RAM: array database management through relational mapping

van Ballegooij, A.R.

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: https://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.
Appendix A

A RAM Example: Sample Likelihood

This Appendix contains the complete translation of a larger example from a RAM expression to three different back-end languages: C++, MIL, and, X100. The example used is the GMM scoring function explained in Section 3.2.8 and used for the optimization experiments in Chapter 6. Apart from the unfolding optimization presented in Section 5.2.1, the example includes the optimizations discussed in Chapter 6.

A.1 The RAM Expression

Consider the GMM-scoring RAM expression presented earlier. The RAM query script consists of three parts. The first part defines the shape, element type, and native names (storage names in the back-end) of the persistent arrays present. The second part defines a number of functions to ease the expression of the query: These functions are merely syntactic sugar and implemented as macros; they are applied through straightforward substitution at query-compile time. The third, and final, part of the script contains the actual query.

# Definition of array-variables
Img  = ([14,1320],dbl,"query_bat")
Mu   = ([14,8,32318],dbl,"mu_bat")
S    = ([14,8,32318],dbl,"sigma_bat")
P    = ([8,32318],dbl,"prior_bat")

# These arrays are defined over axes:
# n = 14 , the number of dimensions of the feature vectors
# c = 8 , the number of components in each Gaussian mixture model
# m = 32318 , the total number of models in the collection
# s = 1320 , the number of samples in query image 'Img'

# Definition of RAM macros
norm(c,m) = 1.0 / (sqrt(pow(2.0 * 3.1415, 14.0)) * prod([ S(n,c,m) | n ]))
activ(c,s,m) = norm(c,m) * exp(-0.5 * sum([ pow(Img(n,s) - Mu(n,c,m), 2.0) / S(n,c,m) | n ]))

# The expression to be evaluated
RES  = [sum([ log(sum([ P(c,m) * activ(c,s,m) | c ]] | s ])) | m ]}

159
During the optimization experiments conducted throughout Chapter 6, two major changes were made to this query plan. First, as described in Section 6.1.1, part of the expression was pre-computed. Normally the optimizer would realize pre-computation by materializing the expression inline and using its values through application at the array-algebra level. The effect of optimization can be mimicked in the RAM expression as follows:

\[
\text{activ}(c, s, m) = \{ \text{norm}(c, m) | c, m \} \ast \\
\exp(-0.5 \ast \text{sum}([\text{pow}(\text{Img}(n, s) - \text{Mu}(n, c, m), 2.0) / S(n, c, m) | n ]))
\]

However, to keep the example concise, we explicitly materialize the sub expression into a persistent array.

\[
\text{NORM} = \{ \text{norm}(c, m) | c, m \}
\]

\[
\text{activ}(c, s, m) = \text{NORM}(c, m) \ast \\
\exp(-0.5 \ast \text{sum}([\text{mahalanobis}(\text{Img}(n, s), \text{Mu}(n, c, m), S(n, c, m)) | n ]))
\]

Second, as described in Section 6.1.1, the sub-query implementing the Mahalanobis distance has been compiled into a user defined function (UDF):

\[
\text{activ}(c, s, m) = \text{NORM}(c, m) \ast \\
\exp(-0.5 \ast \text{sum}([\text{mahalanobis}(\text{Img}(n, s), \text{Mu}(n, c, m), S(n, c, m)) | n ]))
\]

The examples in the remainder of this appendix are based on the query script with both these changes in place:

\[
\#	ext{Definition of RAM macros and pre-computation of the NORM array}
\text{norm}(c, m) = 1.0 / (\sqrt{\text{pow}(2.0 \ast 3.1415, 14.0)}) \ast \text{prod}([S(n, c, m) | n ]))
\text{NORM} = \{ \text{norm}(c, m) | c, m \}
\text{activ}(c, s, m) = \text{NORM}(c, m) \ast \\
\exp(-0.5 \ast \text{sum}([\text{mahalanobis}(\text{Img}(n, s), \text{Mu}(n, c, m), S(n, c, m)) | n ]))
\]

\[
\#	ext{The expression to be evaluated}
\text{RES} = [\text{sum}([\text{log}(\text{sum}([\text{P}(c, m) \ast \text{activ}(c, s, m) | c ])) | s ]) | m ]
\]

### A.2 RAM Query Translation

The RAM system translates queries to its intermediate array algebra before mapping the resulting algebra expression to one of the back-end languages for query evaluation. This section shows three phases in this translation process. First, the query is normalized and flattened, as described in Section 3.4.1. Second, the normalized RAM expression is translated into the array algebra, as described in Section 3.4.2. Finally, the query optimizer, described in Chapter 5, optimizes the algebra expression.

Before a query is translated, the RAM system substitutes all references to macros, in place, with their definition. Substitution of the *activ* macro used, and defined, in the query script results in the following (single) RAM expression:

\[
\text{[sum} \\
\text{[log(sum} \\
\text{[}\ast (\text{P}(c, m), \\
\text{\ast (NORM}(c, m), \\
\text{\exp(\ast ("-0.5", \\
\text{\sum}[\text{mahalanobis}(\text{Img}(n, s), \\
\text{\text{Mu}(n, c, m), \\
\text{\text{S}(n, c, m)) | n ])) \\
\text{\text{S}(n, c, m)) | n ]) \\
\text{\text{S}(n, c, m)) | n ))) \]) \])) \]
\]
A.2. RAM Query Translation

A.2.1 Query Normalization

The first step in the query-translation process is normalization of the query, as described in Section 3.4.1. The key observation here is that all variables have been replaced with explicit references and that as a result of the flattening process a number of additional comprehensions and subsequent applications have been added to the expression:

\[
S(c, s, m) = \sum_{n} \left( c \sum_{i} \left( s \sum_{m} \right) \right)
\]

Compared to the RAM expression presented above, the normalized RAM expression contains three additional comprehensions and six of these comprehensions are dereferenced through application. Additionally, during the flattening process, additional axes have been added to the inner comprehensions to resolve axis dependencies between inner and outer comprehensions. For example, the comprehension directly inside the innermost summation is now specified over (all) four axes whereas the original expression only contained axis \( n \): these axes have been added because they are referenced in the applications of the various arrays inside its value function (the Mahalanobis function).

A.2.2 Producing Array algebra

The second step in the query-translation process is the (straightforward) application of the translation rules described in Section 3.4.2. Application of the translation rules produces the following array-algebra expression:

\[
\text{Apply(}
\text{Aggregate("sum",}
\text{Apply(}
\text{Map("log",}
\text{[Apply(}
\text{Aggregate("sum",}
\text{Apply(}
\text{[Apply(}
\text{Aggregate("sum",}
\text{Apply(}
\end{align*}
\]
A.2.3 Query Optimization

The last step before mapping the algebra expression to any of the back-end languages for evaluation is the application of the RAM optimizer. It is apparent that the optimizer identifies and removes a number of identity transformations from the query plan, evident by the significant reduction in the number of \texttt{Apply} operators. In addition, the optimizer attempts to apply \textit{unfolding}, as described in Sections 5.2.1 and 6.1.1:
However, this Appendix is intended to illustrate the translation rules as presented in Chapter 4. Therefore, we instruct the optimizer not to use the Fold operator, which effectively reverts the expression back to a version that uses the regular Aggregate operator instead:

```plaintext
Aggregate("sum",
  Map("log",
    {Aggregate("sum",
      Map("*",
        {Apply(Variable("priors_bat"),
          Grid([8,1320,32318],0),
          Grid([8,1320,32318],2])},
        Map("*",
          {Apply(Variable("NORM_bat"),
            Grid([8,1320,32318],0),
            Grid([8,1320,32318],2])},
          Map("exp",
            [Const([8,1320,32318],"-0.5"),
             Aggregate("sum",
               Map("mahalanobis",
                 [Apply(Variable("query_bat"),
                   Grid([14,8,1,32318],0),
                   Grid([14,8,1,32318],1),
                   Grid([14,8,1,32318],2])],
                 Const([14,8,32318],"i1")])},
             Apply(Variable("mu_bat"),
               Grid([14,8,32318],0),
               Grid([14,8,32318],1),
               Grid([14,8,32318],2]))},
             Apply(Variable("sigma_bat"),
               Grid([14,8,32318],0),
               Grid([14,8,32318],1),
               Grid([14,8,32318],2])))},
        1}))},
    "i1",
    1320)
)
```
A.3 RAM Array-Algebra Mappings

Given the array-algebra expression derived above, the RAM system has the functionality to produce query plans for a variety of back-end systems. Chapter 4 presents such mappings for a number of different back-end languages: SQL, MIL, X100, Matlab, and C++. In this section we present three of these mappings that represent different platforms: the mapping to C++, a low-level programming language; the mapping to MIL, a main memory relational database language; and the mapping to X100, a fully pipelined relational query language.

Note that, in all three example mappings, the polynomial indexing function discussed in Section 4.2.1 is used to retrieve array elements from a linear storage structure representing a multi-dimensional array.

A.3.1 Mapping to a Low-Level Language: C++

The RAM mapping to C++, presented in Section 4.4.2, produces a code fragment that iterates over the result space computing one value at a time. Aggregates are accumulated incrementally by iterating over the aggregation axes:

```c++
  dbl* res_bat = new dbl[(32318)];
  for(int i0=0;i0<32318;i0++) { // Iterate over the result array
      dbl a1 = 0;
      for(int i1=0;i1<1320;i1++) { // sum(log(...))
          dbl a2 = 0;
          for(int i2=0;i2<8;i2++) { // sum(P * (NORM * exp(-0.5*{...}))
              dbl a3 = 0;
              for(int i3=0;i3<14;i3++) { // sum(mahalanobis(...))
                  a3 += mahalanobis(query_bat[(i3+(14*i1))],
                                  mu_bat[(i3+(14*(i2+(8*i0))))],
                                  sigma_bat[(i3+(14*(i2+(8*i0))))]);
              }
              a2 += prior_bat[(i2+(8*i0))] * (NORM_bat[(i2+(8*i0))] * exp((-0.5*a3)));
          }
          a1 += log(a2);
      }
      res_bat[i0] = a1;
  }
```

A.3.2 Mapping to Main Memory: MIL

The RAM mapping to MIL, presented in Section 4.3.2, produces a query script that explicitly materializes all intermediate results and produces results by processing whole tables (storing these intermediate arrays) at once using bulk operators.

In the following MIL example all variables reference either a constant value, or a Binary Association Table (BAT); BATs are tables with binary tuples. The BATs used in the example all associate an object identifier (type `oid`) with a value (of type `oid`, `int`, or, `dbl`), the object identifier column of the table is called the `head` column and the

---

1Note that the RAM system uses the type `dbl` instead of `double`.

value column is called the tail column. A full reference on the MIL query language can be found on the MonetDB website (http://monetdb.cwi.nl/). The example uses only a few operators:

- The `bat` operator retrieves a persistent BAT by its name.
- The `join` operator performs the relational join over two BATs: \( \text{join}(A, B) = \pi_{(A.\text{head}, B.\text{tail})}(A \bowtie_B B.\text{head}) \).
- The multiplex construct \( [f] \) maps a function over the natural-join result of two BATs: \( [f](A, B) = \pi_{(A.\text{head}, f(A.\text{tail}, B.\text{tail}))}(A \bowtie_B B.\text{head}) \).
- The aggregation construct \( \{g\} \) applies the aggregate function \( g \) over grouped values in a BAT. The groups are defined in a separate BAT, while a third BAT (for optimization reasons) provides the full listing of groups a-priori: \( \{g\}(G, A, C) = \pi_{(C.\text{head}, g(\pi_{A.\text{tail}}(\sigma_{G.\text{tail}}=C.\text{tail})(G \bowtie G.\text{head}=A.\text{head})))}(C) \).
- Finally, the proprietary RAM `milgrid` operator produces a BAT containing indices as defined in Section 4.3.2.

For readability, sections of MIL code that assign constants to variables and subsequently use those variables have been shortened, e.g. the fragment:

```mil
var t2 := lnx(42659760);
var t3 := lnx(8);
var t4 := lnx(1);
var t5 := lnx(0);
var t6 := milgrid(t2,t3,t4,t5);
```

has been replaced with:

```mil
var t6 := milgrid(42659760,8,1,0);
```

The example RAM expression results in the following MIL program:

```mil
# Application of P
var t1 := bat("priors_bat");
var t6 := milgrid(42659760,8,1,0);
var t11 := milgrid(1,32318,10560,0);
var t13 := [+](8,t11);
var t14 := [+]((t6,t13);
var t15 := [oid](t14);
var t16 := join(t15,t1);

# Application of NORM
var t17 := bat("NORM_bat");
var t22 := milgrid(42659760,8,1,0);
var t27 := milgrid(1,32318,10560,0);
var t29 := [+](8,t27);
var t30 := [+]((t29,t29);
var t31 := [oid](t30);
var t32 := join(t31,t17);

# The const array (optimized to a singleton constant)
var t33 := dbl(-0.5);
```

```mil
# Application of Img
var t34 := bat("query_bat");
var t39 := milgrid(341278080,14,1,0);
var t44 := milgrid(32318,1320,112,0);
var t47 := [+]((t39,t46);
var t48 := [oid](t47);
var t49 := join(t48,t34);

# Application of Mu
var t50 := bat("mu_bat");
var t55 := milgrid(341278080,14,1,0);
var t60 := milgrid(42659760,8,14,0);
var t65 := milgrid(1,32318,147840,0);
var t68 := [+]((8,t65);
var t69 := [+]((t60,t68);
var t70 := [+]((14,t69);
```
A.3.3 Mapping to a Pipeline: X100

The RAM mapping to X100, presented in Section 4.3.3, produces a query plan that streams all data through an operator pipeline. The MonetDB/X100 system uses relational algebra as its query language, a full reference on X100 can be found on the MonetDB website (http://monetdb.cwi.nl/) In the example a number of X100 operators are used that warrant clarification:

- The **BatMat** operator retrieves a persistent column by its name.
- The **AlignJoin** operator positionally joins multiple columns into a single multi-column table.
- The **Fetch1** operator performs a special case join operation where it is known that the there is a foreign-key relation: Each value in the first argument occurs exactly once in the second argument.
- The **FixedAggr** operator aggregates elements in a table, grouping its elements in fixed sized groups.
Finally, the proprietary RAM Array operator produces a column with array indices, as defined in Section 4.3.3.

```
AlignJoin( # Post processing: adding an explicit axis column to the result
  Project(
    Array([[i_0_2=dimension(32318)]],
      [i_0=dimension(10)_2]),
    Project( # Grouping and summation of the final summation
      FixedAggr(
        Project( # Grouping and summation of the second summation
          FixedAggr(
            Project( # P * (NORM * exp(...))
              AlignJoin(
                Project( # Application of P
                  Fetch1(
                    Project(
                      Array([[v_0=dimension(8),v_1=dimension(1320),v_2=dimension(32318)]],
                        [Idx_10=+(uidx(v_0),*(uidx('8'),uidx(v_2)))]),
                        Idx_10,
                        BatMat([v_12='priors_bat'])),
                      [v_8=v_12]),
                    Project( # NORM * exp(...)
                      AlignJoin(
                        Project( # Application of NORM
                          Fetch1(
                            Project(
                              Array([[v_0=dimension(8),v_1=dimension(1320),v_2=dimension(32318)]],
                                    [Idx_17=+(uidx(v_0),*(uidx('8'),uidx(v_2)))]),
                                    Idx_17,
                                    BatMat([v_19='NORM_bat'])),
                                  [v_15=v_19]),
                                Project( # 0.5 * sum(mahalanobis)
                                  AlignJoin(
                                    Project( # Grouping and summation of the Mahalanobis subexpression
                                      Array([[10_24=dimension(8),11_24=dimension(1320),
                                              12_24=dimension(32318)]],
                                             [v_23=dbl('-0.5')]),
                                         FixedAggr(
                                          Project( # Application of the Mahalanobis UDF
                                            AlignJoin(
                                              Project( # Application of Img
                                                Fetch1(
                                                  Project(
                                                    Array([[v_0=dimension(14),v_1=dimension(8),
                                                            v_2=dimension(1320),v_3=dimension(32318)]],
                                                          [Idx_30=+(uidx(v_0),*(uidx('14'),uidx(v_2)))]),
                                                          Idx_30,
                                                          BatMat([v_32='query_bat'])),
                                                        [v_28=v_32]),
                                                      Project( # Application of Mu
                                                        Fetch1(
                                                          Project(
                                                            Array([[v_0=dimension(14),v_1=dimension(8),
                                                                     v_2=dimension(1320),v_3=dimension(32318)]],
                                                                   [Idx_35=+(uidx(v_0),*(uidx('14'),*(uidx('8'),uidx(v_3))))]),
                                                                 Idx_35,
                                                                BatMat([v_37='mu_bat'])),
                                                              [v_33=v_37]),
                                                            Project( # Application of Sig
                                                              Fetch1(
                                                                Project(
                                                                 ...
```

```
Appendix A. A RAM Example: Sample Likelihood

Array([v_0=dimension(14), v_1=dimension(8), v_2=dimension(1320), v_3=dimension(32318))],
    [Idx_40=+(uidx(v_0), *(uidx('14'), +(uidx(v_1),
               *(uidx('8'), uidx(v_3))))]],
    Idx_40,
    BatMat([[v_42='sigma_bat']],
            [v_38=v_42]]),
    [v_26=mahalanobis(v_28, v_33, v_38)]),
    []),
    [v_25=sum(v_26),
        14]),
    [v_21=**(v_23, v_25)],
    [v_20=exp(v_21)]),
    [v_13=**(v_15, v_20)],
    [v_6=**(v_8, v_13)]),
    []),
    [v_5=sum(v_6),
        8]),
    [v_4=log(v_5)]),
    []),
    [v_3=sum(v_4),
        1320]),
    [v_1=v_3])}