rasdaman
raster data manager

Query Language Guide

rasdaman version 8.0

rasdaman Version 8.0 Query Language Guide

Preface

Overview

This guide provides information about how to use the rasdaman database management system (in short: rasdaman). The document explains usage of the following interfaces and tools:

- rasql: the rasdaman Query Language, consisting of
    - rasdl: the rasdaman Data Definition Language
    - rasml: the rasdaman Data Manipulation Language

Follow the instructions in this guide as you develop your application which makes use of rasdaman services. Explanations detail how to create type definitions and instances; how to retrieve information from databases; how to insert, manipulate, and delete instances within databases.

Audience

The information in this manual is intended primarily for application developers; additionally, it can be useful for advanced users of rasdaman applications and for database administrators.

Rasdaman Documentation Set

This manual should be read in conjunction with the complete rasdaman documentation set which this guide is part of. The documentation set in its completeness covers all important information needed to work with the rasdaman system, such as programming and query access to databases, guidance to utilities such as the graphical-interactive query tool *rView*, and release notes.

In particular, current restrictions, known bugs, and workarounds are listed in the Release Notes. All documents, therefore, always have to be considered in conjunction with the Release Notes.

The rasdaman Documentation Set consists of the following documents:

- C++ Developer's Guide
- Java Developer's Guide
- Query Language Guide
- Installation and Administration Guide
- PostgreSQL Integration Guide
- Error Messages
- rView Guide
- Release Notes

Table of Contents

1 Introduction

1.1 Multidimensional Data

In principle, any natural phenomenon becomes spatio-temporal array data of some specific dimensionality once it is sampled and quantised for storage and manipulation in a computer system; additionally, a variety of artificial sources such as simulators, image renderers, and data warehouse population tools generate array data. The common characteristic they all share is that a large set of large multidimensional arrays has to be maintained. We call such arrays *multidimensional discrete data* (or short: *MDD*) expressing the variety of dimensions and separating them from the conceptually different multidimensional vectorial data appearing in geo databases.

rasdaman is a domain-independent database management system (DBMS) which supports multidimensional arrays of any size and dimension and over freely definable cell types. Versatile interfaces allow rapid application deployment while a set of cutting-edge intelligent

optimization techniques in the rasdaman server ensures fast, efficient access to large data sets, particularly in networked environments.

1.2 rasdaman Overall Architecture

The rasdaman client/server DBMS has been designed using internationally approved standards wherever possible. The system follows a two-tier client/server architecture with query processing completely done in the server. Internally and invisible to the application, arrays are decomposed into smaller units which are maintained in a conventional DBMS, for our purposes called the *base DBMS*.

On the other hand, the base DBMS usually will hold alphanumeric data (such as metadata) besides the array data. rasdaman offers means to establish references between arrays and alphanumeric data in both directions.

Hence, all multidimensional data go into the same physical database as the alphanumeric data, thereby considerably easing database maintenance (consistency, backup, etc.).

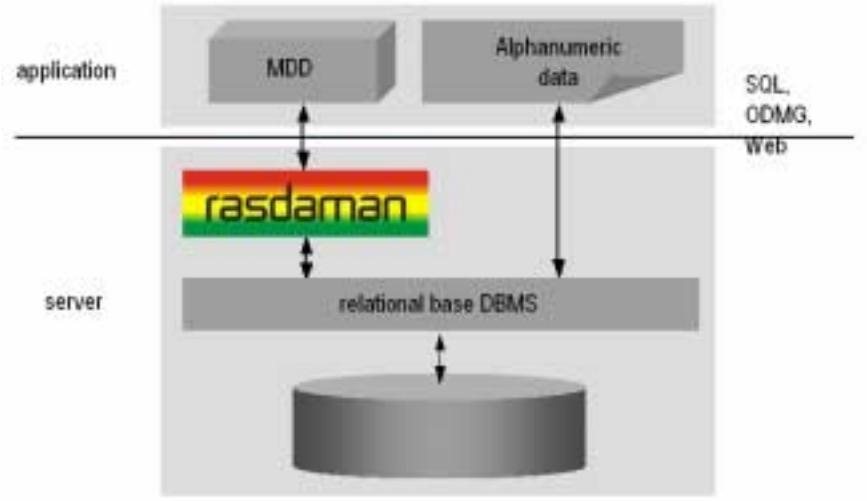

Figure 1 Embedding of rasdaman in IT infrastructure

Further information on application program interfacing, administration, and related topics is available in the other components of the rasdaman documentation set.

1.3 Interfaces

The syntactical elements explained in this document comprise the rasql language interface to rasdaman. There are several ways to actually enter such statements into the rasdaman system:

- By using the rView utility to interactively type in queries and visualize the results.
- By developing an application program which uses the RasLib function `oql_execute()` to forward query strings to the rasdaman server and get back the results.
- By using the rasdl processor to manipulate and inspect the type definitions contained in the dictionary of a rasdaman database.

RasLib and rView are not the subject of this document. Please refer to the *rView Guide* and *C++ Developer's Guide* of the rasdaman documentation set for further information.

1.4 rasql and Standard SQL

The declarative interface to the rasdaman system, the *rasdaman Query Language,* rasql, consists of the *rasdaman Definition Language*, rasdl, and the *rasdaman Manipulation Language*, rasml. The reader may notice the close resemblance of the traditional separation of SQL into a DDL (*Data Definition Language*) and DML (*Data Manipulation Language*). The similarity to traditional query concepts tentatively is kept wherever possible.

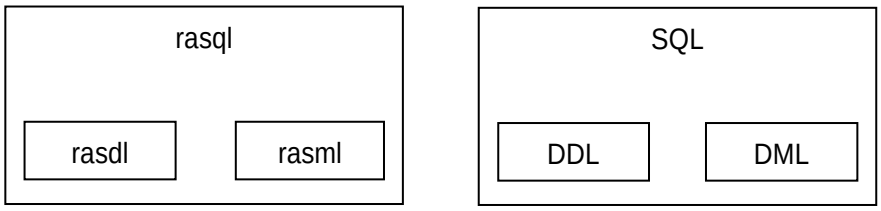

Figure 2 Correspondence between rasql and SQL

Moreover, the rasdaman query language, rasql, is very similar - and in fact relies on - standard SQL. Hence, if you are familiar with SQL, you will quickly be able to use rasql. Otherwise you may want to consult the introductory literature referenced at the end of this chapter.

1.5 Notational Conventions

The following notational conventions are used in this manual:

Program text (under this we also subsume queries in the document on hand) is printed in a `monotype font`. Such text is further differentiated into keywords and syntactic variables. Keywords like **`struct`** are printed in boldface; they have to be typed in as is. On the contrary, syntactic variables like *`structName`* are typeset in italics; they have to be replaced by a name or an expression which evaluates to an entity of the appropriate type.

An optional clause is enclosed in italic brackets; an arbitrary repetition is indicated through italic brackets and an ellipsis:

```
select resultList
from   collName [ as collIterator ]
       [, collName [ as collIterator ] ] ...
[ where  booleanExpr ]
```

It is important not to mix the regular brackets **`[`** and **`]`** denoting array access, trimming, etc., with the italic brackets *`[`* and *`]`* denoting optional clauses and repetition.

Italics are also used in the text to draw attention to the *first instance of a defined term* in the text. In this case, the font is the same as in the running text, not `Courier` as in code pieces.

1.6 Further Reading

S.J. Cannan: *SQL The Standard Handbook*, McGraw-Hill Book Company, London, 1993

R.G.G. Cattell et al.: *The Object Database Standard: ODMG 3.0*, Morgan Kaufmann Publishers, California.

2 Terminology

2.1 An Intuitive Definition

An array is a set of elements which are ordered in space. The space considered here is discretized, i.e., only integer coordinates are admitted. The number of integers needed to identify a particular position in this space is called the *dimension* (sometimes also referred to as *dimensionality*). Each array element, which is referred to as *cell*, is positioned in space through its *coordinates*.

A cell can contain a single value (such as an intensity value in case of grayscale images) or a composite value (such as integer triples for the red, green, and blue component of a color image). All cells share the same structure which is referred to as the *array cell type* or *array base type*.

Implicitly a neighborhood is defined among cells through their coordinates: incrementing or decrementing any component of a coordinate will lead to another point in space. However, not all points of this (infinite) space will

actually house a cell. For each dimension, there is a *lower* and *upper bound*, and only within these limits array cells are allowed; we call this area the *spatial domain* of an array. In the end, arrays look like multidimensional rectangles with limits parallel to the coordinate axes. The database developer defines both spatial domain and cell type in the *array type definition*. Not all bounds have to be fixed during type definition time, though: It is possible to leave bounds open so that the array can dynamically grow and shrink over its lifetime.

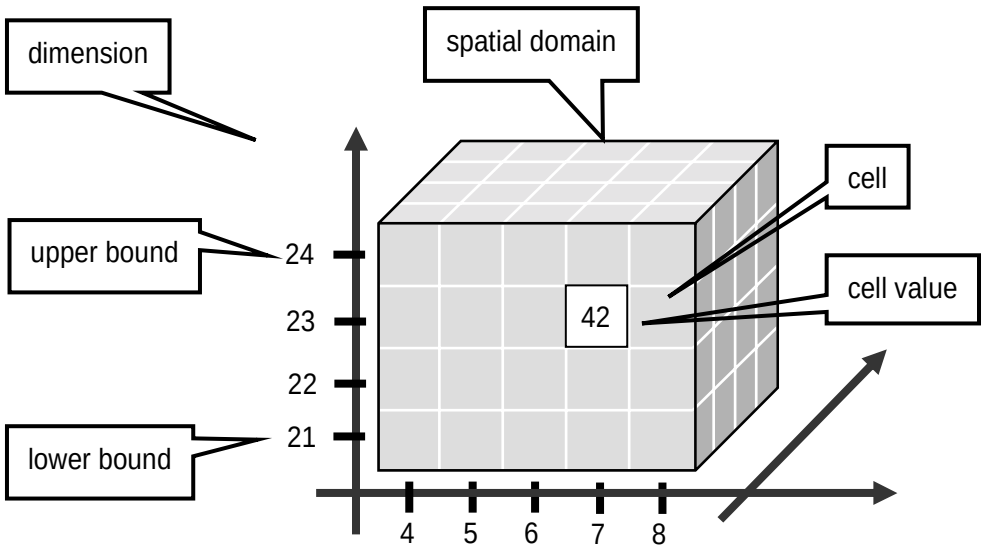

Figure 3 Constituents of an array

Synonyms for the term array are *multidimensional arrays*, *multidimensional data*, *MDD*. They are used interchangeably in the rasdaman documentation.

In rasdaman databases, arrays are grouped into collections. All elements of a collection share the same array type definition (for the remaining degrees of freedom see Section 4.3.2). Collections form the basis for array handling, just as tables do in relational database technology.

2.2 A Technical Definition

Programmers who are familiar with the concept of arrays in programming languages maybe prefer this more technical definition:

An array is a mapping from integer coordinates, the spatial domain, to some data type, the cell type. An array's spatial domain, which is always finite, is described by a pair of lower bounds and upper bounds for each dimension, resp. Arrays, therefore, always cover a finite, axis-parallel subset of Euclidean space.

Cell types can be any of the base types and composite types defined in the ODMG standard and known, for example from C/C++. In fact, every admissible C/C++ type is admissible in the rasdaman type system, too.

In rasdaman, arrays are strictly typed wrt. spatial domain and cell type. Type checking is done at query evaluation time. Type checking can be disabled selectively for an arbitrary number of lower and upper bounds of an array, thereby allowing for arrays whose spatial domains vary over the array lifetime.

3 Sample Database

3.1 Collection mr

This section introduces sample collections used later in this manual. The sample database which is shipped together with the system contains the schema and the instances outlined in the sequel.

Collection `mr` consists of three images (see Figure 4) taken from the same patient using magnetic resonance tomography. Images are 8 bit grayscale with pixel values between 0 and 255 and a size of 256x211.

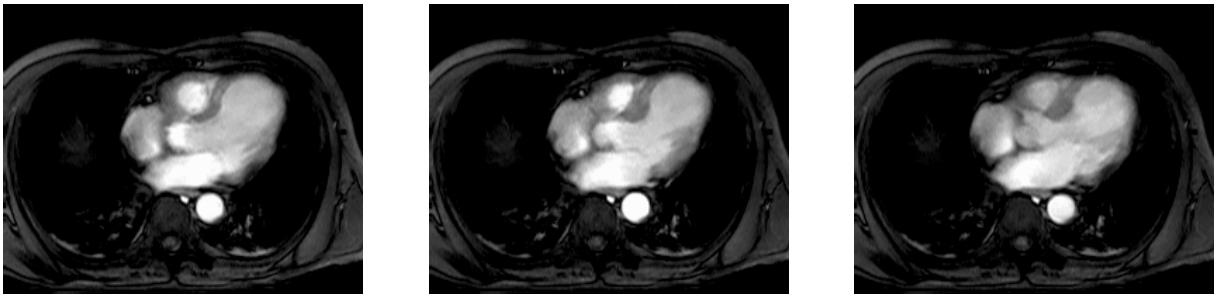

Figure 4 Sample collection `mr`

3.2 Collection mr2

Collection `mr2` consists of only one image, namely the first image of collection `mr`. Hence, it is also 8 bit grayscale with size 256x211.

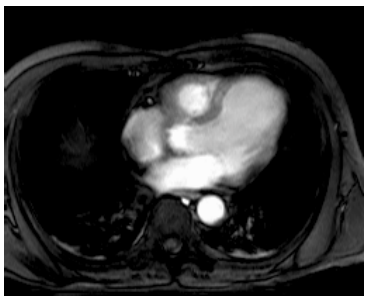

Figure 5 Sample collection `mr2`

3.3 Collection rgb

The last example collection, `rgb`, contains one item, a picture of the anthur flower. It is an RGB image of size 400x344 where each pixel is composed of three 8 bit integer components for the red, green, and blue component, resp.

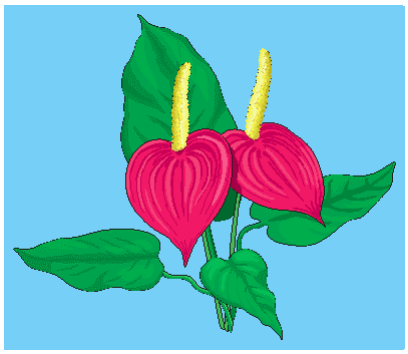

Figure 6 The collection `rgb`

4 Type Definition with rasdl

4.1 Overview

Every instance within a database is described by its data type (i.e., there is exactly one data type to which an instance belongs; conversely, one data type can serve to describe an arbitrary number of instances). Each database contains a self-contained set of such type definitions; no other type information, external to a database, is needed for database access.

A rasdaman schema contains three categories of data types:

- Cell type definitions; these can be base types (such as float) or composite ("struct") types such as red/green/blue color pixels.
- MDD type definitions; these define arrays over some base type and with some spatial domain.
- Collection type definitions; they describe sets over some MDD type; collections of a given type can only contain MDD instances of the MDD type used in the definition.

Types are identified by their name which must be unique within a database; upper and lower case are distinguished. The same names are used in the C++ header files generated by the rasdl processor.

Type handling is done using the rasdl processor which allows to add types to a database schema, to delete types, to display schema information, and to generate C++ header files from database types. rasdl, therefore, is the central tool for maintaining database schemata.

Dynamic Type Definition

New types can be defined dynamically while the rasdaman server is running. This means that new types introduced via rasdl are immediately available to all other applications after rasdl's transaction commit.

Only in some rare cases base DBMSs don't support this. For details consult the corresponding *rasdaman External Products Integration Guide*.

Examples

In Section 4.7 examples are given for the most common rasdl tasks.

Important Note

Extreme care must be exercised on database and type maintenance. For example, deleting a database cannot be undone, and deleting a type still used in the database (i.e., which is needed to describe existing MDD objects) may make it impossible to access these database objects any more.

In general, database administration should be reserved to few persons, all of which should have high familiarity with the operating system, the relational database system, and the rasdaman system.

4.2 Application Development Workflow

The usual proceeding of setting up a database and an application operating this database consists of three major steps.

- First, a rasdl definition is established (best through a file, but also possible via typing in commands interactively) which is fed into the rasdl processor. The output generated from this source consists of C++ header and source files; in parallel, the type information is fed into the target database.
- In the second step, the program source code written by the application developer is compiled using the respective C++ compiler of the target platform. To this end, the header file generated in the first step as well as the RasLib header files are read by the compiler. The output is relocatable object code.
- In the third step, the so created rasdaman application executable can operate on a database with a structure as defined in the rasdl source used.

Figure 7 shows this application development workflow, including all the preprocess, compile and link steps necessary to generate an executable application.

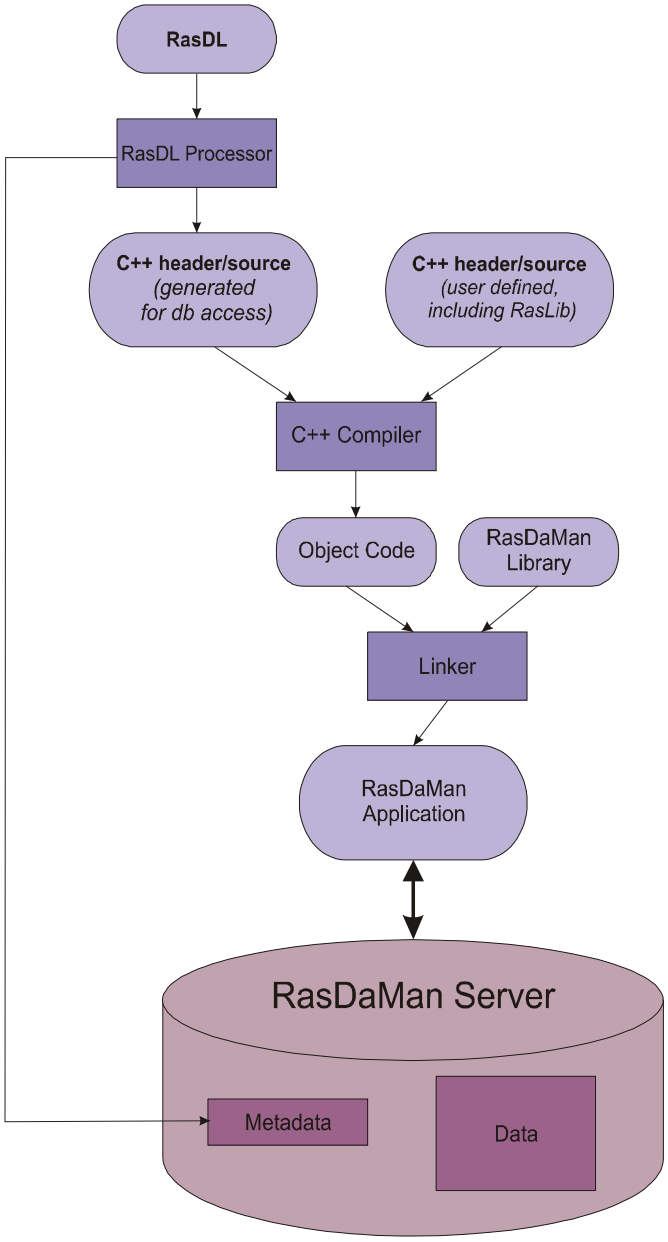

Figure 7 Application development workflow

4.3 Type Definition

rasdl syntax closely follows C/C++ data type definition syntax[1]. In fact, there is only one syntactic extension to ODMG/C++ which allows to conveniently specify spatial domains. The complete syntax specification of rasdl can be found in the appendix.

4.3.1 Base Types

The set of standard data types, which is generated during creation of a database, materializes the base types defined in the ODMG standard (cf.Table 1).

rasdl name	size	description
`octet`	8 bit	signed integer
`char`	8 bit	unsigned integer
`short`	16 bit	signed integer
`unsigned short`	16 bit	unsigned integer
`long`	32 bit	signed integer
`unsigned long`	32 bit	unsigned integer
`float`	32 bit	single precision floating point
`double`	64 bit	double precision floating point
`complex`	64 bit	single precision complex
`complexd`	128 bit	double precision complex
`boolean`	1 bit[2]	`true` (nonzero value), `false` (zero value)

Table 1 rasdl base types

Further cell types can be defined arbitrarily in rasdaman, based on the system-defined base types or other user-defined types. In particular, composite user-defined base types corresponding to **`struct`**s in C/C++ are supported. As a rule, every C/C++ type is admissible as array base type in rasdl with the exception of pointers (these are handled through OIDs, see Section 13), nested arrays, and classes.

The keyword **`struct`** allows to define complex pixel types, **`typedef`** is used for the definition of MDD and set types in the same way as in C++. The *`typeName`* indicating the cell type of the array must be defined within the database schema using it.

---

[1] Actually, rasdl is a subset of ODMG's Object Definition Language (ODL) with the only extension for spatial domain specification within the array template.

[2] memory usage is one byte per pixel

Syntax

```
struct structName
{ attrType_1 attrName_1;
  ...
  attrType_n attrName_n;
};
```

4.3.2 Array Type Definition

A spatial domain can be defined with variable degree of freedom. The most concise way is to explicitly specify a lower and upper bound for each dimension. Such bounds are integer numbers whereby, in each dimension, the lower bound must be less or equal to the upper bound. Negative numbers are allowed, and the lower bound does not have to be 0.

Array ranges defined this way are checked by the server during every access. An attempt to access a cell outside the specified spatial domain will lead to a runtime error.

Syntax

```
typedef marray
< typeName, [ lo_1 : hi_1, ..., lo_n : hi_n  ] >
marrayName;
```

Example

The following definition establishes a 5x5 integer array sitting in the center of the coordinate system. Such matrices can be used, for example, to hold convolution kernels.

```
typedef marray < int, [-2:2, -2:2] > kernel5;
```

Note that the symmetry in the boundaries grounds in the way kernels are defined; there is no constraint on the bounds in rasdl.

A higher degree of freedom in the array boundaries can be specified by indicating an asterisk "*" instead of a lower or upper bound at any position. In this case, range checking is disabled for that particular bound, and dynamic extending and shrinking is possible in that dimension.

Examples

A fax has a fixed width, but an arbitrary length. Modelling this requires to leave open the upper bound of the second dimension:

```
typedef marray <char, [ 1:1728, 1:* ]> G3Fax;
```

An array can have an arbitrary number of variable bounds:

```
typedef marray <char, [ *:*, *:* ]> GreyImage;
```

This extreme case - that all bounds are free - can be abbreviated by indicating, instead of the spatial domain in brackets, only the number of dimensions:

Syntax

```
typedef marray
< typeName, dimension >
marrayName;
```

Example

```
typedef marray <char, 2> GreyImage;
```

To leave open even the dimensionality of an array, even the dimension number can be omitted:

Syntax

```
typedef marray
< typeName >
marrayName;
```

Example

```
typedef set <GreyImage> GreySet;
```

It is recommended to use unbounded arrays with extreme care - all range checking is disabled, and structures may be created in the database which your (or your colleagues') applications don't expect.

4.3.3 Collection Type Definition

An array collection type is defined with the array type as parameter. A collection of such a type can contain an arbitrary number of arrays whereby each of these must conform with the array type indicated.

Syntax

```
typedef set < marrayName > setName;
```

4.3.4 Comments in Type Definitions

Comments are texts which are not evaluated by rasdaman in any way. However, they are useful - and should be used freely - for documentation purposes so that the defined type's meaning will be unambiguously clear to later readers.

Syntax

```
// any text, delimited by end of line
```

Example

```
typedef struct
{ char red,     // red channel of color image
       green,   // green channel of color image
       blue;    // blue channel of color image
} RGBPixel;     // 3 x 8bit color pixels
```

4.4 Sample Database Type Definitions

The following definitions describe the three sample collections introduced earlier.

Collections `mr` and `mr2` share the same structure definition, namely 256x211 8-bit grayscale images. As we don't want to restrict ourselves to a particular image size, we just define the image type as being 2-dimensional. The following fragment accomplishes this.

```
typedef marray <char, 2> GreyImage;
typedef set <GreyImage> GreySet;
```

Equivalently, but more verbosely, we could have specified the image type as

```
typedef marray <char,[ *:*, *:* ]> GreyImage;
```

The last example defines sets of RGB images. The first line defines the cell type as a **`struct`** containing three single-byte components. The next line defines an `RGBImage` as a 2-dimensional array of size 800x600 over base type `RGBPixel`. The last line defines a set of RGB images.

```
struct RGBPixel { char red, green, blue; };
typedef marray <RGBPixel,[0:799, 0:599]> RGBImage;
typedef set <RGBImage> RGBSet;
```

4.5 Deleting Types and Databases

For deleting a type or a whole database the `rasdl –delX` command family is provided where `X` is one of `database`, `basetype`, `mddtype`, or `settype`:

`--deldatabase` *`db`*	delete database *db*[3]
`--delbasetype` *`type`*	delete base type *type* from database
`--delmddtype` *`type`*	delete MDD type *type* from database
`--delsettype` *`type`*	delete set (collection) type *type*

Important Note

Extreme care must be exercised on database and type maintenance. Deleting a database cannot be undone, and deleting a type still used in the database (i.e., which is needed to describe existing MDD objects) may make it impossible to access these database objects any more.

In general, database administration should be reserved to few persons, all of which should have high familiarity with the operating system, the relational database system, and the rasdaman system.

[3] dependent on the relational base DBMS used; please consult the *External Products Integration Guide* for your environment.

Example

The following Unix command line will delete the definition of set type `MyCollectionType` from database `RASBASE`; it is the responsibility of the `rasdl` user (i.e., the database administrator) to ensure that no instances of this set definition exist in the database.

```
rasdl --database RASBASE --delsettype MyCollectionType
```

4.6 rasdl Invocation

As outlined, the rasdl processor reads the specified rasdl source file, imports the schema information into the database, and generates a corresponding C++ header file. A rasdl source file has to be self-contained, i.e., only types which are defined in the same file are allowed to be used for definitions. Types must be defined before use.

Usage:

```
rasdl [options]
```

Options:

`--database` *`db`* name of database
(default: `RASBASE`)

`--connect` *`c`* connect string for base DBMS connection
(default: `/`)

`-c, --createdatabase` *`dbname`*
create database with name *`dbname`* and initialize it with the standard schema

`--deldatabase` *`db`*
delete database *`db`*

`--delbasetype` *`type`*
delete base type *`type`* from database

`--delmddtype` *`type`*
delete MDD type *`type`* from database

`--delsettype` *`type`*
delete set (collection) type *`type`* from database

`-h, --help` help: display invocation syntax

`--connect` *`connectstr`*
connect string for underlying database
(e.g. `test/test@julep`)
default: `/`

`-p, --print` *`db`* print all data type definitions in database *`db`*

`-r, --read` *`file`* read rasdl commands from file *`file`*

`-i, --insert`	insert types into database (`-r` required)
`--hh` *`file`*	generate C++ header file *`file`* (`-r` required)

Notes

Right now, `rasdl` is a server-based utility, which means it must be invoced on the machine where the rasdaman server runs. This limitation will be overcome in future versions.

The `–c` option for database creation is dependent on the base DBMS used – some systems do, a very few don't allow database creation through `rasdl`. Please refer to the *External Product Integration Guide* for limitations due to the respective base DBMS.

4.7 Examples

Default database name is `RASBASE`. It depends upon the base DBMS whether upper and lower case are distinguished (usually they are not).

Create Database

A new database is created through

```
rasdl -c
```

An error message is issued and the operation is aborted if the database exists already.

Delete Database

An existing database is deleted through

```
rasdl --deldatabase
```

All contents will be lost irretrievably. All space in the base DBMS is released, all tables (including rasdaman system tables) are removed.

Read Type Definition File

A type definition file named *`myfile`* are read into the database through

```
rasdl –r myfile –i
```

In particular, the standard types must be read in as part of the database creation process:

```
rasdl –r ~rasdaman/examples/rasdl/basictypes.dl –i
```

Print All Types

An overview on all rasdaman types defined is printed through

```
rasdl -p
```

5 Query Execution with rasql

The rasdaman toolkit offers essentially three ways to communicate with a database through queries:

- By writing a C++ or Java application that uses the rasdaman APIs, raslib or rasj, resp. (see the rasdaman API guides).
- By writing queries using the GUI-based rview tool which allows to visualize results in a large variety of display modes (see the rasdaman rview Guide).
- By submitting queries via command line using rasql; this tool is covered in this section.

The rasql tool accepts a query string (which can be parametrised as explained in the API guides), sends it to the server for evaluation, and receives the result set. Results can be displayed in alphanumeric mode, or they can be stored in files.

5.1 Examples

For the user who is familiar with command line tools in general and the rasql query language, we give a brief introduction by way of examples. They outline the basic principles through common tasks.

- Create a collection `test` of type `GreySet` (note the explicit setting of user `rasadmin`; rasql's default user `rasguest` by default cannot write):

```
rasql –q "create collection test GreySet" \
      --user rasadmin --passwd rasadamin
```

- Print the names of all existing collections:

```
rasql –q "select r from RAS_COLLECTIONNAMES as r" \
      --out string
```

- Export demo collection `mr` into TIFF files `rasql_1.tif`, `rasql_2.tif`, `rasql_3.tif`:

```
rasql –q "select tiff(m) from mr as m" --out file
```

- Import TIFF file *`myfile`* into collection `mr` as new image (note the different query string delimiters to preserve the `$` character!):

```
rasql –q 'insert into mr values $1' –f myfile \
      --user rasadmin --passwd rasadamin
```

- Put a grey square into every `mr` image:

```
rasql –q "update mr as m set m[0:10,0:10] \
          assign marray x in [0:10,0:10] values 127c" \
          --user rasadmin --passwd rasadamin
```

- Verify result of update query by displaying pixel values as hex numbers:

```
rasql –q "select m[0:10,0:10] from mr as m" --out hex
```

5.2 Rasql Invocation

Rasql is invoked as a command with the query string as parameter. Additional parameters guide detailed behavior, such as authentication and result display.

Any errors or other diagnostic output encountered are printed; transactions are aborted upon errors.

Usage:

```
rasql [--query q|-q q] [options]
```

Options:

`-h, --help`	show command line switches
`-q, --query` *`q`*	query string to be sent to the rasdaman server for execution

`-f, --file f` file name for upload through `$i` parameters within queries; each `$i` needs its own file parameter, in proper sequence[4]. Requires `--mdddomain` and `--mddtype`

`--content` display result, if any (see also `--out` and `--type` for output formatting)

`--out t` use display method *t* for cell values of result MDDs where `t` is one of

- `none` do not display result item contents
- `file` write each result MDD into a separate file
- `string` print result MDD contents as char string (only for 1D arrays of type char)
- `hex` print result MDD cells as a sequence of space-separated hex values
- `formatted` reserved, not yet supported

Option `--out` implies `--content`; default: `none`

`--outfile of` file name template for storing result images (ignored for scalar results). Use '`%d`' to indicate auto numbering position, like with printf(1). For well-known file types, a proper suffix is appended to the resulting file name. Implies `--out file`. (default: `rasql_%d`)

`--mdddomain d` MDD domain, format: '`[x0:x1,y0:y1]`'; required only if `--file` specified and file is in data format `r_Array`; if input file format is some standard data exchange format and the query uses a convertor, such as `inv_tiff($1)`, then domain information can be obtained from the file header.

`--mddtype t` input MDD type (must be a type defined in the database); required only if `--file` specified and file is in data format `r_Array`; if input file format is some standard data exchange format and the query uses a convertor, such as `inv_tiff($1)`, then type information can be obtained from the file header.

`--type` display type information for results

`-s, --server h` rasdaman server name or address

(default: `localhost`)

`-p, --port p` rasdaman port number (default: `7001`)

`-d, --database db`

[4] Currently only one `-f` argument is supported (i.e., only `$1`).

	name of database (default: RASBASE)
`--user` *u*	name of user (default: `rasguest`)
`--passwd` *p*	password of user (default: `rasguest`)

6 Overview: General Query Format

6.1 Basic Query Mechanism

rasml provides declarative query functionality on collections (i.e., sets) of MDD stored in a rasdaman database. The query language is based on the SQL-92 standard and extends the language with high-level multidimensional operators.

The general query structure is best explained by means of an example. Consider the following query:

```
select mr[100:150,40:80] / 2
from   mr
where  some_cells( mr[120:160, 55:75] > 250 )
```

In the **from** clause, mr is specified as the working collection on which all evaluation will take place. This name, which serves as an “iterator variable” over this collection, can be used in other parts of the query for referencing the particular collection element under inspection.

Optionally, an alias name can be given to the collection (see syntax below) – however, in most cases this is not necessary.

In the **where** clause, a condition is phrased. Each collection element in turn is probed, and upon fulfillment of the condition the item is added to the query result set. In the example query, part of the image is tested against a threshold value.

Elements in the query result set, finally, can be "post-processed" in the **select** clause by applying further operations. In the case on hand, a spatial extraction is done combined with an intensity reduction on the extracted image part.

In summary, a rasql query returns a set fulfilling some search condition just as is the case with conventional SQL and OQL. The difference lies in the operations which are available in the **select** and **where** clause: SQL does not support expressions containing multidimensional operators, whereas rasql does.

Syntax

```
select resultList
from   collName [ as collIterator ]
       [, collName [ as collIterator ] ] ...
[ where  booleanExpr ]
```

Further information on rasql statements is provided in Section 10. The complete query syntax can be found in the Appendix.

6.2 Select Clause: Result Preparation

Type and format of the query result are specified in the **select** part of the query. The query result type can be multidimensional, a struct, or atomic (i.e., scalar). The `select` clause can reference the collection iteration variable defined in the `from` clause; each array in the collection will be assigned to this iteration variable successively.

Example

Images from collection `mr`, with pixel intensity reduced by a factor 2:

```
select mr / 2
from   mr
```

6.3 From Clause: Collection Specification

In the **from** clause, the list of collections to be inspected is specified, optionally together with a variable name which is associated to each collection. For query evaluation the cross product between all participating collections is built which means that every possible combination of

elements from all collections is evaluated. For instance in case of two collections, each MDD of the first collection is combined with each MDD of the second collection. Hence, combining a collection with `n` elements with a collection containing `m` elements results in `n*m` combinations. This is important for estimating query response time.

Example

The following example subtracts each MDD of collection `mr2` from each MDD of collection `mr` (the binary induced operation used in this example is explained in Section 9.3.2).

```
select mr - mr2
from   mr, mr2
```

Using alias variables `a` and `b` bound to collections `mr` and `mr2`, resp., the same query looks as follows:

```
select a - b
from   mr as a, mr2 as b
```

Cross products

As in SQL, multiple collections in a `from` clause such as

```
from c1, c2, …, ck
```

are evaluated to a *cross product*. This means that the `select` clause is evaluated for a virtual collection that has `n1 * n2 * … * nk` elements if `c1` contains `n1` elements, `c2` contains `n2` elements, and so forth.

Warning:
This holds regardless of the `select` expression – even if you mention only say `c1` in the `select` clause, the number of result elements will be the product of *all* collection sizes!

6.4 Where Clause: Conditions

In the **`where`** clause, conditions are specified which members of the query result set must fulfil. Like in SQL, predicates are built as boolean expressions using comparison, parenthesis, functions, etc. Unlike SQL, however, rasql offers mechanisms to express selection criteria on multidimensional items.

Example

We want to restrict the previous result to those images where at least one difference pixel value is greater than 50 (see Section 9.3.2):

```
select mr - mr2
from   mr, mr2
where  some_cells( mr - mr2 > 50 )
```

6.5 Comments in Queries

Comments are texts which are not evaluated by the rasdaman server in any way. However, they are useful - and should be used freely - for documentation purposes; in particular for stored queries it is important that its meaning will be unambiguously clear to later readers.

Syntax

```
-- any text, delimited by end of line
```

Example

```
select mr      -- this comment text is ignored by rasdaman
from   mr      -- for comments spanning several lines,
               -- every line needs a separate '--' starter
```

7 Constants

7.1 Atomic Constants

Atomic constants are written in standard C/C++ style. If necessary constants are augmented with a one or two letter postfix to unambiguously determine its data type.

The default for integer constants is 'L', for floats it is 'F'. Specifiers are case insensitive.

Example

```
25c
-1700L
.4e-5D
```

Note

Boolean constants `true` and `false` are unique, so they do not need a length specifier.

postfix char	type
c	char
o	octet
s	short
us	unsigned short
l	long
ul	unsigned long
f	float
d	double

Table 2 Data type specifiers

7.2 Composite Constants

Composite constants resemble records ("structs") over atomic constants or other records. Notation is as follows.

Syntax

```
struct
{ const_0,
     ...
  const_n
}
```

where *const_i* can be atomic or a **struct** again.

Example

```
struct{ struct{ 1l, 2l, 3l }, true }
```

Complex numbers

Special built-in structs are `complex` and `complexd` for single and double precision complex numbers, resp. The constructor is defined by

Syntax

complex(*re*, *im* **)**

where *re* and *im* are floating point expressions. The resulting complex constant is of type `complexd` if at least one of the constituent expressions is double precision, otherwise the result is of type `complex`.

Example

```
complex( .35, 16.0d )
```

Component access

See Section 9.2.5 for details on how to extract the constituents from a composite value.

7.3 Array Constants

Small array constants can be indicated literally (see Section 7.3 for a way to describe large array constants). An array constant consists of the spatial domain specification (see Section 9.1) followed by the cell values whereby value sequencing is as follow. The array is linearised in a way that the lowest dimension[5] is the "outermost" dimension and the highest dimension[6] is the "innermost" one. Within each dimension, elements are listed sequentially, starting with the lower bound and proceeding until the upper bound. List elements for the innermost dimension are separated by comma ",", all others by semicolon ";".

The exact number of values as specified in the leading spatial domain expression must be provided. All constants must have the same type; this will be the result array's base type.

Syntax

```
< mintervalExpr
  scalarList_0 ;
  ... ;
  scalarList_n ;
>
```

where *scalarList* is defined as a comma separated list of literals:

scalar_0, scalar_1,... ;

Example

```
< [-1:1,-2:2] 0, 1, 2, 3, 4; 1, 2, 3, 4, 5; 2, 3, 4, 5, 6 >
```

The constant defines the following matrix with cell type `long`:

$$\begin{bmatrix} 0 & 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 6 \end{bmatrix}$$

[5] the dimension which is the *leftmost* in the spatial domain specification

[6] the dimension which is the *rightmost* in the spatial domain specification

7.4 Object Identifier (OID) Constants

OIDs serve to uniquely identify arrays (see Section 13). Within a database, the OID of an array is an integer number. To use an OID outside the context of a particular database, it must be fully qualified with the system name where the database resides, the name of the database containing the array, and the local array OID.

The worldwide unique array identifiers, i.e., OIDs, consist of three components:

- A string containing the system where the database resides (system name),
- A string containing the database (base name), and
- A string containing the local object id within the database.

The full OID is enclosed in '<' and '>' characters, the three name components are separated by a vertical bar '|'. System and database names obey the naming rules of the underlying operating system and base DBMS, i.e., usually they are made up of lower and upper case characters, underscores, and digits. Any additional white space (space, tab, or newline characters) inbetween is assumed to be part of the name, so this should be avoided. The local OID is an integer number.

Syntax

< *systemName* | *baseName* | *objectID* **>**

integerExpr

where *systemName* and *baseName* are string literals and *objectID* is an *integerExpr*.

Example

```
< mySun | Demobase | 2305 >
42
```

7.5 Collection Names

Collections are named containers for sets of MDD objects (see Section 13). A collection name is made up of lower and upper case characters, underscores, and digits. Depending on the underlying base DBMS, names may be limited in length, and some systems (rare though) may not distinguish upper and lower case letters. Please refer to the *rasdaman External Products Integration Guide* for details on your particular platform.

Operations available on name constants are string equality "=" and inequality "=!".

8 Spatial Domain Operations

8.1 One-Dimensional Intervals

One-dimensional (1D) intervals describe non-empty, consecutive sets of integer numbers, described by integer-valued lower and upper bound, resp.; negative values are admissible for both bounds. Intervals are specified by indicating lower and upper bound through integer-valued expressions according to the following syntax:

The lower and upper bounds of an interval can be extracted using the functions `.lo` and `.hi`.

Syntax

integerExpr_1 `:` *integerExpr_2*
intervalExpr`.lo`
intervalExpr`.hi`

A one-dimensional interval with *integerExpr_1* as lower bound and *integerExpr_2* as upper bound is constructed. The lower bound must be

less or equal to the upper bound. Lower and upper bound extractors return the integer-valued bounds.

Examples

An interval ranging from `-17` up to `245` is written as

```
-17 : 245
```

Conversely, the following expression evaluates to 245; note the parenthesis to enforce the desired evaluation sequence:

```
(-17 : 245).hi
```

8.2 Multidimensional Intervals

Multidimensional intervals (*m-intervals*) describe areas in space, or better said: point sets. These point sets form rectangular and axis-parallel "cubes" of some dimension. An m-interval's dimension is given by the number of 1D intervals it needs to be described; the bounds of the "cube" are indicated by the lower and upper bound of the respective 1D interval in each dimension.

From an m-interval, the intervals describing a particular dimension can be extracted by indexing the m-interval with the number of the desired dimension using the operator `[]`.

Dimension counting in an m-interval expression runs from left to right, starting with lowest dimension number 0.

Syntax

```
[ intervalExpr_0, ..., intervalExpr_n ]
[ intervalExpr_0, ..., intervalExpr_n ] [integerExpr ]
```

An n-dimensional m-interval with the specified *`intervalExpr_i`* is built where the first dimension is described by *`intervalExpr_0`*, etc., until the last dimension described by *`intervalExpr_n`*.

Example

A 2-dimensional m-interval ranging from -17 to 245 in dimension 1 and from 42 to 227 in dimension 2 can be denoted as

```
[ -17 : 245, 42 : 227 ]
```

The expression below evaluates to `[42:227]`.

```
[ -17 : 245, 42 : 227 ] [1]
```

...whereas here the result is `42:`

```
[ -17 : 245, 42 : 227 ] [1].lo
```

9 Array Operations

As we have seen in the last Section, *intervals* and *m-intervals* describe n-dimensional regions in space.

Next, we are going to place information into the regular grid established by the m-intervals so that, at the position of every integer-valued coordinate, a value can be stored. Each such value container addressed by an n-dimensional coordinate will be referred to as a *cell*. The set of all the cells described by a particular m-interval and with cells over a particular base type, then, forms the *array*.

As before with intervals, we introduce means to describe arrays through expressions, i.e., to derive new arrays from existing ones. Such operations can change an arrays shape and dimension (sometimes called geometric operations), or the cell values (referred to as value-changing operations), or both. In extreme cases, both array dimension, size, and base type can change completely, for example in the case of a histogram computation.

First, we describe the means to query and manipulate an array's spatial domain (so-called geometric operations), then we introduce the means to query and manipulate an array's cell values (value-changing operations).

Note that some operations are restricted in the operand domains they accept, as is common in arithmetics in programming languages; division by zero is a common example. Section 10 contains information about possible error conditions, how to deal with them, and how to prevent them.

9.1 Spatial Domain

The m-interval covered by an array is called the array's *spatial domain*. Function `sdom()` allows to retrieve an array's current spatial domain. The *current domain* of an array is the minimal axis-parallel bounding box containing all currently defined cells.

As arrays can have variable bounds according to their type definition (see Section 4.3.2), their spatial domain cannot always be determined from the schema information, but must be recorded individually by the database system. In case of a fixed-size array, this will coincide with the schema information, in case of a variable-size array it delivers the spatial domain to which the array has been set. The operators presented below and in Section 12.4 allow to change an array's spatial domain. Notably, a collection defined over variable-size arrays can hold arrays which, at a given moment in time, may differ in the lower and/or upper bounds of their variable dimensions.

Syntax

`sdom( mddExpr )`

Function `sdom()` evaluates to the current spatial domain of *`mddExpr`*.

Examples

Consider an image `a` of collection `mr`. Elements from this collection are defined as having free bounds, but in practice our collection elements all have spatial domain `[0:255, 0:210]`. Then, the following equivalences hold:

```
sdom(a)         = [0:255,0:210]
sdom(a)[0]      = [0:255]
sdom(a)[0].lo   = 0
sdom(a)[0].hi   = 255
```

9.2 Geometric Operations

9.2.1 Trimming

Reducing the spatial domain of an array while leaving the cell values unchanged is called *trimming*. Array dimension remains unchanged.

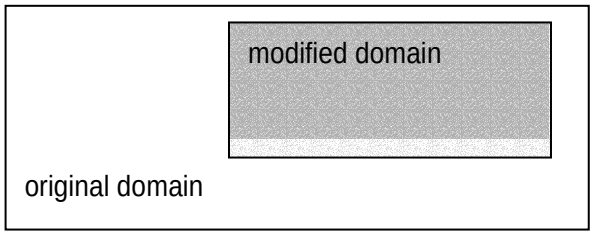

Figure 8 Spatial domain modification through trimming (2-D example)

The *generalized trim operator* allows restriction, extension, and a combination of both operations in a shorthand syntax. This operator does not check for proper subsetting or supersetting of the domain modifier.

Syntax

mddExpr[*mintervalExpr*]

Examples

The following query returns cutouts from the area `[120:160,55:75]` of all images in collection `mr`. (see Figure 9).

```
select mr[ 120:160, 55:75 ]
from   mr
```

Figure 9 Trimming result

9.2.2 Section

A *section* allows to extract lower-dimensional layers ("slices") from an array.

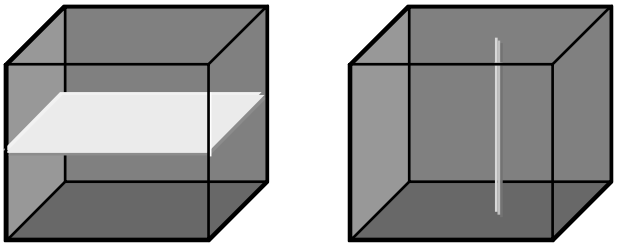

Figure 10 Single and double section through 3-D array, yielding 2-D and 1-D sections.

A section is accomplished through a trim expression by indicating the slicing position rather than a selection interval. A section can be made in any dimension within a trim expression. Each section reduces the dimension by one.

Syntax

mddExpr [*integerExpr_0*, ..., *integerExpr_n*]

This makes sections through *mddExpr* at positions *integerExpr_i* for each dimension *i*.

Example

The following query produces a 2-D section in the 2nd dimension of a 3-D cube:

```
select Images3D[ 0:256, 10, 0:256 ]
from   Images3D
```

Note

If a section is done in *every* dimension of an array, the result is one single cell. This special case resembles array element access in programming languages, e.g., C/C++. However, in rasql the result still is an array, namely one with zero dimensions and exactly one element.

Example

The following query delivers a set of 0-D arrays containing single pixels, namely the ones with coordinate [100,150]:

```
select mr[ 100, 150 ]
from   mr
```

9.2.3 The Array Bound Wildcard Operator "*"

An asterisk "*" can be used as a shorthand for an `sdom()` invocation in a trim expression; the following phrases all are equivalent:

```
a[ *:*, *:* ] = a[ sdom(a)[0], sdom(a)[1] ]
              = a[ sdom(a)[0].lo : sdom(a)[0].hi,
                   sdom(a)[1].lo : sdom(a)[1].hi ]
```

An asterisk "*" can appear at any lower or upper bound position within a trim expression denoting the current spatial domain boundary. A trim expression can contain an arbitrary number of such wildcards. Note, however, that an asterisk cannot be used for specifying a section.

Example

The following are valid applications of the asterisk operator:

```
select mr[ 50:*, *:200 ]
from   mr
```

```
select mr[ *:*, 10:150 ]
from   mr
```

The next is illegal because it attempts to use an asterisk in a section:

```
select mr[ *, 100:200 ]  -- illegal "*" usage in dimension 0
from   mr
```

Note

It is well possible (and often recommended) to use an array's spatial domain or part of it for query formulation; this makes the query more general and, hence, allows to establish query libraries. The following query cuts away the rightmost pixel line from the images:

```
select mr[ *:*, *:sdom(a)[1].hi - 1]      -- good, portable
from   mr
```

In the next example, conversely, trim bounds are written explicitly; this query's trim expression, therefore, cannot be used with any other array type.

```
select mr[ 0:767, 0:1023 ]      -- bad because not portable
from   mr
```

One might get the idea that the last query evaluates faster. This, however, is not the case; the server's intelligent query engine makes the first version execute at just the same speed.

9.2.4 Shifting a Spatial Domain

Built-in function `shift()` transposes an array: its spatial domain remains unchanged in shape, but all cell contents simultaneously are moved to another location in n-dimensional space. Cell values themselves remain unchanged.

Syntax

`shift( mddExpr, pointExpr )`

The function accepts an *`mddExpr`* and a *`pointExpr`* and returns an array whose spatial domain is shifted by vector *`pointExpr`*.

Example

The following expression evaluates to an array with spatial domain `[3:13,4:24]`. Containing the same values as the original array `a`.

```
shift( a[ 0:10, 0:20 ], [ 3, 4 ] )
```

9.2.5 Extending a Spatial Domain

Function `extend()` enlarges a given MDD with the domain specified. The domain for extending must, for every boundary element, be at least as large as the MDD's domain boundary. The new MDD contains null values in the extended part of its domain and the MDD's original cell values within the MDD's domain.

Syntax

```
extend( mddExpr, mintervalExpr )
```

The function accepts an *mddExpr* and a *mintervalExpr* and returns an array whose spatial domain is extended to the new domain specified by *mintervalExpr*, with *mddExpr*'s values in its domain and null values elsewhere. The result MDD has the same cell type as the input MDD.

Precondition:

```
sdom( mddExpr ) contained in mintervalExpr
```

Example

Assuming that MDD `a` has a spatial domain of `[0:50,0:25]`, the following expression evaluates to an array with spatial domain `[-100:100,-50:50]`, `a`'s values in the subdomain `[0:50,0:25]`, and null values at the remaining cell positions.

```
extend( a, [-100:100,-50:50] )
```

9.3 Induced Operations

Induced operations allow to simultaneously apply a function originally working on a single cell value to all cells of an MDD. The result MDD has the same spatial domain, but can change its base type.

Examples

```
img.green + 5c
```

This expression selects component named "`green`" from an RGB image and adds 5 (of type `char`, i.e., 8 bit) to every pixel.

```
img1 + img2
```

This performs pixelwise addition of two images (which must be of equal spatial domain).

Induction and `structs`

Whenever induced operations are applied to a composite cell structure ("`structs`" in C/C++), then the induced operation is executed on every structure component. If some cell structure component turns out to be of an incompatible type, then the operation as a whole aborts with an error.

For example, a constant can be added simultaneously to all components of an RGB image:

```
select rgb + 5
from   rgb
```

Induction and `complex`

Complex numbers, which actually form a composite type supported as a base type, can be accessed with the record component names `re` and `im` for the real and the imaginary part, resp.

Example

The first expression below extracts the real component, the second one the imaginary part from a complex number c:

```
c.re
c.im
```

9.3.1 Unary Induction

Unary induction means that only one array operand is involved in the expression. Two situations can occur: Either the operation is unary by nature (such as boolean `not`); then, this operation is applied to each array cell. Or the induce operation combines a single value (scalar) with the array; then, the contents of each cell is combined with the scalar value.

In any case, sequence of iteration through the array for cell inspection is chosen by the database server (which heavily uses reordering for query optimisation) and not known to the user.

Syntax

mddExpr **binaryOp** *scalarExpr*
scalarExpr **binaryOp** *mddExpr*
unaryOp *mddExpr*

Example

The red images of collection `rgb` with all pixel values multiplied by `2`:

```
select rgb.red * 2c
from   rgb
```

Note that the constant is marked as being of type char so that the result of the two char types again will yield a char result (8 bit per pixel). Omitting the "c" would lead to an addition of long integer and char, the result being long integer with 32 bit per pixel. Although pixel values obviously are the same in both cases, the second alternative requires four times the memory space.

9.3.2 Binary Induction

Binary induction means that two arrays are combined.

Syntax

mddExpr **binaryOp** *mddExpr*

The difference between the images in the `mr` collection and the image in the `mr2` collection:

```
select mr – mr2
from   mr, mr2
```

Note

As in the previous Section, two cases have to be distinguished:

- Both left hand array expression and right hand array expression operate on the same array, for example:

  ```
  select rgb.red - rgb.green
  from   rgb
  ```

  In this case, the expression is evaluated by combining, for each coordinate position, the respective cell values from the left hand and right hand side.

- Left hand array expression and right hand array expression operate on different arrays, for example:

  ```
  select mr – mr2
  from   mr, mr2
  ```

  This situation specifies a cross product between the two collections involved. During evaluation, each array from the first collection is combined with each member of the second collection. Every such pair of arrays then is processed as described above.

  Obviously the second case can become computationally very expensive, depending on the size of the collections involved - if the two collections contain `n` and `m` members, resp., then `n*m` combinations have to be evaluated.

9.3.3 Struct Component Selection

Component selection from a composite value is done with the dot operator well-known from programming languages. The argument can either be a number (starting with 0) or the struct element name. Both statements of the following example would select the green plane of the sample RGB image.

Syntax

```
mddExpr . attrName
mddExpr . intExpr
```

Examples

```
select rgb.green
from   rgb
```

```
select rgb.1
from   rgb
```

Figure 11 RGB image and green component

Note

Aside of operations involving base types such as integer and boolean, combination of complex base types (structs) with scalar values are supported. In this case, the operation is applied to each element of the structure in turn. Both operands then have to be of exactly the same type, which further must be the same for all components of the struct.

Examples

The following expression reduces contrast of a color image in its red, green, and blue channel simultaneously:

```
select rgb / 2c
from   rgb
```

An advanced example is to use image properties for masking areas in this image. In the query below, this is done by searching pixels which are "sufficiently green" by imposing a lower bound on the green intensity and upper bounds on the red and blue intensity. The resulting boolean matrix is multiplied with the original image (i.e., componentwise with the red, green, and blue pixel component); the final image, then, shows the original pixel value where green prevails and is `{0,0,0}` (i.e., black) otherwise (Figure 12)

```
select rgb * ( (rgb.green > 130c) and
               (rgb.red   < 110c) and
               (rgb.blue  < 140c) )
from   rgb
```

Note

This mixing of boolean and integer is possible because the usual C/C++ interpretation of `true` as 1 and `false` as 0 is supported by rasql.

Figure 12 Suppressing "non-green" areas

9.3.4 Induction: All Operations

Below is a complete listing of all cell level operations that can be induced, both unary and binary.

If two different data types are involved, the result will be of the more general type; e.g., float and integer addition will yield a float result.

`is, and, or, xor, not`

For each cell within some Boolean MDD (or evaluated MDD expression), combine it with the second MDD argument using the logical operation `and`, `or`, or `xor`. The `is` operation is equivalent to `==` (see below). The signature of the binary induced operation is

```
is, and, or, xor: mddExpr, intExpr -> mddExpr
```

Unary function `not` negates each cell value in the MDD.

`+, -, *, /`

For each cell within some MDD value (or evaluated MDD expression), add it with the corresponding cell of the second MDD parameter. For example, this code adds two (equally sized) images:

```
img1 + img2
```

As usual, these arithmetic operations are overloaded to expect `mddExpr` as well as `numExpr`, integer as well as float numbers, and single precision as well as double precision values.

`==, <, >, <=; >=, !=`

For two MDD values (or evaluated MDD expressions), compare for each coordinate the corresponding cells to obtain the Boolean result indicated by the operation.

Note that comparison works on all atomic cell types. On composite types, only `==` and `!=` make sense with the meaning that, for two cells with identical structure, all components undergo a pairwise comparison.

`bit(mdd,pos)`

For each cell within MDD value (or evaluated MDD expression) `mdd`, take the bit with nonnegative position number `pos` and put it as a Boolean

value into a byte. Position counting starts with 0 and runs from least to most significant bit. The `bit` operation signature is

```
bit: mddExpr, intExpr -> mddExpr
```

In C/C++ style,

```
bit(mdd,pos)
```

is equivalent to

```
mdd >> pos & 1
```

Overlay

The overlay operator allows to combine two equally sized MDDs by placing the second one "on top" of the first one, informally speaking. Formally, overlaying is done in the following way:

- wherever the second operand's cell value is non-zero[7], the result value will be this value.
- wherever the second operand's cell value is zero, the first argument's cell value will be taken.

This way stacking of layers can be accomplished, e.g., in geographic applications. Consider the following example:

```
ortho overlay tk.water overlay tk.streets
```

When displayed the resulting image will have `streets` on top, followed by `water`, and at the bottom there is the `ortho` photo.

Strictly speaking, the overlay operator is not atomic. Expression

```
a overlay b
```

is equivalent to

```
(b != 0) * b  +  (b == 0) * a
```

However, on the server the overlay operator is executed more efficiently than the above expression.

Trigonometric and exponential functions

The following advanced arithmetic functions are available, with the obvious meaning:

[7] Null means a numerical value of 0 (zero).

```
sqrt()
abs()
exp() log() ln()
sin() cos() tan()
sinh() cosh() tanh()
arcsin() arccos() arctan()
```

cast

Sometimes the desired ultimate scalar type or MDD cell type is different from what the MDD expression would suggest. To this end, the result type can be enforced explicitly through the cast operator.

The syntax is:

```
(newType) generalExpr
```

where `newType` is the desired result type of expression `generalExpr`.

Like in programming languages, the cast operator converts the result to the desired type if this is possible at all. For example, the following scalar expression, without cast, would return a double precision float value; the cast makes it a single precision value:

```
(float) avg_cells( mr )
```

Both scalar values and MDD can be cast; in the latter case, the cast operator is applied to each cell of the MDD yielding an array over the indicated type.

The cast operator also works properly on recursively nested cell structures. In such a case, the cast type is applied to every component of the cell. For example, the following expression converts the pixel type of an (3x8 bit) RGB image to an image where each cell is a structure with three `long` components:

```
(long) rgb
```

Obviously in the result structure all components will bear the same type.

Restrictions

Currently only base types are permitted as cast result types, it is not possible to cast to a struct or `complex` type, e.g.

```
(RGBPixel) rgb                                    -- illegal
```

On base type `complex`, only the following operations are available right now:

```
+   -   *   /
```

9.4 Scaling

Shorthand functions are available to scale multidimensional objects. They receive an array as parameter, plus a scale factor. In the most common

case, the scaling factor is an integer or float number. This factor then is applied to all dimensions homogeneously. For a scaling with individual factors for each dimension, a scaling vector can be supplied which, for each dimension, contains the resp. scale factor.

Syntax

```
scale( mddExpr, intExpr )
scale( mddExpr, floatExpr )
scale( mddExpr, intVector )
```

Examples

The following example returns all images of collection `mr` where each image has been scaled down by a factor of 2.

```
select scale( mr, 0.5 )
from   mr
```

Next, `mr` images are enlarged by 4 in the first dimension and 3 in the second dimension:

```
select scale( mr, [ 4, 3 ] )
from   mr
```

Note

Function `scale()` breaks tile streaming, it needs to load all tiles affected into server main memory. In other words, the source argument of the function must fit into server main memory. Consequently, it is not advisable to use this function on very large items.

9.5 Condensers

Frequently summary information of some kind is required about some array, such as sum or average of cell values. To accomplish this, rasql provides the concept of condensers.

A *condense operation* (or short: *condenser*) takes an array and summarizes its values using a summarization function.

A number of condensers is provided as rasql built-in function. For numeric arrays, `add_cells()` delivers the sum and `avg_cells()` the average of all cell values. Operators `min_cells()` and `max_cells()` return the minimum and maximum, resp., of all cell values in the argument array. For boolean arrays, the condenser `count_cells()` counts the cells containing `true`. Finally, the `some_cells()` operation returns true if at least one cell of the boolean MDD is `true`, `all_cells()` returns true if all of the MDD cells contain `true` as value.

Please keep in mind that, depending on their nature, operations take a boolean, numeric, or arbitrary *`mddExpr`* as argument.

Syntax

```
count_cells( mddExpr )
add_cells( mddExpr )
avg_cells( mddExpr )
min_cells( mddExpr )
max_cells( mddExpr )
some_cells( mddExpr )
all_cells( mddExpr )
```

Examples

The following example returns all images of collection `mr` where all pixel values are greater than `20`. Note that the induction `">20"` generates a boolean array which, then, can be collapsed into a single boolean value by the condenser.

```
select mr
from   mr
where  all_cells( mr > 20 )
```

The next example selects all images of collection `mr` with at least one pixel value greater than `250` in the specified region `[120:160, 55:75]` (Figure 13).

```
select mr
from   mr
where  some_cells( mr[120:160, 55:75] > 250 )
```

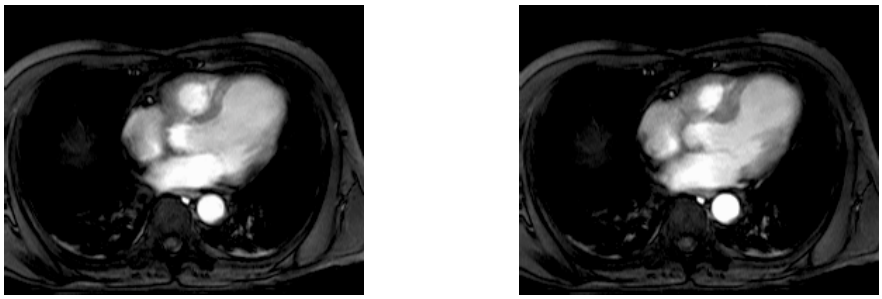

Figure 13 Query result of specific selection

9.6 General Array Condenser

All the condensers introduced above are special cases of a general principle which is represented by the *general condenser* statement.

The general condense operation consolidates cell values of a multidimensional array to a scalar value based on the condensing operation indicated. It iterates over a spatial domain while combining the result values of the *`cellExpr`*s through the *`condenserFunction`* indicated.

Condensers are heavily used in two situations:

- To collapse boolean arrays into scalar boolean values so that they can be used in the **where** clause.
- In conjunction with the **marray** constructor (see next section) to phrase high-level signal processing and statistical operations.

Syntax

```
condense condenserOp
over     var in mintervalExpr
using    cellExpr

condense condenserOp
over     var in mintervalExpr
where    booleanExpr
using    cellExpr
```

The *mintervalExpr* terms together span a multidimensional spatial domain over which the condenser iterates. It visits each point in this space exactly once, assigns the point's respective coordinates to the *var* variables and evaluates *cellExpr* for the current point. The result values are combined using condensing function *condenserOp*. Optionally, points used for the aggregate can be filtered through a *booleanExpr*; in this case, *cellExpr* will be evaluated only for those points where *booleanExpr* is true, all others will not be regarded. Both *booleanExpr* and *cellExpr* can contain occurrences of variables *pointVar*.

Examples

```
condense +
over     x in sdom(a)
using    x[0] * a[ x ]
```

Note

Definition of the specialized condensers in terms of the general condenser statement is as follows:

Restriction

Currently condensers of any kind over cells of type `complex` are not supported.

Array aggregate definition	Meaning

`add_cells(a) =`
`  condense +`
`  over x in sdom(a)`
`  using a[x]`
	sum over all cells in a

`avg_cells(a) =` `  sum_cells(a) / card(sdom(a))`	Average of all cells in a

`min_cells(a) =`
`  condense min`
`  over x in sdom(a)`
`  using a[x]`
	Minimum of all cells in a

`max_cells(a) =`
`  condense max`
`  over x in sdom(a)`
`  using a[x]`
	Maximum of all cells in a

`count_cells(b) =`
`  condense +`
`  over x in sdom(b)`
`  where b[x]`
`  using 1`
	Number of cells in b

`some_cells(b) =`
`  condense or`
`  over x in sdom(b)`
`  using b[x]`
	is there any cell in b
	with value true?

`all_cells(b) =`
`  condense and`
`  over x in sdom(b)`
`  using b[x]`
	do all cells of b
	have value true?

Table 3 Specialized condensers; `a` is a numeric, `b` a boolean array.

9.7 General Array Constructor

The **marray** constructor allows to create n-dimensional arrays with their content defined by a general expression. This is useful

- whenever the array is too large to be described as a constant (see Section 7.3) or
- when the array's cell values are derived from some other source, e.g., for a histogram computation (see examples below).

Syntax

The basic shape of the `marray` construct is as follows.

marray *var* **in** *mintervalExpr* [, *var* **in** *mintervalExpr*]
values *cellExpr*

Iterator Variable Declaration

First, the constructor allocates an array in the server with the spatial domain defined by the cross product of all *mintervalExpr*. For example, the following defines a 2-D 5x10 matrix:

```
marray x in [1:5], y in [1:10]
values ...
```

The base type of the array is determined by the type of *cellExpr*. Variable *var* can be of any number of dimensions.

Iteration Expression

In the second step, the constructor iterates over the spatial domain defined as described, successively evaluating *cellExpr* for each variable combination; the result value is assigned to the cell with the coordinate currently under evaluation. To this end, *cellExpr* can contain arbitrary occurrences of *var*. The syntax for using a variable is:

- for a one-dimensional variable:

  var

- for a higher-dimensional variable

  var[*index-expr*]

  where *index-expr* is a constant expression (no `sdom()` etc.!) evaluating to a non-negative integer; this number indicates the variable dimension to be used.

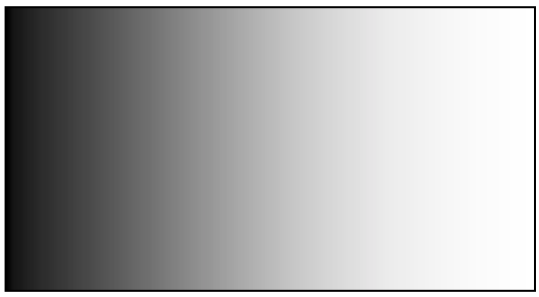

Figure 14 2-D array with values derived from first coordinate

Examples

The following creates an array with spatial domain `[1:100,-50:200]` over cell type char, each cell being initialized to 1.

```
marray x in [ 1:100, -50:200 ]
values 1c
```

In the next expression, cell values are dependent on the first coordinate component (cf. Figure 14)

```
marray x in [ 0:255, 0:511 ]
values x[0]
```

The final two examples comprise a typical marray/condenser combination. The first one takes a sales table and consolidates it from days to week per product. Table structure is as given in Figure 15.

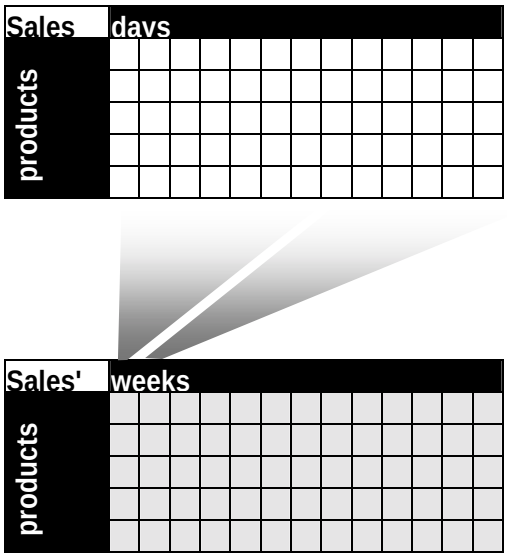

Figure 15 Sales table consolidation

```
select marray tab in [ 0:sdom(s)[0].hi/7, sdom(s)[1] ]
       values condense +
              over  day in [ 0:6 ]
              using s[ day[0] + tab]*7, tab[1] ]
from   salestable as s
```

The last example computes histograms for the `mr` images. The query creates a 1-D array ranging from 0 to 9 where each cell contains the number of pixels in the image having the respective intensity value.

```
select marray v in [ 0: 9 ]
       values condense +
              over  x in sdom(a)
              where mr[x] = v[0]
              using 1
from   mr
```

Shorthand

As a shorthand, variable *var* can be used without indexing; this is equivalent to *var*`[0]`:

```
marray x in [1:5]
values a[ x ]          -- equivalent to a[ x[0] ]
```

Many vs. One Variable

Obviously an expression containing several 1-D variables, such as:

```
marray x in [1:5], y in [1:10]
values a[ x[0], y[0] ]
```

can always be rewritten to an equivalent expression using one higher-dimensional variable, for example:

```
marray xy in [1:5, 1:10]
values a[ xy[0], xy[1] ]
```

Iteration Sequence Undefined

The sequence in which the array cells defined by an `marray` construct are inspected is not defined. In fact, server optimisation will heavily make use of reordering traversal sequence to achieve best performance.

A Note on Expressiveness and Performance

The general condenser and the array constructor together allow to express a very broad range of signal processing and statistical operations. In fact, all other rasql array operations can be expressed through them, for example:

Specialized operation	Specialized constructor	Phrasing with `marray`
Trimming	`a[ *:*, 50:100 ]`	**marray** x **in** sdom(a) **values** a[x]
Section	`a[ 50, *:* ]`	**marray** x **in** sdom(a)[1] **values** a[50, x]
Induction	`a + b`	**marray** x **in** sdom(a) **values** a[x] + b[x]

Table 4 Phrasing of Induction, Trimming, and Section via `marray`

Nevertheless, it is advisable to use the specialized operations whenever possible; not only are they more handy and easier to read, but also internally their processing has been optimized so that they execute considerably faster than the general phrasing.

9.8 Data Exchange Format Conversion

Normally arrays are accepted and delivered in the client's main memory format, regardless of the server architecture. Sometimes, however, it is desirable to use some data exchange format - be it because some device provides a data stream to be inserted in to the database in a particular format, or be it a Web application where particular output formats have to be used to conform with the respective standards.

To this end, rasql provides a series of data format converters (Table 5). They are invoked as pairs of built-in functions `X()` and `inv_X()` which convert to and from format `X` and the corresponding MDD.

Image format	rasql conversion function	Dimension
JPEG	`jpeg(), inv_jpeg()`	2
PNG	`png(), inv_png()`	2
TIFF	`tiff(), inv_tiff()`	2
BMP	`bmp(), inv_bmp()`	2
VFF	`vff(), inv_vff()`	3
HDF 4	`hdf(), inv_hdf()`	2,3
DEM[8]	`dem(), inv_dem()`	2
TOR	`tor(), inv_tor()`	2

Table 5 Data formats supported by rasql

Syntax

***dataFormatIdentifier*(** *mddExpr* **)**
***dataFormatIdentifier*(** *mddExpr*, optionString **)**
inv_*dataFormatIdentifier*(*tiffExpr* **)**

Matching Dimensions and Data Types

For converting an MDD into a data format, the MDD type must match the dimension and data type the image format can handle - it is mandatory that the array to be transformed or generated conforms to the overall structure supported by the particular data exchange format. For example, TIFF can only handle 2-D arrays with a particular subset of supported cell types.

Format Conversion Options

Additional header information ("tags" in TIFF, "chunks" in PNG, etc.) is set to default values; some settings can be done via an optional parameter string containing comma-separated "key=value" pairs. Table 6 lists the options currently implemented.

[8] Digital Elevation Model, i.e., an ASCII file containing lines with white-space-separated x/y/z values per pixel; for 2-D data only.

Image format	rasql conversion function[9]
JPEG	quality=%i (default: 80)
PNG	tRNS=%i tRNS=(%i;%i;%i)
TIFF	comptype=[none|ccittrle|ccittfax3|ccittfax4|lzw|ojpeg|jpeg|next|ccittrlew|packbits|thunderscan|pixarfilm|pixarlog|deflate|dcs|jbig] quality=%i (default:80)
VFF	dorder=[xy|yzx] dimorder=[xy|yzx] vffendian=[0|1]
HDF 4	comptype=[none|rle|huffman|deflate] quality=%i (default: 80) skiphuff=%i (default:0)
DEM[10]	flipx[0|1] (default: 0) flipy=[0|1] (default: 1) startx=%f endx=%f resx=%f start=%f endy=%f resy=%f
TOR	swapendianness=[0|1] (default: 0) rescale=[0|1] (default: 0) domain=[%i:%i,%i:%i]

Table 6 Data format options recognized by rasql (see resp. data format specifications for details on their meaning)

[9] Standard C/C++ notation is used to indicate parameter types: %i for integer (decimal/octal/hex notation), %f for float numbers

[10] Digital Elevation Model, i.e., an ASCII file containing lines with whitespace-separated x/y/z values per pixel; for 2-D data only.

Example

As an example, TIFF conversion is described here. rasql provides the function **tiff()** which to generate TIFF encoding from 2-D arrays. The inverse function is **inv_tiff()**. It takes an MDD in TIFF format as argument and delivers an MDD.

The following query delivers the image contained in the rgb collection as a PNG-encoded byte stream, with transparency set to color (0x77;0xd0; 0xf8):

```
select png( a, "tRNS=(0x77;0xd0;0xf8)" )
from   rgb as a
```

Insertion of a PNG encoded image into this collection is done as follows ($1 represents the input byte stream, see Section 12.4):

```
insert into rgb
values inv_png( $1 )
```

9.9 Object Identifiers

The function oid() gives access to an array's object identifier (OID). It returns the local OID of the database array. The input parameter must be a variable associated with a collection, it cannot be an array expression. The reason is that oid() can be applied to only to persistent arrays which are stored in the database; it cannot be applied to query result arrays - these are not stored in the database, hence do not have an OID.

Syntax

```
oid( variable )
```

Example

The following example retrieves the MDD object with local OID 10 of set mr:

```
select mr
from   mr
where  oid( mr ) == 10
```

The following example is incorrect as it tries to get an OID from a non-persistent result array:

```
select oid( mr * 2 ) -- illegal example: no expressions
from   mr
```

Fully specified external OIDs are inserted as strings surrounded by brackets:

```
select mr
from   mr
where  oid( mr ) == < mySun | DemoBase | 10 >
```

In that case, the specified system (system name where the database server runs) and database must match the one used at query execution time, otherwise query execution will result in an error.

9.10 Expressions

Parentheses

All operators, constructors, and functions can be nested arbitrarily, provided that each sub-expression's result type matches the required type at the position where the sub-expression occurs. This holds without limitation for all arithmetic, Boolean, and array-valued expressions. Parentheses can (and should) be used freely if a particular desired evaluation precedence is needed which does not follow the normal left-to-right precedence.

Example

```
select (rgb.red + rgb.green + rgb.blue) / 3c
from   rgb
```

Operator Precedence Rules

Sometimes the evaluation sequence of expressions is ambiguous, and the different evaluation alternatives have differing results. To resolve this, a set of precedence rules is defined. You will find out that whenever operators have their counterpart in programming languages, the rasdaman precedence rules follow the same rules as are usual there.

Here the list of operators in descending strength of binding:

- dot ".", trimming, section
- unary –
- `sqrt`, `sin`, `cos`, and other unary arithmetic functions
- `*`, `/`
- `+`, `-`
- `<`, `<=`, `>`, `>=`, `!=`, `=`
- `and`
- `or`, `xor`
- `":"` (interval constructor), condense, marray
- overlay
- In all remaining cases evaluation is done left to right.

10 Null Value Handling

Null values can mean many different things – for example, no value given or value not known. For example, during piecewise import of satellite images into a large map, there will be areas which are not written yet. Actually, also after completely creating the map of, say, a country there will be untouched areas, as normally no country has a rectangular shape with axis-parallel boundaries.

rasdaman does not have a notion of distinct null values – every bit pattern in the range of a numeric type can appear in the database, so no bit pattern is left to represent “null”. If such a thing is desired, then the database designer must provide a separate bit map indicating the status for each cell.

To have a clear semantics, the following rule holds:

Uninitialized value handling

A cell value not yet addressed, but within the current domain of an MDD has a value of zero by definition; this extends in the obvious manner to composite cells.

Remark

Note the limitation to the *current* domain of an MDD. While in the case of an MDD with fixed boundaries this does not matter because always *definition domain* = *current domain*, an MDD with variable boundaries can grow and hence will have a varying current domain. Only cells inside the current domain can be addressed, be they uninitialized/null or not; addressing a cell outside the current domain will result in the corresponding exception.

11 Arithmetic Errors and Other Exception Situations

During query execution, a number of situations can arise which prohibit to deliver the desired query result or database update effect. If the server detects such a situation, query execution is aborted, and an error exception is thrown. In this Section, we classify the errors that occur and describe each class.

However, we do not go into the details of handling such an exception – this is the task of the application program, so we refer to the resp. API Guides. For a complete list of all rasdaman error messages, see the *Error Messages Guide*.

11.1 Overflow

Candidates

Add_cells, induced operation such as +

System Reaction

The overflow will be silently ignored, producing a result represented by the bit pattern pruned to the available size. This is in coherence with overflow handling in programming languages.

Remedy

Query coding should avoid potential overflow situations by applying numerical knowledge - simply said, the same care should be applied as always when dealing with numerics.

Example

Obtaining an 8-bit grey image from a 3*8-bit colour image through

```
( a.red + a.green + a.blue ) / 3c
```

most likely will result in an overflow situation after the additions, and scaling back by the division cannot remedy that. Better is to scale before adding up:

```
a.red / 3c + a.green / 3c + a.blue / 3c
```

However, this may result in accuracy loss in the last bits. So the final suggestion is to use a larger data type for the interim computation and push back the result into an 8-bit integer:

```
(char) ( (long)a.red + (long)a.green + (long)a.blue ) / 3 )
```

Obviously, this will be paid with some performance penalty due to the more expensive `long` arithmetics. It is up to the application developer to weight and decide.

11.2 Illegal operands

Candidates

Division by zero, non-positive argument to logarithm, negative arguments to the square root operator, etc. are the well-known candidates for arithmetic exceptions.

System Reaction

As specified in the C++ standard, the result of such an illegal operation is `nan` (not a number) according to the IEEE floating point standard. Query evaluation will continue, and a result will be returned to the client.

If `nan` values are sort of a problem for the application, then either operand MDD objects have to be checked before applying the operation, or the result MDD objects have to be looped through to replace each `nan` value by some other application-chosen value.

Remedy

Make sure that the operation receives valid input across all cells of the MDD objects touched.

11.3 Access Rights Clash

If a database has been opened in read-only mode, a write operation will be refused by the server; “write operation” meaning an insert, update, or delete statement.

12 Database Retrieval and Manipulation

12.1 Collection Handling

12.1.1 Create A Collection

The `create collection` statement is used to create a new, empty MDD collection by specifying its name and type. The type must exist in the database schema. There must not be another collection in this database bearing the name indicated.

Syntax

`create collection` *`collName typeName`*

Example

`create collection` `mr GreySet`

12.1.2 Drop A Collection

A database collection can be deleted using the **drop collection** statement.

Syntax

drop collection *collName*

Example

```
drop collection mr1
```

12.1.3 Retrieve All Collection Names

With the following rasql statement, a list of the names of all collections currently existing in the database is retrieved; both versions below are equivalent:

```
select RAS_COLLECTIONNAMES
from   RAS_COLLECTIONNAMES

select r
from   RAS_COLLECTIONNAMES as r
```

Note that the meta collection name, `RAS_COLLNAMES`, must be written in upper case only. No operation in the `select` clause is permitted. The result is a set of one-dimensional char arrays, each one holding the name of a database collection. Each such `char` array, i.e., string is terminated by a zero value (`'\0'`).

12.2 Select

The select statement allows for the retrieval from array collections. The result is a set (collection) of items whose structure is defined in the select clause. Result items can be arrays, atomic values, or structs. In the where clause, a condition can be expressed which acts as a filter for the result set. A single query can address several collections.

Syntax

```
select resultList
from   collName [ as collIterator ]
       [, collName [ as collIterator ] ] ...

select resultList
from   collName [ as collIterator ]
       [, collName [ as collIterator ] ] ...
where  booleanExpr
```

Examples

This query delivers a set of grayscale images:

```
select mr[100:150,40:80] / 2
from   mr
where  some_cells( mr[120:160, 55:75] > 250 )
```

This query, on the other hand, delivers a set of integers:

```
select count_cells( mr[120:160, 55:75] > 250 )
from   mr
```

12.3 Insert

MDD objects can be inserted into database collections using the **insert** statement. The array to be inserted must conform with the collection's type definition concerning both cell type and spatial domain. One or more variable bounds in the collection's array type definition allow degrees of freedom for the array to be inserted. Hence, the resulting collection in this case can contain arrays with different spatial domain.

Syntax

```
insert into collName
values mddExpr
```

collName specifies the name of the target set, *mddExpr* describes the array to be inserted.

Example

Add a black image to collection `mr1`.

```
insert into mr1
values marray x in [ 0:255, 0:210 ]
       values 0c
```

See the *rView Guide* and the programming interfaces described in the *rasdaman Developer's Guides* on how to ship external array data to the server using `insert` and `update` statements.

12.4 Update

The **update** statement allows to manipulate arrays of a collection. Which elements of the collection are affected can be determined with the **where** clause; by indicating a particular OID, single arrays can be updated.

An update can be *complete* in that the whole array is replaced or *partial*, i.e., only part of the database array is changed. Only those array cells are affected the spatial domain of the replacement expression on the right-hand side of the `set` clause. Pixel locations are matched pairwise according to the arrays' spatial domains. Therefore, to appropriately position the replacement array, application of the `shift()` function (see Section 9.2.4) can be necessary.

As a rule, the spatial domain of the righthand side expression must be equal to or a subset of the database array's spatial domain.

See the *rView* manual and the programming interfaces described in the *rasdaman Developer's Guides* on how to ship external array data to the server using `insert` and `update` statements.

Syntax

```
update collName as collIterator
set    updateSpec assign mddExpr

update collName as collIterator
set    updateSpec assign mddExpr
where  booleanExpr
```

where *updateSpec* can optionally contain a restricting minterval (see examples further below):

```
var
var [ mintervalExpr ]
```

Each element of the set named *collName* which fulfils the selection predicate *booleanEpxr* gets assigned the result of *mddExpr*. The right-hand side `mddExpr` overwrites the corresponding area in the collection element; note that no automatic shifting takes place: the spatial domain of `mddExpr` determines the very place where to put it.

Example

An arrow marker is put into the image in collection `mr2`. The appropriate part of `a` is selected and added to the arrow image which, for simplicity, is assumed to have the appropriate spatial domain.

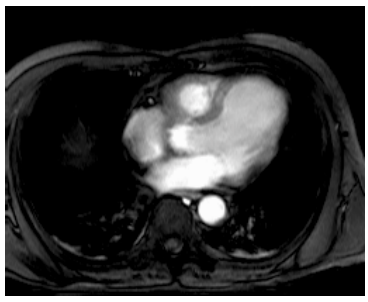

Figure 16 original image of collection `mr2`

```
update mr2 as a
set    a assign a[0:179, 0:54] + $1/2c
```

The argument `$1` is the arrow image (Figure 16) which has to be shipped to the server along with the query. It is an image showing a white arrow on a black background. For more information on the use of `$` variables you may want to consult the language binding guides of the rasdaman Documentation Set.

Figure 17 arrow used for updating

Looking up the **mr2** collection after executing the update yields the following result:

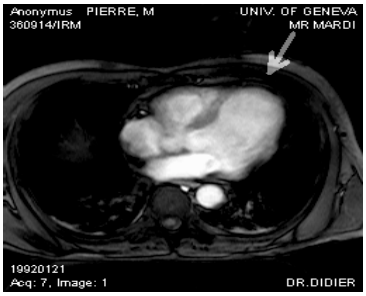

Figure 18: updated collection `mr2`

Note

The replacement expression and the MDD to be updated (i.e., left and right-hand side of the **assign** clause) in the above example must have the same dimensionality. Updating a (lower-dimensional) section of an MDDs can be achieved through a section operator indicating the "slice" to be modified. The following query appends one line to a fax (which is assumed to be extensible in the second dimension):

```
update fax as f
set    f[ *:*, sdom(f)[1].hi+1 ] assign $1
```

12.5 Delete

Arrays are deleted from a database collection using the **delete** statement. The arrays to be removed from a collection can be further characterized in an optional **where** clause. If the condition is omitted, all elements will be deleted so that the collection will be empty afterwards.

Syntax

```
delete from collName [ as collIterator ]
[ where  booleanExpr ]
```

Example

```
delete from mr1 as a
where  all_cells( a < 30 )
```

This will delete all "very dark" images of collection `mr1` with all pixel values lower than 30.

13 Linking MDD with Other Data

13.1 Purpose of OIDs

Each array instance and each collection in a rasdaman database has a identifier which is unique within a database. In the case of a collection this is the collection name and an object identifier (OID), whereas for an array this is only the OID. OIDs are generated by the system upon creation of an array instance, they do not change over an array's lifetime, and OIDs of deleted arrays will never be reassigned to other arrays. This way, OIDs form the means to unambiguously identifiy a particular array. OIDs can be used several ways:

- In rasml, OIDs of arrays can be retrieved and displayed, and they can be used as selection conditions in the condition part.
- OIDs form the means to establish references from objects or tuples residing in other databases systems to rasdaman arrays. Please refer for further information to the language-specific *rasdaman Developer's*

Guides and the *rasdaman External Products Integration Guide* available for each database system to which rasdaman interfaces.

Due to the very different referencing mechanisms used in current database technology, there cannot be one single mechanism. Instead, rasdaman employs its own identification scheme which, then, is combined with the target DBMS way of referencing. See Section 7.4 of this document as well as the *rasdaman External Products Integration Guide* for further information.

13.2 Collection Names

MDD collections are named. The name is indicated by the user or the application program upon creation of the collection; it must be unique within the given database. The most typical usage forms of collection names are

- as a reference in the from clause of a rasml query
- their storage in an attribute of a base DBMS object or tuple, thereby establishing a reference (also called foreign key or pointer).

13.3 Array Object Identifiers

Each MDD array is world-wide uniquely identified by its object identifier (OID). An OID consists of three components:

- A string containing the system where the database resides (system name),
- A string containing the database (base name), and
- A number containing the local object id within the database.

The main purposes of OIDs are

- to establish references from the outside world to arrays and
- to identify a particular array by indicating one OID or an OID list in the search condition of a query.

14 Appendix A: rasdl Grammar

This appendix presents a simplified list of the main rasdl grammar rules used in the rasdaman system. The grammar is described as a set of production rules. Each rule consists of a non-terminal on the left-hand side of the colon operator and a list of symbol names on the right-hand side. The vertical bar "|" introduces a rule with the same left-hand side as the previous one. It is usually read as *or*. Symbol names can either be non-terminals or terminals, the latter ones written in bold face. Terminals either represent keywords, or identifiers, or number literals.

```
typeDef      : structDef
             | marrayDef
             | setDef

structDef    : struct structName { attrList } ;

structName   : ident

attrList     : attrType attrName ; attrList
             | attrType attrName ;

attrType     : ident

attrName     : ident

marrayDef    : typedef marray < typeName >
             | typedef marray
               < typeName, spatialDomain > marrayName ;

typeName     : ident

spatialDomain: [ spatialExprList ]

spatialExprList :
               spatialExprList , spatialExpr
             | spatialExpr

spatialExpr  : integerExpr | intervalExpr

intervalExpr : boundSpec : boundSpec

boundSpec    : integer | *

setDef       : typedef set < marrayName > setName ;

setName      : ident
```

15 Appendix B: rasml Grammar

This appendix presents a simplified list of the main rasml grammar rules used in the rasdaman system. The grammar is described as a set of production rules. Each rule consists of a non-terminal on the left-hand side of the colon operator and a list of symbol names on the right-hand side. The vertical bar "|" introduces a rule with the same left-hand side as the previous one. It is usually read as *or*. Symbol names can either be non-terminals or terminals (the latter ones printed in bold face). Terminals represent keywords of the language, or identifiers, or number literals.

```
query       : createExp
            | dropExp
            | selectExp
            | updateExp
            | insertExp
            | deleteExp

createExp   : create collection namedCollection typeName

dropExp     : drop collection namedCollection

selectExp   : select resultList
              from collectionList
```

```
                  where generalExp
                | select resultList
                  from collectionList

updateExp       : update iteratedCollection set updateSpec
                  assign generalExp
                  where generalExp
                | update iteratedCollection set updateSpec
                  assign generalExp

insertExp       : insert into namedCollection values generalExp

deleteExp       : delete from iteratedCollection
                  where generalExp

updateSpec      : variable
                | variable mintervalExp

resultList      : resultList , generalExp
                | generalExp

generalExp      : mddExp
                | trimExp
                | reduceExp
                | inductionExp
                | functionExp
                | integerExp
                | condenseExp
                | variable
                | mintervalExp
                | intervalExp
                | generalLit

integerExp      : generalExp . lo
                | generalExp . hi

mintervalExp    : [ spatialOpList ]
                | sdom ( collectionIterator )

spatialOpList   : /* empty */
                | spatialOpList2

spatialOpList2  : spatialOpList2 , spatialOp
                | spatialOp

spatialOp       : generalExp

intervalExp     : generalExp : generalExp
                | * : generalExp
                | generalExp : *
                | * : *

condenseExp     : condense condenseOpLit
                  over condenseVariable in generalExp
                  where generalExp using generalExp
                | condense condenseOpLit
```

```
                over condenseVariable in generalExp
                using generalExp

condenseOpLit: +
             | -
             | *
             | /
             | and
             | or

functionExp  : oid ( collectionIterator )
             | shift ( generalExp , generalExp )
             | scale ( generalExp , generalExp )
             | bit ( generalExp , generalExp )
             | tiff ( generalExp , StringLit )
             | tiff ( generalExp )
             | bmp ( generalExp , StringLit )
             | bmp ( generalExp )
             | hdf ( generalExp , StringLit )
             | hdf ( generalExp )
             | jpeg ( generalExp , StringLit )
             | jpeg ( generalExp )
             | png ( generalExp , StringLit )
             | png ( generalExp )
             | vff ( generalExp , StringLit )
             | vff ( generalExp )
             | tor ( generalExp , StringLit )
             | tor ( generalExp )
             | dem ( generalExp , StringLit )
             | dem ( generalExp )
             | csv ( generalExp )
             | inv_tiff ( generalExp , StringLit )
             | inv_tiff ( generalExp )
             | inv_bmp ( generalExp , StringLit )
             | inv_bmp ( generalExp )
             | inv_hdf ( generalExp , StringLit )
             | inv_hdf ( generalExp )
             | inv_jpeg ( generalExp , StringLit )
             | inv_jpeg ( generalExp )
             | inv_png ( generalExp , StringLit )
             | inv_png ( generalExp )
             | inv_vff ( generalExp , StringLit )
             | inv_vff ( generalExp )
             | inv_tor ( generalExp , StringLit )
             | inv_tor ( generalExp )
             | inv_dem ( generalExp , StringLit )
             | inv_dem ( generalExp )
             | inv_csv ( generalExp )
```

```
structSelection :
                . attributeIdent
              | . intLitExp

inductionExp : sqrt ( generalExp )
             | abs ( generalExp )
             | exp ( generalExp )
             | log ( generalExp )
             | ln ( generalExp )
             | sin ( generalExp )
             | cos ( generalExp )
             | tan ( generalExp )
             | sinh ( generalExp )
             | cosh ( generalExp )
             | tanh ( generalExp )
             | arcsin ( generalExp )
             | arccos ( generalExp )
             | arctan ( generalExp )
             | generalExp . re
             | generalExp . im
             | not generalExp
             | generalExp overlay generalExp
             | generalExp is generalExp
             | generalExp and generalExp
             | generalExp or generalExp
             | generalExp xor generalExp
             | generalExp plus generalExp
             | generalExp minus generalExp
             | generalExp mult generalExp
             | generalExp div generalExp
             | generalExp equal generalExp
             | generalExp < generalExp
             | generalExp > generalExp
             | generalExp <= generalExp
             | generalExp >= generalExp
             | generalExp != generalExp
             | + generalExp
             | - generalExp
             | ( castType ) generalExp
             | ( generalExp )
             | generalExp structSelection

castType     : bool
             | char
             | octet
             | short
             | ushort
             | long
             | ulong
             | float
             | double
```

```
                | unsigned short
                | unsigned long

collectionList  :
                  collectionList , iteratedCollection
                | iteratedCollection

iteratedCollection :
                  namedCollection as collectionIterator
                | namedCollection collectionIterator
                | namedCollection

reduceExp       : reduceIdent ( generalExp )

reduceIdent     : all
                | some
                | count_cells
                | add_cells
                | avg_cells
                | min_cells
                | max_cells

trimExp         : generalExp mintervalExp

mddExp          : marray ivList
                  values generalExp
ivList          : ivList , marrayVariable in generalExp
                | marrayVariable in generalExp

intLitExp       : IntegerLit

generalLit      : scalarLit
                | mddLit
                | StringLit
                | oidLit

oidLit          : < StringLit >

mddLit          : < mintervalExp dimensionLitList >
                | $ IntLit

dimensionLitList :
                  dimensionLitList ; scalarLitList
                | scalarLitList

scalarLitList: scalarLitList , scalarLit
                | scalarLit

scalarLit       : complexLit
                | atomicLit

complexLit      : { scalarLitList }
                | struct { scalarLitList }

atomicLit       : BooleanLit
                | IntegerLit
                | FloatLit
                | complex ( FloatLit , FloatLit )
```

```
variable     : Identifier

namedCollection :
               Identifier

collectionIterator :
               Identifier

attributeIdent :
               Identifier

typeName     : Identifier

marrayVariable :
               Identifier

condenseVariable :
               Identifier
```