has resulted in sophisticated in-memory algorithms that are CPU-tuned, with an accompanying architecture to ensure the CPU needs not to idle, wasting precious time.
The vertical fragmentation of MonetDB makes it a member of the class of column-oriented data-stores among [64, 47]. In MonetDB, every relational table is represented as a collection of *Binary Association Tables (BATs)*. For a relation $R$ of $k$ attributes, there exist $k$ BATs, each BAT storing the respective column as a collection of key-attr pairs. The system-generated key identifies the relational tuple that attribute value attr belongs to, i.e. all attribute values of a single tuple are assigned the same key. Typically, key values form a dense ascending sequence representing the position of an attribute value in the column. This enables MonetDB to use fast positional lookups in a BAT given a key (such as for tuple reconstruction) and to avoid materialising the key part of a BAT in many situations completely. BATs are stored as dense tuple sequences, to enable fast in-memory processing.

The database kernel consists of a large collection of highly tuned algorithms to evaluate basic binary relational operators. All operators work independently and produce a fully materialized result. Each operator includes a runtime optimiser to exploit storage properties of operands, like sortedness, uniqueness and data type. Heavy code expansion is used to further reduce the overhead of interpretation.

**MAL** The primary textual interface to the MonetDB kernel is a simple, assembler-like language, called MAL. The language reflects the virtual machine architecture around the kernel libraries and has been designed for speed of parsing, ease of analysis and ease of target compilation by query compilers. The language is not meant as a primary programming or scripting language, such use is even discouraged.

Furthermore, a MAL program is considered a specification of intended computation and data flow behaviour. It should be understood that its actual evaluation depends on the execution paradigm chosen in the scenario. The program blocks can both be interpreted as ordered sequences of assembler instructions or as a representation of a data-flow graph that should be resolved in a data flow driven manner. The language syntax uses a functional style definition of actions and mark those that affect the flow explicitly. Flow of control keywords identify a point to change the interpretation and denote a synchronisation point.

### 8.2.1 MonetDB/SQL

A user typically uses the SQL language to interact with databases. This language is in general easier to understand for humans than programming or scripting languages are. However, being a language geared towards humans, it is not
sufficient for execution by a machine. Hence, the SQL query is compiled into the lower level MAL language, that in turn does a step deeper downwards to machine language.

An SQL query is compiled into MAL statements. For this compilation, logical table and column properties need to be known, such that can be checked whether a query is using existing objects, and whether they are of the right type for what they are used for. This information is typically contained in the catalog of the SQL database. In MonetDB/SQL the catalog is implemented by a few system tables that in principle are ordinary SQL tables containing information about all objects present in the database. The vertical fragmented nature of MonetDB has the effect of having at most single column tables. The MonetDB/SQL implementation hides this limitation, by using a BAT for each column of an SQL table, and keeping the administration of the set of BATs that make up a single SQL table.

The output produced by an SQL compiler consists of sizeable MAL programs, mostly comprising binary relational algebra operations. They have already been optimised using the basic relational rewrite rules, such as selection push-downs by the SQL compiler. The MAL program also takes care of glueing together the relational containers with pending inserts, deletes and updates to represent the latest consistent snapshot. Furthermore, the MAL program is decorated with all information needed to optimise and execute the query without access to the SQL catalog. This information takes the form of function calls with all-constant arguments and properties linked with variables.

Figure 8.2 illustrates the MAL plan produced by the compiler for the query
\[\text{SELECT } \text{count}(*) \text{ FROM } R, S \text{ WHERE } R\.key = S\.key \text{ AND } R\.Key < 23.\]
The query is translated into a cached function, which is called with argument 23. The body of the function is a linear representation of the logical expression. The first section locates the BATs that represent the two tables R and S. It also obtains the reference to the pending updates and deletes, which are consolidated in the algebraic section. The major part is the binary relational algebra plan.

The effective result of the SQL compilation phase is a MAL plan that references columns that are necessary to compute the answer to the original SQL query. The Armada implementation at the MAL level targets individual BATs. Since the compiled SQL queries address BATs, this is where the Armada comes in. Obviously, as each Armada in this case is a single column, vertical fragmentation is not possible in this scheme.
function user.s2_0(A0:sht):void;
  _2:bat[:oid,:int]{rows=1:lng,notnil=true} := sql.bind("sys","r","key",0);
  _7:bat[:oid,:int]{rows=0:lng,notnil=true} := sql.bind("sys","r","key",1);
  _10:bat[:oid,:int]{rows=0:lng,notnil=true} := sql.bind("sys","r","key",2);
  _14:bat[:oid,:oid]{rows=0:lng} := sql.bind_dbat("sys","r",1);
  _29:bat[:oid,:int]{rows=1:lng,notnil=true} := sql.bind("sys","s","key",0);
  _31:bat[:oid,:int]{rows=0:lng,notnil=true} := sql.bind("sys","s","key",1);
  _33:bat[:oid,:int]{rows=0:lng,notnil=true} := sql.bind("sys","s","key",2);
  _36:bat[:oid,:oid]{rows=0:lng} := sql.bind_dbat("sys","s",1);
  _9 := algebra.kunion(_2,_7);
  _12 := algebra.kdifference(_9,_10);
  _13 := algebra.kunion(_12,_10);
  _15 := bat.reverse(_14);
  _16 := algebra.kdifference(_13,_15);
  _17 := A0;
  _18 := calc.int(_17);
  _19 := algebra.uselect(_2,nil:int,_18,false,false);
  _22 := algebra.uselect(_7,nil:int,_18,false,false);
  _23 := algebra.kunion(_19,_22);
  _24 := algebra.kdifference(_23,_10);
  _25 := algebra.uselect(_10,nil:int,_18,false,false);
  _26 := algebra.kunion(_24,_25);
  _27 := algebra.kdifference(_26,_15);
  _28 := algebra.semijoin(_16,_27);
  _32 := algebra.kunion(_29,_31);
  _34 := algebra.kdifference(_32,_33);
  _35 := algebra.kunion(_34,_33);
  _37 := bat.reverse(_36);
  _38 := algebra.kdifference(_35,_37);
  _39 := bat.reverse(_38);
  _40 := algebra.join(_28,_39);
  _41 := calc.oid(000);
  _43 := algebra.markT(_40,_41);
  _44 := bat.reverse(_43);
  _45 := aggr.count(_44);
  sql.exportValue(1,"sys.","count_","int",32,0,6,_45,"");
end s2_0;

Figure 8.2: SELECT count(*) FROM R, S WHERE R.key = S.key AND R.key < 23.
8.3 Static MAL Armada implementation

MAL code produced by the SQL compiler follows in general a sequence of bind calls, BAT operations and a final result for the user. The bind calls refer to local BATs for the SQL compiler that are maintained and kept aligned where necessary. Since these bind calls are just visible in the MAL code, code transformers can detect and replace these bind calls with Armada specific calls to activate special use of BATs as part of an Armada. Since this is done by code transformers, the SQL compiler is unaware of this change. This is the starting point for an Armada at the level of MAL code.

To have Armada working at the MAL level, we need to encode the Armada model in MAL structures. The MAL language has the notion of function routines that have one or multiple return values after a call. We can use functions to represent a box. Normal BATs are not sufficient to represent a box, since in the Armada model boxes can become inactive, which means such boxes are redirects to other boxes. BATs cannot have this behaviour, but functions, on the other hand, can be changed to reflect this state. Assuming that we have an armada name space, box functions with an unique name can be placed in this name space. An Armada “BAT” is addressable via its box name only. This means that a BAT is wrapped by a function stored in the armada name space.

**Wrapper Function** In Figure 8.3 such function is shown. It contains extra information that in later stages can be used to steer the process of query execution. Starting with the declaration of the function, we see two properties, inline and active. The first property is a hint that the contents of the function can (and should) be easily inlined in other code which calls the function. Inlining allows code transformers and optimisers to easily consider the function as a part of the original code and do changes that go beyond the scope of the inlined function.
The second property, *active*, represents the state of the box, whether it is containing data (active) or whether it is just a redirect to other boxes (inactive) after an operation has been applied. Recall from the Armada model that a box can only participate in an operation once. Hence, when the box is marked as inactive, its contents (redirects) never changes.

**Assertions** The first statement in the function is an assertion statement, that ensures the function is representing an active box. While this may seem superfluous, its use becomes apparent in the light of function caching. A function that is once read by a remote site may be cached or implicitly cached due to the inlining property. A box can become inactive, hence so can the function. This happens when the function is for instance chunked. Caching a function which represents an active box leads to problems once the box becomes inactive. A site which uses an outdated (cached) version of the function does not notice that the used BAT is no longer the full dataset, but only a part instead. Obviously this is very much undesirable, hence it needs to be ensured that the cached function is legally used in case of an active box. armada.ensureActive is a no-operation function for the program in the function. When it fails to ensure that the given function is active, it forces a reconsideration of the query (MAL) plan, starting from the last known state where the function which failed to ensure being active was introduced. We discuss the details of this trap in the optimiser framework further on in Section 8.3.1.

**Remote Retrieval** The next line in the function gets a BAT from the remote site and assigns its copy to b0. The get effectively copies the data from the remote BAT to the local site and makes it available as b0. The copying can be delayed to the first moment that b0 is used. Further optimisations regarding fetching only parts (to avoid doing the entire BAT) are of later concern, at the stage where we know upfront that we only need a part of the BAT. For now, it keeps the functionality of the get function limited to making the remote BAT available. Note that the get call is constructed in such a way that also when the contents of the function is inlined on another site, the statement still works and results in the same data. The mechanism should be clever enough to simply issue a local bind in case the “remote” host is the same as the local host.

**Function Encoding** To encode the functions that are applied to the boxes, as defined in the Armada model, special meta-statements are necessary. Since a box does not contain any more data than their function describes, additional
selections over the boxes that match the function are a waste of efforts. The adding such select statement in the function technically allows for optimisers to detect the selection boundaries and take them into account, such ordinary select statements are hard to distinguish from those statements that are really necessary to perform. To avoid this potential performance pitfall, we use meta-statements for this that describe what data can possibly be in a given BAT. The armada.chunk statement does this for BAT b0 in the above MAL function. It describes a range chunk function and specifies what (numeric) range is being used.

Returning Finally, b0 is being returned to the caller. Note that b0 carries the properties remote and armada. The first marks the BAT as being a copy of a remote BAT somewhere. The second marks the BAT being a result of and owned by Armada. Both properties can be used later in the process to treat the BAT properly on optimisations and operations.

8.3.1 Use of Optimiser Plan Stacks

The MonetDB distribution comes with a large collection of optimiser modules. They are developed up to the point that they could be used to experiment with the optimiser software infrastructure. They are highly targeted at a particular problem. Table 8.1 shows the modules forming the optimiser pipeline for SQL plans.

The MAL language does not imply a specific optimiser to be used. Instead, calls to specific optimiser routines is part of the MAL program produced by the front-end compiler. They are, however, evaluated during the optimiser phase only.

Plan Stacks Optimisers have the freedom to change the code, provided it is known that the plan derived is invariant to changes in the environment. In particular, an optimiser may leave behind calls to other optimiser routines. When all optimiser calls have been dealt with, the query plan is cached and ready for execution. The alternative plans are collected as a stack of MAL program blocks. The plan stack can be inspected for a posterior analysis of optimiser behaviour. Alternatively, the stack may be pruned and re-optimised when appropriate from changes in the environment. An example of such change is whether a BAT is empty or not. Big parts of the code may be disabled and removed if an input BAT is empty, however, if the BAT becomes non empty later on when reusing the
inline inlines functions identified as such
remap locates hardwired multiplex operations
costModel inspects the SQL catalog for size information
coercions performs static coercions
emptySet removes all empty set expressions
access modes ensures that BATs for update are writeable
aliases removes alias assignments
common terms searches for common terms and retains one only
accumulators re-uses BATs to hold the result of an expression
joinPath searches multiple joins and glues them together
deadcode removes all code not leading to used results
reduce reduces the stack space for faster calls
garbageCollector injects calls to free up space
multiplex translates multiplex operations to iterators

Table 8.1: The MonetDB/SQL optimiser pipeline.

same (optimised) plan, a re-evaluation of the original plan is necessary, since it obviously is not valid any more. Re-evaluation may, however, not always have to be done completely from scratch. It may only be necessary to re-evaluate from a given point in the stack. This is typically the case for the armada.ensureActive function. In case the check fails, a re-evaluation has to be made starting from the point where the statement was introduced. For the Armada case, this is the point where the remote function was inlined in the query plan.

Labels When the armada.ensureActive statement is inlined, its invocation is changed to refer to the plan right before the inline operation. This allows the operation to jump back to the right plan in the stack to start re-evaluating. For this to work, each plan in the stack is labelled. MAL properties control whether a reference to a label should be made or whether the statement has to be kept as is. The transformer that sets the label for the ensureActive statement looks for the properties func and label. If one or both are missing or the value of func is not equal to the current function name, the statement is considered to be new. In this case the func property is set to the name of the current MAL function. The label property is set to the label of the current plan in the stack, that is the label of the last known plan. This is not the plan that is currently being generated by the transformer. Using this scheme the jump point for the ensureActive statement is being set once it is introduced or inlined into
8.3. STATIC MAL ARMADA IMPLEMENTATION

```plaintext
sql oranges => columns "id" is t_id, "name" is t_name, "value" is t_value
bat t_id => "december" on "amalia"
bat t_name => "june" on "alexia"
bat t_value => "april" on "ariane"
```

Figure 8.4: The catalog state.

another plan. This results in the desired behaviour where a jump is being made to the right label and an optimiser is able to simply determine whether the jump label has to be set or not.

8.3.2 Analysis Use Case

Using the previously described techniques to manipulate functions on the MAL level, we can build an Armada system as follows. For this example we assume that the catalog contains information about an SQL table “oranges”, having three columns, “id”, “name” and “value”. Each of the three columns are mapped to a named Armada BAT, “december” on host “amalia”, “june” on host “alexia” and “april” on host “ariane” respectively. Figure 8.4 schematically represents this catalog.

The state of the catalog specifies that there exist three sites that have an Armada BAT. Those BATs are initially set up on their sites, and made accessible by a wrapper function. The wrapper functions are like Figure 8.3, but for “december” on “amalia”, a chunk operation was applied. The three transitions the function “december” made are depicted in Figure 8.5.

In our example we only consider the last state of the “december” function to be used. In other words our example runs only after “december” has been chunked. As can be seen in Figure 8.5, “december” has become an inactive box, marked by the absence of the “active” property. Any following changes do not affect the box being active or not, following the Armada model. The definition of the newly used boxes, such as “january” on host “maxima” has been inlined in the “december” function, including the guarding assertion to make sure no out of date definition of the function is used. The armada.chunk operations indicate how the chunk was performed using a range function. The guards may cause the function to be re-evaluated starting from the second function. As can be seen, the armada.ensureActive call for the “december” function itself is dropped in the second function, as it is no longer necessary to ensure that the box is active. The box is inactive, which means its actual contents is not going
function amalia.december{inline,remote,active}():bat[oid:int];
  armada.ensureActive("amalia.december");
  b1 := remote.bind("amalia","b7");
  ret := armada.chunk(b1,0,-1);
  return ret;
end december;

function amalia.december{inline,remote}():bat[oid:int];
  b1 := amalia.regina();
  b2 := maxima.january();
  return ret := algebra.sunion(b1,b2);
end december;
  armada.resolve("amalia.regina");
  armada.resolve("maxima.january");

function amalia.december{inline,remote}():bat[oid:int];
  armada.ensureActive("amalia.regina");
  b1 := remote.bind("amalia","b7");
  b3 := armada.chunk(b1,0,20);
  armada.ensureActive("maxima.january");
  b2 := remote.bind("maxima","b32");
  b4 := armada.chunk(b2,20,-1);
  return ret := algebra.sunion(b3,b4);
end december;

Figure 8.5: Transitions for the function “december” on host “amalia”.
function user.s1_0():void;
  _8:bat[:,oid,:int] := armada.bind("t_id");
  _15:bat[:,oid,:str] := armada.bind("t_name");
  _18:bat[:,oid,:int] := armada.bind("t_value");
  _11 := algebra.markT(_8,0@0);
  _12 := bat.reverse(_11);
  _13 := algebra.join(_12,_8);
  _16 := algebra.join(_12,_15);
  _19 := algebra.join(_12,_18);
  _20 := sql.resultSet(3,1,_13);
  sql.rsColumn(_20,"armada.oranges","id","int",32,0,_13);
  sql.rsColumn(_20,"armada.oranges","name","varchar",24,0,_16);
  sql.rsColumn(_20,"armada.oranges","value","int",32,0,_19);
  sql.exportResult(_20,"");
end s1_0;
  armada.resolve("user.s1_0");

Figure 8.6: Simplified initial SQL query plan for SELECT * FROM oranges.

to become wrong, at most out of date. Hence, the function can be freely copied
and cached. Of course the newly added guards in the third function make the
function body itself potentially incorrect again. This is due to the contents of
the referenced functions being inlined.

Query Execution In the described setting, an agent called “trix” executes the
SQL query SELECT * FROM oranges. This query translates to MAL code as in
Figure 8.6. The figure shows simplified code, leaving out the delta administra-
tion for inserts and deletes. Also, some optimisers have been run. Normally
also the empty set optimiser runs, but in this example it has been disabled,
for it would remove empty results, thereby making it harder to see what the
code is doing. The code in the figure essentially just binds to the columns and
prepares them to be aligned before being added to a final result where addi-
tional metadata is stored. At the bottom of the code an armada.resolve call
is found. It processes the armada.bind calls that are inserted as a replacement
of sql.bind calls. This processing either results in the call being replaced by a
simple sql.bind call for non-Armada BATs, or the (remote) function that maps
the requested Armada BAT.

Figure 8.7 depicts the situation after resolving the armada.bind calls. In
function user.s1_0():void;

_8:bat[:oid,:int] := amalia.december();
_15:bat[:oid,:str] := alexia.june();
_18:bat[:oid,:int] := ariane.april();
_11 := algebra.markT(_8,0@0);
_12 := bat.reverse(_11);
_13 := algebra.join(_12,_8);
_16 := algebra.join(_12,_15);
_19 := algebra.join(_12,_18);
_20 := sql.resultSet(3,1,_13);
sql.rsColumn(_20,"armada.oranges","id","int",32,0,_13);
sql.rsColumn(_20,"armada.oranges","name","varchar",24,0,_16);
sql.rsColumn(_20,"armada.oranges","value","int",32,0,_19);
sql.exportResult(_20,"");
end s1_0;

optimizer.inliner("user.s1_0");

Figure 8.7: Resolved query from Figure 8.6.

our case, all binds are replaced with calls to our previously defined Armada BAT functions. Not surprisingly, these calls can be inlined to further expand the plan, like we did in Figure 8.5. Eventually, after all code has been expanded, the final plan becomes as in Figure 8.8. In that plan, in total three armada.ensureActive statements are present. These statements are potentially invalidating the plan. More importantly, they need to be checked during execution, which requires (network) communication with the involved site. This obviously is an inevitable pity, since it adds extra network costs. When the plan is being constructed, the site is contacted for the first time. Then before the data is being requested, it is checked whether the plan is up-to-date by checking with the site. Finally, the site is contacted for the last time to retrieve the data. In case any up-to-date check fails, even more communication costs are involved. Obviously, it is desirable to reduce the number of communications with sites.

8.4 Stepwise Dynamic Inlining

While the before described approach is clear for analysis of queries to be performed on an Armada, it imposes a different agent strategy than the Armada model proposes. In terms of agent communications with remote servers the Ar-
function user.s1_0():void;
  armada.ensureActive("amalia.regina");
  _1 := remote.bind("amalia","b7");
  _2 := armada.chunk(_1,0,20);
  armada.ensureActive("maxima.january");
  _3 := remote.bind("maxima","b32");
  _4 := armada.chunk(_3,20,-1);
  _8:bat[:oid,:int] := algebra.sunion(_2,_4);
  armada.ensureActive("alexia.june");
  _14 := remote.bind("alexia","b26");
  _15:bat[:oid,:str] := armada.chunk(_14,0,nil:str);
  armada.ensureActive("ariane.april");
  _17 := remote.bind("ariane","b10");
  _18:bat[:oid,:int] := armada.chunk(_17,0,-1);
  _11 := algebra.markT(_8,0@0);
  _12 := bat.reverse(_11);
  _13 := algebra.join(_12, _8);
  _16 := algebra.join(_12, _15);
  _19 := algebra.join(_12, _18);
  _20 := sql.resultSet(3,1,_13);
  sql.rsColumn(_20,"armada.oranges","id","int",32,0,_13);
  sql.rsColumn(_20,"armada.oranges","name","varchar",24,0,_16);
  sql.rsColumn(_20,"armada.oranges","value","int",32,0,_19);
  sql.exportResult(_20,"");
end s1_0;

Figure 8.8: Final expanded query plan.
mada agent is more efficient, and also efficiency through caching is significantly easier. The key of an Armada agent is that it detects an outdated box when it asks it for its data. Not only does this mean no total plan is generated before execution, but also that an agent has no notion of whether a box is active or inactive. When an agent requests a box for its data, it simply receives this data, or a redirection where to find this data. The agent simply refines its plan at runtime with this received redirect. Based on this property, another approach was devised that inhibits this plan refinement at runtime behaviour.

Dynamic environments require dynamic execution models to be efficient. *Stepwise dynamic inlining* is a query execution strategy where the query plan is incrementally built during query execution. It can be seen as a refinement of the query plan that is made during execution. The plan is seen as just an approximation or as out of date.

**Incremental Optimisations** The architecture imposed by the Armada model requires an agent to be an active player during query execution. Not only does this mean the agent is doing a lot of work, but also that this work implies communication with the servers involved. Optimisations in this area are indispensible to reduce expensive network calls. However, optimisations cannot be made without a plan for all actions to take. This yields in a contradicting situation where on the one hand the full plan needs to be expanded and on the other hand the costs of unnecessarily expanding the plan needs to be avoided. A conventional approach to query execution requires a full plan to be made before the actual execution starts, such that optimisations aimed at the full execution can be made. In the case of Armada, the tree-shaped lineage tree offers the opportunity to minimise the granularity of the plan gradually as the agent executes it. With the agent contacting servers to ask for the data, every time it receives a query plan back, it can update its query plan and do incremental optimisations during the execution phase.

Figure 8.9 depicts the architecture of the approach taken to obtain stepwise dynamic inlining. The behaviour of the Armada agent to possibly change its query plan during execution suggests it has to rewrite the currently running code. While in Armada this means that one instruction is replaced by a number of others (inlining) at runtime it imposes a serious administration burden to do so. Many references from and to values placed on the stack may need to be shifted in the current context. While an implementation like that has many potentials to cause random *crashes* in the implementation of MAL which was not designed to support this, a multi-staged solution was used as shown in the figure.
8.4. STEPWISE DYNAMIC INLINING

Figure 8.9: Stepwise Dynamic Inlining in action.

Agent

Server A

Server B

function b3():bat;
  r := bbp.bind("b");
  return r;
end b3;

function b1():bat;
  e := armada.redirect(b2, b3);
  raise e;
end b1;

function stub1():bat;
  l := stub2();
  r := stub3();
  b := algebra.sunion(l, r);
  return b;
end stub1;

r := remote.exec("b1");
catch RedirectException re;
  r := armada.rewrite(re);
end stub1;

...
Like in the analytic approach, the query of an agent is transformed such that requests for data in an Armada are made available in the plan. In the dynamic approach stub functions are used instead of armada.bind calls. The purpose of the stub functions is to implement dealing with redirects encountered at runtime. The function stub1 tries to execute a function on the remote site to obtain the data. It is prepared to receive an exception which like a redirect contains enough information to create two new stub functions stub2 and stub3 and to redefine function stub1 as a union of the result of the two new stub functions. After the rewrite the new function is called, which may iterate the same process again to obtain the data.

**Dynamic Rewriting** Going back to the abstract level of Figure 8.9, a box on a server is represented by a function that initially returns the data in the box as a BAT. The function remains at the server and is not meant to be shipped to any other site, or its implementation to become visible to others. This characteristic forces an agent to call the (remote) function when it wants to obtain the data from the box. Whether or not such call is cached as query plan, the real data being retrieved is always correct, as in, not out-of-date. When the box is chunked, its function is replaced by one that raises an exception which includes redirect information. An agent calling the function to retrieve the data’s box, then receives the raised exception which allows to handle the redirection in the query execution.

From the agent’s perspective the redirection exception raised by the remote function initiates a procedure to follow the redirection in the query. Since this situation occurs at run-time, a workaround to avoid rewriting the currently executing query plan is made through use of the aforementioned stub functions. Internally, when a new function is called, a new execution environment is used. For this reason, the soon to execute function can be created or modified just before execution. Also, functions can be overwritten, thereby “changing” the definition of the function for the next caller of that function. These ingredients are used by armada.rewrite to effectuate the redirect received from the server. First it creates two new stub-functions for the new boxes received via the redirect. These stub-functions are created by a template which simply tries to retrieve the box’ data as BAT, or handles the redirection exception. Because those stub-functions are specific for a given box, the location (not shown in the figure) and name of the function to call are hardcoded in the stub-functions. Stub-functions cannot be reused, since they are overwritten when a redirect is encountered. Hence, hardcoding the remote function information in the stub
is not limiting in any way from a code and architecture point of view. Finally, armada.rewrite overwrites the function it was called from with a simple function which calls the two new stub-functions and returns the union of both BATs. Since execution of the original function where armada.rewrite was called is not affected, armada.rewrite calls the new function which overwrites the old stub and returns that result to the original stub-function, which is written in such a way that the data is finally delivered to the user's program. Note that the original stub-function stub1 is created as part of the particular Armada initialisation on the system. It is part of the Armada catalog that enables availability of Armada BATs in the database system.

8.4.1 Caching

With stepwise dynamic inlining, each remotely known BAT from an Armada has a corresponding stub function on the agent. The stub functions are a necessity, but have a side effect of generating a cache on the agent. Each following use of stub1 benefits from the already previously retrieved stub functions, as local access is cheaper and faster than constructing new stub functions from a remote redirect.

As the Armada grows, the agent has to store more stub functions. As a result, for each query the entire stub function tree has to be traversed. While still being local, these eventually long chains cause a lot of function call overhead. In addition, each query performed, has to execute the same path again, since they start at stub1. Figure 8.10 depicts such a trail of stub functions. Not surprisingly, such trail carefully follows the lineage trails of boxes. As such the same conditions of active and inactive stub functions hold. This means that stub-functions rewritten by armada.rewrite are not going to change and hence are eligible for inlining.
begin stub1():bat;
    _7 := stub6();
    _8 := stub7();
    _9 := algebra.sunion(_7, _8);
    _4 := stub4();
    _5 := _9;
    _6 := algebra.sunion(_4, _5);
    _1 := _6;
    _2 := stub3();
    _3 := algebra.sunion(_1, _2);
return _3;
end stub1;

Figure 8.11: A fully inlined stub1 function.

**Inlining Stubs** Inlining the stub functions has the largest benefit when this is done in stub1, since each query using it then immediately accesses the fully inlined plan. Figure 8.11 depicts an inlined version of stub1 in the situation depicted by Figure 8.10. Here, all possible stub functions are inlined. The resulting plan is flattened with the only functions not inlined, those stubs that represent active boxes, and hence may change in the future. Note that the inlining process has inlined the stubs in chronological order. As such originally stub2 was assigned to variable _1. Further optimisations can reduce the size of the code by removing assignments such as _1 := _6; by propagating them through the code. This is commonly referred to as *alias removal*. The inlined plan in this state can be cached and reused for subsequent queries on the Armada BAT. Important detail here is that the inlining and optimisations are done on stub1, and not in the program that calls stub1. This is quite uncommon, but is the only way in which subsequent queries benefit from the work previously done. The contents of stub1 may not be inlined into the calling program, since that would complicate matters in case one of the stub functions appears to be out of date.

Having rewritten stub functions being inlined in the top-level stub function, allows for generic optimisations to be applied only once, instead of for each query that uses the stub. Since active stub functions are not inlined, armada.rewrite which operates on those stubs, does not have to change the way in which it rewrites the stub functions. As a stub function gets rewritten, it effectively gets replaced without taking the original contents into ac-
8.4. STEPWISE DYNAMIC INLINING

... a := bbp.bind("some_bat");
b := bbp.bind("another_bat");
...
p := bat.select(a, 10, 20);
q := bat.select(b, 40, 60);
...
t := algebra.join(p, q);
...

Figure 8.12: Example selection code.

count. This is no problem as long as the stub functions are not changed before armada.rewrite operates. The result of a rewritten stub-function can of course be inlined as well. For instance when stub6 from Figure 8.10 gets rewritten, upon a next query it may get inlined into the stub1 function. This scheme effectuates a self-reorganisation after a dynamic query refinement.

8.4.2 Volume Optimisation

In a distributed setting, optimisations on network costs always play a prominent role. In Armada, two different kinds of optimisation can be made in this respect. First, avoiding a call to a stub function yields in a full reduction of network communication. As side effect, it also avoids code expansion costs due to dynamic rewriting. Second, the data being retrieved can be minimised by pushing selections down into the remote server that holds the data. While this takes extra communication to retrieve only the selection, it potentially pays off against the number of data tuples that do not have to be shipped between the sites. For both optimisations it is necessary to have the Armada functions from the model. Without them, no information on what data is contained where is available, and hence no box can be assumed not to have the data. Encoding the functions in MAL can be done by assigning properties to the BATs returned by the stub functions. Properties are from a human code consumption point of view natural elements to tag objects with certain characteristics. Also for optimisers, for example minimum and maximum values as properties of a BAT are easy to consume and process.
tmp1{pmin=0,pmax=50} := stub2();
tmp2{pmin=50} := stub3();
tmp3{pmin=0} := algebra.sunion(tmp1{pmin=0,pmax=50}, tmp2{pmin=50});
b{pmin=5,pmax=20} := bat.select(tmp3{pmin=0}, 5, 20);
io.print(b{pmin=5,pmax=20});

Figure 8.13: An Armada chunk stub function using properties.

Code Optimisation Consider the example in Figure 8.12. There are no properties in the code, but it is not hard to imagine that a and b have undefined minimum and maximum values, whereas p and q have limits between 10, 20 and 40, 60, respectively. The final join is guaranteed to have an empty result, as p and q cannot have any tuples in common. End result is that the bbp.bind calls can be removed from the plan, thereby reducing actual work to do. Existing optimisers are already able to effectuate this, based on the previously mentioned code analysis. Reflecting this on the stub functions, inserting select calls on the result of the remote.exec calls would allow for the same optimisation without introducing an Armada specific optimiser. This contrasts the armada.chunk function used in the statical analysis approach.

Consider Figure 8.13 which depicts a selection made over a typical union of two stub functions. The figure shows the use of properties for the BATs in use. The possible minimum (pmin) and maximum (pmax) properties for the results of the stub functions are part of the stub function's definitions. Function stub2 covers a range from 0 till (not including) 50. Since stub3 has no upper limit on its range, only the minimum possible value (50) is encoded as property. In the generality of the property management, analysis of the property sets upon operations result in properties on return values. In the case of the algebra.sunion a "union" of both BATs theirs pmin and pmax properties results in the range of 0 till infinity. Of course the result of a bat.select call results in a BAT with pmin and pmax equalling the selection criteria. With all these properties available, a few conclusions can be made, when looking at the code from the bottom to the top. The selection on tmp3 can possibly result in something, since the input BAT possibly holds data in the selection range. If this were not the case the bat.select statement could be replaced by an empty set. Such operation can yield in more statements becoming void, which are removed by the empty set optimiser.

Another optimisation based on the previous example is to reduce the size of the sub-results, in particular because they need to be fetched from a remote...
8.4. STEPWISE DYNAMIC INLINING

tmp1{pmin=0,pmax=50} := stub2();
tmp4{pmin=5,pmax=20} := bat.select(tmp1{pmin=0,pmax=50}, 5, 20);
tmp2{pmin=50} := stub3();
tmp6{rows=0} := bat.select(tmp1{pmin=50}, 5, 20);
tmp3{pmin=5,pmax=20} := algebra.sunion(tmp4{pmin=5,pmax=20}, tmp6{rows=0});
io.print(tmp3{pmin=5,pmax=20});

Figure 8.14: Selection push-down applied on the example of Figure 8.13.

In Figure 8.14 the selection is pushed through the union operation such that the selection is done right after the stub functions over tmp1 and tmp2. Due to the properties, it can be seen that the selection over tmp2 is going to be empty, resulting in an empty set, indicated by the property rows=0 on BAT tmp5. An effect of this knowledge to the empty set optimiser is that the union operation is now useless, and the entire operation can be skipped, with the result being just tmp4. The bat.select on tmp1 can be removed as part of this, since the operation does not have any effect. In turn, the dead code optimiser detects that the stub3 function call is needless, as its result is never used. This finally leads to avoidance of any network communication performed by function stub3.

Runtime Optimisation  The set of regular optimisers work fine here on the inlined code, as shown before. An important problem, however, is that the stub1 code cannot be inlined as we concluded before. Newly rewritten functions would not benefit from the optimisations, and optimisations applied to existing code possibly breaks for other queries. Regardless, selection optimisation as done above has the desirable effect of reducing network communication. Not inlining stub1 makes the stub function a black box to the optimiser. Nothing can be done to it, the same situation that characterises a just rewritten stub-function, and selection optimisation therein. Temporarily inlining only works for the stub-functions representing an inactive box, the active ones can get rewritten at runtime. Still, it means a temporary copy is made which is modified for the query at hand. Since the dynamic nature of the Armada query strategy forces a runtime based strategy, static (analytic) optimisers per definition fail to meet the requirements. Instead of them, runtime guards to avoid unnecessary work can provide generic avoidance of expensive network calls, at the expense of slightly higher execution costs. To do these runtime checks, more information is necessary at runtime in the stub functions. Starting with the user’s original query again, we can use optimisers and static analysis to find if there
b := stub1(0, 10);
c := algebra.select(b, 0, 10);
io.print(c);
...

Figure 8.15: Optimisers can “push” the selection criteria through the stub functions.

is a selection made on the BAT retrieved by the stub function being called. This selection range then can be made an argument to the stub function call, such as in Figure 8.15.

The low and high values of the selection range are put as arguments of the stub1 function, making the selection operation on b unnecessary. It is left here in the figure for explanation purposes. When no selection is made or its range cannot be determined, the default low and high values of nil are filled in which denote an unlimited range. A range for example cannot be determined when the selection is done using variables filled in at runtime. Though to a certain extent, also these variables can in some cases be used as arguments for the stub function. In some cases execution order may be changed such that the selection value is known in time for the stub function at runtime, but this may be too complicated to derive, or simply impossible. With the possible selection criteria the stub function can now use this information to possibly skip consulting remote servers that do not have the requested data. Instead of using properties that aid in statical analysis of the plans, the data coverage of the stub functions is now encoded in the code by means of the guards.

In Figure 8.16, a quick exit is encoded in a stub function to avoid performing the remote function call to retrieve the data. To not to disturb the calling code, an empty BAT is returned otherwise. Upon rewrite of the stub-function, the guards can be placed in the rewritten function as well, such as depicted in Figure 8.17.

With this code, it is no longer necessary to have any guards in the active stub functions, since they are not called when they cannot produce data matching the selection criteria. Note, however, that the selection range is now also given as argument to the remote.exec call. This allows the remote site to only return what is necessary for the selection. This is a further reduction of network traffic that comes for free now the selection range is pushed through all stub functions being called. New stub functions generated by the armada.rewrite call hence
function stub1(low:int, high:int):bat;
    barrier e := low > 10 || high < 0;
    r := bat.empty();
    return r;
    exit e;
    try;
    r := remote.exec("b1", low, high);
    catch RedirectException re;
    r := armada.rewrite(re);
    exit re;
    return r;
end stub1;

Figure 8.16: Runtime guards in active stub functions.

function stub1(low, high):bat;
    barrier e := low > 10 || high < 0;
    lb := bat.empty();
    opposite e;
    lb := stub2(low, high);
    exit e;
    barrier e := high < 10;
    rb := bat.empty();
    opposite e;
    rb := stub3(low, high);
    exit e;
    b := algebra.sunion(lb, rb);
    return(b);
end stub1;

Figure 8.17: Guards placed in an inactive stub function.
include low and high arguments and the respective guards. The low and high arguments are passed on to the remote.exec call. Because stub1 is generated by the Armada initial ritual, it can contain the guards as shown before, such that calls to stub1 in user code need no transformation to add the guards. This means that analytic compilers can do their job on the user’s code up to the point where they know what the selection criteria are for the stub1 function. From there the execution dynamically deals with the optimisations at runtime.

Summary

An Armada implementation requires more functionality from a generic database engine, than it can deliver via the SQL layer. To overcome this functionality shortage, we chose MAL, the algebraic MonetDB Assembler Language, as target for our next exploration of an Armada implementation.

Our implementation focusses on the columns present in the database, and by means of wrapper functions for those columns, Armada boxes are simulated. Since the functions can be changed, an operation is administered in such function by changing its code, and creating new functions representing the new boxes. The lineage in these functions is encoded by the call to other functions, and available to others by copying the functions.

We put some attention to optimisations in the query execution over an Armada. Compile time variants of MAL plans are able to reduce a lot of work, but they are hampered by the execution phase which can find a situation not accounted for. Instead, runtime variants are necessary to limit the communication with other sites to a minimum, by requiring only a single call to either get an answer or redirect.

Caching the MAL functions received from other sites allow to speed up the process by requiring less communication with other sites. Many optimisation efforts can be performed on the cached plans, resulting in a high benefit due to reuse.

With the final proposed implementation on the MAL language, we have shown that it is possible to implement an Armada agent conforming to the model. This implementation respects the autonomy of the involved sites, supports the decentralisation, and hence does not block the evolution of the cluster.