Making an Array Database Language Server-Side Extensible

Otoniel José Campos Escobar
Jacobs University Bremen
Bremen, Germany
o.camposescobar@jacobs-university.de

Dimitar Misev
rasdaman GmbH
Bremen, Germany
misev@rasdaman.com

Peter Baumann
Jacobs University Bremen
Bremen, Germany
p.baumann@jacobs-university.de

2020 IEEE International Conference on Big Data (Big Data) | 978-1-7281-6251-5/20/$31.00 ©2020 IEEE | DOI: 10.1109/BigData50022.2020.9378108

***Abstract*—Server-side extensibility through dynamically linked external code is a common method in relational databases. In the field of Array Databases such User-Defined Functions (UDFs) sometimes even represent the architectural cornerstone for array functionality. On the downside, UDF implementation often suffers from high coding complexity.**

The rasdaman Array DBMS is a full-stack C++ implementation, so does not rely on some generic UDF mechanism. This allowed designing such an API from scratch, with particular emphasis on UDF coder convenience. The rasdaman UDFs rely on the general C++ client API classes. Based on a straightforward UDF interface definition adapter code is generated automatically. Experimental evaluation shows encouraging results, and the mechanism is going to be used in research and under operational conditions. We present the approach and motivate it through practical use cases.

***Index Terms*—array database, user-defined functions, rasdaman**

I. Introduction

Array Databases [1] [2] [3] [8] [24] close a gap in the database ecosystem by adding modeling, storage, and processing support on multi-dimensional arrays, also called *datacubes*. Such "datacubes" appear as Spatio-temporal sensor, image, simulation, and statistics data in all science and engineering domains, and beyond. For example, 2-D satellite imagery, 3-D x/y/t image time series and x/y/z geophysical voxel data, and 4-D x/y/z/t climate data contribute to today's data deluge in the Earth sciences. Virtual observatories in the Space sciences routinely generate Petabytes of such data. Life sciences deal with microarray data, confocal microscopy, human brain data. Some experts consider matrices a suitable paradigm for the processing of large graphs.

Obviously, all of these are candidates for Big Data, so "shipping code to data" is indispensable. Declarative query languages provide safer ways for such code shipping than procedural code (such as python), but have certain limitations. First, users may not want to rewrite existing algorithms in the given query language, in particular, if complex simulation or Linear Algebra operations are under consideration. Second (and related), there may exist efficient, specialized packages that should be made available as part of the database service. Third, the query language may even not support some operations, such as explicit loops which often are avoided in database query languages to prevent a class of denial-of-service attacks. Finally, sometimes functionality should be provided for free use, but without disclosing the underlying code.

In response to this, database languages early on have been enhanced with User-Defined Functions (UDFs) [12] [13] [14] so as to allow invocation of external code from within a query, executing the associated code on the server through dynamic linking or similar techniques. (We disregard UDFs implemented in a query language like [15] does.) Distinguishing criteria for UDF APIs are ease of development ("How difficult is it to write a UDF?"), safety of the API ("How error-prone is it? Are there simple debugging aids?"), power of the API ("What kind of data can be exchanged?"), and performance ("What overhead is encountered? Are UDFs integrated in performance management, like optimization?").

Array DBMSs have previously used UDFs to implement array functionality. SciDB [17] centers around an modified Postgres kernel where significant array functionality is realized through UDFs. EXTASCID [19] similarly uses UDFs to map array operations to some generic, array-agnostic parallelizing kernel. Similar approaches are taken by PostGIS Raster [21], Teradata Arrays [13], etc., all using generic UDF definition via the system's general object-relational capabilities. However, we believe that classic mechanisms are not always optimal for arrays as these have some characteristic properties that make them different:

- As opposed to tuples or records the arrays considered in Array Databases are huge, often exceeding the main memory of the server. Therefore, tiles get stored, retrieved, and processed in a partitioned ("tile-based") fashion. UDFs may operate on arrays or partitions, depending on the next facet.
- Only some array operations can be executed on partitions naively, many important operations require for each array cell computation a neighborhood of the cell; examples include convolution kernels, matrix multiplication, and regression.
- Arrays are not data types, but data type constructors (in C++: templates), much like stacks and sets. Therefore, a complete implementation through object-relational data types is not possible. They will always need to be

specialized in order to support a hardcoded set of array type instances.

The rasdaman Array DBMS [16] follows a different approach in that it is a full-stack implementation in C++, without any generic object-relational capabilities. The from-scratch design required gives the opportunity of rethinking mechanisms to tune them to arrays. In the rasdaman UDF design handling of partitioned arrays has been adopted directly from the C++ API classes which encapsulate details like partitioning and extent management. Further, hints have been added allowing the engine to adjust evaluation strategy.

In this contribution, we present the rasdaman UDF approach, discuss examples, and present a performance evaluation. The results show that performance is very satisfactory and can be easily scalable. The contribution of this paper is a novel way of supporting array UDFs, including a thorough evaluation considering both usability and performance aspects, indicating that the approach taken is worth considering.

The remainder is organized as follows. In the next section, we give a brief introduction to array handling in databases, followed by the presentation of rasdaman UDFs in Section III. State of the Art is discussed in Section IV. Finally, we have an evaluation in Section V and Section VI concludes the plot.

II. Array Queries in rasdaman

In this section, we give a brief overview of array queries; for further details see [4]. The situation is complicated by the fact that rasdaman has two query interfaces: the domain-agnostic SQL-style rasql language and the domain-specific geo datacube language, WCPS.

A. Database Query Model

Arrays in rasdaman are characterized through their number of dimensions, their extent per axis, and the array cell data type, which is a record structure. Along with this logical definition, a tiling scheme and further physical parameters can be fixed which apply to array instances of that type. This model is embedded in the relational model in that array instances appear as attribute values in some relational tables. In other words, an array type can serve to define a table column.

The query language, *rasql*, adds declarative array operators to SQL to allow retrieval, filtering, and processing on arrays. The language, which in 2019 has been adopted almost verbatim by ISO SQL [1], is based on Array Algebra [3]. At its heart, it defines two second-order operators, an array constructor and a condenser (i.e., aggregator), together with a series of convenience operators derived from those generic ones. As usual in SQL, arbitrarily complex expressions can be built by combining arrays, scalars, etc.

The *constructor* takes an n-D array extent and an expression and builds an array whose cells are filled by evaluating the expression for each array position. For example, the following creates a 100x100 matrix filled with the pairwise difference of cells taken from existing arrays `a` and `b`:

```
MARRAY p in [ 0:99, 0:99 ]
VALUES a[p]-b[p]
```

This can be abbreviated as `a-b`. Any general index computation is possible, though, such as determining changes in an x/y/t timeseries `tx`:

```
MDARRAY x in [ 0:99 ],
        y in [ 0:99 ],
        t in [ 0:99 ]
VALUES  ts[x,y,t]-ts[x,y,t-1]
```

The condenser is somewhat dual in that it iterates over some array area and aggregates based on some aggregation function which is one of the usual suspects `count`, `sum`, `avg`, `min`, `max`, `some`, and `all`. The following expression determines the maximum value from n-D array `a`:

```
MDCONDENSE max
OVER       p in sdom(a)
USING      a[p]
```

Again, there is a shorthand for this simple case, written as `mdmax(a)`. And as before, general expressions and addressing schemes are possible.

Altogether, a typical array SQL query looks like below. Table `LandsatImageTimeseries` contains an attribute `data` constituting an array with many satellite image spectral bands, including `red` and `nir`. From this, the difference of two bands is computed for every tuple, restricted to the x/y/t coordinates indicated in brackets. The result gets encoded in NetCDF, so the query response overall is a (possibly empty) set of NetCDF files.

```
SELECT encode( ls.data.red - ls.data.nir)
               [ x0:x1, y0:y1, t0:t1 ],
               "application/netcdf" )
FROM LandsatImageTimeseries as ls
```

This language allows expressing operations on vectors, matrices, and tensors up to the Discrete Fourier Transform. What cannot be expressed are algorithms that are inherently iterative, such as matrix inversion. Adding iterative power to the language, ultimately enabling complete Linear Algebra, while retaining termination guarantees is an area of active research [8].

B. Geo Service Query Model

Specifically for geo services a business layer, programmed in Java, realizes the semantics of space-time, knows about regular and irregular grids, etc. Its main interface is the Open Geospatial Consortium (OGC) Web Coverage Processing Service (WCPS) geo datacube analytics language [9] [10]. WCPS at its heart has the same processing model as rasql, with two differences: the aforementioned addition of space/time semantics, reflected by the data model defined in the OGC Coverage Implementation Schema (CIS); and a different syntax flavor, geared towards XQuery rather than SQL to better integrate with the mostly XML-based geo metadata handling. Inside

rasdaman, with the help of some geo metadata, WCPS queries are translated into rasql queries. We mention WCPS because UDFs have to support this layer as well, as we will discuss later.

Rather than introducing the language, the following example may serve to give a flavor: "From MODIS satellite scenes M1, M2, M3, return the difference between the red and nir (near-infrared) bands, encoded as TIFF - but only those where nir exceeds 127 somewhere."

```
for $c in ( M1, M2, M3 )
where some( $c.nir > 127 )
return
 encode( $c.red - $c.nir,
         "image/tiff" )
```

Figure 1 shows the result of a WCPS request, visualized in Microsoft Cesium.

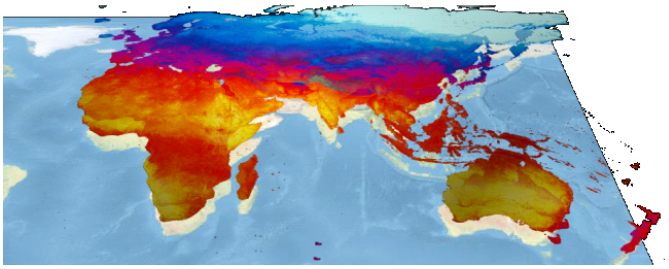

Fig. 1. Sample WCPS query result.

C. Architecture

We present a very brief overview of the rasdaman architecture (Figure 2), as much as is needed for the UDF discussion in the next section. More details can be found in [7] and [5].

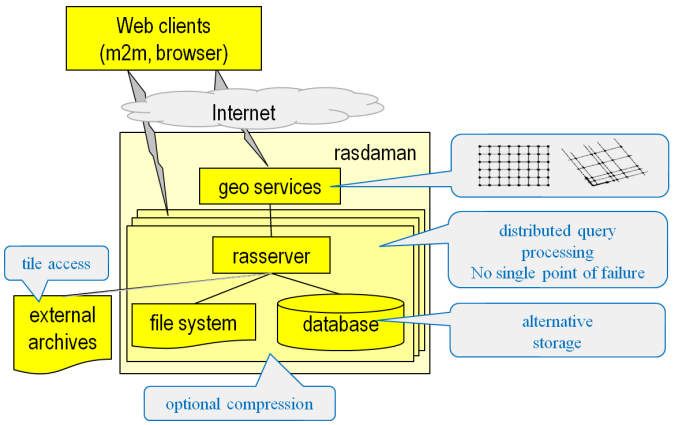

Fig. 2. rasdaman architecture overview.

The rasdaman architecture resembles a standard DBMS architecture implemented in heavily templated C++ with every component specifically optimized for tiled arrays. These components include client APIs (such as C++), query parsing, optimizing, and execution, storage and index management, and several more. Queries are translated into logical trees, then to physical trees, and finally to executable code.

Several languages are supported by rasdaman APIs, including C++, Java, and JavaScript. The C++ API, which is compliant with the ODMG object database standard [6], centers around class `GMArray`, *General Multi-Dimensional Array*. An object of this class contains its structure definition, together with a tiled representation of the array. A cloud of auxiliary functions adds convenience to array handling. This API normally is used to create arrays (maybe by reading contents from files) and forward these for insertion into the database, or to receive arrays in the client for visualization or further processing by the client.

III. Dynamic Extensibility of Array Queries

UDFs are provided on two levels, the geo datacube layer with its WCPS language and the generic database layer with its rasql language. Given the implementation languages of both layers, Java and C++, UDFs need to support these both. We inspect them in turn.

A. rasql UDFs

UDFs in rasdaman can be used in place of any function invocation in both WCPS and rasql. Organisationally, UDFs are grouped into bundles of functionality with designated namespaces. In a query, namespace and function together serve to identify the function (Figure 3).

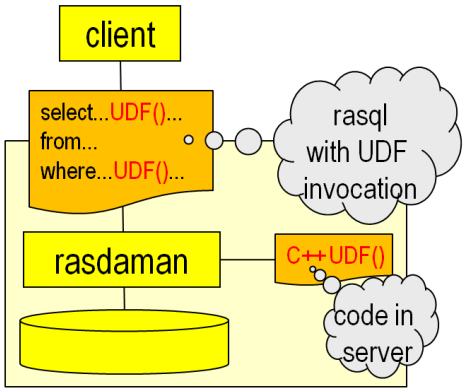

Fig. 3. rasql UDF architecture overview.

The following two sample rasql queries may illustrate syntax and expression embedding, assuming functions `math.fib` and `stat.avg`:

```
SELECT A / sqrt( math.fib( 10 ) ) FROM A
SELECT stat.avg( A[0,*:*,*:*] ) FROM A
```

Definition of a UDF follows common syntax:

```
CREATE FUNCTION namespace . funcName
( typeName1 var1, ..., typeNameN varN )
RETURNS typeName
LANGUAGE cpp
```

`codeSpec`

The code can be provided in place, such as in the following definition:

```
CREATE FUNCTION math.fib ( ulong n )
RETURNS ulong
LANGUAGE cpp
BEGIN
  return n <= 1 ? n
                : fib(n - 1) + fib(n - 2);
END
```

The code provided gets compiled into a dynamic library which gets loaded upon invocation of the UDF.

Alternatively to inlining the code, a reference to some existing dynamic library can be given, assumed to sit in a designated system directory. An example would be:

```
CREATE FUNCTION stat.avg( array a )
RETURNS ushort
LANGUAGE cpp
EXTERN "stat/average.so"
```

The code realizing this function obviously must adhere to the function signature indicated in the definition; for the above example this would be

```
extern "C"
unsigned short avg( r_GMarray *a );
```

Additional adapter code gets generated automatically before compiling and linking; this is necessary for direct plugging into the query operator tree. Common atomic types, such as numbers and strings, are handled in a straightforward manner. Data type `array` internally is interpreted as an object of C++ type `GMArray`. The `extern "C"` keyword inhibits C++ name mangling.

Two keywords give hints to the evaluation engine and optimizer. `DETERMINISTIC` hints the optimizer that the defined function behaves in a deterministic manner, i.e.: it will return the same output on the same input every time; when avoiding this flag the optimizer assumes that the function behaves nondeterministic, through side effects like file read/write, and cautiously avoids some rewriting of the expression.

If `NONBLOCKING` is specified then the engine will optimize execution by generating a series of UDF invocations, one for each tile. Obviously, such a situation is amenable to tile streaming and can be nicely parallelized, commonly referred to as ”embarrassingly parallel”. Keyword `BLOCKING` indicates that an array must be presented to the UDF in its entirety; examples include convolution functions which address into a neighborhood of each pixel and, hence, may miss cells sitting in another tile.

Convenience functions allow the administrator to manage the UDFs, such as listing UDFs:

```
SELECT VIEW FUNCTION LIST
SELECT GET FUNCTION math.fib
```

B. WCPS UDFs

Queries arriving via rasdaman’s geo frontend, in the WCPS language, need to have UDF support as well. Two cases can be distinguished (Figure 4):

- The UDF call actually addresses a rasql UDF; in this case, the call has to be translated while doing a proper translation from the geo objects to arrays.
- the UDF call refers to code actually executed in the geo layer, so directly invoking Java code.

The situation here is different from the C++ environment. The WCPS engine, implemented in Java, runs in a servlet container and, therefore, is subject to the rules of this framework. Invocation of a UDF in a WCPS query is straightforward as the following example shows:

```
for $c in (Sentinel)
return
  encode( math.Quantile( $c.green, 1000 ),
          "json" )
```

The UDF code first gets implemented in the Java language; it must realize the interface

```
WcpsResult
handle( List<WcpsResult> arguments )
```

where `WcpsResult` is a helper class. As before, UDFs are identified by a package and a function name. By convention, the Java package name represents the namespace, the Java class name represents the function name. Also as before, the corresponding jar file needs to be provided in a designated directory, known to the servlet container. No separate registration is required.

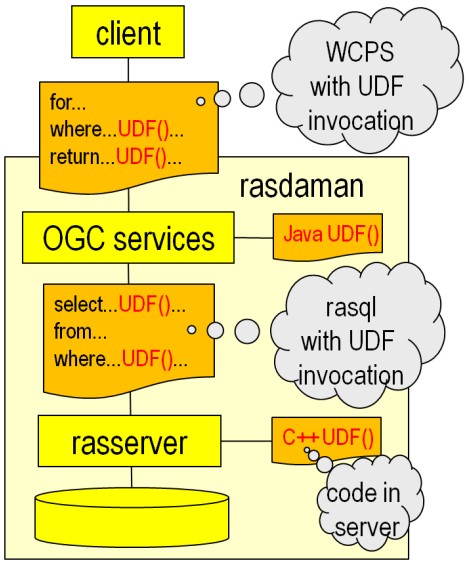

Fig. 4. WCPS UDF architecture overview.

IV. State of the Art

In this section, we will go through what do the major Array DBMS players have to offer in UDF functionality.

A. General UDFs

UDFs have been introduced commonly with object-relational database extensions, emerging from object-oriented databases [6].

Paradise [11] has a very general, powerful UDF mechanism. However, writing and coupling UDFs is described as highly involved even by the Paradise developers themselves.

Parallelization of UDF invocations in an object-relational DBMS has also been studied by Ng et al [12], however not in the context of tiled arrays. Teradata has reported about dynamic scheduling of UDFs evaluating BLOBs, so large objects like arrays are, but again without specific language semantics nor data type knowledge applied [13].

B. Array UDFs

From the large and increasing wealth of array processing systems [2] we explicitly focus on Array DBMSs offering common database functionality on arrays such as a query language, optimization, concurrency control, storage management, etc. These systems may be subdivided into two groups.

- **Full-stack Array Databases.** Systems implemented from the scratch, e.g., rasdaman [16], SciDB [17], ChronosDB [18].
- **Add-ons to existing database systems.** May be implemented by adding extra layers to existing DBMSs (e.g., EXTASCID [19]), performing direct DBMS kernel coding (e.g., SciQL [20]), or providing object-relational extensions (e.g., PostGIS Raster [21], Teradata Arrays [13], Oracle GeoRaster [22]).

SciDB is an Array DBMS following the tradition of rasdaman [2]. It features two query language interfaces: the Array Query Language (AQL) and the Array Functional Language (AFL). It runs on top of a modified Postgres kernel plus UDFs (User-Defined Functions) implementing array functionality and also effecting parallelization. UDFs themselves are defined using AFL. Checking at the latest open-source version at the time of this study, SciDB Community 19.11 [25], the current support for UDFs comes in the form of user-defined macros written into a text file that is later loaded into SciDB by using the `load_module` operator. Below an example of a user-defined macro text that calculates the Euclidean distance between two points [26]:

```
distance(x1,y1,x2,y2) = sqrt(sq(x2-x1) +
sq(y2-y1)) where
{
  sq(x) = x * x;
};
```

Loading a module into SciDB syntax is as follows:

```
load_module('module_pathname');
```

Once loaded in SciDB, the user can invoke the function by specifying the array, or a portion of it, and the user-defined macro name as parameters to the `build` function which will produce a result array. The following is an example of user-defined macro invocation:

```
<AFL> build(<v:double>[i=0:2; j=0:2],
distance(i,i,j,j));
```

User-defined macros in SciDB provide the expressive power to extend the functionality of the Array Database, but only within the boundaries defined by the language itself and the functions that are already provided by the DBMS. To the best of our knowledge, there is no support for user-written UDFs in some external language comparable to the approach we suggest. Consequently, full procedural power is not supported as the SciDB language does not contain any explicit iteration constructs; rasdaman, in contrast, allows the full power of the C/C++ language. SciDB, therefore, requires implementation through an extra layer on top of the database capable of implementing iteration constructs. Performance analysis of the SciDB UDFs has not yet been published.

EXTASCID is an extensible system for scientific data processing [14]. It is capable of supporting natively both arrays as well as relational data. Complex processing is handled by a meta operator that can execute any user code. EXTASCID is built around the massively parallel GLADE architecture for data aggregation. It also provides a robust array data storage model, but it some shortcomings like no portable window-like operator for UDFs.

"Raster" is a data type in the object-relational PostgreSQL geo extension, PostGIS, for storing and analyzing geo raster data [2]. Like PostGIS in general, it is implemented using the extension capabilities of the PostgreSQL object-relational DBMS.

Oracle GeoRaster is a feature of Oracle Spatial that allows storing, indexing, querying, analyzing, and delivering raster image and gridded data and its associated metadata. [2] One important limitation for Oracle GeoRaster UDFs is that they do not allow to pass BLOBs in and out of UDFs.

As mentioned before, Teradata has reported about dynamic scheduling of UDFs evaluating BLOBs [13]. Remarkably, in this context, Teradata does not handle UDFs in its V2 Optimizer.

Optimization of array queries, also involving UDFs, has been studied in [23], albeit without a performance analysis for UDFs. While rasdaman performs such optimization too; this is out of the scope of this paper. In an Array DBMS geared towards astronomical data, the Microsoft SQL server has been used, with UDFs heavily used [24]. However, no rigorous performance evaluation is given so its efficiency remains unclear.

Altogether, while UDFs have been used on arrays in other systems it still remains unclear how efficient they are. The comparative benchmark in [2], revealing that rasdaman can be up to 304x faster than related systems like SciDB, suggests that UDFs in many systems incur a visible performance penalty indeed.

V. Experimental Evaluation

In this section, we describe a performance evaluation of the UDF API. The question to be answered is:

- How much overhead is created by using a UDF?

A. Test Design

We conducted a series of experiments by measuring the elapsed time of passing arrays of 1-dimensional randomly generated 8-bit unsigned integers, *char* atomic type in rasdaman, into a UDF and the native rasdaman implementation of a *max* function that returns the maximum of a given input array. Since we could not modify the source code we adopted the following strategy. Starting from the test arrays, randomly generated binary files were created, in powers of ten increments up to 1GB, using the pseudorandom number generator special file */dev/urandom* which is present in most Unix-like operating systems. Once created, these files were ingested into rasdaman; one collection per file with no tiling in order to avoid possible tile retrieval and memory allocation overhead. Below we show the object metadata retrieved from the *dbinfo()* function in rasdaman for the 100 bytes char collection. Notice *tileNo* = 1; and the *totalSize* and *tileSize* are equal. More information on rasdaman *dbinfo()* function and data types in rasdaman Query Guide [4].

```
Query result collection has 1 element(s):
  Result object 1: {
    "oid": "133121",
    "baseType": "marray <char, [*:*]>",
    "setTypeName": "GreySet1",
    "mddTypeName": "GreyString",
    "tileNo": "1",
    "totalSize": "100",
    "tiling": {
      "tilingScheme": "regular",
      "tileSize": "100",
      "tileConfiguration": "[0:99]"
  },
    "index": {
      "type": "rpt_index",
      "PCTmax": "4096",
      "PCTmin": "2048"
    }
  }
```

The next step was to create the UDF function, the objective of this function was to keep it as simple and concise as possible so no unnecessary overhead time could be introduced into the measurements. Implementation of this UDF `char_max()` function in C++ is as follows:

```
#include "rasodmg/gmarray.hh"
#include "raslib/basetype.hh"
#include "raslib/odmgtypes.hh"
#include "raslib/error.hh"

extern "C"
unsigned char char_max(r_GMarray* m)
{
  const auto size =
      m->spatial_domain().cell_count();

  unsigned char* mv =
      reinterpret_cast<unsigned char*>
      (m->get_array());

  return *std::max_element(mv, mv + size);
}
```

Recall from rasql UDFs subsection, data type `array` is internally interpreted as an object of C++ type `GMArray` by rasdaman. In this implementation we used the `max_element()` function from the C++ standard library which returns a pointer to the maximum value of the array contained in the `GMArray` object. Complexity of `max_element()` is linear in one less than the number of elements compared.

After compiling and generating the *shared object .so* file, we created the UDF inside rasdaman. When creating the UDF we used the default *Blocking* feature which specifies that the function needs to see all tiles in order to run, and the *Determinism* feature which specifies the same result will always be delivered when invoked with the same parameters.

```
create function char_max(array m)
returns char
language cpp
extern "char_max.so"'
```

This `char_max()` UDF was later benchmarked against rasdaman built-in `max_cells()` function that, in a very similar way, returns the maximum of all cell values in the argument array.

As mentioned before, it was not possible to modify the source code, so we adopted the following strategy. We measured the overall elapsed time of both the parameter passing between the database engine and UDF code; and the execution of the code itself. Measurements were taken using the following query:

```
SELECT test.char_max( TestObject )
FROM TestObject
```

Where *TestObject* refers to the name of the collection where the 1D char arrays are stored. This query is run several times and the elapsed run time is measured. To measure the elapsed run time, we used the Linux shell keyword *time* and calculated the mean of all elapsed times for UDF `char_max()` and rasdaman `max_cells()`. Initially, we hypothesized that parameter passing time could be isolated and therefore calculated by first defining the function to return a scalar, hence the decision of using an aggregation function such as *max*, in order to not incur into overhead time as a result of encoding the result and transferring it to the client. We also hypothesized that, to avoid overhead from the query engine, the execution time difference between the built-in function `max_cells()` and the UDF variant could be calculated and thus the overall query overhead eliminated. However, given the fact that rasdaman native `max_cells()` implementation is a generic algorithm and our UDF implementation is specialized for *chars*, we got elapsed time measurements were UDF outperformed the native implementation; meaning parameter passing couldn't be effectively calculated. We considered also

adapting our UDF to make it more generic but discarded the idea as it would bring further complexity into the algorithm that would have not been present in the native implementation. Given this, we, therefore, decided to benchmark the elapsed time of both UDF and native implementation separately and study their behavior as data volume increased.

The computer used for testing was a Dell Inspiron 15 7000 Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz with one physical processor and 2 cores, 4 threads with 8GB of RAM, a page size of 4096 bytes, 128GB of SSD memory, and a swap file of 20GB. The operating system was Ubuntu 18.04.5 LTS with rasdaman Enterprise v10.0.

All queries have been executed from a cold state, with the DBMS restarted and file system caches cleared before every run. Rasdaman was used in *–experimental* mode in order to make use of the latest engine features.

B. Results and Analysis

Our experiments show almost constant execution time until we reach the 10-megabyte threshold (10^7); after this point, execution times grew linearly; which can be expected because data is being copied and consequently its transport dominates the cost. For better visualization, we show the execution times in a logarithmic scale as the data volume increases.

Figure 5 compares the execution time of both the rasdaman `max_cells()` and UDF `char_max()` implementations. Notice from 1-byte (10^0) to 1KB (10^3) rasdaman outperfoms UDF *max* implementation by a margin of 7.07 milliseconds (at 1 byte) to 0.1 millisecond (at 1KB), then from 10KB (10^4) to 100KB (10^5) UDF performs better, rasdaman later outperforms once again at 1MB (10^6); and finally from 10MB (10^7) to 1GB (10^9) UDF clearly outperforms rasdaman as data volumes increase. With larger amounts of data, UDF can play out in its advantage.

It is a well known fact in classical databases [27] that significant time is spent latching and copying data. This could be avoided in order to keep constant time behavior with all data sizes. A possible method could be to execute the query twice so that in the second run all the data is already sitting on main memory and no data copying cost is measured, the resulting elapsed time will only consist of parameter passing through the `GMArray` object.

It also worth considering the fact that UDF `char_max()` is a specialized implementation, only deals with *chars*, and rasdaman `max_cells()` is a generic implementation that is designed to work with any data type. Specialized implementations are known to outperform generic ones as algorithmic complexity is reduced and thus execution time will be reduced as well.

In the case of rasdaman, if we subtract the noise of the homegrown UDF implementation; it will not be faster than rasdaman as we can see on the results produced by the first 4 experiments on Figure 5.

In summary, the rasdaman implementation of UDF linkage turns out rather efficient as the benchmark shows, with an invocation overhead of 7.07 milliseconds. Parameter passing

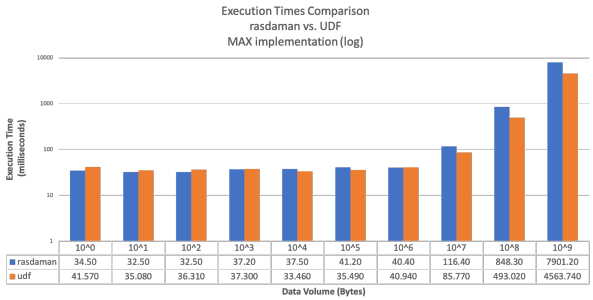

Fig. 5. (Log) Mean Execution Time of rasdaman `char_max()` vs. UDF `max_cells()`.

is negligible up to a size of about 10 megabytes (10^7) and then grows practically linearly with input parameter size, which we find reasonable. Altogether, UDFs can be used freely and without a performance impediment also for complex invocation patterns with many calls to the UDF.

VI. Conclusions

In this contribution, we have presented an interface for C++ UDFs as implemented in the rasdaman Array DBMS. As a further result, we presented a benchmark for UDF implementation; to the best of our knowledge, such a benchmark did not exist yet. As it is non-intrusive to the database kernel it can be transposed to any array-handling DBMS.

The need for implementing a UDF API from scratch, as opposed to re-using existing UDF mechanisms, gave the opportunity to rethink some principles. One design decision was to reuse the existing C++ client API for interfacing of the UDF code with the DBMS engine so that, aside from atomic data types, only a single object class for general multi-dimensional arrays needs to be known by the developer.

The specific nature of arrays - being substantially larger than tuples and database pages, and having a clearly defined neighborhood relation between pixels which operations heavily rely upon - required to differentiate between blocking operations, where array tiles are collected into one object before UDF invocation, and non-blocking operations, where tiles can be streamed into UDFs, possibly in parallel.

Current work includes linking in Linear Algebra packages into rasdaman so that Linear Algebra becomes available on large multi-dimensional arrays, embedded in the general array query language. This is applied, among others, in the DeepRain project where rain forecast in mountainous areas is improved through Deep Learning techniques with the rasdaman Array DBMS adding scalability in a supercomputing environment.

Acknowledgment

This work was supported by EU H2020 EarthServer-2 and German BMBF DeepRain.

References

[1] ISO: Information technology — Database languages — SQL — Part 15: Multi-dimensional arrays (SQL/MDA). ISO IS 9075-15:2019.

[2] P. Baumann, D. Misev, V. Merticariu, B. Pham Huu, B. Bell, K.-S. Kuo: Array Databases: Concepts, Standards, Implementations. RDA Array Database Assessment Working Group Report, 2018, https://rd-alliance.org/system/files/Array-Databases_final-report.pdf. Accessed on 23 Aug 2020.

[3] P. Baumann: A Database Array Algebra for Spatio-Temporal Data and Beyond. Proc. Intl. Workshop on Next Generation Information Technologies and Systems (NGITS), July 5-7, 1999, Zikhron Yaakov, Israel, Springer LNCS 1649, 1999.

[4] rasdaman: Query Language Guide., `https://doc.rasdaman.org/04_ql-guide.html`. Accessed on 23 Aug 2020.

[5] rasdaman: C++ Developer Guide. `https://doc.rasdaman.org/08_dev-guide-cpp.html`. Accessed 23 Aug 2020.

[6] R. G. G. Cattell and Douglas K. Barry: The Object Data Standard: ODMG 3.0. Morgan Kaufmann 2000.

[7] P. Baumann, A.P. Rossi, B. Bell, O. Clements, B. Evans, H. Hoenig, P. Hogan, G. Kakaletris, P. Koltsida, S. Mantovani, R. Marco Figuera, V. Merticariu, D. Misev, B. Pham Huu, S. Siemen, J. Wagemann: Fostering Cross-Disciplinary Earth Science Through Datacube Analytics. In. P.P. Mathieu, C. Aubrecht (eds.): Earth Observation Open Science and Innovation - Changing the World One Pixel at a Time, International Space Science Institute (ISSI), 2017, pp. 91 - 119.

[8] S. Villarroya and P. Baumann: On the Integration of Machine Learning and Array Databases. 2020 IEEE 36th International Conference on Data Engineering (ICDE), Dallas, TX, USA, 2020, pp. 1786-1789, doi: 10.1109/ICDE48307.2020.00170.

[9] P. Baumann: The OGC Web Coverage Pro¬cess¬ing Service (WCPS) Standard. Geoinformatica 14(4)2010, pp. 447 – 479.

[10] P. Baumann: OGC Web Coverage Processing Service (WCPS) Language Interface Standard, version 1.1. OGC document 08-068r3, 2010, https://www.ogc.org/standards/wcps. Accessed 23 Aug 2020.

[11] DeWitt, D. J., Kabra, N., Luo, J., Patel, J. M., & Yu, J. B.: Client-Server Paradise. Proc. VLDB Vol. 94, 1994, pp. 558-569.

[12] K. W. Ng and R. R. Muntz: Parallelizing User-Defined Functions in Distributed Object-Relational DBMS. Proc. International Database Engineering and Applications Symposium (IDEAS), Montreal, Canada, 1999, pp. 442-450.

[13] Cariño, F., & O'Connell, W.: Plan-Per-Tuple Optimization Solution-Parallel Execution of Expensive User-Defined Functions. InProc. VLDB 1998, pp. 690-695.

[14] Dong, Bin, et al. ArrayUDF: User-Defined Scientific Data Analysis on Arrays. Proc. Intl. Symposium on High-Performance Parallel and Distributed Computing, 2017, p. 53-64.

[15] Ramachandra, Karthik, and Kwanghyun Park: BlackMagic: Automatic Inlining of Scalar UDFs into SQL Queries with Froid. Proc. VLDB 2019, p. 1810-1813.

[16] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann, "The multidimensional database system rasdaman," in Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data, ser. SIGMOD '98. New York, NY, USA: ACM, 1998, pp. 575–577. [Online]. Available: http://doi.acm.org/10.1145/276304.276386

[17] M. Stonebraker, P. Brown, A. Poliakov, and S. Raman, "The architecture of scidb," in Proceedings of the 23rd International Conference on Scientific and Statistical Database Management, ser. SSDBM'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 1–16. [Online]. Available: http://dl.acm.org/citation.cfm?id=2032397. 2032399

[18] Ramon Antonio Rodriges Zalipynis. 2018. ChronosDB: distributed, file based, geospatial array DBMS. ¡i¿Proc. VLDB Endow.¡/i¿ 11, 10 (June 2018), 1247–1261. DOI:https://doi.org/10.14778/3231751.3231754

[19] Y. Cheng and F. Rusu, "Astronomical data processing in EXTASCID." in Proceedings of the 25th International Conference on Scientific and Statistical Database Management, ser. SSDBM. New York, NY, USA: ACM, 2013, pp. 47:1–47:4. [Online]. Available: http://doi.acm.org/10.1145/2484838.2484875

[20] Y. Zhang, M. Kersten, M. Ivanova, and N. Nes, "Sciql: Bridging the gap between science and relational dbms," in Proceedings of the 15th Symposium on International Database Engineering & Applications, ser. IDEAS '11. New York, NY, USA: ACM, 2011, pp. 124–133. [Online]. Available: http://doi.acm.org/10.1145/2076623.207663.

[21] PostGIS, PostGIS Raster Manual, 2019. [Online]. Available: http://postgis.net/docs/manual-dev/using raster dataman.html.

[22] GeoServer, Oracle Georaster User Manual, 2019. [Online]. Available: https://docs.geoserver.org/latest/en/user/data/raster/oraclegeoraster.html

[23] Cornacchia, R., van Ballegooij, A., & de Vries, A. P. (2004, June). A case study on array query optimisation. In Proceedings of the 1st international workshop on Computer vision meets databases (pp. 3-10).

[24] Dobos, L., Szalay, A., Blakeley, J., Budavári, T., Csabai, I., Tomic, D., ... & Jovanovic, A. (2011, March). Array requirements for scientific applications and an implementation for microsoft sql server. In Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases (pp. 13-19).

[25] SciDB Documentation 19.11. `https://paradigm4.atlassian.net/wiki/spaces/scidb/overview`. Accessed on 19 Nov 2020.

[26] SciDB Documentation 19.11: `load_module`. `https://paradigm4.atlassian.net/wiki/spaces/scidb/pages/730268777/load+module`. Accessed on 19 Nov 2020.

[27] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. 2008. OLTP through the looking glass, and what we found there. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (emphSIGMOD '08). Association for Computing Machinery, New York, NY, USA, 981–992. DOI:https://doi.org/10.1145/1376616.1376713