On the Efficient Evaluation of Array Joins

Peter Baumann
Jacobs University
Bremen, Germany
28759 Bremen, Germany
e-mail: p.baumann@jacobs-university.de

Vlad Merticariu
rasdaman GmbH
28759 Bremen, Germany
e-mail: merticariu@rasdaman.com

***Abstract*—Array Databases close a gap in the database ecosystem by adding modeling, storage, and processing support on multi-dimensional arrays. Declarative queries provide processing of arrays of regularly massive size, such as Tera- to Petabyte datacubes, while allowing internal degrees of freedom in partitioning the large arrays into tractable sub-arrays. Among the important new operations is the array Theta-Join, such as overlaying two images. Evaluation of such joins is complicated by the fact that the participating arrays likely do not align in their partitioning schemes. This can lead to inefficient multiple reads of sub-arrays.**

We introduce array joins and present an efficient way of pairing corresponding sub-arrays. As a byproduct, this technique delivers information on optimal data placement for parallel join evaluation. The method is implemented in the Array DBMS rasdaman which is in operational use at data centers and mapping agencies.

Array database; scientific database; join; rasdaman

I. Introduction

In the era of NoSQL and NewSQL databases, multi-dimensional arrays meantime are accepted as an additional data category that needs to be supported by databases, next to sets, trees, and graphs. For the representation of sensor, image, simulation output, and statistics data they arguably comprise the larger part of today's Big Data in virtually all science and engineering application domains and beyond. Examples in the Earth Sciences include 2-D satellite image maps, 3-D x/y/t image timeseries and x/y/z geological voxel models, as well a as four-dimensional x/y/z/t weather forecast datacubes. In the Life Sciences we find microarray data, gene expression data, as well as a variety of image modalities like CAT scans. In the Astro Sciences we encounter large-scale cosmological simulations as well as optical and radio telescope observations.

In query processing, arrays behave quite differently from traditional tuples. While in classical relational systems tuples are well below database page size, single array objects easily exceed today's server RAM, such as 4-D climate simulation data cubes with dozens of Terabytes. Hence, partitioning schemes have been introduced which allow to optimize subsetting and to process arrays piecewise, such as tiling [3] [12] and chunking [19].

The most important requirement on any partitioning scheme is to preserve the spatial proximity on array cells induced by the well-defined Euclidean neighborhood of cells. This is a decisive factor for array query performance as array disk access patterns almost always are a function of this neighborhood. In plain words, when a particular cell is accessed it is extremely likely that its neighbor pixels will get accessed, too.

Aside from this spatial clustering of array cells there is a wide open space for partitioning strategies; a systematic study has been conducted by Paula Furtado [12]. A good partitioning strategy will minimize the number of disk accesses and the amount of data to be read from disk for some given access pattern induced by the query workload under consideration. Ideally, a query can be answered with just one disk access. In practice, though, access patterns may conflict.

Consider for example, a satellite image timeseries stored as a 3-D x/y/t array (Figure 1.). One common access pattern is to extract time slices, such as "sea surface temperature in the Tyrrhenian Sea at timestamp 2002-04-12T12:36" (Figure 1. top right). This pattern is best supported by the classical storage in data centers where each incoming image is stored as a, say, GeoTIFF image. Another common pattern gaining much importance in particular for environmental monitoring is "sea surface temperature at position x/y" (Figure 1. bottom right). Such kind of queries obviously are best accommodated by a partitioning that stretches along the time dimension while narrow (in the extreme case: 1x1 pixel) horizontally. Figure 2. shows an early experimental interface to these data (left) and a modern portal on a 130+ TB database hosted by the European Space Agency [5] (right), all using rasdaman.

However, there is no single optimal partitioning, rather solutions have to be determined relative to a given workload. Even a single workload may contain patterns suggesting conflicting partitionings, in which case a tradeoff has to be found. Figure 3. shows some partitionings which are meant to outline the wide range of possible options. Sometimes these may even be non-disjoint and overlapping [21].

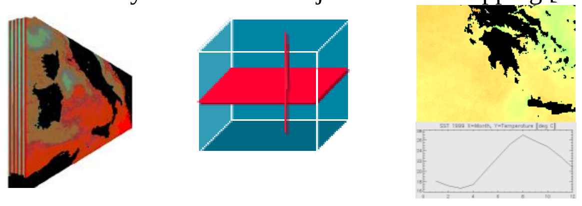


Figure 1. Satellite image timeseries: 3-D extract (left), schematic spatial and temporal datacube subsetting (center), x/y and t cutouts (right); source: rasdaman screenshot, data courtesy DLR-DFD

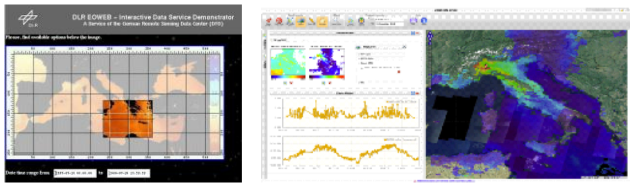


Figure 2. Portal interface for spatio-temporal data selection and processing: early experimental portal by DLR-DFD (left) and recent ESA portal done by MEEO s.r.l.

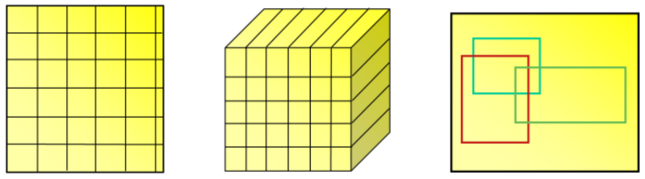


Figure 3. Sample partitioning strategies [12].

Among the various operations an array query language has to offer (see [16] for operators under consideration for the forthcoming ISO SQL array extension) there is the array join which combines two arrays through some operation. Unlike the relational join where a join predicate (such as *R.a=S.b*) determines the tuples retained from the cross product of the operand sets, an array join retains the cells of all locations the arrays share and performs some operation on each matching cell pair. Practical applications are manifold, such as overlaying two images, computing the vegetation index from red and infrared bands, etc.

For example, in Figure 4. a standard situation in Web mapping is shown. The conceptual model consists of 2D image layers stacked on top of each other, each one derived from a base map. The server generates the layers, overlays them, and delivers a single RGB image to the client[1]. In the map overlay on hand, the stack consists of the following layers, from top to bottom:

- an elevation layer coded in red, yellow, green, and transparent to highlight areas endangered by floods.
- two so-called "thematic maps" for water lines and water areas, respectively, stored as bit masks which get colored on the fly in two distinct shades of blue.
- at the bottom, a grayscale airborne image.

Image, thematic, and elevation data are obtained from different sources with individual spatial resolution and are scaled to the output resolution before overlaying.

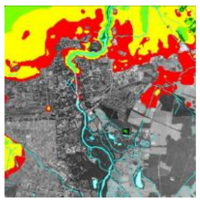


Figure 4. Map generated from four base layers; source: rasdaman screenshot; data courtesy Zwickau municipality, Germany.

[1] Alternatively, clients may assemble the layers into the final image, but our discussion focuses on server-side processing – in particular, as in this server-side approach substantially less data have to be transferred to the client.

This is the simplest case; more complex combinations of arrays – such as matrix multiplications and other Linear Algebra – are conceivable and practically meaningful; however, we claim that the basic problem remains the same: pairwise combining cells form the input arrays based on their matching positions. Hence, the problem might be translated into relational algebra as a join on the coordinates, as we will discuss lateron. In practice, the partitioning supports joins by naturally providing a spatial access method respecting spatial clustering.

During evaluation of array joins under partitioning the engine loads matching partitions from disk as input to the combination function (such as the overlay above). This is straightforward in case both arrays share the same underlying partitioning scheme. In general, however, the patterns will differ, for reasons of different spatial resolution, optimization of each array for different access patterns, etc. Treatment of partially overlapping partitions obviously makes it difficult to retain the optimal situation where each partition is loaded only once, unless we assume that all partitions can be loaded into the engine's main memory, which is unrealistic in view of today's "Big Data".

In this paper we introduce a technique which, based on an arbitrary partitioning of the operand arrays in a join, guarantee a minimal number of multiple reads for the partitions involved. We show that the remaining multiple reads can be determined in advance, and that the algorithm is optimal in that the remaining excessive reads cannot be avoided. The resulting scheduling algorithm for array join evaluation additionally yields a distribution criterion for join parallelization under a shared-nothing regime. The remainder of this contribution is organized as follows. In the next Section, we briefly introduce array management in databases, as far as necessary for our discussion. Section 3 discusses related work. Our array join evaluation method is presented in Section 4, which undergoes a complexity analysis in Section 5. Optimization opportunities are assessed in Section 6, followed by an evaluation in Section 7. Finally, Section 8 concludes the paper.

II. Arrays in Databases

This section gives a sketchy description of array modeling and querying, sufficient for the purpose of this paper. While the concepts discussed in this paper are universal and independent from a particular array model or language we will use the notation of the ISO Array SQL standard under development [15], which is implemented in the the rasdaman ("raster data manager") Array DBMS. In addition to the relevance this language has it is backed by a thorough algebraic formalization suitable for describing queries, optimization, and storage mapping, Array Algebra [6], which is a prerequisite for the array join formalization we undertake. A comprehensive introduction to array queries can be found in [18]; a comparison of different array models has been published in [4] and [9].

A. *Array Modelling and Querying*

Formally, a *d*-dimensional array is a function a: D → V where the domain consists of the *d*-fold cross product of closed integer intervals:

$D = \{lo_1, \dots, hi_1\} \times \dots \times \{lo_d, \dots, hi_d\}$ with $lo_i \leq hi_i$ for $1 \leq i \leq d$

V is some non-empty value set, also called the array's *cell type*. Single elements in such an array we call *cells*.

In terms of operations on arrays, we rely on Array Algebra [6], a minimal formal framework of well understood expressiveness. An array constructor, *marray*, and an aggregator called *condenser* form the two core operations.

The `marray` operator creates an array of a given extent and assigns values to each cell through some expression which may contain occurrences of the cell's coordinate. An example of a unary `marray` operation is deriving the logarithm of some input array of given domain extent *D*:

```
marray x in D
values log( a[x] )
```

An example for a binary operator is addition of two images:

```
marray x in D
values a[x] + b[x]
```

In fact, any binary operation defined on the input arrays' cell types this way "induce" corresponding array operations. We require that both operand arrays share the same spatial extent so that the pairwise matching of array cells is defined. Syntactically, we abbreviate such `marray` operations to the simpler form of

```
a+b
```

This is the kind of operations which introduce array joins, and we will focus on these in the sequel.

Another operation we will need later is array concatenation. For arrays *a* with domain *A* and *b* with *B* we define

```
a concat b := marray x in A ∪ B
              values if x∈A then a[x]
                             else b[x]
```

Obviously, the direct sum of the input domains must be a valid array domain again. (Again, overlapping techniques [21] have no impact on this discussion – by definition they only replicate values, so that the value for each array coordinate remains unambiguously defined.) As an example, consider array *a* with extent [-10:5,-2:2] and *b* with extent [6:10,-2:2]. The resulting domain extent is [-10:10,-2:2] as Figure 5. shows.

It is straightforward to extend concatenation to an *n*-ary function provided the input array domains altogether form a valid array partition (see later for a formal definition).

Only for completeness we briefly mention condensers which act similar to SQL aggregates by iterating over an array and consolidating all values into a single result scalar.

The condensing operation is mentioned explicitly, together with some preprocessing expression which again may contain occurrences of the coordinates visited:

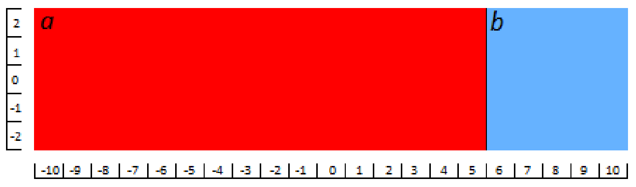


Figure 5. Concatenation of two arrays (color code as in Section 4)

```
condense +
over     x in D
using    A[x]
```

which this case can be abbreviated to `add_cells(A)`.

In passing we note that array operations, being 2nd order with functions as parameters, introduce functionals.

Following ISO SQL we embed arrays into the relational model as a new column type; this approach is shared by the majority of systems. This offers several practical advantages, such as a clear separation of concerns in query optimization and evaluation which eases mixed optimization [17]. Assuming tables *R* and *S* each containing an array-valued attribute *A* and *B*, respectively, we can retrieve a result array set containing the θ-combination of both arrays by writing

```
select R.A θ S.B
from   R, S
```

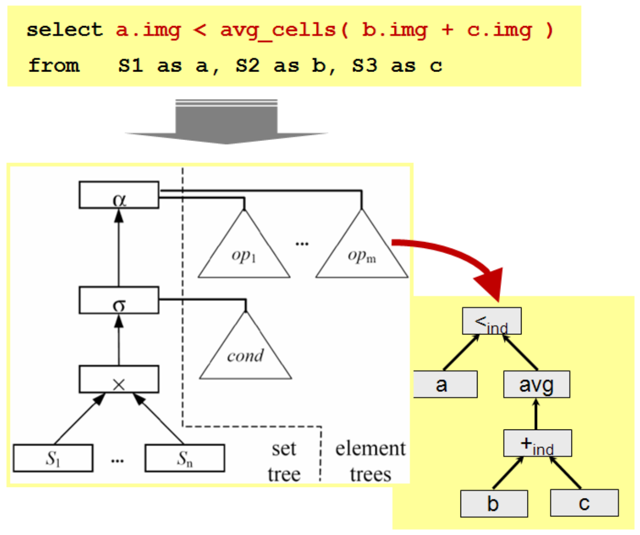
During evaluation the set (i.e., relational) engine first establishes the cross product of *R.A* and *S.B* as usual leading, after projection, to a set of pairs (*R.A*,*S.B*). Each pair are then is forwarded to the array engine as input operands for the array expression. The array engine responds with a single output array for each input pair which the set engine recombines into the final list of result arrays.

This approach has the attractive property that the set mechanics as such remains unaffected (see Figure 6.), the novel part is on micro-level where the concrete input array instances are already provided by the set engine and the result is computed from those. As such, array-valued attributes can be rather massive as compared to the small (i.e., below page size) alphanumeric attribute values their processing deserves particular efficiency considerations.

B. *Array Processing under Partitioning*

A *partitioning* *P* of array *A* with extent *E* is given as a finite set of non-overlapping extents $E_1,\dots,E_n$ for some n∈**N** which together cover *E* completely. In other words, *E* is the direct sum of *P*.

Consider again two arrays *A* and *B* which in this case share a common partitioning scheme *E*. Then, we can rewrite *A* θ *B* as follows:



```
select a.img < avg_cells( b.img + c.img )
from   S1 as a, S2 as b, S3 as c
```

Figure 6. Query tree with set tree and embedded array subtree.

```
A θ B
= marray x in D
  values A[x] θ B[x]
= concat(
    marray x in E1 values A[x]θB[x],
    …,
    marray x in En values A[x]θB[x] )
```

Partitions form the unit of access to persistent storage. Query windows are assessed against the partitioning of the target array by consulting a spatial index. The resulting set of tiles is fetched by the query engine. In order to efficiently process arrays larger than server main memory the physical operators are designed in a way that at any instant in time only a limited number of partitions has to be kept in RAM; this technique is called *tile streaming* [2]. Obviously, there is a large non-deterministic component, as a large number of different iteration sequences is possible. Task on hand, therefore, is to find a sequence which is optimal in the sense that only a minimal number of partition reads have to be performed. While this is trivial for unary operations it is more involved with binary operations as in general the two arrays engaged will have different partitionings. Note, though, that even combining two arrays with identical partitioning may turn out nontrivial. One possible reason for misalignment between the arrays is a different origin in real world (such as a geographic offset) which requires shifting one array before joining them. Another possible reason is a different resolution (such as when combining a Digital Elevation Model with satellite imagery).

III. RELATED WORK

Both mature [1] and younger systems [23][8] perform partitioning into sub-arrays called *tiles* [7] or *chunks* [19][21]. Theoretical foundations for array operations, optimization, and tiling have been laid in Array Algebra [6].

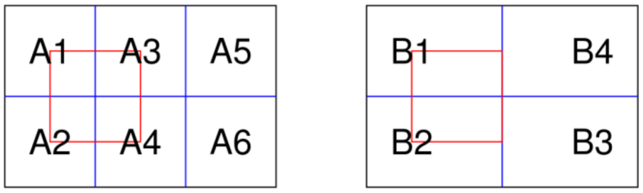


Figure 7. Array partitioning and query windows.

Array DBMSs storage managers usually rely on a partitioning in sub-arrays, with a varying degree of flexibility depending on the system. Furtado et al [12] have established a classification of partitioning schemes for n-D arrays. It is based on the number of common border lines an array's partitions share (Figure 8.). Without discussing the partitioning in detail we notice that it is unlikely that two independently optimized arrays share the same alignment.

SciDB uses regular chunking [19], splitting the array objects into fixed sized chunks whose extent per dimension is specified at ingestion time (Figure 8. left) whereas rasdaman allows arbitrary partitioning (Figure 8. right). rasdaman provides workload specific tiling strategies. The storage organization of an array is dependent on the access pattern used to access its cells. As different queries use different access patterns, there is no partitioning strategy that performs best for all possible queries. However, higher degrees of freedom allow for better adaptation, hence: performance. In the best case scenario, only tiles containing cells relevant to the query are accessed. In the worst case, the entire set of tiles needs to be accessed. Only SciQL [26], a prototype Array DBMS query interface extending MonetDB, relies on its builtin column store manager for serializing arrays. From a formal viewpoint, this can be seen as a border case where partitions uniformly contain exactly one array cell each.

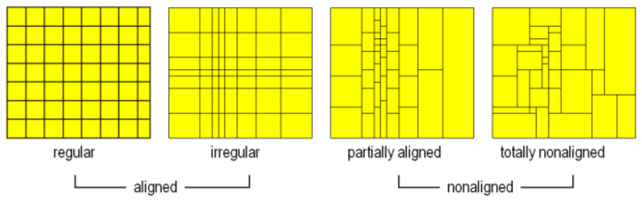


Figure 8. Classification of array partitioning [12].

Partitioning represents a physical tuning parameter available to the database designer or administrator, for example through a storage layout language [3]. ArrayStore [21] integrates this physical property with array definition in a clause that specifies a common partition size and shape:

```
create array myArray
<x:double> [i=0:99,10,0]
```

The statement creates a 1-dimensional array with pixel type double, of domain 0:99, with a chunk size of 10 pixels and overlap 0.

In rasdaman, a dedicated storage layout sub-language allows, as part of the *insert* statement, defining a range of strategies. The above ArrayStore partitioning looks as follows in rasdaman, corresponding to a regular tiling strategy:

```
insert into arrayCollection
values myArray
tiling regular [0:9]
```

If only some hotspots are known (which may well overlap) then the *area of interest* strategy is adequate. Here the system will generate an optimal tiling in the sense that non-overlapping hot spots will be put into a single tile each to provide access through a single disk access; overlapping hotspots will be stored in a minimal set of tiles. For the remaining areas a suitable tiling will be generated automatically. In the *insert* statement below, which generates a new array tuple from reading in a NetCDF file, only the hot spots *area*$_1$, …, *area*$_n$ need to be listed:

```
insert into …
values decode( $1, "netcdf" )
tiling area of interest area1,…, arean
```

Most systems hide the internal partitioning and provide a uniform large array to users. As an exception, PostGIS Raster [27] exposes the tiling structure in the query so that it is the user's task to take care of recombining arrays in joins.

A special case is SciQL [26]. Arrays are stored in the MonetDB column store, that is: they are linearized on disk. As this effectively inhibits spatial clustering, subarray extraction will be inefficient in all directions but one. Likely a dedicated array store will be added at some point.

Storage managers like rasdaman [7] allow arbitrary partitioning. While this flexibility benefits specific user patterns, it makes mismatches in join operations even more likely.

NASA is considering "data rods" [24] for spatio-temporal data cubes made up from satellite images. In this approach a subdivision is chosen with a 1x1 extent in space and the full length of the timeseries along the time axis. This again is a special case of partitioning.

To support so-called focal operations, such as convolutions which require a neighborhood around each pixel for deriving the target pixel, overlapping partitions have been proposed and implemented [23]. For our discussion this feature is not relevant; first, we focus on the pairwise mapping done in the array join, which is known as a local operation in Map Algebra [25]. Second, convolution kernels consist of very small arrays (for example, 3x3), hence they are fully kept in RAM and there is no traversal problem.

In the relational databases world, graph-theoretic models for optimizing disk page access in join operations have been investigated, among others, by Pramanik and Ittner [28]. They propose representing disk pages as graph nodes, two nodes being connected when tuples residing on both are joined. This gives us a first hint on how to represent the array join as a graph problem, however, due to the completely different data models (relational tuples as opposed to arrays), the subsequent steps do not apply in our case.

Effectively, to the best of our knowledge no complete solution is known for the array join problem.

IV. Array Join Algorithm

Assume two arrays A and B with partitioning extent $E_A = \{A_1, \ldots, A_n\}$ and $E_B = \{B_1, \ldots, B_m\}$, respectively, for some $m, n > 0$. Assume further some binary array operation $A \theta B$, such as $A+B$, to be executed on A and B. If both operands fit in main memory, all partitions can be loaded from disk ahead of processing so that cheap random access to every cell can be performed. However, we are interested in situations where the arrays are exceeding RAM so that some partitions have to be swapped out after processing to make way for the next partitions. Unfortunately, in face of inhomogeneous partitioning of the operands it can happen that partitions have to be read and discarded more than once. As it turns out, some partition access sequences require more repeated reads than others.

In the sample situation depicted in Figure 7. , the red query window requires partitions A_1 through A_4 to be combined with B_1 and B_2. The query engine would start, say, with loading A_1 and B_1 to extract the required part from each partition and perform θ cell by cell. After that, the engine may decide to load A_3 next. The area of A_3 is completely contained in B_1 which still is available in RAM, so no new access to B is required. After that, A_2 might get loaded which requires B_2 as its counterpart. As before, advancing to A_4 allows exploiting B_2 completely, after which we are done. Hence, the final load sequence is $<A_1,B_1,A_3,A_2,B_2,A_4>$. It is complete because every partition has been evaluated, and it is minimal because no partition is accessed more than once.

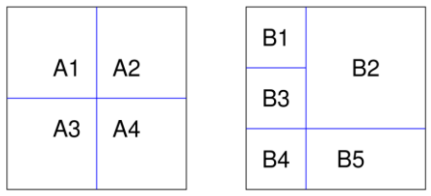


Figure 9. Array join pattern P1.

The task on hand, therefore, is to find a partition traversal sequence which minimizes disk reads, ideally accessing every partition exactly once.

This problem does not necessarily have a unique solution: Any permutation of the load sequence is acceptable, as we do not consider placement on disk etc. For example, the sequence shown before is equivalent to the following (non-exhaustive) list of sequences:

$< A1, B1, A3, B2, A2, A4 >$, $< B1, A1, A3, B2, A2, A4 >$,
$< A1, B1, A3, A2, B2, A4 >$, $< B1, A1, A3, A2, B2, A4 >$

However, the problem does have always at least one solution: there is always a brute-force approach which loads every pair if matching partitions, regardless of possibly multiple loads. For our running example, one suboptimal solution is the following: $< A1,B1,A2,B2,A3,B1,A4,B2>$. With 8 partition accesses it is considerably longer and, hence, more expensive than the above solution of length 6.

A. Partition Traversal as a Graph Problem

Our approach to the array join problem is to map partition correspondences to a graph and apply a graph-theoretic method to determine a traversal sequence efficiently.

Let the *array join graph* $G = (V, E)$ be given by a set of vertices, V, and a set of edges, E. V consists of all partitions participating in the join – in other words, each node represents a partition. The first operand's partition nodes we color in red, the second operand's partition nodes in blue. Any two partitions which are combined during the join – such as A_3 and B_1 – induce an edge in E. Figure 10. shows the graph corresponding to *P1* shown in Figure 9. .

On the side we note that obviously G is a bipartite graph: any edge ends in nodes of different colors if we color partitions according to their array containment (see Figure 10.).

The original problem can now be rephrased to: "Find a complete edge traversal that minimizes the number of nodes visited". This is equivalent to the well-known Königsberg Bridge problem [11] which in fact let Euler to pioneer graph theory. Hierholzer and Wiener have proven that a necessary and sufficient condition for the existence of a solution, a so-called *Euler path*, is that the graph is connected and has exactly zero or two nodes of odd degree [13]. Further, they showed that a closed walk – where start and end node are identical – exists if all nodes have an even degree; this is called an *Euler circuit*.

We enter design of the algorithm devising a suitable traversal by observing that array join graphs can be disconnected, as the example in Figure 11. shows; we overcome this by treating each connected graph separately. Candidate solutions for the array join problem, then, are all permutations of the solutions to the individual separate graph problems.

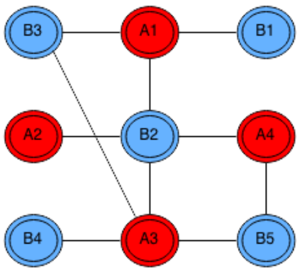


Figure 10. Array join graph G1.

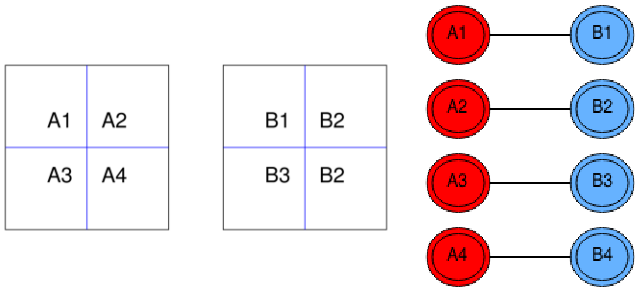


Figure 11. Partitioning P3 and corresponding graph G3.

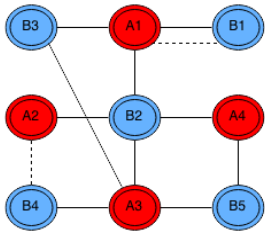


Figure 12. Array join graph G5 with auxiliary edges.

In the sequel we concentrate on connected subgraphs. Obviously, array join graphs can contain any number of nodes with odd degree which prevents to apply the Hierholzer approach. To overcome this we amend the graph with auxiliary edges so that all nodes have an even degree. This allows us to establish a traversal path for each connected component of the graph. In a final cleanup step we optimize the paths by removing the extra partition loads coming from the auxiliary edges as much as possible. This leads us to the algorithm shown in Algorithm 1.

As an example, reconsider situation P1. In the corresponding join graph G5, edges <A1,B1> and <A2,B4> constitute such auxiliary edges, shown as dotted lines in Figure 12. . We note in passing that multiple edges are allowed.

On the resulting graph G5, following Hierholzer and Wiener, an Euler circuit exists. Function *buildJoinGraph*() creates the join graph from the input objects. In practice, this means accessing each array's spatial index with the query bounding box to determine the tiles affected and their extents. The next function, *isolateConnectedComponents*(), splits input graph *G* into a set of connected components to be treated individually in the sequel. Function *buildEulerGraph*() augments some given connected graph in a way that each node has an even degree. This is achieved by adding auxiliary edges in a way that provides best chances for removing them again in the final cleanup stage of *buildTraversalPath*(). The algorithm is shown in Algorithm 2.

ALGORITHM 1: BUILDPATHSET

```
Input:    PA, PB: lists of array partition extents
Output:   TS: set of partition identifier lists
begin
    G := buildJoinGraph( PA, PB );
    set IC := isolateConnectedComponents( G );
    set TS := ∅;
    forall C ∈ IC
    do    C.buildEulerGraph();
          set T := buildTraversalPath( C );
          TS.add( T );
    done;
    return TS;
end
```

ALGORITHM 2: BUILDEULERGRAPH

```
Input:    C: connected join (sub-) graph
Output:   C': augmented join graph
begin
    Odd := { Node n: degree(C,n) mod 2 != 0 };
    # first connect odd nodes sharing an edge:
    forall {odd1,odd2}∈Odd with C.connected(odd1,odd2)
    do    add inner aux edge (odd1, odd2);
          Odd.subtract( { odd1, odd2 } );
    done
    # connect remaining odd nodes of different color:
    forall {odd1,odd2}∈Odd with odd1.color()≠odd2.color
    do    C.addInnerAuxiliaryEdge (odd1, odd2);
          Odd.subtract( { odd1, odd2 } );
    done
    # step 3: connect all remaining nodes, even same color:
    forall {odd1,odd2}∈Odd
    do    C.addOuterAuxEdge (odd1, odd2);
          Odd.subtract( { odd1, odd2 } );
    done
    return C;
end
```

Following this preparation, the traversal path can be established in the core function, *buildTraversalPath*(). Remember that the connected input graph G has been augmented to have only even-degree nodes. This means that every incoming edge to a node will also have a (different) outgoing edge. Following Hierholzer and Wiener we can walk the graph from any starting point without going any edge more than once while still visiting each node (possibly more than once). In other words, whatever walk we take we cannot get stuck at any node and eventually will return to our starting node. We skip this algorithm as it is straightforward.

Based on this, function *buildTraversalPath* () extracts a path involving all nodes and all edges, which exists according to Hierholzer. Note that this is highly nondeterministic as obviously there are many such paths possible.

The overall result delivered by *buildPathSet*(), finally, is a set of paths, one for each connected component of the array join graph. Output of the algorithm is a set of independent traversal sequences which can be executed in any sequence, or in parallel. Within each traversal sequence, the algorithm has made a non-deterministic choice.

The output path set is characterized by the following properties. Path length is |*E*|+1+|*X*| where *E* is the number of partitions involved and *X* is the set of outer edges added. *X* is bounded by the number of nodes with odd degree. This path length is minimal, according to Hierholzer/Wiener plus the strategy of preferring inner auxiliary nodes (which do not contribute to paths) over outer auxiliary nodes: no shorter path is possible under the constraints given.

Traversal paths are cycle-invariant, i.e.: any of the nodes in the sequence may be used instead of the (randomly picked) starting point; when the last element in the sequence is reached a wrap-around to the first one is performed. This degree of freedom may be exploited in the DBMS engine to accommodate some other advantageous criterion, such as minimizing partition lock conflicts, considering cached partitions, etc. Note that all functions uniformly operate on main memory data where each partition is represented only by its bounding extent, which in practice is just a few bytes.

B. Examples

In this section we provide several examples to explain the mechanics of traversal path generation. For a warmup, let us reconsider the case where arrays *A* and *B* share the same partitioning (Figure 11.). We find four connected components which conveniently can be evaluated in parallel. Each component trivially requires loading of one *A* and one *B* partition. The corresponding path set might be:

{ <A1,B1>, <A2,B2>, <A3,B3>, <A4,B4> }

Next, we reinspect the case of G5. In Figure 12. we had added edges A1 – B1 and A2 – B4 to achieve an Euler graph. Paths have a length of 11, one possible choice being:

<B4,$\underline{\text{A3}}$,B2,A1,B1,B3,$\underline{\text{A3}}$,B5,A4,B2,A2>

The outer auxiliary edge leads to a double load of A3, marked by underlining. To see the effect of inner versus outer auxiliary nodes we choose different edges, such as <A2, B1> and <A1,B4> as shown in Figure 13. . This way we obtain two outer auxiliary edges and no inner auxiliary edges. A corresponding traversal path is the following:

<B4,$\underline{\text{A3}}$,B3,$\underline{\text{A1}}$,B1,A2,B2,A4,B5,$\underline{\text{A3}}$,B2,$\underline{\text{A1}}$>

We observe that two nodes, A1 and A3, now get loaded twice (see underlines), leading to an overall path length of 12. This situation might be graphically represented by the unfolded ring in Figure 14. .

C. Array Join

Let us now put all pieces together. The above method forms the core of array join evaluation in that it determines further processing steps. As part of the physical query optimization process, the join operator will first determine the operand arrays' partition by querying the index. Based on this information, *buildPathSet*() establishes the concrete iteration. When it comes to join evaluation, the traversal paths will be used to request partitions in a sequence that allows to combine overlapping parts of the corresponding operand partitions into result partitions. At this time, the result object gets established piecemeal. As we will see later there is ample opportunity for parallelization in this step. Optionally, a retiling of the result object will be performed to avoid partition size underflow.

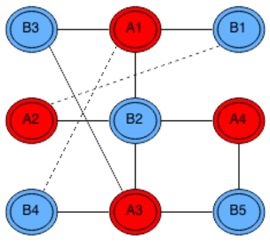


Figure 13. Array join graph G5 with different auxiliary edges.

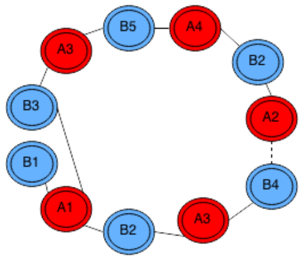


Figure 14. Array join graph G5 unfolded to a ring.

V. Complexity Analysis

Let us briefly analyze the complexity of the resulting algorithm. We consider only disk reads – the number of cells actually combined in the Θ operation obviously is always the same, so performance as well as buffer memory needs, are determined by disk access.

The best case we achieve when every partition of both *A* and *B* is read exactly once. This is achieved when no auxiliary edges have to be added to achieve an Euler circuit. Complexity here is $|E_A|+|E_B|$. In the worst case, *A* is partitioned into compartments of thickness 1 along some axis x_1. Each such compartment stretches across the full array in some axis $x_2 \neq x_1$. Array *B* is subdivided in a way that along axis x_2 partitions have thickness 1 while stretching over the full extent of *B* along axis x_1. Figure 15. shows this for the 2-D case. Data read complexity now becomes $|E_A|*|E_B|$ as every vertical partition has to be matched with every horizontal partition on every overlap. In terms of the join graph we have a situation that every node has an odd degree, hence an according number of auxiliary edges has to be added to achieve an Euler circuit (Figure 16.).

Intuitively, one might think that adding more dimensions leads to even more complicated situations. This, however, is not the case as the above argument shows: 2 orthogonally partitioned dimensions constitute the worst case in that they make all graph nodes have an odd degree. Hence, $|E_A|*|E_B|$ indeed represents a firm upper bound on complexity.

VI. Optimization Opportunities

The information gained form constructing the graph gives important insight into parallelization opportunities. Disconnected parts of the join graph do not need any information exchange for join processing, hence they can advantageously be sent to different compute nodes. If a join between two particular objects is known to happen frequently, or is otherwise important, then the two operand tile sets can be accordingly materialized on different shared-nothing nodes in advance.

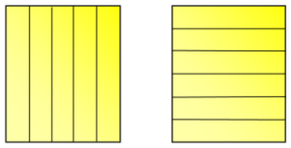


Figure 15. Worst case join scenario on 2-D arrays.

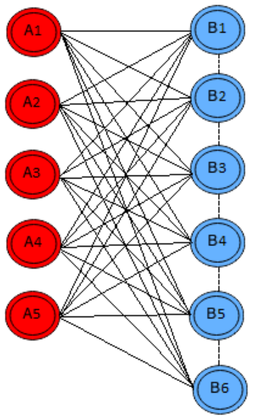


Figure 16. Worst case join graph with auxiliary edges.

Hence, preparatory analysis of the query workload can yield exact data distribution hints. We are currently researching on ways to determine optimal distribution schemes. Even within a connected subgraph there is parallelization potential. Processing a connected graph component can be parallelized locally across the cores of a compute node; the shared-all situation in particular allows sharing of buffers.

Coming back to the extra reads imposed by the traversal, we can take advantage of the path information for buffer management. This is possible because the path contains information about how often a tile load is required. Our implicit assumption so far was that we want to hold only one operand partition in RAM at a time, corresponding to an input buffer of size 1 on each operand. However, by establishing a histogram of each partition's use in a given path we can obtain the number of buffers needed to not reload any partition ever. Even more sophisticated, given operand buffers of some size N_A and N_B the path allows deducing the number of reloads required. This way, a fine-grain tradeoff between main memory and performance is possible.

Currently all partitions, regardless of their size, are considered to bear the same costs for loading. While this is a good first approximation for a well-tuned array database [22], this assumption might be revisited. In particular for cases where some partitions have to be fetched over a networks the cost weights associated to each node will need to be individual.

Further optimizations are currently under investigation, exploiting the degree of freedom given by the nondeterminism of path construction.

VII. Evaluation

We have implemented the array join in the commercial variant of the rasdaman Array DBMS, rasdaman enterprise. We discuss this implementation and compare it against alternatives. The fully naïve approach to joining two arrays would be to materialize both in main memory. While this guarantees minimal disk reads it requires substantial memory because, for some size S of an operand, the memory required is $3S$. Hence, this does not scale to array sizes beyond main memory.The semi-naïve approach taken by rasdaman (both community, the open-source version, and enterprise) fully materializes the first operand and then iterates in a tile-by-tile fashion over the second operand (recall that partitions in rasdaman are called tiles). While this reduces the memory amount from $3S$ to $S+\tau_1+\tau_2$, where τ_i are tile sizes, there remains a main memory limit on the array size on principle. Further, this was an ad-hoc implementation without the conceptual fundament established in this paper. Our new approach overcomes this by requiring only a small, plannable number of tiles to be materialized at any moment in time, thereby overcoming the main memory sizing barrier. Notably, this join is non-blocking: result array tiles can be streamed upwards as they become available.

We have measured the memory usage of rasdaman's execution tree when performing a simple binary operation (in this case addition), between two array objects under different partitioning schemes. First we have used the semi-naïve method described above, and then the new approach proposed in this paper. Both arrays represent red-green-blue images of 18000 by 18000 pixels of type (byte,byte,byte), with a size of approximately 1GB each.

In the first iteration, both arrays have been partitioned using the same scheme: regular tiling [12]. Figure 17. shows the results obtained when the arrays have been partitioned into 4 tiles of approximately 250MB each. The loading sequence of the tiles is shown for each step of the execution. The memory used by the resulting tiles (when performing addition between tiles of each object, the result is written in a separate tile) is also accounted for. However, as soon as the execution step ends, the resulting tile is passed on (either to the next node in the tree if existing or, in our case, to the client) and corresponding memory is freed.

The same operation, this time executed following the graph approach, uses only half of the memory, without performing extra read operations. Moreover, after each execution step, memory usage drops to 0, which indicates that we could execute all the steps in parallel without imposing a shared-memory architecture. The execution steps as well as the memory usage are shown in Fig. 18.

Tile sizes of 250MB are not typical for our applications, however they have been chosen so that the loading sequence can be followed easily in our assessment. In a more practically relevant example, the same objects have been partitioned into regular tiles of 20MB each. Fig. 19 shows the memory usage in the semi-naïve case, which corresponds to our expectations: first operand is fully materialized (so around 1GB of memory), and the second one is loaded tile by tile to produce the result. Fig. 20 displays the memory usage for the same operation, executed following the graph approach, and shows that the maximum amount of memory that is used is below 70MB (which corresponds to roughly two operand tiles + one resulting tile).

Our second experiment targeted the same objects, but partitioned into tiles along perpendicular dimensions (similar to the example in Fig. 15). This means that every tile of the first object intersects every tile of the second object, which implies that every combination of tiles will need to be in memory at some point. Fig. 21 shows the behavior of the semi-naïve algorithm. By comparison, the graph based algorithm uses less memory, however it performs an extra read operation (Fig. 22; A1 is read as step 1, discarded at step 2 and read again at step 8), thus it takes longer.

In order to allow a configurable trade-off between the number of extra reads and maximum memory usage, a buffer that can hold a given number of tiles is implemented. In our case, if we allow the buffer to hold tile A1 between steps 3 and 7, the extra read is not required anymore; results are shown in Figure 23. . Thus, one may choose, depending on the available hardware, a buffer size that ensures a balance between execution time and memory consumption.

In summary, we see that by allowing a modest, controllable amount of extra reads we obtain a scalable algorithm which provides extra information useful for a priori buffer management parallelization.

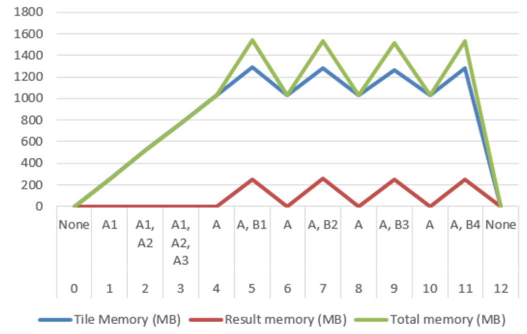


Figure 17. Memory usage: semi-naïve join, tile size 250MB

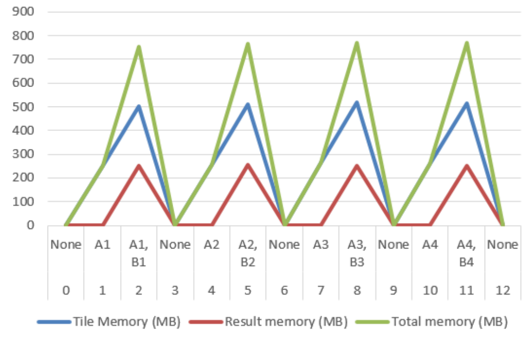


Figure 18. Memory usage: graph-based join, tile size 250MB

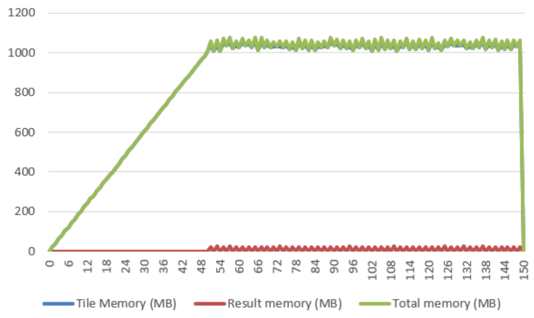


Figure 19. Memory usage: semi-naïve join, tile size 20MB

Finally, we observe that joining two arrays with highly divergent partitioning schemes will result in an array that has a very fine partition granularity. In the worst case, arrays with $|E_A|$ and $|E_B|$ partitions, respectively, will yield an array with $|E_A|*|E_B|$ partitions. This may lead to undesirably small partitions. To avoid this, a repartitioning of the result object may be advisable before continuing query evaluation.

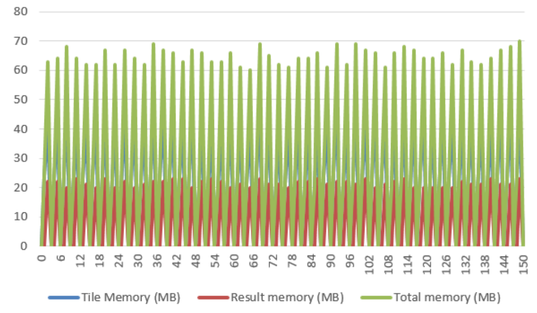


Figure 20. Memory usage: graph-based join, tile size 20MB

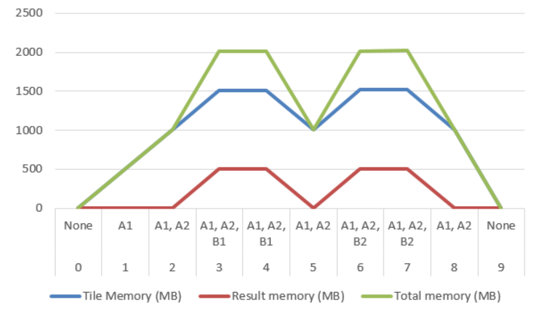


Figure 21. Memory usage: semi-naïve join, perpendicular tiles

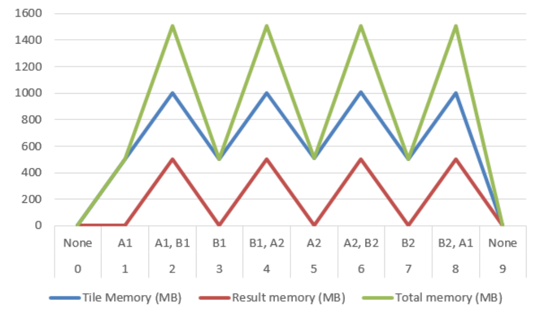


Figure 22. Memory usage: graph-based join, perpendicular tiling, no buffer

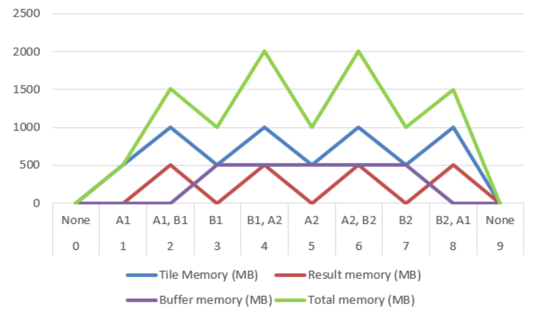


Figure 23. Memory usage: graph-based join, perpendicular tiling, with buffer

VIII. Summary

Increasingly it is recognized that the effort of reorganizing arrays during ingestion pays off substantially with retrieval. With this new degree of freedom, though, the alignment problem in array joins occurs even more frequently.

Combining two arrays into a new one constitutes an array join, a common operation class in Array DBMSs. In this paper, we have proposed a method for finding an efficient traversal on the partitions of the participating arrays. Efficiency is important in face of the large volume even single arrays can comprise. To the best of our knowledge this problem has not been addressed before.

We map the arrays to a bipartite graph where nodes correspond to partitions and edges indicate that the two partitions connected overlap, hence have to be combined during join evaluation. From this graph the array join algorithm deduces the pairings between partitions and finds a minimal sequence of partition accesses. As our graph approach is agnostic of the directions of overlap, this method works for any number of dimensions. The traversal path allows easily determining the number of partition buffers necessary to prevent multiple reads. Conversely, when a certain number of partition buffers is made available for join processing then the traversal path naturally induces an eviction strategy. Furthermore, the graph approach provides valuable information for mixed shared-nothing / shared-all parallelization.

We expect useful insights from the manifold operational rasdaman installations which, in the case of the European Space Agency, already exceed 130 Terabyte of satellite image timeseries datacubes [5]; in the intercontinental EarthServer initiative, intercontinental federations between large-scale satellite and climate data centers are being established with datacubes exceeding 1 Petabyte each. Being able to join such datacubes efficiently will be of critical importance.

Acknowledgement

Part of this work has been supported through European funded projects EarthServer and PublicaMundi supported by the European Commission. Nikolce Kolev's elaboration of examples is appreciated.

References

[1] P. Baumann et al.: *The multidimensional database system RasDaMan*. ACM SIGMOD Record 27(2)1998

[2] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, N. Widmann: *Spatio-Temporal Retrieval with RasDaMan*. Proc. VLDB'99, September 7-10, 1999, Edinburgh, Scotland, pp. 746-749

[3] P. Baumann, S. Feyzabadi, C. Jucovschi: *Putting Pixels in Place: A Storage Layout Language for Scientific Data*. Proc. IEEE ICDM Workshop on Spatial and Spatiotemporal Data Mining (SSTDM-10), December 14, 2010, pp. 194 – 201

[4] P. Baumann, S. Holsten: *A Comparative Analysis of Array Models for Databases. International Journal of Database Theory and Application*, 5(1)2012, pp. 89 – 120

[5] P. Baumann, P. Mazzetti, J. Ungar, R. Barbera, D. Barboni, A. Beccati, L. Bigagli, E. Boldrini, R. Bruno, A. Calanducci, P. Campalani, O. Clement, A. Dumitru, M. Grant, P. Herzig, G. Kakaletris, J. Laxton, P. Koltsida, K. Lipskoch, A.R. Mahdiraji, S. Mantovani, V. Merticariu, A. Messina, D. Misev, S. Natali, S. Nativi, J. Oosthoek, J. Passmore, M. Pappalardo, A.P. Rossi, F. Rundo, M. Sen, V. Sorbera, D. Sullivan, M. Torrisi, L. Trovato, M.G. Veratelli, S. Wagner: *Big Data Analytics for Earth Sciences: the EarthServer Approach*. Intl. Journal of Digital Earth, 2015

[6] P. Baumann: *A Database Array Algebra for Spatio-Temporal Data and Beyond*. Workshop on Next Generation Information Technologies and Systems (NGITS), 1999, Zikhron Yaakov, Israel, LNCS 1649, Springer Verlag, pp. 76 – 93

[7] P. Baumann: *On the Management of Multidimensional Discrete Data*. VLDB Journal, Special Issue on Spatial Database Systems, 1999, pp. 401-444

[8] J.B. Buck, N. Watkins, J. LeFevre, K. Ioannidou, C. Maltzahn, N. Polyzotis, S. Brandt. *SciHadoop: Array-based Query Processing in Hadoop*. Proc. 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC), pp. 66:1–66:11

[9] Y. Cheng, F. Rusu. *EXTASCID: An Extensible System for the Analysis of Scientific Data* (poster). Extremely Large *Databases* (XLDB), Standford, California, USA, September 10 – 13, 2012

[10] EarthServer. www.earthserver.eu, seen on 2015-10-05

[11] L. Euler: *Solutio problematis ad geometriam situs* pertinentis. Commentarii academiae scientiarum Petropolitanae, 1741

[12] P. Furtado, P. Baumann: *Storage of Multidimensional Arrays Based on Arbitrary Tiling*. Proc. IEEE ICDE International Conference on Data Engineering, Sydney, Australia, March 23 – 26, 1999

[13] C. Hierholzer, C. Wiener: *Über die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu* umfahren. Mathematische Annalen 1873, 6: 30–32

[14] M.A. Hong, W.M. Olson, M. Ubell, M. Stonebraker: *Query Processing in a Parallel Object-Relational Database System.* Data Engineering: 3, 1996

[15] J. Melton, P. Baumann: *Information technology – Database languages – SQL – Part 15: Multi-Dimensional Arrays (SQL/MDA).* ISO/IEC JTC1/SC32/WG3 (in preparation)

[16] D. Misev, P. Baumann: *Extending the SQL Array Concept to Support Scientific Analytics*. Proc. Scientific and Statistical Database Management (SSDBM), 2014, Aalborg, Denmark, paper #10

[17] D. Misev, P. Baumann: *Enhancing Science Support in SQL*. Proc. Workshop on Data and Computational Science Technologies for Earth Science Research (co-located with IEEE BigData), Santa Clara, US, October 29, 2015

[18] Rasdaman *QL Guide*. www.rasdaman.org, seen on 2015-10-05

[19] S. Sarawagi, M. Stonebraker. *Efficient Organization of Large Multi-dimensional Arrays*. ICDE, Washington, USA, 1994, pp. 328-336

[20] N.n.: *Seven Bridges of Königsberg*. Wikipedia, http://en.wikipedia.org/wiki/Seven_Bridges_of_K%C3%B6nigsberg, seen on 2015-10-05

[21] E. Soroush, M. Balazinska, D. Wang: *Arraystore: a Storage Manager for Complex Parallel Array Processing*. ACM SIGMOD, 2011

[22] S. Stancu-Mara, P. Baumann, V. Marinov: *A Comparative Benchmark of Large Objects in Relational Databases*. Proc. IDEAS 2008, Coimbra, Portugal, November 10 - 13, 2008

[23] SciDB. www.scidb.org, seen on 2015-10-095

[24] W. Teng et al: *Bridging the Digital Divide to Enhance Access to and Use of NASA Data for the Hydrological Community*. 11th Earth Science Data Systems WG (ESDSWG), Nov. 13 – 15, 2012

[25] C.D. Tomlin: *Geographic Information Systems and Cartographic Modeling*. Prentice Hall 1990

[26] Y. Zhang, M. Kersten, S. Manegold. *SciQL: Array Data Processing Inside an RDBMS.* Proc. ACM SIGMOD, 2013

[27] R. O. Obe, L. S. Hsu: *PostGIS in Action*. Manning Pubs., 2011

[28] S. Pramanik, D.Ittner: *Use of Graph-Theoretic Models for Optimal Relational Database Accesses to Perform Join*. ACM ToDS, March 1985, pp. 57-74.