Diplomarbeit in Informatik

In Memory Implementation of Vector Primitives

Tobias Müller

Aufgabensteller: Prof. Dr. Torsten Grust
Betreuer: Jeroen Weijers
Datum: 27. Februar 2013
4.1.3. Strings Implementation ........................................ 65
  4.1.3.1. Charset .............................................. 67
  4.1.3.2. Reference Management .................................. 67

4.2. Vector Primitives ............................................. 69
  4.2.1. Parallelization Categories ............................... 69
  4.2.2. Atomic Primitives ......................................... 70
    4.2.2.1. singletonDescr ...................................... 71
    4.2.2.2. lengthA .............................................. 72
  4.2.3. Fragmented Loop Primitives ............................. 73
    4.2.3.1. toDescr .............................................. 74
  4.2.4. Shared Variables Primitives ........................... 75
    4.2.4.1. vecSum ............................................... 76
    4.2.4.2. vecMin and vecMax .................................. 78
    4.2.4.3. Advanced for Loop ................................. 83
    4.2.4.4. distDesc ............................................ 85
    4.2.4.5. restrictVec ......................................... 87
  4.2.5. Sorting Primitives ...................................... 88
    4.2.5.1. Parallel Quicksort ................................. 89
    4.2.5.2. Comparison Interface ............................... 91
    4.2.5.3. unique .............................................. 92
  4.2.6. Pseudo Primitives ...................................... 94
    4.2.6.1. R[1-3] ............................................... 94
    4.2.6.2. constructLiteralTable/Value ...................... 94

4.3. Control Flow .................................................. 96
  4.3.1. Phase One, Initialisation ............................. 96
  4.3.2. Phase Two, Execution .................................. 97

5. Conclusion ....................................................... 100
  5.1. Future Work .................................................. 100
    5.1.1. Performance Improvements ............................. 100

Appendices .......................................................... 103
A. Example Execution ................................................ 103
B. Source Code for restrictVec .................................... 105
List of Listings .................................................... 109
List of Figures and Tables ......................................... 110
Bibliography ........................................................ 111
Abstract

DSH is a Haskell library for database supported program execution. It turns the database into a coprocessor for the Haskell runtime. Compilation of Haskell code for the database is a multi-step process. One of the intermediate languages is the vector algebra which is of interest to get a better understanding of the performance behaviour of DSH. We have implemented an in memory execution environment which executes vector algebra programs natively. The execution environment allows us to investigate the performance behaviour of each vector operation individually and without database context.
1. Introduction

DSH [1] is a Haskell library for database supported program execution. The library extends the responsibility of the database for not only being a data storage but also a coprocessor to the Haskell runtime. This database coprocessor deals with data-intensive computations and supports nested data-parallelism.

For performant execution of nested data-parallelism DSH uses loop lifting [2]. The currently developed new version of DSH uses the flattening transformation [3] which translates nested data-parallel constructs of Haskell (utilizing lists and tuples) into flat data-parallel constructs (vector operations).

![Diagram of DSH compilation chain]

Figure 1.1.: Major components of the new DSH compilation chain

Figure 1.1 provides a fragmentary overview of the compilation process in the new DSH version. As it is still under development, up to now no publications have been made.

Haskell program parts intended to be executed on the database coprocessor are translated into the flat kernel language using the flattening transformation. In the vectorisation step the program is translated into vector algebra. In this work we describe the in memory execution environment which executes vector algebra programs natively. This allows us to do research on the performance behaviour of vector algebra. Each vector operation can be analyzed individually and without database context.

Normally vector algebra is translated into relational algebra for execution on relational databases. Another compilation path leads to X100, a proprietary high-performance column-store DBMS.
1. Introduction

1.1. Outline

The goal of this work is to implement a performant in memory execution environment for vector algebra. Performant execution is of high interest for being able to make meaningful comparisons to execution in databases. It is aimed at doing a data parallel implementation which exploits multiprocessing computer systems (this includes variations like multi-core CPUs).

Using the execution environment to research the performance behaviour of the vector algebra remains as future work.

In Chapter 2 the vector algebra is specified. It provides the basis for the functionality to be implemented. Chapter 3 describes the main programming tools used for implementation. The implementation work itself is described in Chapter 4 and a discussion of future work can be found in Chapter 5.
The Vector algebra is an intermediate language within the DSH compilation chain. It consists of vectors and vector primitives manipulating these. Up to now there are no publications about the vector algebra.

Vector algebra programs are flat (vectors cannot contain vectors) but they are the transformation result of Haskell programs which in general contain nested constructs. Being a flat language vector algebra programs can be executed efficiently on actual hardware although they indirectly process nested data structures.

In the remainder of this section a short introduction to vectors and vector primitives is given. In Section 2.1.2 segments are introduced which are required for back-transformation of vectors into nested data structures. Sections 2.2, 2.3 and 2.4 contain a comprehensive specification of vectors, data types and vector primitives.

**Vectors** A vector in the context of the vector algebra is a multi-column structure, see example (2.1). This vector has two columns containing metadata \((d, p)\) and one column containing data values of integer type \((i_1)\). In general more variations for columns exist.

\[
\begin{pmatrix}
  d & p & i_1 \\
  1 & 1 & 10 \\
  1 & 2 & 20 \\
  1 & 3 & 30 \\
  1 & 4 & 40 \\
  1 & 5 & 50 \\
\end{pmatrix}
\]  

(2.1)

**Vector Primitives** A vector primitive executes certain computations on input vectors. It will always return one or more vectors as a result. Some forty different vector primitives exist.

**Example** Equation (2.2) illustrates how a vector primitive called \(compExpr2L\) appears; Equation (2.3) is a plausible notation of what this primitive along with the plus sign does: a vector addition.

\[
v_{res} = compExpr2L(+, v_1, v_2) \\
= v_1 +^\uparrow v_2
\]  

(2.2)  

(2.3)
2. Vector Algebra

For sample input vectors:

\( \mathbf{v}_1 \):

\[
\begin{pmatrix}
  d & p & i_1 \\
  1 & 1 & 10 \\
  1 & 2 & 20 \\
  1 & 3 & 30 \\
  1 & 4 & 40 \\
  1 & 5 & 50 \\
\end{pmatrix}
\]

\( \mathbf{v}_2 \):

\[
\begin{pmatrix}
  d & p & i_1 \\
  1 & 1 & 2 \\
  1 & 2 & 2 \\
  1 & 3 & 2 \\
  1 & 4 & 4 \\
  1 & 5 & 4 \\
\end{pmatrix}
\]

Equation (2.2) results in:

\( \mathbf{v}_{\text{res}} \):

\[
\begin{pmatrix}
  d & p & i_1 \\
  1 & 1 & 12 \\
  1 & 2 & 22 \\
  1 & 3 & 32 \\
  1 & 4 & 44 \\
  1 & 5 & 54 \\
\end{pmatrix}
\]

Please note that \texttt{compExpr2L} only processes the columns labeled with \( i_1 \). The other columns (\( d \) and \( p \)) do not contain data but meta information and stay unchanged in this case.
2. Vector Algebra

2.1. Segments

Conversion of Haskell programs into vector algebra programs is based on the flattening transformation introduced by Guy E. Blelloch [3]. In this section we provide the intuition of the transformation as it is not part of this work. Nonetheless having an intuition of how vectors correspond to lists and tuples is helpful for understanding the work in this thesis.

2.1.1. Lists and Tuples

Lists and tuples are the most prominent data types of Haskell for creating nested structures. Lists and tuples can be nested arbitrarily.

Elements of a certain list always share the same type (e.g. all elements are integers or all elements are lists) while elements of a certain tuple can be of different types. See Listing 2.1 for examples.

```
flat list containing integers:
[ 10 , 20 , 30 ]

nested lists:

flat tuple containing elements of different types:
( 10 , "abc" , -17.0 )

list nested in a tuple:
( 10 , "abc" , [10,20,30] )
```

Listing 2.1: Haskell lists and tuples examples

2.1.2. Segments in Vector Algebra

A key element of expressing nested data in flat data are segments [3]. Segments can be understood to be sub-vectors which correspond to sub-lists or sub-tuples.

Formally segments are specified within the $d$ columns of vectors. Two components of a vector are residing within the same segment if their corresponding segment identifiers are identical. Values of the segment column are ascending and the smallest possible value is 1.

Segments Examples Vector (2.4) is effectively not segmented. The only segment identifier is 1 and it is shared by both components.

\[
\begin{pmatrix}
  d & p & i_1 \\
  1 & 1 & 10 \\
  1 & 2 & 20 \\
\end{pmatrix}
\]  

(2.4)

Vector (2.5) consists of three segments. Segment 1 does not occur. Segments 2 and 4 have each a length of two and segment 5 has a length of one.
2. Vector Algebra

\[
\begin{pmatrix}
  d & p & i_1 \\
  2 & 1 & 10 \\
  2 & 2 & 20 \\
  4 & 3 & 30 \\
  4 & 4 & 40 \\
  5 & 5 & 50 \\
\end{pmatrix}
\] (2.5)

2.1.3. Conversion of Lists and Tuples

The flattening transformation transforms complete programs. In this section we demonstrate how lists and tuples are flattened.

**Flat List Example**  Input for this example is a flat list (2.6). After transformation this results in vector (2.11) having a single segment with a shared identifier of 1. Position and data values are directly copied into the vector.

\[ [10, 20, 30, 40] \] (2.6)

\[
\begin{pmatrix}
  d & p & i_1 \\
  1 & 1 & 10 \\
  1 & 2 & 20 \\
  1 & 3 & 30 \\
  1 & 4 & 40 \\
\end{pmatrix}
\] (2.7)

**Nested Lists Example**  A Haskell requirement is that all elements of a list share the same type. For List (2.8) the type is *list of list of integers*. Accordingly all inner lists are *lists of integers*: [10, 20], [30, 40] and []. In the flattening step the inner lists are merged into vector (2.10). For each inner list another segment id is used. The third inner list (being empty) does not show up in vector (2.10).

The outer list described by Vector (2.9) does not contain integers (nor does the outer list) but references to the three inner lists: the \( p \) column of vector (2.9) maps onto the \( d \) column of vector (2.10) (both columns are printed next to each other in this example). In the third component of Vector (2.9) the third inner list shows up (which has no counterpart in Vector (2.10)).

If input List (2.8) had a third layer of nesting, a third vector would be required. Nesting to any depth is possible.

\[ [ [10, 20], [30, 40], [] ] \] (2.8)

\[
\begin{pmatrix}
  d & p \\
  1 & 1 \\
  1 & 2 \\
  1 & 3 \\
\end{pmatrix}
\] (2.9)

\[
\begin{pmatrix}
  d & p & i_1 \\
  1 & 1 & 10 \\
  1 & 2 & 20 \\
  2 & 3 & 30 \\
  2 & 4 & 40 \\
\end{pmatrix}
\] (2.10)
2. Vector Algebra

**Flat Tuple Example**  Tuples may consist of different data types as demonstrated with Tuple (2.11). The translation result is Vector (2.12) which is strictly speaking a scalar and not a vector (more about this in the next section). It has length 1 and as many item columns as required for storing the tuple’s values.

\[
(10, "abc", -17.0) \quad (2.11)
\]

\[
\begin{pmatrix}
d & p & i_1 & i_2 & i_3 \\
1 & 10 & "abc" & -17.0
\end{pmatrix} \quad (2.12)
\]

**Nested Tuple Example**  This example includes heterogeneous nesting of lists and tuples. However it is similar to the previous “Nested Lists Example”. Input (2.13) contains now an additional tuple layer between inner and outer lists. Primitive values of that tuples now go into (2.14). The lists are treated the same as in the previous example.

If only based on Vectors (2.14) and (2.15), the backward transformation to Haskell constructs is ambiguous. The translator does not know if the original tuples were of type (string, integer) or (integer, string). This additional information has to be kept until the backward transformation is done.

\[
[("A", [10, 20]), ("B", [30, 40])] \quad (2.13)
\]

\[
\begin{pmatrix}
d & p & i_1 \\
1 & 1 & "A"
\end{pmatrix} \quad (2.14)
\]

\[
\begin{pmatrix}
d & p & i_1 \\
1 & 1 & 10 \\
1 & 2 & 20 \\
2 & 3 & 30 \\
2 & 4 & 40
\end{pmatrix} \quad (2.15)
\]
2. Vector Algebra

2.2. Vectors

In the vector algebra four vector types and one scalar type exist. But in perspective of this work with its focus on implementation we consider the scalar type as a vector, too. (see Section 4.1)

This perception makes five vector types in total, see Table 2.1. Unless stated otherwise, throughout this document the term vector will apply to all five types, including the scalar type (ScalarValue). The presented abbreviations are used within this document and in the source code.

<table>
<thead>
<tr>
<th>Vector type</th>
<th>Abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>DescriptionVector</td>
<td>DV</td>
</tr>
<tr>
<td>ValueVector</td>
<td>VV</td>
</tr>
<tr>
<td>ScalarValue</td>
<td>VS</td>
</tr>
<tr>
<td>PropagationVector</td>
<td>PV</td>
</tr>
<tr>
<td>RenameVector</td>
<td>RV</td>
</tr>
</tbody>
</table>

Table 2.1.: Vector types overview

**Notation**  For better readability vectors will generally be written with an additional header line (example: \(d, p, i_1\)). It is not part of the actual vector.

2.2.1. DescriptionVector

A DescriptionVector consists of exactly two columns. Its length is zero or more.

The first column, \(d\), describes a segment structure, called descriptor. Segments were already described in Section 2.1.2.

The second column, \(p\), serves as a row identifier, called position. Its numbering is dense (=no skipping of values), strictly ascending, starts with 1 and only contains positive integer values. It is implicitly given by the vector’s length.

**Examples**  Vectors (2.16) and (2.17) are two description vectors sharing the same length: 2. Their position columns are identical being determined by the vectors’ lengths. Vector (2.18) is another example. All three examples describe different segment structures.

\[
\begin{pmatrix}
  d & p \\
  1 & 1 \\
  1 & 2 \\
\end{pmatrix} \quad (2.16)
\]

\[
\begin{pmatrix}
  d & p \\
  2 & 1 \\
  2 & 2 \\
\end{pmatrix} \quad (2.17)
\]
2. Vector Algebra

\[
\begin{pmatrix}
 d & p \\
 1 & 1 \\
 1 & 2 \\
 1 & 3 \\
 4 & 4 \\
 5 & 5 \\
\end{pmatrix}
\]  
(2.18)

2.2.2. ValueVector

*ValueVectors* are *DescriptionVectors* with an additional list of *item columns* \(i_1...i_n\) (dense numbering). Each item column has a type. Values inside of that column are all of that particular type, for example string or double. The available types will be specified in Section 2.3.

If a ValueVector has zero item columns it is actually a DescriptionVector.

**Examples**  Vector (2.19) contains one item column with boolean values. Vector (2.20) contains two item columns with different data types (integer + string).

\[
\begin{pmatrix}
 d & p & i_1 \\
 2 & 1 & t \\
 2 & 2 & t \\
 4 & 3 & f \\
 4 & 4 & f \\
 5 & 5 & t \\
\end{pmatrix}
\]  
(2.19)

\[
\begin{pmatrix}
 d & p & i_1 & i_2 \\
 1 & 1 & 10 & "This" \\
 1 & 2 & 20 & "is" \\
 1 & 3 & 30 & "an" \\
 2 & 4 & 40 & "example" \\
 3 & 5 & 50 & "vector." \\
\end{pmatrix}
\]  
(2.20)

2.2.3. ScalarValue

*ScalarValues* are considered as atomic values on conceptual layer. They are not of variable length like a ValueVector. They contain exactly one descriptor value, one position value and one or more data elements (elements here are columns in a value vector).

**Examples**  ScalarValues having one (2.21) and three (2.22) item columns. In every case (not only in these examples) the position value is 1.

\[
\begin{pmatrix}
 d & p & i_1 \\
 1 & 1 & t \\
\end{pmatrix}
\]  
(2.21)

\[
\begin{pmatrix}
 d & p & i_1 & i_2 & i_3 \\
 2 & 1 & f & 42 & 3.14 \\
\end{pmatrix}
\]  
(2.22)
2. Vector Algebra

2.2.4. PropagationVector

A *PropagationVector* consists of exactly two columns. The first is called *position old* \((p_o)\), the second *position new* \((p_n)\). As the naming suggests, they are used for recording position updates. For example, when sorting a ValueVector its components will be reordered. A PropagationVector can describe these changes.

Both columns contain positive integers bigger than zero (just like position columns). Values in \(p_n\) have to be ascending.

**Example**  Let Vector (2.23) be an unsorted ValueVector containing integer data and Vector (2.24) the numerically sorted counterpart. Then PropagationVector (2.25) describes the shift of positions.

\[
\begin{pmatrix}
d & p & i_1 \\
1 & 1 & 20 \\
1 & 2 & 50 \\
1 & 3 & 30 \\
1 & 4 & 10 \\
1 & 5 & 40 \\
\end{pmatrix}
\]  \hspace{1cm}  (2.23)

\[
\begin{pmatrix}
d & p & i_1 \\
1 & 1 & 10 \\
1 & 2 & 20 \\
1 & 3 & 30 \\
1 & 4 & 40 \\
1 & 5 & 50 \\
\end{pmatrix}
\]  \hspace{1cm}  (2.24)

\[
\begin{pmatrix}
p_o & p_n \\
4 & 1 \\
1 & 2 \\
3 & 3 \\
5 & 4 \\
2 & 5 \\
\end{pmatrix}
\]  \hspace{1cm}  (2.25)

2.2.5. RenameVector

A *RenameVector* is similar to a PropagationVector. It has more restrictive conditions on ordering: \(p_o\) is ascending and \(p_n\) is strictly ascending.

Thus, every RenameVector is a PropagationVector - but not the other way round.

**Example**  Vector (2.26) is a Rename- and a PropagationVector. Vector (2.25) (see last section) is a PropagationVector but not a RenameVector. It violates the condition of an ascending \(p_o\) column.
2. Vector Algebra

\[
\begin{pmatrix}
p_0 & p_m \\
1 & 2 \\
2 & 3 \\
3 & 4 \\
4 & 5 \\
5 & 6 \\
\end{pmatrix}
\]

(2.26)
2. Vector Algebra

2.3. Data Types

The vector algebra supports six data type primitives. They are listed in Table 2.2.

<table>
<thead>
<tr>
<th>Type name</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>signed integer values</td>
<td>+10, -20, +30, ...</td>
</tr>
<tr>
<td>nat</td>
<td>unsigned integer values</td>
<td>0, 1, 2, 3, ...</td>
</tr>
<tr>
<td>double</td>
<td>floating point values</td>
<td>+1.0, -1.1, +1.2, ...</td>
</tr>
<tr>
<td>string</td>
<td>unicode format</td>
<td>&quot;abcd&quot;, &quot;xyz&quot;, ...</td>
</tr>
<tr>
<td>bool</td>
<td>logical values</td>
<td>true, false</td>
</tr>
<tr>
<td>unit</td>
<td>has cardinality = 1</td>
<td>()</td>
</tr>
</tbody>
</table>

Table 2.2.: Data types

The type unit has cardinality one, which means it does not actually carry information. Every value of type unit is the same, written as ()

The integer types (int and nat) are conceptually of unlimited length in Haskell. However the execution environment will only support 64bit values.

Vector 2.27 is an example ValueVector which has columns for each of the six types: int, nat, double, bool, string, unit.

\[
\begin{pmatrix}
  d & p & i_1 & i_2 & i_3 & i_4 & i_5 & i_6 \\
  1 & 1 & +10 & 1 & 1.0 & t & "abc" & () \\
  1 & 2 & -20 & 2 & 2.0 & t & "def" & () \\
  1 & 3 & +30 & 3 & 3.0 & f & "ghi" & () \\
  1 & 4 & -40 & 4 & 4.0 & t & "jkl" & () \\
  1 & 5 & +50 & 5 & 5.0 & f & "mno" & ()
\end{pmatrix}
\]

(2.27)
2. Vector Algebra

2.4. Vector Primitives

A vector primitive is a function in the mathematical sense, i.e. it has no side effects. As argument types vectors and non-vectors do occur. The arguments are immutable. Each vector primitive returns one or more vectors.

In the next section the complete specifications for all 47 primitives are provided. They are in alphabetical order, making it easy to be used as a reference.

Example: compExpr2L (binary, partitioned loop)

In the header of each primitive a classification is provided:

1. Arity: In the example it is a binary operator which has two input vectors. There are nullary (zero), unary (one), binary (two) and ternary (three) operators. Only vector arguments count to the arity.

2. Parallelization type: There are atomic, partitioned loop, shared variables, sorting and pseudo primitives. For implementation the challenging categories are shared variables and sorting.

2.4.1. Specifications

append (binary, shared variables)

\[
(VV_{\text{res}}, RV_1, RV_2) = \text{append}(VV_1, VV_2)
\]

Append takes two vectors (VV_1, VV_2) and merges them into one (VV_{res}). Item columns and descriptors do not get altered. The result vector’s length is the sum of both input vectors’ lengths. It requires that both input vectors have the same number of item columns and the same types.

The merging itself is based on segments. If a certain segment id only exists in one of the input vectors, the according vector components get copied into the result vector as one contiguous piece. If both input vectors contain a segment with the same id, the two segments get merged into one. Components of VV_1 come first, then all components of VV_2.

Additionally, two rename vectors are computed. RV_1 maps the old and new positions of components from VV_1. RV_2 does the same on VV_2.

Example Computation example with one item column of integer type.

\[
VV_1 : \begin{pmatrix}
  d & p & i_1 \\
  1 & 1 & 10 \\
  1 & 2 & 20 \\
  2 & 3 & 30 \\
  3 & 4 & 40 \\
  3 & 5 & 50 \\
\end{pmatrix}
\]
2. Vector Algebra

$$VV_2:\begin{pmatrix}
    d & p & i_1 \\
    1 & 1 & 15 \\
    1 & 2 & 25 \\
    2 & 3 & 35 \\
    2 & 4 & 45
\end{pmatrix}$$

$$VV_{res}:\begin{pmatrix}
    d & p & i_1 \\
    1 & 1 & 10 \\
    1 & 2 & 20 \\
    1 & 3 & 15 \\
    1 & 4 & 25 \\
    2 & 5 & 30 \\
    2 & 6 & 35 \\
    2 & 7 & 45 \\
    3 & 8 & 40 \\
    3 & 9 & 50
\end{pmatrix}$$

$$RV_1:\begin{pmatrix}
    p_o & p_n \\
    1 & 1 \\
    2 & 2 \\
    3 & 5 \\
    4 & 8 \\
    5 & 9
\end{pmatrix}$$

$$RV_2:\begin{pmatrix}
    p_o & p_n \\
    1 & 3 \\
    2 & 4 \\
    3 & 6 \\
    4 & 7
\end{pmatrix}$$

**combineVec (ternary, shared variables)**

$$(VV_{res}, RV_1, RV_2) = append(VV_c, VV_1, VV_2)$$

CombineVec does a conditional merge as instructed by $VV_c$ (c is short for conditions). $VV_1$ and $VV_2$ are the data sources which have to share the types of their item columns. $VV_{res}$ is where the data is copied to.

Merging rules are based on the contents of $VV_c$. The vector $VV_c$ contains exactly one item column filled with booleans. The length of that column is equal to the length of $VV_{res}$. For every component $x$ of $VV_{res}$ that is to be written, the algorithm inspects the corresponding boolean value in $VV_c$. If that value is true, the copy source for component $x$ is the next available component in $VV_1$; else it is the next available component of $VV_2$.

$R_1$ and $R_2$ reference old and new positions for each vector component.
Example  Computation example with one item column of integer type.

$$VV_{c} : \begin{pmatrix} d & p & i_{1} \\ 1 & 1 & t \\ 1 & 2 & f \\ 1 & 3 & t \\ 1 & 4 & f \\ 1 & 5 & t \end{pmatrix}$$

$$VV_{1} : \begin{pmatrix} d & p & i_{1} \\ 1 & 1 & 10 \\ 1 & 2 & 20 \\ 2 & 3 & 30 \end{pmatrix}$$

$$VV_{2} : \begin{pmatrix} d & p & i_{1} \\ 1 & 1 & 15 \\ 2 & 2 & 25 \end{pmatrix}$$

$$VV_{res} : \begin{pmatrix} d & p & i_{1} \\ 1 & 1 & 10 \\ 1 & 2 & 15 \\ 1 & 3 & 20 \\ 2 & 4 & 25 \\ 2 & 5 & 30 \end{pmatrix}$$

$$RV_{1} : \begin{pmatrix} p_{o} & p_{n} \\ 1 & 1 \\ 2 & 3 \\ 3 & 5 \end{pmatrix}$$

$$RV_{2} : \begin{pmatrix} p_{o} & p_{n} \\ 1 & 2 \\ 2 & 4 \end{pmatrix}$$

compExpr2 (binary, atomic)

$$VS_{res} = \text{compExpr2} (\text{oper}, VS_{1}, VS_{2}) = VS_{1} \circ VS_{2}$$

CompExpr2 (short form of: compute expression; 2 arguments) evaluates an arithmetic term. It has a non-vector input which is the binary operator type like plus (⊕) or minus (⊖).

See Table 2.3 for available binary operators and the corresponding data types they can be used with. Both input vectors are required to have one column which has to be the same data type for both.
2. Vector Algebra

<table>
<thead>
<tr>
<th></th>
<th>int</th>
<th>nat</th>
<th>double</th>
<th>bool</th>
<th>string</th>
<th>unit</th>
<th>result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>as input</td>
</tr>
<tr>
<td>sub</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>as input</td>
</tr>
<tr>
<td>div</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>as input</td>
</tr>
<tr>
<td>mul</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>as input</td>
</tr>
<tr>
<td>mod</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>as input</td>
</tr>
<tr>
<td>eq</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>bool</td>
</tr>
<tr>
<td>gt</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>bool</td>
</tr>
<tr>
<td>gte</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>bool</td>
</tr>
<tr>
<td>lt</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>bool</td>
</tr>
<tr>
<td>lte</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>bool</td>
</tr>
<tr>
<td>conj</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>bool</td>
</tr>
<tr>
<td>disj</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>bool</td>
</tr>
</tbody>
</table>

Table 2.3.: Operator table for compExpr2 and compExpr2L

With comparison operators, string values get compared lexicographically. An unit value can only be compared to itself. Logical consequence is that gt and lt always evaluate to false, the remaining ones always evaluate to true.

The result descriptor is copied from VS₁.

**Example**  Plus operation on integer types.

*operator* : ⊕

VS₁ :

\[
\begin{pmatrix}
  d & p & i_1 \\
  1 & 1 & 20
\end{pmatrix}
\]

VS₂ :

\[
\begin{pmatrix}
  d & p & i_1 \\
  1 & 1 & 22
\end{pmatrix}
\]

VSₚᵣₑₛ :

\[
\begin{pmatrix}
  d & p & i_1 \\
  1 & 1 & 42
\end{pmatrix}
\]

**compExpr2L (binary, partitioned loop)**

\[
VV_{res} = compExpr2L(oper, VV₁, VV₂)
\]

\[
= VV₁ \bigcirc VV₂
\]

*CompExpr2L* is the lifted version of compExpr2. Operators are all the same (such as a plus operator ⊕), but they are applied to all components of a vector (lifted). This is indicated by an arrow next to the operator symbol (⊕↑).

The descriptor for VVₚᵣₑₛ is copied from VV₁.
Example  Plus operation on integer types.

\[ \text{operator} : \oplus \]

\[
VV_1 : \begin{pmatrix} d & p & i_1 \\ 1 & 1 & 1 \\ 1 & 1 & 2 \\ 2 & 1 & 3 \\ 2 & 1 & 4 \end{pmatrix}
\]

\[
VV_2 : \begin{pmatrix} d & p & i_1 \\ 1 & 1 & 1 \\ 1 & 1 & 0 \\ 2 & 1 & 0 \\ 2 & 1 & 2 \end{pmatrix}
\]

\[
VV_{\text{res}} : \begin{pmatrix} d & p & i_1 \\ 1 & 1 & 2 \\ 1 & 1 & 2 \\ 2 & 1 & 3 \\ 2 & 1 & 6 \end{pmatrix}
\]

constructLiteralValue (nullary, pseudo)

\[ VS_{\text{res}} = \text{constructLiteralValue}("\text{scalar fragments}\") \]

ConstructLiteralValue creates a vector by assembling its fragments. It is used to copy actual data into vector algebra space.

constructLiteralTable (nullary, pseudo)

\[ VV_{\text{res}} = \text{constructLiteralTable}("\text{vector fragments}\") \]

ConstructLiteralTable is the lifted version of constructLiteralValue.

descToRename (unary, partitioned loop)

\[ RV_{\text{res}} = \text{descToRename}(DV_{\text{in}}) \]

DescToRename converts a DescriptionVector into a RenameVector: \( p \) becomes \( p_o \) and \( d \) becomes \( p_n \).
2. Vector Algebra

Example

\[ DV_{in} : \begin{pmatrix} d & p \\ 1 & 1 \\ 1 & 2 \\ 2 & 3 \\ 2 & 4 \end{pmatrix} \]

\[ RV_{res} : \begin{pmatrix} p_o & p_n \\ 1 & 1 \\ 2 & 1 \\ 3 & 2 \\ 4 & 2 \end{pmatrix} \]

distDesc (binary, shared variables)

\[(VV_{res}, PV_{res}) = distDesc(VV_{data}, DV_{len})\]

\(DistDesc\) copies the contiguous content of \(VV_{data}\) into \(VV_{res}\). This is done as many times as the length \(l\) of \(DV_{len}\) indicates. The current value of \(l\) is written into the descriptor column of \(VV_{res}\).

\(PV_{res}\) references for each copied component the original \(p_o\) and the final \(p_n\) positions.

Example

\[ VV_{data} : \begin{pmatrix} d & p & i_1 \\ 1 & 1 & 10 \\ 1 & 2 & 20 \\ 1 & 3 & 30 \end{pmatrix} \]

\[ DV_{len} : \begin{pmatrix} d & p \\ 1 & 1 \\ 1 & 2 \\ 1 & 3 \end{pmatrix} \]

\[ VV_{res} : \begin{pmatrix} d & p & i_1 \\ 1 & 1 & 10 \\ 1 & 2 & 20 \\ 1 & 3 & 30 \\ 2 & 4 & 10 \\ 2 & 5 & 20 \\ 2 & 6 & 30 \\ 3 & 7 & 10 \\ 3 & 8 & 20 \\ 3 & 9 & 30 \end{pmatrix} \]
2. Vector Algebra

$PV_{res}$:

$$
\begin{pmatrix}
1 & 1 \\
2 & 2 \\
3 & 3 \\
1 & 4 \\
2 & 5 \\
3 & 6 \\
1 & 7 \\
2 & 8 \\
3 & 9 \\
\end{pmatrix}
$$

$distLift$ (binary, partitioned loop)

$$(VV_{res}, PV_{res}) = distLift(VV_{data}, DV_{pos})$$

$DistLift$ creates a result vector $VV_{res}$ with a copy of the descriptor column $p$ of $DV_{pos}$. The descriptor column $p$ is interpreted as a reference to the positions in $VV_{data}$. The according components of $VV_{data}$ get copied to $VV_{res}$.

$PV_{res}$ references the corresponding positions of $VV_{data}$ ($p_o$) and $VV_{res}$ ($p_n$).

Example

$VV_{data}$:

$$
\begin{pmatrix}
d & p & i_1 \\
1 & 1 & 10 \\
1 & 2 & 20 \\
1 & 3 & 30 \\
\end{pmatrix}
$$

$DV_{pos}$:

$$
\begin{pmatrix}
d & p \\
1 & 1 \\
1 & 2 \\
2 & 3 \\
3 & 4 \\
\end{pmatrix}
$$

$VV_{res}$:

$$
\begin{pmatrix}
d & p & i_1 \\
1 & 1 & 10 \\
1 & 2 & 10 \\
2 & 3 & 20 \\
3 & 4 & 30 \\
\end{pmatrix}
$$

$PV_{res}$:

$$
\begin{pmatrix}
p_o & p_n \\
1 & 1 \\
1 & 2 \\
2 & 3 \\
3 & 4 \\
\end{pmatrix}
$$
2. Vector Algebra

**distPrim (binary, partitioned loop)**

\[(V V_{res}, P V_{res}) = distPrim(V S_{data}, D V_{len})\]

*DistPrim* replicates the component found in *V S*\(_{data}\) as many times as the length of *D V*\(_{len}\) indicates.

Additionally a PropagationVector *P V*\(_{res}\) is assembled. It references the original position (=1) of every component to its new positions in *V V*\(_{res}\).

**Example**

\[V S_{data}: \begin{pmatrix} d & p & i_1 & i_2 \\ 1 & 1 & 10 & t \end{pmatrix}\]

\[D V_{len}: \begin{pmatrix} d & p \\ 1 & 1 \\ 1 & 2 \\ 2 & 3 \\ 2 & 4 \end{pmatrix}\]

\[V V_{res}: \begin{pmatrix} d & p & i_1 & i_2 \\ 1 & 1 & 10 & t \\ 1 & 2 & 10 & t \\ 2 & 3 & 10 & t \\ 2 & 4 & 10 & t \end{pmatrix}\]

\[P V_{res}: \begin{pmatrix} p_o & p_n \\ 1 & 1 \\ 1 & 2 \\ 1 & 3 \\ 1 & 4 \end{pmatrix}\]

**falsePositions (unary, shared variables)**

\[V V_{res} = falsePositions(V V_{in})\]

*FalsePositions* filters components having a boolean *true* in its first item column of *V V*\(_{in}\). No other columns are allowed.

The remaining components form a new vector *V V*\(_{res}\) which is shorter than *V V*\(_{in}\) if filtering happens. An additional row numbering is created for *V V*\(_{in}\) (see in example: *p*\(^{'}\)) which is dense and ascending like a normal position column but starts over for every segment.

For each component that passes the filter its descriptor value and *p*\(^{'}\) value are copied into *V V*\(_{res}\) where *p*\(^{'}\) becomes *i*\(_{1}\). Type of *i*\(_{1}\) is integer.
2. Vector Algebra

**Example** The additional \( p' \) column is not part of the input vector. Filtered components are printed on black background.

\[
VV_{\text{in}} : \begin{pmatrix}
\begin{array}{ccc}
d & p & i_1 \\
1 & 1 & t \\
1 & 2 & f \\
1 & 3 & f \\
2 & 4 & t \\
2 & 5 & t \\
3 & 6 & f \\
5 & 7 & t \\
5 & 8 & f \\
\end{array}
\end{pmatrix}
\]

\[
VV_{\text{res}} : \begin{pmatrix}
\begin{array}{ccc}
d & p & i_1 \\
1 & 1 & 2 \\
1 & 2 & 3 \\
3 & 3 & 1 \\
5 & 4 & 2 \\
\end{array}
\end{pmatrix}
\]

**groupBy (binary, sorting)**

\[(DV_{\text{res}}, VV_{\text{res}}, PV_{\text{res}}) = groupBy(VV_c, VV_{\text{data}})\]

*groupBy* does the same lifted sorting as sortWith (the sorting conditions are in \( VV_c \) and the data is in \( VV_{\text{data}} \)). Additionally it computes a new descriptor column for \( VV_{\text{res}} \). \( DV_{\text{res}} \) describes how old and new descriptor are related.

Computation of the new descriptor is based on grouping. When sorting a certain segment in \( VV_c \), all components having identical data values (without considering the position column) will be considered as one group. The group numbering starts by 1 for the group with the smallest data values and the value increments for each new group.

**Example** In segment 1 of \( VV_c \) the first group consists of components sharing a *false* value (for \( p = 2 \)). Accordingly in the helper column \( d' \) a 1 is inserted. Group 2 consists of *true* values (\( p = 1 \) and \( p = 3 \)) and will achieve a 2. Group 3 again consists of *true* values but resides in the next segment. Because of this it becomes a separate group with id=3.

After reordering \( d' \) according to the sorting result it will look like column \( d \) of \( VV_{\text{res}} \).

\( d' \), \( p' \) and \( d_{\text{old}} \) are not part of the vectors and only inserted for better understanding. (\( p' \) again is the new order after sorting - as explained in sortWith)
### 2. Vector Algebra

#### $VV_c$:

$$
\begin{pmatrix}
2 & 1 & 1 & t \\
1 & 1 & 2 & f \\
2 & 1 & 3 & t \\
3 & 2 & 4 & t \\
3 & 2 & 5 & t \\
4 & 3 & 6 & f \\
6 & 4 & 7 & t \\
5 & 4 & 8 & f \\
6 & 4 & 9 & t \\
6 & 4 & 10 & t \\
\end{pmatrix}
$$

#### $VV_{\text{data}}$:

$$
\begin{pmatrix}
1 & 1 & i_1 \\
1 & 2 & 10 \\
1 & 3 & 20 \\
2 & 4 & 30 \\
2 & 5 & 40 \\
3 & 6 & 50 \\
4 & 7 & 60 \\
4 & 8 & 70 \\
4 & 9 & 80 \\
4 & 10 & 90 \\
\end{pmatrix}
$$

#### $DV_{\text{res}}$:

$$
\begin{pmatrix}
1 & 1 \\
1 & 2 \\
2 & 3 \\
3 & 4 \\
4 & 5 \\
4 & 6 \\
\end{pmatrix}
$$

#### $VV_{\text{res}}$:

$$
\begin{pmatrix}
1 & 1 & i_1 \\
1 & 2 & 20 \\
1 & 2 & 10 \\
1 & 2 & 30 \\
2 & 3 & 40 \\
2 & 3 & 50 \\
3 & 4 & 60 \\
4 & 5 & 80 \\
4 & 6 & 70 \\
4 & 6 & 90 \\
4 & 6 & 10 & 100 \\
\end{pmatrix}
$$
2. Vector Algebra

\[ PV_{\text{res}} : \]
\[
\begin{pmatrix}
  p_0 & p_n \\
  2 & 1 \\
  1 & 2 \\
  3 & 3 \\
  4 & 4 \\
  5 & 5 \\
  6 & 6 \\
  8 & 7 \\
  7 & 8 \\
  9 & 9 \\
  10 & 10 \\
\end{pmatrix}
\]

\text{intToDoubleA (unary, atomic)}

\[ VS_{\text{res}} = \text{intToDoubleA}(VS_{\text{data}}) \]

\text{IntToDoubleA} converts the column type from integer to double. The input vector must have exactly one column.

\textbf{Example}

\[ VS_{\text{data}} : \]
\[
\begin{pmatrix}
  d & p & i_1 \\
  1 & 1 & 42 \\
\end{pmatrix}
\]

\[ VS_{\text{res}} : \]
\[
\begin{pmatrix}
  d & p & i_1 \\
  1 & 1 & 42.0 \\
\end{pmatrix}
\]

\text{intToDoubleL (unary, partitioned loop)}

\[ VV_{\text{res}} = \text{intToDoubleL}(VV_{\text{data}}) \]

Lifted version of \text{intToDoubleA}.

\textbf{Example}

\[ VV_{\text{data}} : \]
\[
\begin{pmatrix}
  d & p & i_1 \\
  2 & 1 & 42 \\
  2 & 2 & 4 \\
  5 & 3 & 2 \\
\end{pmatrix}
\]

\[ VV_{\text{res}} : \]
\[
\begin{pmatrix}
  d & p & i_1 \\
  2 & 1 & 42.0 \\
  2 & 2 & 4.0 \\
  5 & 3 & 2.0 \\
\end{pmatrix}
\]
2. Vector Algebra

**lengthA (unary, atomic)**

\[ \mathbf{VS}_{\text{res}} = \text{lengthA}(\mathbf{DV}_{\text{len}}) \]

*LengthA* returns the input vector’s length (data type is integer).

Its parallelization category depends on how \( \mathbf{DV}_{\text{len}} \) is held in memory. In context of this work each vector has a dedicated length variable which makes this primitive a member of the *atomic* category.

**Example**

\[
\begin{pmatrix}
  d & p \\
  2 & 1 \\
  4 & 2 \\
  8 & 3 \\
  9 & 4 \\
  9 & 5 \\
  9 & 6 \\
\end{pmatrix}
\]

\[
\mathbf{VS}_{\text{res}}:
\begin{pmatrix}
  d & p & i_1 \\
  1 & 1 & 6 \\
\end{pmatrix}
\]

**lengthSeg (unary, shared variables)**

\[ \mathbf{VV}_{\text{res}} = \text{lengthSeg}(\mathbf{DV}_{\text{len}}, \mathbf{DV}_{\text{data}}) \]

*LengthSeg* is a lifted version of *lengthA* and computes the individual length per segment.

The contents of \( \mathbf{DV}_{\text{len}} \) are completely ignored. Only its length \( l \) is important (\( l = 6 \) for the example below). Variable \( l \) states that for all segments \( s \) of \( \mathbf{DV}_{\text{data}} \) with \( 1 \leq s \leq l \) the length is to be computed. If certain segments in range are empty their computed length will be zero.

The result column is of integer type.

**Example**

\[
\begin{pmatrix}
  d & p \\
  1 & 1 \\
  2 & 2 \\
  3 & 3 \\
  4 & 4 \\
  5 & 5 \\
  7 & 6 \\
\end{pmatrix}
\]
2. Vector Algebra

\[ DV_{\text{data}} : \begin{pmatrix} d & p \\ 1 & 1 \\ 1 & 2 \\ 3 & 3 \\ 4 & 4 \\ 4 & 5 \\ 4 & 6 \end{pmatrix} \]

\[ VV_{\text{res}} : \begin{pmatrix} d & p & i_1 \\ 1 & 1 & 2 \\ 2 & 2 & 0 \\ 3 & 3 & 1 \\ 4 & 4 & 3 \\ 5 & 5 & 0 \\ 6 & 6 & 0 \end{pmatrix} \]

notPrim (unary, atomic)

\[ VS_{\text{res}} = \text{notPrim}(VS_{\text{data}}) \]

NotPrim computes the logical negation of a boolean input value. Multiple columns are not allowed.

Example

\[ VS_{\text{data}} : \begin{pmatrix} d & p & i_1 \\ 1 & 1 & t \\ \end{pmatrix} \]

\[ VS_{\text{res}} : \begin{pmatrix} d & p & i_1 \\ 1 & 1 & f \end{pmatrix} \]

notVec (unary, partitioned loop)

\[ VV_{\text{res}} = \text{notVec}(VV_{\text{data}}) \]

NotVec is the lifted version of notPrim.

Example

\[ VV_{\text{data}} : \begin{pmatrix} d & p & i_1 \\ 1 & 1 & t \\ 3 & 2 & t \\ 4 & 3 & f \\ 5 & 4 & f \\ 5 & 5 & t \end{pmatrix} \]
2. Vector Algebra

\[ VV_{\text{res}} : \begin{pmatrix}
  d & p & i_1 \\
  1 & 1 & f \\
  3 & 2 & f \\
  4 & 3 & t \\
  5 & 4 & t \\
  5 & 5 & f \\
\end{pmatrix} \]

pairA (binary, atomic)

\[ VS_{\text{res}} = \text{pairA}(VS_1, VS_2) \]

PairA merges two ScalarValues by their columns. In the first (conceptual) step a copy of \( VS_1 \) is created. Then all columns of \( VS_2 \) are appended in the same order as they originally were in \( VS_2 \). The descriptor is copied from \( VS_1 \).

Example

\[ VS_1 : \begin{pmatrix}
  d & p & i_1 & i_2 \\
  1 & 1 & 42 & () \\
\end{pmatrix} \]

\[ VS_2 : \begin{pmatrix}
  d & p & i_1 \\
  1 & 1 & 9.9 \\
\end{pmatrix} \]

\[ VS_{\text{res}} : \begin{pmatrix}
  d & p & i_1 & i_2 & i_3 \\
  1 & 1 & 42 & () & 9.9 \\
\end{pmatrix} \]

pairL (binary, partitioned loop)

\[ VV_{\text{res}} = \text{pairL}(VV_1, VV_2) \]

PairL is the lifted version of pairA. If \( VV_1 \) and \( VV_2 \) have different lengths, the shorter one is selected as result length. Truncation - if necessary - happens at the end of the longer input vector. The descriptor is copied from \( VV_1 \).

Example  Component 4 of \( VV_2 \) is dropped because \( VV_1 \) only has length 3.

\[ VV_1 : \begin{pmatrix}
  d & p & i_1 & i_2 \\
  1 & 1 & 42 & () \\
  2 & 2 & 43 & () \\
  4 & 3 & 45 & () \\
\end{pmatrix} \]
2. Vector Algebra

\[
VV_2 :
\begin{pmatrix}
d & p & i_1 \\
1 & 1 & 9.9 \\
2 & 2 & 9.8 \\
3 & 3 & 9.0 \\
4 & 4 & 8.9 \\
\end{pmatrix}
\]

\[
VV_{\text{res}} :
\begin{pmatrix}
d & p & i_1 & i_2 & i_3 \\
1 & 1 & 42 & () & 9.9 \\
2 & 1 & 43 & () & 9.8 \\
4 & 1 & 45 & () & 9.0 \\
\end{pmatrix}
\]

**projectA (unary, atomic)**

\[
VS_{\text{res}} = \text{projectA}(VS_{\text{data}}, \text{ProjectionList})
\]

`ProjectA` is distantly related to the projection operator \( \pi \) in Relational Algebra. Columns can get duplicated, reordered or dropped ("projected away"). The non-vector argument `ProjectionList` is a list of column ids. The corresponding columns in \( VS_{\text{data}} \) are copied into the result vector. The ordering is kept.

**Example**

`ProjectionList` : \([4, 4, 1] \]

\[
VS_{\text{data}} :
\begin{pmatrix}
d & p & i_1 & i_2 & i_3 & i_4 \\
1 & 1 & 42 & () & 17.0 & t \\
\end{pmatrix}
\]

\[
VS_{\text{res}} :
\begin{pmatrix}
d & p & i_1 & i_2 & i_3 \\
1 & 1 & t & t & 42 \\
\end{pmatrix}
\]

**projectL (unary, partitioned loop)**

\[
VV_{\text{res}} = \text{projectL}(VV_{\text{data}}, \text{ProjectionList})
\]

`ProjectL` is the lifted version of `projectA`.

**Example**

`ProjectionList` : \([4, 4, 1] \]

\[
VV_{\text{data}} :
\begin{pmatrix}
d & p & i_1 & i_2 & i_3 & i_4 \\
3 & 1 & 42 & () & 17.0 & t \\
6 & 2 & 39 & () & 18.2 & f \\
7 & 3 & 38 & () & 19.0 & t \\
7 & 4 & 37 & () & 19.1 & t \\
\end{pmatrix}
\]
2. Vector Algebra

\[ VV_{\text{res}} : \begin{pmatrix} d & p & i_1 & i_2 & i_3 \\ 1 & 1 & t & t & 42 \\ 2 & 2 & f & f & 39 \\ 3 & 3 & t & t & 38 \\ 4 & 4 & t & t & 37 \end{pmatrix} \]

\text{propFilter (binary, shared variables)}

\[(VV_{\text{res}}, RV_{\text{res}}) = \text{propFilter}(RV_{\text{in}}, VV_{\text{data}})\]

PropFilter together with propRename and propReorder work on layer of segments. Together they support segment renaming, filtering and reordering. PropFilter is capable of doing segment renaming and filtering.

Renaming is the replacement of segment ids in \(VV_{\text{data}}\) with new segment ids provided by \(RV_{\text{in}}\). More details on segment renaming can be found in the section of propRename.

Additionally propFilter can filter complete segments of the input vector \(VV_{\text{data}}\). Filtering of a certain segment \(s\) happens if \(s\) it is not listed in the \(p_o\) column of \(RV_{\text{in}}\).

The listed segments \(s\) get copied into the result vector \(VV_{\text{res}}\) (and depending on the \(p_n\) column of \(RV_{\text{in}}\) are renamed).

\(RV_{\text{res}}\) is another result vector and references old and new positions for the copied components.

\textbf{Example} Segment 2 is filtered (data value = 30). However in \(VV_{\text{res}}\) a new segment 2 shows up; it is the old segment 3 after renaming.

\[ RV_{\text{in}} : \begin{pmatrix} po & pn \\ 1 & 1 \\ 3 & 2 \\ 4 & 5 \end{pmatrix} \]

\[ VV_{\text{data}} : \begin{pmatrix} d & p & i_1 \\ 1 & 1 & 10 \\ 1 & 2 & 20 \\ 2 & 3 & 30 \\ 3 & 4 & 40 \\ 4 & 5 & 50 \end{pmatrix} \]

\[ VV_{\text{res}} : \begin{pmatrix} d & p & i_1 \\ 1 & 1 & 10 \\ 1 & 2 & 20 \\ 2 & 3 & 40 \\ 5 & 4 & 50 \end{pmatrix} \]
2. Vector Algebra

\[ RV_{\text{res}} : \begin{pmatrix} po & pn \\ 1 & 1 \\ 2 & 2 \\ 4 & 3 \\ 5 & 4 \end{pmatrix} \]

**propRename (binary, shared variables)**

\[ VV_{\text{res}} = \text{propRename}(RV_{\text{in}}, VV_{\text{data}}) \]

*PropRename* does segment renaming. It copies the contents of \( VV_{\text{data}} \) into \( VV_{\text{res}} \) with exception of the descriptor column \( d \).

For each segment id in \( d \) a matching entry in \( po \) of \( RV_{\text{in}} \) exists (if not an error is triggered). The according \( pn \) value becomes the new segment id.

**Example**

\[ RV_{\text{in}} : \begin{pmatrix} po & pn \\ 1 & 2 \\ 2 & 3 \\ 4 & 5 \\ 5 & 6 \end{pmatrix} \]

\[ VV_{\text{data}} : \begin{pmatrix} d & p & i_1 \\ 1 & 1 & 10 \\ 1 & 2 & 20 \\ 2 & 3 & 30 \\ 4 & 4 & 40 \\ 5 & 5 & 50 \end{pmatrix} \]

\[ VV_{\text{res}} : \begin{pmatrix} d & p & i_1 \\ 2 & 1 & 10 \\ 2 & 2 & 20 \\ 3 & 3 & 30 \\ 5 & 4 & 40 \\ 6 & 5 & 50 \end{pmatrix} \]

**propReorder (binary, shared variables)**

\[ (VV_{\text{res}}, PV_{\text{res}}) = \text{propReorder}(PV_{\text{in}}, VV_{\text{data}}) \]

*PropReorder* does segment renaming (as described in propRename), filtering (as described in propFilter) and reordering.

If reordering of a segment \( s \) happens, the components of the vector \( VV_{\text{data}} \) associated with \( s \) are moved upwards or downwards the vector in relation to
other segments. The reorder instructions are read from $PV_{in}$: the $p_{o}$ column describes the old ordering and the $p_{n}$ column the final ordering.

The result of reordering, filtering and renaming is written to $V V_{res}$. The position changes are referenced by $PV_{res}$.

**Example**  Actual reordering takes place (in the result 50 occurs before 10). The descriptor column in vector $V V_{res}$ is still consistent because of proper segment renaming.

$PV_{in}:$

\[
\begin{pmatrix}
po & pn \\
4 & 1 \\
1 & 2 \\
2 & 3
\end{pmatrix}
\]

$V V_{data}:$

\[
\begin{pmatrix}
d & p & i_1 \\
1 & 1 & 10 \\
2 & 2 & 20 \\
2 & 3 & 30 \\
3 & 4 & 40 \\
4 & 5 & 50
\end{pmatrix}
\]

$V V_{res}:$

\[
\begin{pmatrix}
d & p & i_1 \\
1 & 1 & 50 \\
2 & 2 & 10 \\
3 & 3 & 20 \\
3 & 4 & 30
\end{pmatrix}
\]

$PV_{res}:$

\[
\begin{pmatrix}
po & pn \\
5 & 1 \\
1 & 2 \\
2 & 3 \\
3 & 4
\end{pmatrix}
\]

**R1 (**ary, pseudo)**

\[V_{res} = R1(V_1, ... V_n)\]
\[= V_1\]

R1 returns the first result of a multi-vector result ($n = 1$ is allowed). $V$ can be any vector type.

**R2 (**ary, pseudo)**

\[V_{res} = R2(V_1, V_2, ... V_n)\]
\[= V_2\]

R2 is like R1, but at least a two vector input is required.
2. Vector Algebra

**R3 (**ary, pseudo)**

\[ V_{\text{res}} = R3(V_1, V_2, V_3, \ldots V_n) \]
\[ = V_3 \]

R3 is like R1, but at least a three vector input is required.
R4 or higher are not specified (no primitive creates a four-vectors result).

**restrictVec (binary, shared variables)**

\[ (V V_{\text{res}}, R V_{\text{res}}) = \text{restrictVec}(V V_c, V V_{\text{data}}) \]

*RestrictVec* filters \( V V_{\text{data}} \) according to \( V V_c \). \( V V_c \) contains a single column of boolean type. Both vectors share the same length.

For every component of \( V V_{\text{data}} \) the according boolean value in \( V V_c \) is investigated. If \textit{true}, the component is copied to \( V V_{\text{res}} \) (including descriptor). The remaining components get dropped.

\( R V_{\text{res}} \) references old and new positions of the copied components.

**Example**

\( V V_c : \)

\[
\begin{pmatrix}
 d & p & i_1 \\
 1 & 1 & t \\
 1 & 2 & f \\
 1 & 3 & t \\
 1 & 4 & f \\
\end{pmatrix}
\]

\( V V_{\text{data}} : \)

\[
\begin{pmatrix}
 d & p & i_1 \\
 1 & 1 & 10 \\
 1 & 2 & 20 \\
 2 & 3 & 30 \\
 2 & 4 & 40 \\
\end{pmatrix}
\]

\( V V_{\text{res}} : \)

\[
\begin{pmatrix}
 d & p & i_1 \\
 1 & 1 & 10 \\
 2 & 2 & 30 \\
\end{pmatrix}
\]

\( R V_{\text{res}} : \)

\[
\begin{pmatrix}
 p o & p n \\
 1 & 1 \\
 3 & 2 \\
\end{pmatrix}
\]

**reverseA (unary, partitioned loop)**

\[ (V V_{\text{res}}, P V_{\text{res}}) = \text{reverseA}(V V_{\text{data}}) \]

*ReverseA* reverses the vertical order of components in \( V V_{\text{data}} \). \( P V_{\text{res}} \) references old and new positions.
2. Vector Algebra

Example

\[ VV_{\text{data}} : \begin{pmatrix} d & p & i_1 \\ 1 & 1 & 10 \\ 1 & 2 & 20 \\ 2 & 3 & 30 \\ 2 & 4 & 40 \end{pmatrix} \]

\[ VV_{\text{res}} : \begin{pmatrix} d & p & i_1 \\ 1 & 1 & 40 \\ 1 & 2 & 30 \\ 2 & 3 & 20 \\ 2 & 4 & 10 \end{pmatrix} \]

\[ PV_{\text{res}} : \begin{pmatrix} po & pn \\ 4 & 1 \\ 3 & 2 \\ 2 & 3 \\ 1 & 4 \end{pmatrix} \]

`reverseL` (unary, shared variables)

\[ (VV_{\text{res}}, PV_{\text{res}}) = reverseL(VV_{\text{data}}) \]

`ReverseL` does the same as `reverseA` but per segment.

Example

\[ VV_{\text{data}} : \begin{pmatrix} d & p & i_1 \\ 1 & 1 & 10 \\ 1 & 2 & 20 \\ 2 & 3 & 30 \\ 2 & 4 & 40 \end{pmatrix} \]

\[ VV_{\text{res}} : \begin{pmatrix} d & p & i_1 \\ 1 & 1 & 10 \\ 1 & 2 & 20 \\ 2 & 3 & 30 \\ 2 & 4 & 30 \end{pmatrix} \]

\[ PV_{\text{res}} : \begin{pmatrix} po & pn \\ 2 & 1 \\ 1 & 2 \\ 4 & 3 \\ 3 & 4 \end{pmatrix} \]
2. Vector Algebra

**segment (unary, partitioned loop)**

\[ VV_{res} = \text{segment}(VV_{data}) \]

This primitive creates exactly one segment for each component of \( VV_{data} \). The segment numbering is identical to the positions column. All item columns get copied.

**Example**

\[ VV_{data} : \begin{pmatrix} d & p & i_1 \\ 1 & 1 & 10 \\ 2 & 3 & 30 \\ 2 & 4 & 40 \end{pmatrix} \]

\[ VV_{res} : \begin{pmatrix} d & p & i_1 \\ 1 & 1 & 10 \\ 2 & 2 & 20 \\ 3 & 3 & 30 \\ 4 & 4 & 40 \end{pmatrix} \]

**selectPos (binary, partitioned loop)**

\[ (VV_{res}, RV_{res}) = \text{selectPos}(VS_c, oper, VV_{data}) \]

`selectPos` filters vector components of \( VV_{data} \) depending on their position. `Oper` is a comparison operator (out of: \( >, \geq, <, \leq, = \)) and \( VS_c \) provides an integer to compare with (exactly one integer column is expected). If a negative integer value is provided, it will be interpreted as plus one.

An additional result vector of type \( RV_{res} \) is assembled. It maps the shift of positions.

**Example** All components having positions \( > 4 \) get selected: [5, 6, 7].

\[ VS_c : \begin{pmatrix} d & p & i_1 \\ 1 & 1 & 4 \end{pmatrix} \]

\( oper : > \)

\[ VV_{data} : \begin{pmatrix} d & p & i_1 \\ 1 & 1 & 10 \\ 2 & 2 & 20 \\ 3 & 3 & 30 \\ 4 & 4 & 40 \\ 5 & 5 & 50 \\ 5 & 6 & 60 \\ 5 & 7 & 70 \end{pmatrix} \]
2. Vector Algebra

$VV_{res}$:
\[
\begin{pmatrix}
  d & p & i_1 \\
  5 & 1 & 50 \\
  5 & 2 & 60 \\
  5 & 3 & 70 \\
\end{pmatrix}
\]

$RV_{res}$:
\[
\begin{pmatrix}
  po & pn \\
  5 & 1 \\
  6 & 2 \\
  7 & 3 \\
\end{pmatrix}
\]

**selectPosLift (binary, shared variables)**

$\left(VV_{res}, RV_{res}\right) = selectPosLift(VS_c, oper, VV_{data})$

*SelectPosLift* is the lifted version of *selectPos*: filtering takes place on segment level.

For each segment, position counting starts with one (see example Vector (2.28) and the additional $p'$ column). Comparisons are carried out against the value in $p'$.

**Example**  All components with $p' \geq 3$ get selected.

$VS_c$:
\[
\begin{pmatrix}
  d & p & i_1 \\
  1 & 1 & 3 \\
\end{pmatrix}
\]

$oper : \geq$

$VV_{data}$:
\[
\begin{pmatrix}
  d & p & i_1 & p' \\
  1 & 1 & 10 & 1 \\
  1 & 2 & 20 & 2 \\
  2 & 3 & 30 & 1 \\
  2 & 4 & 40 & 2 \\
  2 & 5 & 50 & 3 \\
  2 & 6 & 60 & 4 \\
  3 & 7 & 70 & 1 \\
  3 & 8 & 80 & 2 \\
  3 & 9 & 90 & 3 \\
  4 & 10 & 100 & 1 \\
\end{pmatrix}
\]  \hspace{1cm} (2.28)

$VV_{res}$:
\[
\begin{pmatrix}
  d & p & i_1 \\
  2 & 1 & 50 \\
  2 & 2 & 60 \\
  3 & 3 & 90 \\
\end{pmatrix}
\]
2. Vector Algebra

\[ RV_{\text{res}} : \begin{pmatrix} po & pn \\ 5 & 1 \\ 6 & 2 \\ 9 & 3 \end{pmatrix} \]

singletonDescr (nullary, atomic)

\[ DV_{\text{res}} = \text{singletonDescr}() \]

This is a constant vector primitive (zero arguments).

Return value

\[ DV_{\text{res}} : \begin{pmatrix} d & p \\ 1 & 1 \end{pmatrix} \]

sortWith (binary, sorting)

\[ (VV_{\text{res}}, PV_{\text{res}}) = \text{sortWith}(VV_c, VV_{\text{data}}) \]

SortWith basically is sorting an input vector \( VV_{\text{data}} \). However the sorting criteria are found in \( VV_c \). Both vectors have to share the same length.

Primary sorting criteria is the descriptor column of \( VV_c \). Secondary criteria is the first item column (at least one item column has to exist). If equal values in a certain item column occur, the next item column is investigated. If all values of two components are equal, the \( p \) column will be the last (and always unambiguous) sorting criteria. In example Vector 2.4.1 the additional \( p' \) column contains the position values after sorting.

After sorting the corresponding components of \( VV_{\text{data}} \) are copied into \( VV_{\text{res}} \). SortWith requires that both \( VV_c \) and \( VV_{\text{data}} \) have the same descriptor. It is copied to \( VV_{\text{res}} \).

\( RV_{\text{res}} \) references old and new positions for vector components in \( VV_{\text{data}} \) and \( VV_{\text{res}} \).

Example  Segment one: gets sorted by \( i_1 \). Segment two: \( i_1 \) values are all the same, need to sort by \( i_2 \). Segment three: gets sorted by \( p \).

\[ VV_c : \begin{pmatrix} d & p & i_1 & i_2 & p' \\ 1 & 1 & 20 & "C" & 4 \\ 1 & 2 & 40 & "A" & 1 \\ 1 & 3 & 30 & "B" & 3 \\ 1 & 4 & 10 & "D" & 2 \\ 2 & 5 & 0 & "C" & 7 \\ 2 & 6 & 0 & "B" & 6 \\ 2 & 7 & 0 & "A" & 5 \\ 3 & 8 & 50 & "E" & 8 \\ 3 & 9 & 50 & "E" & 9 \end{pmatrix} \]
2. Vector Algebra

\[ VV_{\text{data}} : \begin{pmatrix}
1 & 1 & 20 \\
1 & 2 & 40 \\
1 & 3 & 30 \\
1 & 4 & 10 \\
2 & 5 & 1 \\
2 & 6 & 2 \\
2 & 7 & 3 \\
3 & 8 & 80 \\
3 & 9 & 90 \\
\end{pmatrix} \]

\[ VV_{\text{res}} : \begin{pmatrix}
1 & 1 & 10 \\
1 & 2 & 20 \\
1 & 3 & 30 \\
1 & 4 & 40 \\
2 & 5 & 3 \\
2 & 6 & 2 \\
2 & 7 & 1 \\
3 & 8 & 80 \\
3 & 9 & 90 \\
\end{pmatrix} \]

\[ PV_{\text{res}} : \begin{pmatrix}
4 & 1 \\
1 & 2 \\
3 & 3 \\
2 & 4 \\
7 & 5 \\
6 & 6 \\
5 & 7 \\
8 & 8 \\
9 & 9 \\
\end{pmatrix} \]

toDescr (unary, partitioned loop)

\[ DV_{\text{res}} = \text{toDescr}(VV_{\text{data}}) \]

ToDescr turns a value vector into a description vector by copying the descriptor (and ignoring all item columns).

Example  Item columns \(i_1\) and \(i_2\) are ignored.
2. Vector Algebra

\[ VV_{\text{data}} : \begin{pmatrix} d & p & i_1 & i_2 \\ 3 & 1 & 10 & u \\ 6 & 2 & 20 & u \\ 7 & 3 & 30 & u \\ 7 & 4 & 40 & u \end{pmatrix} \]

\[ DV_{\text{res}} : \begin{pmatrix} d & p \\ 3 & 1 \\ 6 & 2 \\ 7 & 3 \\ 7 & 4 \end{pmatrix} \]

unique (unary, sorting)

\[ VV_{\text{res}} = \text{unique}(VV_{\text{data}}) \]

*Unique* filters duplicate components. Two components are duplicates if all of their data values in corresponding item columns are equal. The component with the smallest position id is kept.

**Example**  Components five and six get dropped. Their data values (10 / 20) already occurred on position one and two.

\[ VV_{\text{data}} : \begin{pmatrix} d & p & i_1 \\ 1 & 1 & 10 \\ 1 & 2 & 20 \\ 1 & 3 & 30 \\ 1 & 4 & 40 \\ 2 & 5 & 10 \\ 2 & 6 & 20 \\ 2 & 7 & 70 \\ 2 & 8 & 80 \\ 2 & 9 & 90 \end{pmatrix} \]

\[ VV_{\text{res}} : \begin{pmatrix} d & p & i_1 \\ 1 & 1 & 10 \\ 1 & 2 & 20 \\ 1 & 3 & 30 \\ 1 & 4 & 40 \\ 1 & 5 & 70 \\ 1 & 6 & 80 \\ 1 & 7 & 90 \end{pmatrix} \]
uniqueL (unary, sorting)

$$VV_{res} = uniqueL(VV_{data})$$

Lifted version of unique. Duplicates are only filtered if they occur inside the same segment.

**Example**  Same input as in unique example. Nothing gets dropped because of different segments.

$$VV_{data} :$$

$$
\begin{pmatrix}
  d & p & i_1 \\
  1 & 1 & 10 \\
  1 & 2 & 20 \\
  1 & 3 & 30 \\
  1 & 4 & 40 \\
  2 & 5 & 10 \\
  2 & 6 & 20 \\
  2 & 7 & 70 \\
  2 & 8 & 80 \\
  2 & 9 & 90 \\
\end{pmatrix}
$$

$$VV_{res} :$$

$$
\begin{pmatrix}
  d & p & i_1 \\
  1 & 1 & 10 \\
  1 & 2 & 20 \\
  1 & 3 & 30 \\
  1 & 4 & 40 \\
  2 & 5 & 10 \\
  2 & 6 & 20 \\
  2 & 7 & 70 \\
  2 & 8 & 80 \\
  2 & 9 & 90 \\
\end{pmatrix}
$$

unSegment (unary, partitioned loop)

$$VV_{res} = unSegment(VV_{data})$$

UnSegment copies all item columns of $$VV_{data}$$. But it ignores the descriptor and creates a new one consisting of one big segment with id=1.

It is not a reverse function to segment because unSegment does not recreate the original segment structure.

**Example**
2. Vector Algebra

\[ VV_{\text{data}} : \begin{pmatrix} d & p & i_1 \\ 1 & 1 & 10 \\ 2 & 2 & 20 \\ 3 & 3 & 30 \\ 4 & 4 & 40 \end{pmatrix} \]

\[ VV_{\text{res}} : \begin{pmatrix} d & p & i_1 \\ 1 & 1 & 10 \\ 2 & 2 & 20 \\ 3 & 3 & 30 \\ 4 & 4 & 40 \end{pmatrix} \]

**vecMax (unary, shared variables)**

\[ VS_{\text{res}} = vecMax(VV_{\text{data}}) \]

*VecMax* requires exactly one item column and length > 0. *VecMax* returns the maximum value found in the column. For booleans, *true* is considered as maximum.

**Example**

\[ VV_{\text{data}} : \begin{pmatrix} d & p & i_1 \\ 1 & 1 & 10 \\ 2 & 2 & 20 \\ 3 & 3 & 30 \\ 4 & 4 & 40 \end{pmatrix} \]

\[ VS_{\text{res}} : \begin{pmatrix} d & p & i_1 \\ 1 & 1 & 40 \end{pmatrix} \]

**vecMaxLift (unary, shared variables)**

\[ VV_{\text{res}} = vecMaxLift(VV_{\text{data}}) \]

*VecMaxLift* is the lifted version of *vecMax*. It returns a maximum per segment.

**Example**

\[ VV_{\text{data}} : \begin{pmatrix} d & p & i_1 \\ 1 & 1 & 10 \\ 2 & 2 & 20 \\ 3 & 3 & 30 \\ 4 & 4 & 40 \end{pmatrix} \]
2. Vector Algebra

\[ VV_{\text{res}} : \begin{pmatrix} d & p & i_1 \\ 1 & 1 & 20 \\ 2 & 2 & 40 \end{pmatrix} \]

**vecMin (unary, shared variables)**

\[ VS_{\text{res}} = \text{vecMin}(VV_{\text{data}}) \]

VecMin is the opposite to vecMax.

**Example**

\[ VV_{\text{data}} : \begin{pmatrix} d & p & i_1 \\ 1 & 1 & 10 \\ 1 & 2 & 20 \\ 2 & 3 & 30 \\ 2 & 4 & 40 \end{pmatrix} \]

\[ VS_{\text{res}} : \begin{pmatrix} d & p & i_1 \\ 1 & 1 & 10 \end{pmatrix} \]

**vecMinLift (unary, shared variables)**

\[ VV_{\text{res}} = \text{vecMinLift}(VV_{\text{data}}) \]

VecMinLift is the opposite to vecMaxLift.

**Example**

\[ VV_{\text{data}} : \begin{pmatrix} d & p & i_1 \\ 1 & 1 & 10 \\ 1 & 2 & 20 \\ 2 & 3 & 30 \\ 2 & 4 & 40 \end{pmatrix} \]

\[ VV_{\text{res}} : \begin{pmatrix} d & p & i_1 \\ 1 & 1 & 10 \end{pmatrix} \]

**vecSum (unary, shared variables)**

\[ VS_{\text{res}} = \text{vecSum}(VV_{\text{data}}, \text{type}) \]

VecSum sums up all values of the input column in \( VV_{\text{data}} \). Exactly one column is required and it has to be of a numeric type (int, nat, double).

The \textit{type} argument determines the result type if the input vector does not (because of length=0). This corner case only exists on Haskell side.
2. Vector Algebra

Example

\[ VV_{\text{data}} : \]
\[
\begin{pmatrix}
 d & p & i_1 \\
 1 & 1 & 10 \\
 1 & 2 & 20 \\
 2 & 3 & 30 \\
 2 & 4 & 40 \\
\end{pmatrix}
\]

\[ VS_{\text{res}} : \]
\[
\begin{pmatrix}
 d & p & i_1 \\
 1 & 1 & 100 \\
\end{pmatrix}
\]

\[ \text{vecSumLift (binary, shared variables)} \]

\[ VV_{\text{res}} = \text{vecSumLift}(DV_{\text{len}}, VV_{\text{data}}) \]

VecSumLift is the lifted version of vecSum. It computes a total per segment. The length \( l \) of \( DV_{\text{len}} \) determines the segments to be processed. For each segment with \( id \leq l \) the total is computed. If a certain segment does not occur in \( VV_{\text{data}} \), the total of it will be 0.

Example  Length of \( DV_{\text{len}} \) is 3 which means computation of segments 1, 2 and 3 will be carried out.

\[ DV_{\text{len}} : \]
\[
\begin{pmatrix}
 d & p \\
 1 & 1 \\
 1 & 2 \\
 1 & 3 \\
\end{pmatrix}
\]

\[ VV_{\text{data}} : \]
\[
\begin{pmatrix}
 d & p & i_1 \\
 1 & 1 & 10 \\
 1 & 2 & 20 \\
 2 & 3 & 30 \\
 2 & 4 & 40 \\
\end{pmatrix}
\]

\[ VV_{\text{res}} : \]
\[
\begin{pmatrix}
 d & p & i_1 \\
 1 & 1 & 30 \\
 2 & 2 & 70 \\
 3 & 3 & 0 \\
\end{pmatrix}
\]

\[ \text{zipL (binary, shared variables)} \]

\[ (VV_{\text{res}}, RV_1, RV_2) = \text{zipL}(VV_1, VV_2) \]

ZipL merges two value vectors by concatenating their item columns (same principle as in pairL). \( VV_1 \) and \( VV_2 \) are not required to share the same length.
or descriptors. This is made possible by an additional filtering step which is explained next.

ZipL determines for both input vectors the length of their individual segments (see Table 2.4 in example). If a segment does not exist its length is assumed as zero. For each segment pair the minimum is computed. All minima together define the segment structure of the result vector. This vector is created and its item columns get filled in appropriately. If a certain result segment is too short to contain all components of an input segment - the backmost components will be dropped.

$RV_1$ and $RV_2$ reference the old and new positions (if a new position exists) for components of $VV_1$ and $VV_2$.

**Example**

$VV_1$:

\[
\begin{pmatrix}
1 & 1 & 10 \\
1 & 2 & 20 \\
2 & 3 & 30 \\
4 & 4 & 40 \\
4 & 5 & 50 \\
4 & 6 & 60 \\
4 & 7 & 70
\end{pmatrix}
\]

$VV_2$:

\[
\begin{pmatrix}
1 & 1 & \text{"A"} \\
1 & 2 & \text{"B"} \\
2 & 3 & \text{"C"} \\
3 & 4 & \text{"D"} \\
4 & 5 & \text{"E"} \\
4 & 6 & \text{"F"}
\end{pmatrix}
\]

<table>
<thead>
<tr>
<th>Segment id</th>
<th>Length in $VV_1$</th>
<th>Length in $VV_2$</th>
<th>Minimum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 2.4.: Computation of segment lengths for $VV_{res}$

$VV_{res}$:

\[
\begin{pmatrix}
1 & 1 & 10 & \text{"A"} \\
1 & 2 & 20 & \text{"B"} \\
2 & 3 & 30 & \text{"C"} \\
4 & 4 & 40 & \text{"E"} \\
4 & 5 & 50 & \text{"F"}
\end{pmatrix}
\]
2. Vector Algebra

\[ RV_1 : \begin{pmatrix} po & pn \\ 1 & 1 \\ 2 & 2 \\ 3 & 3 \\ 4 & 4 \\ 5 & 5 \end{pmatrix} \]

\[ RV_2 : \begin{pmatrix} po & pn \\ 1 & 1 \\ 2 & 2 \\ 3 & 3 \\ 5 & 4 \\ 6 & 5 \end{pmatrix} \]
3. Implementation Tools

In this chapter we discuss the implementation tools used. The programming language for the implementation is C++. A selection of language features is described in Section 3.1.

An important part of this work is the performant execution on multi-CPU computers. The main tool selected for creating parallel code is OpenMP and will be discussed in Section 3.2. Another candidate for parallelization was ArBB but after doing the first implementation steps it turned out being not suitable for this work. A short overview and why it has been rejected can be found in Section 3.3.

JsonCpp is a library for parsing and generating JSON-formatted data. A short description of this library can be found in Section 3.4.
3. Implementation Tools

3.1. C++

C++ provides good performance and certain high-level language features.

A subset of the properties of C++ will be described in this section. In Section 3.1.1 we discuss fundamental data types. All types of the vector algebra will be mapped onto them. Section 3.1.2 illustrates what function templates are and how they can be used to avoid redundant code parts. Subsection 3.1.3 deals with semaphores of the C++ STL (Standard Template Library).

Not described in a section on its own but worth mentioning is the OOP support of C++ (the object-oriented programming paradigm). Within this work OOP features are used for the implementation of vectors. Certain properties (e.g. descriptors) of the different vector types are shared and therefore inheritance can be used to avoid redundant code.

3.1.1. Data Types

In C++ the exact properties of data types are depending on compiler and hardware. However typical 64 bit computers should share the type sizes as presented in Table 3.1. These types will be used to implement five of the six vector algebra types. The sixth type is called unit and has no requirement for data storage.

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>1 byte</td>
<td>true or false</td>
</tr>
<tr>
<td>char</td>
<td>1 byte</td>
<td>depends on interpretation</td>
</tr>
<tr>
<td>double</td>
<td>8 bytes</td>
<td>$2.22507-10^{308}$ to $1.79769-10^{+308}$</td>
</tr>
<tr>
<td>long int</td>
<td>8 bytes</td>
<td>$-2^{64}$ to $+2^{64}-1$</td>
</tr>
<tr>
<td>unsigned long int</td>
<td>8 bytes</td>
<td>$0$ to $2^{64}-1$</td>
</tr>
</tbody>
</table>

Table 3.1.: Subset of fundamental types in C++

Typedef The C typedef construct is used for creating new abbreviations of known types. Listing 3.1 contains two examples.

```c
1 typedef long int intval;
2 typedef String* stringval;
```

Listing 3.1: Typedef examples

On line 1 a new type name called “intval” is introduced. This new type will behave exactly like the fundamental C++ type. But writing “intval” instead of “long int” is in our case beneficial for readability of code. Furthermore a typedef creates an abstraction layer for types which improves code maintainability.

It is even possible to hide pointers as done on line 2. The type “stringval” is a pointer but it looks like a non-pointer.
Hiding a pointer is useful if uniform access to pointer- and non-pointer types is desired. For example when working with templates (see next section).

### 3.1.2. Function Templates

Function templates are a means for generic programming. In source code a function template looks like an ordinary function with additional type parameters. In C++ they are written in angle brackets: `<integer, string>`.

A typical use case for function templates is to let the type parameter determine the data type a function does deal with.

Listing 3.2 contains a simple example. It has a type parameter called `T`. The arguments `a` and `b` (which are of type `T`) are compared against each other and the bigger one is returned.

```cpp
1 template <class T>
2 T max(T a, T b) {
3    if (a>b) return a;
4    else return b;
5 }
```

Listing 3.2: Function template example-1

Template function `max()` is called as demonstrated in Listing 3.3. Variable `T` will be initialized with the type parameter provided in angle brackets.

```cpp
1 // integer comparison
2 // T = int
3 int a_int = 1;
4 int b_int = 2;
5 int res_int = max<int>(a_int, b_int);
6
7 // double comparison
8 // T = double
9 double a_double = 1.0;
10 double b_double = 2.0;
11 double res_double = max<double>(a_double, b_double);
```

Listing 3.3: Function template example-2

Another form of templates in C++ are `class templates`. They share the same concept but work on the class layer.

### 3.1.3. Semaphores

A `semaphore` is a synchronization construct for parallel programming. Semaphores can be used to implement mutual exclusion between threads and therefore pro-
3. Implementation Tools

tect shared resources from being damaged due to multiple read/write access at the same time.

Central data structure of a semaphor is an integer variable which can be accessed atomically (this includes not only read/write access but increments/decrements as well). Interruptions by other threads are not possible. This is achieved by certain hardware measures like blocking the memory bus for other threads (CPUs). The variable can then be used for book keeping the number of threads being currently inside of a critical code region.

The C++ STL offers a semaphore API. In the next paragraphs an example usage of semaphores is illustrated. The explanations focus on the conceptual layer - the inner workings of STL semaphores may be slightly different but are not of importance here.

In the initialization example of Listing 3.4 we choose a value of 1 because the semaphore is intended to be used as a mutual exclusion lock.

```c
// set max. allowed thread number for critical region
int maxThreads = 1;

// allocate semaphore
sem_t busy;

// initialize semaphore
//(the 0 indicates no sharing between processes)
sem_init(&busy, 0, maxThreads);
```

Listing 3.4: Semaphore initialisation

Listing 3.5 contains a corresponding example of how to use our recently created semaphore. Any thread with the desire to enter the critical code section at first has to pass line 4. The function `sem_wait()` checks if our semaphore variable is bigger than 0 (in our example it can be only 1 or 0). If yes, the variable is decremented and the thread enters the critical code region. If not, the thread (maybe more than one) has to wait. As long as another thread already is inside of the critical code region, no other thread must enter.

As soons as the currently active thread leaves the critical code region, it executes `sem_post()` and triggers an increment of the semaphore variable. Additionally, a wakeup signal for one of the sleeping threads is created. Only now the next thread can enter the critical code region.
As a semaphore’s counter can be set to an arbitrary number it is not limited on implementing mutual exclusion locks. In Section 4.2.5 on sorting another use case is documented.
3. Implementation Tools

3.2. OpenMP

OpenMP (Open Multi-Processing) [4] is a parallelization API for C/C++ and Fortran. It is not only a library as compiler support is required. In OpenMP parallelism instructions (like creation of threads) are separate from ordinary C++ code. This is achieved by means of compiler directives. In the simplest case the software developer writes a sequential algorithm in C++. After testing it, he can embedd certain compiler directives into his code and will get a parallel algorithm without changing the original code. A “separation of concerns” approach is realised.

Listing 3.6 contains such a sequential code fragment. An array is allocated and its entries are initialized with zeroes. When executing as provided, it will make the CPU loop over the 100 array entries. If multiple CPUs are available (and idle) they cannot improve performance. At least when working with larger arrays having millions of entries it will be desired to exploit all CPUs.

```
1 // allocation
2 int length = 100;
3 int* array = new int[length];
4
5 // initialization
6 for (int i=0; i< length; i++) {
7    array[i] = 0;
8 }
```

Listing 3.6: OpenMP example: serial loop

As the array contents are completely independent of each other it would be simple to split up the array in multiple ranges and let each available CPU work on a certain range. The OpenMP way of implementing this is by adding a compiler directive (see line 1 of Listing 3.7).

```
# pragma omp parallel for
2 for (int i=0; i< length; i++) {
3    array[i] = 0;
4 }
```

Listing 3.7: OpenMP example: parallel loop

During compilation the OpenMP extension of the compiler will rewrite the loop for appropriate execution among multiple CPUs.

At runtime the number of available CPUs will be detected and for example if there are two CPUs two threads will be created and each one will be assigned to compute 50 contiguous iterations. The mapping of threads to CPUs is not controlled by OpenMP but depends on the scheduling policy of the operating system.

---

1. A list of supported compilers can be found here: http://openmp.org/wp/openmp-compilers/ (14.2.2013)
3. Implementation Tools

Figure 3.1.: Execution model: example execution

system. Because of this and other pseudo-random factors it will probably result in one thread being earlier done with its array range than the other thread. OpenMP takes care of this by blocking all threads at line 4 until the work is completely done.

The example may look like a universal panacea but it does not work for every loop. For example if \( \text{array}[i] \) depends on \( \text{array}[i - 1] \) the algorithm will in general be wrong. OpenMP provides additional tools to the software developer for dealing with this. The API specifications can be found in [5].

3.2.1. Execution Model

The OpenMP execution model known as fork-join model of parallel execution interprets any program as consisting of alternating sequential and parallel code regions. See Figure 3.1 for an example.

Every program starts with one master (=initial) thread only. When a parallel code region is encountered it will create an additional number of threads (the concrete number depends on several factors and rules) and together all threads will execute the code found inside of the parallel region. After the threads have done their computations individually, an additional merging phase of the results may be required. In the end all additional threads will dissipate and the master thread alone will continue with serial program execution.

Additionally OpenMP allows nesting of parallel regions. Basically this leads to an exponential growing number of threads and to the requirement for growth limitations. In our implementation of quicksort we use semaphores to limit the total number of threads (see Section 4.2.5.1).

3.2.2. Preprocessor Directives

The C++ compilation process consists of multiple steps of which the first step is calling the C++ preprocessor. It handles certain preparation tasks before the actual compilation starts.

A software developer can interact directly with the preprocessor by embedding preprocessor directives into his source code. A well-known directive is
3. Implementation Tools

#include which has a filename argument and instructs the preprocessor to replace the include line with the contents of the file it points at.

The C++ standard defines a special preprocessor directive called pragma. It “causes the implementation to behave in an implementation-defined manner” (page 396 of [6]).

OpenMP uses the pragma interface for introducing several new preprocessor directives. They all start with the prefix #pragma omp and if occurring in source code configure the OpenMP behaviour.

3.2.3. parallel Directive

This is the most basic construct of OpenMP. It starts a parallel code region which gets executed by multiple threads. It is simple and unrestricted and therefore powerful and suitable for implementing complex algorithms.

Each thread of example 3.8 will print a “Hello, world!” message on screen preceded by an integer value identifying the thread. Please note that there is no thread synchronization. When executing this code piece it is probable that the thread’s outputs will get mixed up with each other’s.

```
1  // serial region
2
3 #pragma omp parallel
4 {
5     // parallel region
6     cout << "Thread no. " << omp_get_thread_num()
7     << " says: Hello, world!" << "\n";
8 }
9
10 // serial region
```

Listing 3.8: Hello world example

The numbering scheme for threads always starts with 0 (which is the master thread) and increments by one for each additional thread of the current team.

On line 6 a library function omp_get_thread_num() is used. It is worth pointing out that the OpenMP API exploits two layers: preprocessor directives and library functions. From the programmer’s perspective both work seamlessly together.

Vector Addition Example  Listing 3.9 already does something useful. Let \(a\) and \(b\) contain the \(x\), \(y\) and \(z\) components of two 3d vectors (no vector algebra here but ordinary vectors). On line 7 a fixed thread amount is defined which is 3 for this example. Remark that if the thread amount does not exploit the number of available CPUs the program will underperform. Accordingly 3 threads will be created and each of them computes one component of the vector addition.
On line 9 each thread creates a variable \textit{idx} and writes its numerical identifier into it for later use. The question may arise whether this leads to a race condition. It does not because of the implicit OpenMP semantics for parallel regions. Variables declared outside (before) a parallel region are shared between all threads (like \textit{a,b, res}). Variables declared inside of a parallel region are thread-private and only accessible as long as the parallel section is not over. In other words, the variable \textit{idx} will be created three times and consumes memory space for three variables.

This second example was brought up to demonstrate how the parallel construct works. In production code one would better use a for construct so that it does not depend on exactly 3 threads.

\subsection*{3.2.4. \textit{for} Directive}

The \textit{for} directive already occurred in the introduction example of Section 3.2. It splits up a for loop into multiple contiguous ranges and lets each thread work on one of these ranges.

The directive \#pragma omp parallel for used in the introduction example is actually just a shortcut of the two nested pragmas of Example 3.10.
3. Implementation Tools

```c
int* a = new int[3];
int* b = new int[3];
int* res = new int[3];

// ... initialize a and b

#pragma omp parallel
{
    #pragma omp for
    for (int i=0; i<3; i++) {
        res[idx] = a[idx] + b[idx];
    }
}
```

Listing 3.10: Vector addition with for construct

This is a better version of vector addition because it is independent of the number of threads. Actually the program can be compiled completely without OpenMP support. The behaviour of the C++ preprocessor is to ignore all pragmas it does not recognize. Ignoring the pragmas of this example results in a sequential for loop.

3.2.5. Other Directives

The `parallel` and `for` constructs described in the last two sections are the most important ones used for the implementation work. A `for` is very inflexible but very convenient when working on parallelization of loops.

There are more directives which can be combined with `parallel`. The following directives can show up anywhere inside a parallel code region (incomplete list):

- `#pragma omp single`: can be used to define a sequential region within a parallel region. The code inside of it will be executed by exactly one thread.

- `#pragma omp critical`: a code region executed by all threads of the parallel region - mutual exclusive (one thread after the other). Good for merging computation results.

- `#pragma omp barrier`: threads encountering this directive will wait here until all other threads arrive. This directive is useful when one part of an algorithm has to be finished by all threads before moving on to the next part.

3.2.6. Clauses and Environment Variables

Compiler directives can be modified using `clauses`. Clauses can determine the scope of variables, how variables get initialized and how variables are to be merged at the end of a particular parallel section.
3. Implementation Tools

Clauses can be appended to a subset of compiler directives. Examples are:

- **private(list)** variables being *private* are copied into private memory of each thread.
- **shared(list)** variables declared as being *shared* are accessible by all threads. If not properly used, race conditions may occur.
- **reduction(operator : list)** merges variables before leaving a parallel section. A restricted set of operators is available (for example +, -, *). Since OpenMP 3.1 **min, max** are also available.

*Environment Variables* define the program-wide behaviour of OpenMP. Examples are:

- **OMP_NUM_THREADS** sets the number of threads to be used in parallel regions.
- **OMP_NESTED** allows or disallows creation of additional threads in nested parallel regions.
3.3. ArBB (rejected)

Before OpenMP was selected as parallelization tool, the first implementation approach was using ArBB [7].

ArBB (Intel® Array Building Blocks) is a C++ library for both multi-CPU and vector parallelism. ArBB exploits vector units of modern CPU architectures which allow SIMD (single instruction, multiple data) execution. For example, when computing a vector addition the vector unit can compute all vector components in one step rather than one component after the other.

Program Structure Core element of programming with ArBB is the transition from ordinary C++ into ArBB space. See Figure 3.2 for an illustration.

![Figure 3.2.: Effect of the call() function](image)

The transition is triggered by issuing the \texttt{arbb::call()} function. It has two pairs of parantheses of which the first one takes a \textit{function name} denoting the function to be called on ArBB side. The second one is for the \textit{parameter list}.

ArBB space is where the parallelization takes place. It is a managed environment supporting garbage collection and further optimization of data structures in order to achieve good performance.

Custom functions and types can be passed over to \texttt{call()} but there are restrictions on them. One restriction is that data fields of a custom type must consist of ”scalar types, container types, or other valid user-defined types” (page 77 of [8]). ArBB offers replacement types for the C++ types, for example \texttt{i64} which is a type appropriate for 64-bit unsigned integers.

Implementation Issues The main reason why ArBB was rejected is because after investing reasonable efforts on research no way was found to implement our vectors of vector algebra in a suitable way. In general there are ArBB types which support multi-dimensional data structures (e.g. dense containers) but our vectors consist of columns with different data types. Dynamic allocation of columns using an array of pointers together with type casts is not possible since this is no ArBB type.

After asking in the official ArBB support forum a suggestion was to use dense containers of structs.\footnote{forum thread: http://software.intel.com/en-us/forums/topic/292439 (link validated on 16.2.13)} Whether this suggestion would have worked is not completely clarified. From the author’s perspective structs do not help because
of the unpredictable inner structure of vectors (variable length, column amount and column types). Again, this requires dynamic allocation of memory. It seemed not appropriate to do more research on this.

**Summary** A possible workaround to the type restrictions would have been to build a compiler instead of an interpreter. The compiler adds an additional execution step in which C++ is created with hardcoded vector sizes into it. In a second compilation step together with the ArBB library the binary is created and finally executed.

However after a first glance at OpenMP it was decided to switch over. With OpenMP one can write nearly unrestricted C++ programs. In this perspective OpenMP is very much the opposite of ArBB which has a big impact on the programming style. A possible downside is the shorter support for vector parallelization.

**Discontinuation of ArBB** According to the official webpage [7] (date: 26.2.2013) the ArBB project has been retired in October 2012. This happened after we decided to reject it.
3. Implementation Tools

3.4. JsonCpp

JsonCpp [9] is a C++ library for parsing and formatting data in the JSON data format.

JSON is short for “JavaScript Object Notation” and defines a textual data format. JSON is valid JavaScript syntax but has a good circulation outside of the JavaScript world. Vector algebra programs being input to the execution environment of this work are in JSON format.

In the implementation JSON and JsonCpp only occur inside of the input/output interfaces.
4. Implementation

The *execution environment* has been built as an interpreter for vector algebra programs. It follows a simple IPO model of input, processing and output. The *input* is a JSON-formatted file containing a vector algebra program; *processing* is the execution of this program; the *output* again is a JSON-formatted file and contains the computed result vectors. See Figure 4.1 for a graphical illustration.

![Figure 4.1.: Diagram of the execution environment](image)

Sections 4.1 and 4.2 are about how vectors and vector primitives have been implemented. The section about primitives includes data parallel programming issues.

Section 4.3 is about the internal components of the execution environment, how the components are related and what the control flow is.
4. Implementation

4.1. Vectors

In the OOP design of the vector types exists one abstract base class called \textit{Vector}. The five non-abstract vector types defined by vector algebra are derived from it. Figure 4.2 is a class diagram of the class relations including important data fields and methods. The naming scheme for vectors is as described in Section 2.2.

Important members of the base class are \textit{length()} and \textit{type()}. All vector data is stored within arrays of a certain length. \textit{Type()} returns an integer encoding the vector type.

![UML class diagram](image)

The diagram consists of two major branches: \textit{PropagationVector}, which contains two columns (\textit{p}_o and \textit{p}_n). And \textit{ValueVector}, which has a \textit{descriptor} column and an \textit{items} object containing the item columns.

However a positions column (part of the specification) is not implemented in ValueVector. It is implicitly given because all columns are implemented as arrays having 0-based indices. Positions which are 1-based can be derived by adding 1 to the current array index.

\textit{RenameVector} is modelled as a subtype of PropagationVector with the difference of having stricter conditions on the orderings of \textit{p}_o and \textit{p}_n. All RenameVectors can be casted to PropagationVectors.

And \textit{DescriptionVectors} having no item columns can be casted to ValueVectors which can have an arbitrary number of item columns.

However \textit{ScalarValue} is semantically completely different from ValueVector or even Vector (in vector algebra ScalarValue is considered as atomic value).
Nonetheless it inherits from ValueVector. Reasons are the similarities between them on layer of implementation. Both have a descriptor and columns of the six data types. Identical algorithms can be applied to both, for example adding two integers.

For these reasons, a ScalarValue is, on the implementation layer, a special ValueVector with a fixed length of one and a column count of \( \geq 1 \). In source code these conditions are included in the constructors.

A downside of this design decision is that casting from ScalarValue to ValueVector must be avoided. It would be a totally legal and even implicit operation in OOP perspective but not in perspective of vector algebra.

ScalarValue implements an additional method `getDescriptorA()` which returns the descriptor value as an integer (there exists exactly one entry in the descriptor array).

### 4.1.1. Item Columns

Item columns are required for implementing ValueVectors (and formally for its two descendants). Figure 4.3 contains a class diagram depicting the relations between `ValueVector`, `Items` and `Column` which is the abstract base class for item columns.

![UML class diagram of Items and its components](image)

Columns which belong to a common vector are bundled together inside an `Items` object. Each `Items` object has fields for storing the number of columns belonging to it (\( hsize \)) and an array which holds pointers to these objects (\( columns[] \)).

Every Item object is mapped to a ValueVector.
4. Implementation

4.1.2. Column Types

*Column* is an abstract base class for the actual six column types. See Figure 4.4 for the available types.

![UML Class diagram of available column types.](image)

*Column* itself defines generic attributes like *length* and *type*. There are additional type-specific member functions like *toDouble()* which does a conversion of integer to double values. Another one is *complement()* which computes the logical complement of boolean values. Certain vector primitives (integerToDoubleA, notPrim) and their lifted counterparts depend on these functions.

Five of the column types define arrays which contain the actual data, in the diagram named *intval[]*, *natval[]* and so on. These abbreviations are typedefs and with exception of *stringval* (see next section) the typedefs do a mapping onto the fundamental C types which have been introduced in Section 3.1.1. Another exception is the unit type which does not need to store any data. The type information itself is enough.

Table 4.1 lists the mappings used for implementation.

4.1.3. Strings Implementation

*Strings* are different from the other data types because they do not exist as a fundamental C type. It is a common way to store strings in char arrays. Another common way, if programming in C++, is to use the STL and its std::string class.
4. Implementation

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>C type</th>
</tr>
</thead>
<tbody>
<tr>
<td>intval</td>
<td>long int</td>
</tr>
<tr>
<td>natval</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>doubleval</td>
<td>double</td>
</tr>
<tr>
<td>boolval</td>
<td>bool</td>
</tr>
<tr>
<td>unitval</td>
<td>(does not occur)</td>
</tr>
<tr>
<td>stringval</td>
<td>(see section 4.1.3)</td>
</tr>
</tbody>
</table>

Table 4.1.: Typedefs for vector algebra data types

As table 4.2 illustrates, strings are double pointers when referencing arrays or single pointers when referencing single values, i.e. char arrays and objects.

Concerning the number of pointer layers there is no difference between the recently discussed two string variants. The issue is that there is a difference compared to the fundamental types: strings have generally one pointer layer more. This is an issue because it hinders the software developer from working in a convenient way with C++ template functions. For example when implementing a minimum search it is a good strategy to create exactly one template function which can carry out the computation for all available data types. Nowhere inside of the template function a type switch appears.

This problem was solved using the typedef construct. See Table 4.3 for the final list of typedefs (unit is not required). After applying the typedefs all data types share the same pointer depth (on an abstract layer).

Table 4.2.: Pointer depth for data types

As table 4.2 illustrates, strings are double pointers when referencing arrays or single pointers when referencing single values, i.e. char arrays and objects.

Concerning the number of pointer layers there is no difference between the recently discussed two string variants. The issue is that there is a difference compared to the fundamental types: strings have generally one pointer layer more. This is an issue because it hinders the software developer from working in a convenient way with C++ template functions. For example when implementing a minimum search it is a good strategy to create exactly one template function which can carry out the computation for all available data types. Nowhere inside of the template function a type switch appears.

This problem was solved using the typedef construct. See Table 4.3 for the final list of typedefs (unit is not required). After applying the typedefs all data types share the same pointer depth (on an abstract layer).

Table 4.3.: Typedefs for vector algebra data types

Another solution would have been to encapsulate all fundamental data types in container objects. For performance reasons this solution was rejected.
4. Implementation

4.1.3.1. Charset

Part of the specification of vector algebra is handling of strings in unicode format. Unicode is a modern and comprehensive standard for digital representation of characters. It contains more than 100,000 character numbers each of them being unique for a certain character from the most important alphabets in the world.

UTF-8 (part of the unicode standard) is the encoding format used by the compilation chain. The encoding of a single character has a variable length of 1 to 4 bytes. Compared to encodings with constant length (like extended ASCII where each character has 1 byte) this makes handling strings more difficult.

However there is an useful characteristic of UTF-8; quoting from [10]:

The byte-value lexicographic sorting order of UTF-8 strings is the same as if ordered by character numbers.

This property is exploited within this work and avoids any need for character interpretation. The minimum/maximum/equals comparisons which are carried out can be handled on byte level. When sorting strings in this manner, the resulting order will be as implied by the character number order in the unicode standard.

In the implementation of the String class (as occured in Table 4.3) character arrays are used to store the UTF-8 data.

4.1.3.2. Reference Management

Vector algebra makes strings generally immutable. Two integers for example may get added and result in a third integer different from the two original ones. But two strings will never get concatenated or produce any kind of a different string. There is just an initial set of strings being part of the input program.

To exploit that and because strings can be long and thus copying them becomes very time-consuming in general, the execution environment does not copy strings itself but only pointers to strings. For example when sorting a vector containing strings the sorted output effectively contains the same strings as the input vector. All that has to be done by the execution environment is to copy the pointers and increase a reference counter per string.

On the other hand when deleting a vector, the reference counters for the according strings get decremented. And if a reference counter goes down to 0, the string object itself can be deleted.

Listing 4.1 contains the code for reference increment. The semaphor responsible for mutual exclusion of threads is a local variable per string. Making it local minimizes the risk for threads to get blocked.
4. Implementation

In Listing 4.2 is the code responsible for reference decrements. Compared to the increment it additionally contains the delete statement which is executed if the reference counter reaches 0. Deleting the current object is safe if the control flow leaves the object directly afterwards. For this reason the delete statement is not within the critical code region. If it were inside, the control flow would try to execute line 5 (after deletion) and access a semaphor that does no longer exist.

If the reference counter reaches 0 there will be no calls of refUp() afterwards. This is a property of the control flow: first produce new vectors, afterwards delete the old ones. There is no random access. For this reason there is no race condition when placing the delete statement outside of the synchronized code region.

Listing 4.1: Implementation of String::refUp()

```
String* String::refUp() {
    sem_wait(&mutex_);
    references_++;
    sem_post(&mutex_);
    return this;
}
```

Listing 4.2: Implementation of String::refDown()

```
void String::refDown() {
    sem_wait(&mutex_);
    references_--;
    sem_post(&mutex_);
    if (references_ == 0) {
        delete this;
    }
    return;
}
```
4. Implementation

4.2. Vector Primitives

This chapter is about how the vector primitives have been implemented. A subset of them will be discussed here.

Five categories have been created to group primitives of the same parallelization type together.

4.2.1. Parallelization Categories

Atomic     No parallel execution.

Fragmented Loop     Simple parallelization using for loops and the for directive of OpenMP.

Shared Variables     During execution threads have to share data. The parallel directive is utilized.

Sorting     These primitives deal with sorting of vectors.

Pseudo     No actual computation is carried out. These primitives are just for data handling.
4. Implementation

4.2.2. Atomic Primitives

This category does not deal with parallelization. There are two reasons not to do it.

The first one is because too few computation steps are carried out. A good example for this case is `singletonDescr` which writes exactly one descriptor value and besides of object instantiation does nothing more.

The second reason is about vectors of type ScalarValue and operations on columns. It is a design decision to relinquish parallelization when it is only about columns each of them having length of 1. Downside of this decision is that processing vectors consisting of millions of columns (each of them of length 1) will not exploit multiple CPUs. Nonetheless millions of columns is considered as a corner case. Advantage of this decision is to have only one matter left to focus on: parallel handling of vector components.

Primitives List

- `compExpr2`
- `intToDoubleA`
- `lengthA`
- `notPrim`
- `pairA`
- `projectA`
- `singletonDescr`
4. Implementation

4.2.2.1. singletonDescr

SingletonDescr has no arguments which makes it a constant function. It creates the DescriptionVector 4.1 shown below.

\[
\begin{pmatrix}
  d & p \\
  1 & 1
\end{pmatrix}
\]

(4.1)

In code, a descriptor array descr is allocated and in the second step a vector object res is instantiated. See Listing 4.3 for the complete source code of singletonDescr.

```cpp
const DV* NullaryOp::singletonDescr () {
    bigindex* descr = new bigindex[1];
    descr[0] = 1;
    DV* res = new DV(descr, 1);
    return res;
}
```

Listing 4.3: Implementation of singletonDescr

All code listings make use of bigindex (component ids), smallindex (column ids) and nodeindex (node ids) variables. These are again typedefs and mapped onto different integer types of appropriate size.

Function Prototype  For this very first example, the relevant part of the according header file is provided in Listing 4.4. The remaining primitives are implemented accordingly.

For each primitive arity, a separate class was created. In this case, it is NullaryOp. These classes have no constructors or variables and do not get instantiated. The classes only contain public methods and each of the prototypes inside of the header file declares a vector primitive. All of them are static which makes the function executable without object context. Another keyword frequently used is const. It makes the return value (in this case a pointer to a DescriptionVector) a constant. This is desired for vectors of vector algebra since they do not change after their creation.

```cpp
class NullaryOp {
    public:
    
    static const DV* singletonDescr();
    
    // ... remaining prototypes
    // no class variables
};
```

Listing 4.4: Prototype of singletonDescr in the nullary class
4. Implementation

4.2.2.2. lengthA

LengthA returns the length $l$ of an input vector $a$. In our implementation a dedicated length variable exists for all vectors, which makes this a trivial operation. The length $l$ is of type natural and gets casted into an integer $l_{\text{int}}$ as specified by the vector algebra. See Listing 4.5 for the C++ code.

```
const VS* UnOp::lengthA(const DV* a) {
    bigindex l = a->length();
    intval l_int = (intval) l; // cast to integer
    return VS::createFromIntval(l_int);
}
```

Listing 4.5: Implementation of lengthA
4. Implementation

4.2.3. Fragmented Loop Primitives

_Fragmented Loop_ refers to the execution pattern for vector primitives of this category. An input array gets fragmented into multiple equal-sized ranges and each range can be processed by a different CPU (thread). Each thread works on its dedicated array range completely on itself. No synchronization or sharing of variables is needed.

**Primitives List**

- compExpr2L
- descToRename
- distLift
- distPrim
- intToDoubleL
- notVec
- pairL
- projectL
- reverseA
- segment
- selectPos
- toDescr
- unSegment
4. Implementation

4.2.3.1. toDescr

toDescr turns a ValueVector \( a \) into a DescriptionVector by copying the descriptor and ignoring all other columns. The C++ code for doing this is printed in Listing 4.6. The variable \( \text{source} \) is the descriptor array of \( a \) and is about to be copied. On line 5 an equally-sized array \( \text{res} \) is defined which will be part of the result vector. The copying takes place in the for loop which has been parallelized using an OpenMP compiler directive.

```cpp
const DV* UnOp::toDescr(const VV* a) {
    const bigindex* source = a->getDescriptor();
    bigindex length = a->length();
    bigindex* res = new bigindex[length];

    #pragma omp parallel for
    for (bigindex i=0; i<length; i++) {
        res[i] = source[i];
    }

    return new DV(res, length);
}
```

Listing 4.6: Implementation of toDescr

More primitives of this category will be not discussed. All other primitives follow the same pattern.
4. Implementation

4.2.4. Shared Variables Primitives

For implementation of vector primitives from this category the OpenMP parallel directive is used. It provides more flexibility than the for directive from the last category and can make threads work closely together on a fine-grained level.

However not all primitives of this category share variables (as the category name suggests). DistDesc and propRename are the two exceptions which have been put into this category because they performe better when using the parallel directive (for distDesc, a modulo operation can be moved outside the for loop).

The remaining vector primitives do variable sharing between their threads. For example in the implementation of vecSum each thread computes the sum for a disjunct subset of the vector components. In the end all threads together compute a total out of their individual sums. To do this, the total variable is shared between the threads.

Primitives List

- append
- combineVec
- distDesc
- falsePositions
- lengthSeg
- propFilter
- propRename
- propReorder
- restrictVec
- reverseL
- selectPosLift
- vecMax
- vecMin
- vecSum
- vecSumLift
- vecMaxLift
- vecMinLift
- zipL
4. Implementation

4.2.4.1. vecSum

VecSum is an example for doing an additional merging step before leaving the parallel code region. Also it can process three different data types, hence a template function is used.

Listing 4.7 contains the first part of vecSum. On line 2 an assertion is used to make sure ValueVector a has exactly one data column - which then is copied into input (more precisely, only its pointer is copied). Its type is determined and fed into the switch statement. Inside there, a template function vecSumExec is called and the appropriate type parameters are provided (data type T and column type C).

```cpp
const VS* UnOp::vecSum(const VV* a) {
    assert(a->hsize() == 1);

    const Column* input = a->getItems()->getColumn(0);
    int type = input->type();

    Column* res;

    switch (type) {
        case TYPE_INT:
            res = UnOp::vecSumExec<intval, ColumnInt>(input);
            break;
        case TYPE_NAT:
            res = UnOp::vecSumExec<natval, ColumnNat>(input);
            break;
        case TYPE_DOUBLE:
            res = UnOp::vecSumExec<doubleval, ColumnDouble>(input);
            break;
        case TYPE_ERROR:
        default:
            throw xtype(type);
    }

    return new VS(new Items(res, 1), 1);
}
```

Listing 4.7: Implementation of vecSum

In Listing 4.8 the template function is presented. The type variables T and C have been set by the outer function. The first statement on line 3 casts the generic column into a concrete column, for example col could now be a ColumnInt.

Line 7 initialises the result variable sum with 0 (the neutral element for addition). Please notice, that sum is declared outside of the parallel for loop. Normally this means that all threads of the following parallel code region have access to this variable (and the danger of race conditions arises). However
4. Implementation

In this case the **reduction** clause is applied to `sum` and triggers another behaviour. Each thread will have its private copy of `sum` and therefore carries out the summations at line 11 without any danger of race conditions. In the end all private `sum` variables will be merged (correctly synchronized) using the specified operator of the reduction clause on line 9 (+).

At the end of the template function `sum` will be put into a result column object and returned.

```cpp
template <class T, class C>
static C* vecSumExec(const Column* input) {
    const C* col = (const C*) input;

    bigindex length = col->length();
    const T* data = col->getData();
    T sum = 0;

    #pragma omp parallel for reduction(+:sum)
    for (bigindex i = 0; i < length; i++) {
        sum += data[i];
    }

    T* res = new T[1];
    res[0] = sum;

    return new C(res, 1);
}
```

Listing 4.8: Implementation of vecSumExec
4. Implementation

4.2.4.2. vecMin and vecMax

The implementation of vecMin and vecMax is similar to the one of vecSum but the reduction clause of OpenMP does not support minimum or maximum operators (recent version of OpenMP support them, but the development environment did not).

What we are going to discuss is how to implement manually what an OpenMP reduction clause does automatically. Again a template function is used but this time it deals with all six data types. Moreover vecMin and vecMax are merged into one function.

Listing 4.9 contains the code for the entry function of vecMax. The function itself does nothing else than calling vecMin with an additional false parameter. This parameter will let vecMin behave differently and compute the maximum.

```cpp
const VS* UnOp::vecMax(const VV* a) {
  return vecMin(a, false);
}
```

Listing 4.9: Implementation of vecMax

In Listing 4.10 is the code for vecMin. In the function header minmax shows up which has a default parameter. If true, the minimum is computed; in the other case the maximum.

The rest of vecMin() is similar to vecSum() which has already been discussed. New is the treatment for unit inputs. Having no distinguishable data values the only possible result can be returned at once (line 27: the 1 is for the column length). All remaining five types get handled by the template function vecMinMaxExec().
4. Implementation

```cpp
const VS* UnOp::vecMin(const VV* a, bool minmax=true) {
    assert(a->hsize() == 1);
    assert(a->length() > 0);
    const Column* input = a->getItems()->getColumn(0);
    int type = input->type();
    Column* res;
    switch (type) {
        case TYPE_INT:
            res = UnOp::vecMinMaxExec
                <intval, ColumnInt>(input, minmax);
            break;
        case TYPE_NAT:
            res = UnOp::vecMinMaxExec
                <natval, ColumnNat>(input, minmax);
            break;
        case TYPE_DOUBLE:
            res = UnOp::vecMinMaxExec
                <doubleval, ColumnDouble>(input, minmax);
            break;
        case TYPE_BOOL:
            res = UnOp::vecMinMaxExec
                <boolval, ColumnBool>(input, minmax);
            break;
        case TYPE_STRING:
            res = UnOp::vecMinMaxExec
                <stringval, ColumnString>(input, minmax);
            break;
        case TYPE_UNIT:
            res = new ColumnUnit(1);
            break;
        case TYPE_ERROR:
            default:
                throw xType(type);
            }
    return new VS(new Items(res, 1), 1);
}
```

Listing 4.10: Implementation of vecMin

Before discussing the template function, another issue is going to be treated. As described in Section 4.1.3, strings have one pointer layer more than fundamental C types. Accordingly this requires one dereference step more when accessing a string value (the strings we are talking of here are objects imple-
4. Implementation

implementing overloaded operators, hence there is no to think about what happens on character layer right now). For example when applying a comparison operators like < it can be used on two integer variables directly by writing \(a < b\) (\(a, b\) are of type `intval`). The same can be done with variables of type double, boolean, or natural. But not on two string pointers. If a comparison operator is applied on string pointers, it will not compare the strings inside but the two memory addresses. What we need is an automatic dereferenciation if and only if we are comparing strings.

The perfect solution to this issue has not been found because inside a template function different types are all treated in the same way (which is the concept of a template function). Avoiding the template function for strings is not a good option too, since this would result in writing redundant code.

The solution that has been selected is to create overloaded functions for all combinations of comparison operators and types. Example implementations can be found in Listing 4.11. In the listing two implementations of `CompDeref::gt()` can be found. Each of them with the same number of arguments but with different type signatures. Depending on the argument types, values \(a\) and \(b\) will get dereferenced or not. An additional argument `dummy` has been introduced to make sure no implicit casting occurs.

```cpp
1 bool CompDeref::gt(ColumnInt* dummy, intval a,
2               intval b) {
3       return a > b;
4  }

5 bool CompDeref::gt(ColumnString* dummy,
6             stringval a, stringval b) {
7       return (*a) > (*b); // dereferencing
8  }
```

Listing 4.11: Examples for `gt()` implementations

Listing 4.12 contains the code for `vecMinMaxExec()`. This is the first example for having different parallel code regions. Additionally it contains the switches for the `minmax` variable (if `true`, the minimum is computed; if `false`, the maximum) and it contains code required for string treatment. The relevant lines for string treatment are line 11 where the `dummy` variable is created and line 33 where the reference counter for the minimum (or maximum) string is incremented by one.

Concerning parallel programming the conceptual structure is mostly the same as it was for the implementation of `vecSum` but this time more explicit. Outside of the parallel code region we initialize the variable `valShared` which will be shared between all threads. It is initialized with the first data value of the input array (the array has at least length 1 which is ensured by an assertion, see Code Listing 4.10). It does not matter which value is used for initialization. It just needs to originate from the `data` array.

The variable `valPrivate` is initialised in the same way (line 15) but being inside the parallel code region each thread creates its own variable and does not
share it with the other threads.

The first subsection in the parallel region is a parallelised for loop in which each thread searches the minimum (maximum) for the array range the thread has been assigned to work on. The `nowait` clause lets each thread continue individually after it is done with its part of the loop.

The second subsection makes use of the `critical` directive of OpenMP. It makes all threads execute this subsection one after the other (synchronized). In this section each thread merges its private result into the shared result variable `valShared`. Synchronization is essential to avoid a race condition on `valShared`.
4. Implementation

template <class T, class C>
static C* vecMinMaxExec(const Column* input, bool minmax) {
  // prepare input vars
  const C* a = (const C*) input;
  const T* data = a->getData();

  // prepare result vars
  T init = data[0];
  T valShared = init;
  C* dummy; // do not read from

  #pragma omp parallel
  {
    // intermediate result per thread
    T valPrivate = init;

    #pragma omp for nowait
    for (bigindex i=0; i<a->length(); i++) {
      if (minmax == true && CompDeref::gt(dummy, valPrivate, data[i])) valPrivate = data[i];
      if (minmax == false && CompDeref::lt(dummy, valPrivate, data[i])) valPrivate = data[i];
    }

    #pragma omp critical
    {
      // merge intermediate results
      if (minmax == true && CompDeref::gt(dummy, valShared, valPrivate)) valShared = valPrivate;
      if (minmax == false && CompDeref::lt(dummy, valShared, valPrivate)) valShared = valPrivate;
    }
  }

  // closing parallel bracket
  T* res = new T[1];
  res[0] = valShared;
  String::refUpIfString(valShared);

  return new C(res, 1);
}

Listing 4.12: Inner function of vecMin and vecMax
4. Implementation

4.2.4.3. Advanced for Loop

Most of the primitives of the shared variables category make use of the parallelization pattern described in this subsection.

When working with vectors, the frequent occurrence of for loops is quite natural. However the for directive of OpenMP is too inflexible for most vector primitives of this category. The replacement method is to use the parallel directive and based on it manually implement a more flexible parallel for loop.

Listing 4.13 contains the relevant code parts which have been repeatedly used for the implementation. Input parameter is the length variable which denotes the number of iteration steps to be carried out. Outside of the parallel code region three (empty) variables are declared: threadNum stores the number of threads. threadOffset and threadLength are arrays each of them containing one value per thread. They tell each thread at which index to start its computations (threadOffset) and how many iterations are to be done (threadLength).

Within the parallel region a single directive shows up. The corresponding code will be executed by one thread only: at first threadNum is set to the current number of threads. The function newThreadRanges() computes based on length and threadNum an equal workload for each thread. The algorithm is not provided here but it will compute appropriate offset and length values for each thread and write them into threadOffset and threadLength (and previously allocate arrays of the fitting size). In corner cases threadNum is updated. For example if threadNum = 4 and length = 1 then only 1 thread can do some work, the others are idle. Making sure they do not interfere is realised on lines 17+18.

If a certain thread myId has work to do, it will start with preparational tasks, then executes the loop (which loops over the thread-specific range) and in the end executes merging tasks if existing.
4. Implementation

```c
// provided variable: length

// threading vars
int threadNum;
bigindex* threadOffset;
bigindex* threadLength;

#pragma omp parallel
{
    #pragma omp single
    {
        threadNum = omp_get_num_threads();
        newThreadRanges(threadNum, length,
                        threadOffset, threadLength);
    }

    int myId = omp_get_thread_num();
    if (myId < threadNum) {
        // initialise thread-private variables
        for (bigindex i=threadOffset[myId];
             i<threadOffset[myId]+threadLength[myId];
             i++) {
            // actual work is done here
        }
        // merging steps (optional)
    } // myId bracket
} // parallel bracket
delete[] threadOffset;
delete[] threadLength;
```

Listing 4.13: Parallelization pattern
4. Implementation

4.2.4.4. distDesc

DistDesc is an example for an application of the advanced for loop described in the last section. DistDesc replicates the contiguous components of an input ValueVector a for a certain number of iterations (determined by the length of DescriptionVector b).

Listing 4.14 contains the nearly complete source code for distDesc. Not listed are the treatment of the corner case of empty input vectors and the delete statements for cleaning up temporary variables. In the function header the Entity type shows up which is used for bundling multiple return vectors. In this case these are a ValueVector and a PropagationVector.

The most important variable within the preparation statements is res which will contain the replicated data. It is created based on sourceItems (same data column types) but completely empty. For this vector primitive the length of the result vector (newlength) can be computed right at the beginning of the function. It is used to initialize res.

The code between the variable preparations and line 27 is as explained in the previous section. On line 27 we are in the parallel code region and are about to prepare the private variables for each thread. The two variables innerOffset and segId are important meta information for execution of the for loop on line 31. The loop iterates over all components of the result vector and fills them with values. However in general there will be threads which have the assignment to start at a certain position in the middle of the result vector and to copy the components which belong into a certain range beginning at this starting position. Doing that requires two pieces of information: 1) where to start reading the components of vector a (this is called innerOffset) and 2) the number of the current replication cycle, which is needed for writing the descriptor column (this is called segId).

The variable innerOffset is the result of a modulo operation: arguments are the start position for this thread and the length of vector a. For segId it is calculated backwards to where the last copy cycle started and then a division by the length of vector a will return the number of the copy cycle. In the end an increment is done to convert it into a segment number (which are 1-based).

In the loop res, descriptor and p_o are written. These are the three result arrays which are the assembly parts for the result vectors. The member function copyRow() has three arguments: sourceItems (the items object to copy from), innerOffset (source position of the component to be copied) and i (target position of the component to be copied).

The second part of the loop increments innerOffset until fresh copy cycles start. In the end of the function the two result vectors are assembled and returned.

If the length of input vector a is 0, the division on line 29 (and the modulo operation) will trigger a division by zero error. The appropriate corner case handling is not provided in the code listing.
4. Implementation

```cpp
const Entity* BinOp::distDesc(const VV* a, const DV* b) {
  // prepare input variables
  bigindex lengthA = a->length();
  bigindex lengthB = b->length();
  bigindex newlength = lengthA * lengthB;
  const Items* sourceItems = a->getItems();
  Items* res = sourceItems->newEmptyCopy(newlength);
  bigindex* descriptor = new bigindex[newlength];
  bigindex* po = new bigindex[newlength];

  // prepare threading variables
  bigindex* threadOffset;
  bigindex* threadLength;
  int threadNum;

  #pragma omp parallel
  {
    #pragma omp single
    {
      threadNum = omp_get_num_threads();
      newThreadRanges(threadNum, newlength,
                      threadOffset, threadLength);
    }

    int myId = omp_get_thread_num();
    if (myId < threadNum) {
      bigindex innerOffset = threadOffset[myId] % lengthA;
      bigindex segId = (threadOffset[myId] - innerOffset)
                        / lengthA + 1;

      for (bigindex i=threadOffset[myId];
           i<threadOffset[myId]+threadLength[myId]; i++) {
        res->copyRow(sourceItems, innerOffset, i);
        descriptor[i] = segId;
        po[i] = innerOffset + 1;
        innerOffset++;
        if (innerOffset >= lengthA) {
          innerOffset = 0;
          segId++;
        }
      } //myId bracket
    } //parallel bracket

  PV* pv = new PV(po, newlength);
  VV* vv = new VV(res, descriptor, newlength);
  return new Entity(vv, pv);
}
```

Listing 4.14: Implementation of distDesc (relevant parts)
4. Implementation

4.2.4.5. restrictVec

RestrictVec is a similar but more advanced example and thus only a high level overview is provided here. Parameter \( a \) is a vector containing a column of booleans. It is used for filtering another vector \( b \) of the same length. If the according boolean in \( b \) is \textbf{true}, the component from \( a \) is copied into the result vector.

When starting to execute this primitive, it is unknown what the length \textbf{newLength} of the result vector will finally be. However the length is required to allocate result arrays of the appropriate size before copying the values. For this reason, two loops will be implemented: a first one iterates over the input vector and determines \textbf{newLength}. Then the result vector can be created. In a second loop the data values are copied into the new vector.

Additionally an optimization has been included to avoid a complete second iteration over the input vectors. The optimization is, to copy the index numbers of all not filtered components into a temporary array \textbf{temp}. In the second loop these indices are used to copy the corresponding components directly from the source to the result vector. In general \textbf{temp} will have less entries than the length of the input vector is and this makes the execution faster (if nothing is filtered, the execution will be slower).

The source code for this vector primitive can be found in Appendix B.
4. Implementation

4.2.5. Sorting Primitives

Vector primitives of this category reorder of vectors. Unique and uniqueL actually remove duplicate vector components but sorting is a common strategy to implement duplicate recognition. This has the advantage that four primitives can make use of the same algorithm.

The sorting algorithm we selected for the backend is quicksort. Its implementation is short and can be parallelized well. Mergesort was another candidate with the advantage of being a stable algorithm.

A custom implementation of quicksort was done rather than using a library. This has the advantage of not being dependent on the interface of the library. Moreover, the implementation can be done consistently with the vector primitives by using again OpenMP.

Primitives List

- unique
- uniqueL
- sortWith
- groupBy
4. Implementation

4.2.5.1. Parallel Quicksort

Our quicksort [11] implementation is based on nesting of parallel code regions found in [12]. However the actual implementation is different.

Listing 4.15 contains the relevant code parts of the parallel quicksort algorithm. The semaphore busy is used for counting (limiting) the number of threads. It can for example be initialised with 8 (for 8 CPUs) and accordingly a maximum of 8 locks (threads) will be allowed at the same time. The if statement beginning on line 7 (see below - Reducing Locking Overhead - for details) is responsible for creation of new threads. The conditions inside of the if statement are used to acquire a lock. On failure, the current thread does not get blocked but it can go ahead and do the sequential recursion.

If the lock could be acquired, a new thread is allowed to be created. This is done below in the parallel code region which forms a team of 2 threads (which is 1 thread more than before). Inside the parallel region an OpenMP sections directive is used to assign different tasks to the two different threads. In our case, the different tasks are the two recursive function calls of quicksort.

Thanks to the nowait clause on line 22, the first thread which returns from recursion is allowed to leave the sections code block independently from the other thread which is probably still working (it is improbable that both threads get finished right at the same time). The first thread (and only the first) will enter the single code block below and give back its lock. The lock can then be used somewhere else in the recursion tree to create a new parallel branch.

It is important to return the lock immediately because the thread being done at first will now have to wait (sleep) at the end of the parallel section until the second thread is done, too.

Finally only the master thread will leave the parallel section. It still holds one lock (as it held when it entered the function) and will return with it.

Reducing Locking Overhead The if statement for acquiring new threads consists of two parts. On line 7 the counter variable of busy is investigated without synchronization (no locking overhead but perhaps the return value is wrong or outdated). If an available thread is indicated, only then the condition of line 8 will be tested and synchronization will happen.

This saves the threads from doing (mostly) useless locking attempts.

Pivot Selection Three elements of the input array are selected (first, middle, last) and the median is computed.
4. Implementation

```c
void Sort::qsRecursion(/* arguments not printed */) {
    // work: select pivot and do partitioning step

    // try to acquire an additional lock
    bool threadAvailable = false;
    if (busy.__align > 0 // unsynchronized
        && sem_trywait(&busy) != -1) // synchronized
        threadAvailable = true;

    // if not available: execute sequentially
    if (!threadAvailable) {
        qsRecursion(/* left */);
        qsRecursion(/* right */);
        return;
    }

    // if available: nest
    #pragma omp parallel num_threads(2)
    {
        #pragma omp sections nowait
        {
            #pragma omp section
            {
                qsRecursion(/* left */);
            }
            #pragma omp section
            {
                qsRecursion(/* right */);
            }
        } // sections bracket

        #pragma omp single
        {
            sem_post(&busy);
        }
    } // parallel bracket

    Listing 4.15: Partial code for parallel quicksort
```
4. Implementation

4.2.5.2. Comparison Interface

Comparisons for sorting are done on the level of vector components: two components of the same vector are compared with each other. This includes values in item columns (with \(i_1\) being the primary sorting criteria, \(i_2\) the second, etc.) and the positions column (which is the least relevant sorting criteria).

Descriptors are not considered for sorting itself. However the sorting algorithm we designed can optionally be invoked on a certain segment of a vector only. If doing this repeatedly and for all available segments it will result in a sorting operation considering the descriptor column as most relevant sorting criteria (for example \(uniqueL\) needs the descriptor criteria).

Because of these considerations the comparison interface has been built into the \texttt{Items} class where item columns and position ids are available. Listing 4.16 contains the C++ code for a \texttt{gt} comparison. Input parameters \(a\), \(b\) are the array indices of the two components to be compared. A for loop iterates over the available item columns and triggers a comparison for the requested array indices \(a\), \(b\). In case of equality the next column is investigated. If all data values are equal, the array indices are compared on line 6 (which effectively is a comparison of the position values).

```cpp
1 bool Items::rowIsGt(bigindex a, bigindex b) const {
2     for (smallindex i=0; i<hsize_; i++) {
3         if (((data_[i])->valuesAreEq(a, b)) continue;
4         else return (data_[i])->valueIsGt(a, b);
5     }
6     return a > b;
7 }
```

Listing 4.16: Example for comparison implementations
4. Implementation

4.2.5.3. unique

Unique filters duplicate tuples but keeps the first occurrence of a tuple. Its implementation consists of three steps of which a pseudocode version is presented in Listing 4.17.

The first major step is sorting which has been discussed already. The result will be an integer array \( s \) containing the position ids for a sorted vector \( a \).

In the second step an iteration over \( a \) is done and for each component \( c \) a binary search is carried out. It returns the first occurrence of \( c \). If the first occurrence is equal to the position of \( c \) in \( a \) it has passed the filter and the position is stored in \( \text{temp} \).

The third step is to create the result vector by copying the values which have passed.

All three computation parts are implemented in parallel. Theoretical runtimes in \( O \) notation are (\( n \) is the vector length):

- quicksort: \( O(n \cdot \log(n)) \) (average case)
- binary search: \( O(n \cdot \log(n)) \) (additional factor \( n \): each component is searched)
- copying: \( O(n) \)

For quicksort the average runtime has been considered (binary search and copying are worst case). This is because quicksort usually performs well in practice. Its worst case complexity of \( O(n^2) \) occurs for example when using a bad strategy of pivot selection (on sorted inputs).

Overall it makes a complexity of \( O(n \cdot \log(n)) \) for the unique operation.
function unique(ValueVector a)

// 1) sorting
int[] s = sort(a);
int l = 0;
int[] temp = int[a.length];

// 2) binary search
for (int i=0; i<a.length; i++):
    component c = a.get(i);
    int posInA = binsearchFirst(s, c);
    if (posInA == i):
        temp[l] = posInA;
        l++;

ValueVector res = a.cloneWithEmptyColumns(l);

// 3) copy
for (int i=0; i<l; i++):
    res.set(i) = a.get(i);

end function

Listing 4.17: Pseudo code for unique implementation
4. Implementation

4.2.6. Pseudo Primitives

The *pseudo primitives* category is documented here for completeness. These vector primitives are for data management and do not actually compute something.

List:

R1
R2
R3
constructLiteralValue
constructLiteralTable

4.2.6.1. R[1-3]

Some vector primitives return multiple vectors as result (for example distDesc returns two vectors). In general there are 1, 2 or 3 result vectors. If a vector out of the result is to be passed to another primitive, the selection on the layer of vector algebra is carried out using an R* primitive.

On the implementation layer, an additional container class *Entity* has been developed. It stores a variable amount of result vectors and offers the required getter function for accessing them individually. As an example the implementation of R3 is provided in Listing 4.18.

After execution of R3, a new Entity object exists which only contains a copy of the third vector of the original Entity object. For consistency reasons, vector results containing only one vector are also put into Entity objects.

```cpp
const Entity* UnOp::r3(const Entity* a) {
    const Vector* v = a->getThird()->newCopy();
    return new Entity(v);
}
```

Listing 4.18: Implementation of R3

4.2.6.2. constructLiteralTable/Value

*ConstructLiteralTable* and its counterpart *constructLiteralValue* are responsible for creating vectors from the raw vector algebra program input. Listing 4.19 contains the C++ code for creation of a ValueVector from its assembly parts. The assembly parts are in our implementation generated by the JSON parser. They consist of the Items object, the descriptor array and its length.
4. Implementation

```cpp
const VV* NullaryOp::constructLiteralTableCompute(
    Items* argItems, bigindex* argDescriptor,
    bigindex argLength) {
    VV* res = new VV(argItems, argDescriptor, argLength);
    return res;
}
```

Listing 4.19: Implementation of constructLiteralTable
4. Implementation

4.3. Control Flow

Execution of vector algebra programs splits up into two phases. Both of them are described in the following. The program parts discussed here are all executed sequentially. Parallelisation only exists on the layer of vector primitives where the time-consuming computations are done.

The advantages are that developing sequential code is in most cases simpler and no locking overhead occurs.

4.3.1. Phase One, Initialisation

The first phase is illustrated in Figure 4.5. Central part is the JSON parsing which instantiates two objects and one important array.

![Figure 4.5.: Components of phase one](image)

Node List is the array and contains the node ids which are to be computed and are the result of the vector algebra program. These ids are found in a preamble being the first part of the JSON-formated data (called the shape).

Instructions is basically a copy of the input program but without JSON formatting. Each computation node of the program consists of a list of references which determine the input vectors and a litteral identifier for the vector primitive to be executed (also some vector primitives posess non-vector parameters which have to be treated; they are not illustrated here).

However the parser does not blindly copy everything. It recursively iterates trough the input program beginning with the nodes found in the shape and then following the references to all child nodes required for computation (graph cycles do not occur). Doing it this way has two effects: first all superfluous nodes (artifacts from the compilation chain) are ignored. Second the parser creates a references table which contains for each occurring node the number of outgoing references. This table will in phase two be used for memory management.

Vector Cache (the second object) is at creation nearly empty. Its main constituent is an array of pointers to node results and gets initialized with the
4. Implementation

appropriate size. The only thing the parser puts into the vector cache is the references tables.

4.3.2. Phase Two, Execution

In phase two the actual computation of vectors takes place. The flow chart of Figure 4.6 provides an overview of the execution cycle. The initial node ids used to start the process have been found in phase one in the shape and are now used to trigger the computations. For every computation of a certain node, it is first tested if the result already exists in the cache. If not, the according instructions are used to fetch the required child nodes (here recursion occurs) and then call the requested vector primitive to compute the node.

As mentioned in the description of phase one, the references table is stored in the Cache object and is used for memory management. For each access to a certain node result, the Cache decrements its reference counter. If it reaches zero, the result is deleted (timing issues have to be considered, not illustrated here).

The main code for the implementation of phase 2 is printed in Listing 4.20. The omitted parts are annotated. A switch statement has been used to distinguish between vector primitives of different arity. For each arity, a dispatch() function is implemented (called on lines 11, 18, and 31). Inside of dispatch() is another switch which branches into the parallel implementations of the actual vector primitives. The Instructions object is passed over (omitted in the listing) and provides all the necessary information for dispatch().

For each vector primitive an additional helper function exists where type checks and conversions (between Entities and Vectors) are carried out. The details are not documented here.
4. Implementation

Figure 4.6.: Flowchart diagram of phase two
const Entity * compute(nodeindex nid /* ... */) {
    const Entity * res = cache->get(nid);
    if (res != 0)
        return res; // cache hit

    int primClass = instructions->getChildrenCount(nid);
    switch (primClass) {
        case 0:
            res = NullaryOp::dispatch(nid /* ... */);
            break;

        // case 1 omitted

        case 2: {
            nodeindex childOneNid = instructions->getChildOneId(nid);
            nodeindex childTwoNid = instructions->getChildTwoId(nid);
            const Entity * childOne = compute(childOneNid /* ... */); // recursion
            const Entity * childTwo = compute(childTwoNid /* ... */);
            res = BinOp::dispatch(nid, childOne, childTwo /* ... */);

            // here delete is triggered if refcount=0
            cache->refDown(childOneNid);
            cache->refDown(childTwoNid);
            break;

        // case 3 omitted
        }
    }
    cache->put(nid, res);
    return res;
}

Listing 4.20: Main program code of phase two
5. Conclusion

The in memory execution environment has been implemented. It is able to do an in memory execution of vector algebra programs. The vector primitives make use of data-parallelism and exploit multi-CPU computer systems.

Using OpenMP as a tool for writing data-parallel algorithms has turned out as a good decision. Unlike ARBB the full flexibility of C++ programming is maintained and exploited (e.g. pointers and OOP).

We implemented a special treatment for strings. They have been brought onto the same abstraction layer as fundamental data types. This was carried out without encapsulating non-string data types into additional container classes. The overhead for container management and additional pointer dereferencing is avoided.

5.1. Future Work

Tests for correctness are not done completely. The execution environment described in this document is still in beta phase. All features for running vector algebra programs are implemented and first tests have been carried out. A program consisting of more than 80 nodes has been executed successfully on a double core CPU. An example run can be found in the Appendix.

Performance evaluation of the execution environment is not done yet. Additional performance improvements are to be implemented (see next section). To exploit vector units of modern CPUs the influence of compilation flags will be evaluated.

Afterwards the execution environment will be used to do research on the performance behaviour of the vector primitives.

5.1.1. Performance Improvements

Integrate R[1-3] into Reference Management. The pseudo primitives R[1-3] copy a certain result vector (contained in an Entity object) into a new entity object in which the new vector will be the only one. The new entity is then ready to be fed into a vector primitive.

Instead of copying the whole vector it would be faster to only copy its pointer instead. To achieve this a better reference management for entities is required. As described in Section 4.3.2, all node results which are not referenced any longer are deleted (including the vectors inside them). If one of the entities containing a vector which is pointed at from multiple entities gets deleted, invalid pointers will remain.

One solution could be to add a reference pointer for each entity. If one of its vectors is copied, the reference pointer is increased and will protect this entity
5. Conclusion

from being deleted while its vectors are still in use. Accordingly decrements are performed when one of the vector copies is deleted.

**Store Segment Metadata** There are lifted vector primitives for which execution depends on having the segment lengths for its input vectors at hand. The current implementation is, that they are computed on demand. A more efficient way would probably be to keep the segment lengths together with the vector in an additional data structure. They are computed when the vector is computed.
Appendices
Appendix A.

Example Execution

For an example execution a vector algebra program consisting of 87 nodes has been selected. The program requests the result of node 87. The original (JSON-formatted) computation instructions for node 87 are presented in Listing A.1.

The instructions are to take the result of node 84 and feed it into a vector primitive called Unsegment.

```
[  
  87, 
  { 
    "UnOp" : [ 
      { 
        "Unsegment" : [] 
      }, 
      84 
    ] 
  } 
]
```

Listing A.1: Example node

In Listing A.2 is the unformatted result for node 87 after program execution (without conversion to JSON). Columns types from left to right are: descriptor, positions, string data, integer data. This makes it a ValueVector.

```
node 87:
1 1 1 MS 170
1 2 Ed 85
1 3 Yale 60
```

Listing A.2: Example result

Listing A.3 is the console output of the execution environment during execution of the example described above. In Listing A.1 a short piece of the input file BigQuery.json can be found. The result file Result.json is not provided here. Basically the file contains the vector of Listing A.2 in JSON format.

The two lists of node ids occurring in the program output refer in phase 1 to the parsing order and in phase 2 to the computation order. Node 2 which is the first node of computation is an actual vector input.
Appendix A. Example Execution

$ ./a.out exec BigQuery.json Result.json
running in exec mode with 2 threads

1st phase: parsing json and creating
references table

nodes in list: 87
biggest node index: 87

nodes requested by shape: 87,
parsing: 87
87,84,83,80,79,17,16,15,12,11,8,7,6,2,5,10,4,3,
78,82,77,76,74,68,67,66,59,58,57,52,51,50,31,30,
29,19,18,27,26,25,23,22,21,40,64,61,41,33,32,65,
47,46,43,42,45,39,38,36,35,63,73,72,62,75,71,

will compute 65 out of 87 nodes
2nd phase: computing nodes
computing: 87
2,3,4,10,5,6,7,8,11,12,15,16,17,71,75,18,19,21,
22,23,25,26,27,29,30,31,32,33,41,50,51,52,57,61,
62,72,73,63,35,36,38,39,45,40,42,43,46,47,65,64,
58,59,66,67,68,74,76,77,82,78,79,80,83,84,87,
done
result has been written to file: Result.json

Listing A.3: Example execution on a dual-core CPU (2 threads)
Appendix B.

Source Code for restrictVec

The complete source code for the vector primitive restrictVec is provided here. A brief discussion of its implementation approach is in Section 4.2.4.5. The semantics of restrictVec is to replicate the contents of input vector b as many times as indicated by input vector a.

The implementation consists of two loops (see comments in the source) separated by an OpenMP barrier directive. The first loop does the filtering and determines the length of the result vectors. The second loops fills the result vector with content. Additionally an array temp is computed by the first loop: it contains the array indices for the components which are to be copied in the second loop.

```c
const Entity* BinOp::restrictVec(const VV* a, const VV* b) {
    bigindex length = a->length();
    assert(length == b->length());
    assert(a->hsize() == 1);
    const Column* c = a->getItems()->getColumn(0);
    assert(c->type() == TYPE_BOOL);
    const boolval* bools = ((ColumnBool*) c)->getData();
    const Items* items = b->getItems();
    const bigindex* descriptor = b->getDescriptor();

    // shared vars
    bigindex** temp; // indices of passed components
    bigindex* rowCount; // number of passed components
    bigindex newLength; // total of passed components

    // results vars
    Items* res;
    bigindex* dor;
    bigindex* po;
    bigindex* pn;

    // threading vars
    int threadNum1;
    int threadNum2;
```
Appendix B. Source Code for restrictVec

```
bigindex* threadOffset = new bigindex[length];
bigindex* threadLength = new bigindex[length];

#pragma omp parallel
{
    #pragma omp single
    {
        threadNum1 = omp_get_num_threads();
        newThreadRanges(threadNum1, length, 
                        threadOffset, threadLength);
        temp = new bigindex*[threadNum1];
        rowCount = new bigindex[threadNum1];
        newLength = 0;
    }

    int myId = omp_get_thread_num();
    if (myId < threadNum1) {
        temp[myId] = new bigindex[threadLength[myId]];
        rowCount[myId] = 0;

        // loop1: * count number of 'true' rows
        //       * store their indices in temp
        for (bigindex i=threadOffset[myId]; 
             i<threadOffset[myId]+threadLength[myId]; 
             i++) {
            if (bools[i] == true) {
                bigindex ptr = rowCount[myId];
                temp[myId][ptr] = i;
                rowCount[myId] = ptr+1;
            }
        }
    }

    // all threads will wait here
    // until the last thread arrives
    #pragma omp barrier

    // the code block below is executed
```
Appendix B. Source Code for restrictVec

```c
// by exactly 1 thread
#pragma omp single
{
    delete[] threadOffset;
    delete[] threadLength;
    threadNum2 = omp_get_num_threads();
    newThreadRanges(threadNum2, newLength, 
                    threadOffset, threadLength);

    // create arrays for result vectors
    res = items->newEmptyCopy(newLength);
    dor = new bigindex[newLength];
    po = new bigindex[newLength];
    pn = new bigindex[newLength];
}

myId = omp_get_thread_num();
if (myId < threadNum2) {

    // prepare loop 2
    // outerOffset: length between array's first
    // index and where a new
    // contiguous replication of
    // vector b's contents starts
    // innerOffset: offset within the
    // current replication
    bigindex outerOffset = 0;
    int tid = 0;
    while (tid < threadNum1 && threadOffset[myId] >= 
           outerOffset+rowCount[tid]) {
        outerOffset += rowCount[tid];
        tid++;
    }
    bigindex innerOffset = 
        threadOffset[myId] - outerOffset;

    // loop 2: copy values to result vector
    for (bigindex i=threadOffset[myId];
         i<threadOffset[myId]+threadLength[myId];
         i++) {
        bigindex fromId = temp[tid][innerOffset];
        res->copyRow(items, fromId, i); // copy data
dor[i] = descriptor[fromId];
        po[i] = fromId+1; // convert po to 1–basis
        pn[i] = i+1;
    }
}
```

107
innerOffset ++;
if (tid < threadNum1
    && innerOffset >= rowCount[tid]) {
    outerOffset += rowCount[tid];
    tid ++;
    innerOffset = 0;
}
}
}

// myId bracket
}
// parallel bracket
}

// delete temporary variables
delete[] threadOffset;
delete[] threadLength;

for (int i = 0; i < threadNum1; i++) {
    delete[] temp[i];
}
delete[] temp;
delete[] rowCount;

// assemble result vectors
VV* vv = new VV(res, dor, newLength);
RV* rv = new RV(po, pn, newLength);
return new Entity(vv, rv);
}

Listing B.1: Implementation of restrictVec
Listings

2.1. Haskell lists and tuples examples ...................................... 9

3.1. Typedef examples .......................................................... 49
3.2. Function template example-1 ............................................. 50
3.3. Function template example-2 ............................................. 50
3.4. Semaphore initialisation .................................................. 51
3.5. Semaphore usage ........................................................... 52
3.6. OpenMP example: serial loop .......................................... 53
3.7. OpenMP example: parallel loop ....................................... 53
3.8. Hello world example ...................................................... 55
3.9. Vector addition example ................................................ 56
3.10. Vector addition with for construct ................................... 57

4.1. Implementation of String::refUp() .................................... 68
4.2. Implementation of String::refDown() .................................. 68
4.3. Implementation of singletonDescr ..................................... 71
4.4. Prototype of singletonDescr in the nullary class ................. 71
4.5. Implementation of lengthA .............................................. 72
4.6. Implementation of toDescr .............................................. 74
4.7. Implementation of vecSum .............................................. 76
4.8. Implementation of vecSumExec ........................................ 77
4.9. Implementation of vecMax .............................................. 78
4.10. Implementation of vecMin .............................................. 79
4.11. Examples for gt() implementations .................................. 80
4.12. Inner function of vecMin and vecMax ............................... 82
4.13. Parallelization pattern .................................................. 84
4.14. Implementation of distDesc (relevant parts) ..................... 86
4.15. Partial code for parallel quicksort .................................. 90
4.16. Example for comparison implementations ........................... 91
4.17. Pseudo code for unique implementation ............................ 93
4.18. Implementation of R3 ................................................... 94
4.19. Implementation of constructLiteralTable .......................... 95
4.20. Main program code of phase two .................................... 99

A.1. Example node .............................................................. 103
A.2. Example result ............................................................ 103
A.3. Example execution on a dual-core CPU (2 threads) .............. 104

B.1. Implementation of restrictVec ........................................ 105


List of Figures

1.1. Major components of the new DSH compilation chain . . . . . 5
3.1. Caption for LOF . . . . . . . . . . . . . . . . . . . . . . . . . . 54
3.2. Effect of the call()() function . . . . . . . . . . . . . . . . . . . 59
4.1. Diagram of the execution environment . . . . . . . . . . . . . . . 62
4.2. UML class diagramm of inheritance relations between vector types 63
4.3. UML class diagram of Items and its components . . . . . . . . 64
4.4. UML Class diagram of available column types. . . . . . . . . . 65
4.5. Components of phase one . . . . . . . . . . . . . . . . . . . . . . . 96
4.6. Flowchart diagram of phase two . . . . . . . . . . . . . . . . . . . 98

List of Tables

2.1. Vector types overview . . . . . . . . . . . . . . . . . . . . . . . . . . 12
2.2. Data types . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
2.3. Operator table for compExpr2 and compExpr2L . . . . . . . . . 20
2.4. Computation of segment lengths for $VV_{res}$ . . . . . . . . . . 46
3.1. Subset of fundamental types in C++ . . . . . . . . . . . . . . . . 49
4.1. Typedefs for vector algebra data types . . . . . . . . . . . . . . . 66
4.2. Pointer depth for data types . . . . . . . . . . . . . . . . . . . . . 66
4.3. Typedefs for vector algebra data types . . . . . . . . . . . . . . . 66
Bibliography


Selbständigkeitserklärung

Ich erkläre, daß ich die vorliegende Diplomarbeit einschließlich des beiliegen-
den Programmcodes selbständig und nur unter Verwendung der angegebenen
Literatur und Hilfsmittel angefertigt habe.

Tübingen, den

Tobias Müller