Location and Processing Aware Datacube Caching

Veranika Liaukevich*
Google Inc.
Montreal, Canada
liaukevich@gmail.com

Dimitar Mišev
Jacobs University
Bremen, Germany
d.misev@jacobs-university.de

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

Vlad Merticariu
Jacobs University
Bremen, Germany
v.merticariu@jacobs-university.de

ABSTRACT

Array databases are used to manage and query large N-dimensional arrays, such as sensor data, simulation models and imagery, as well as various time-series. Modern database systems and database applications make extensive use of caching techniques to improve performance. Research on array databases on the other hand has not explored the potential benefits of caching in query processing on big arrays. In this work we propose a design for a content-aware cache for array databases which allows to reuse results of previously evaluated queries. Besides identical query matching, our method also takes into account *spatially overlapping* queries and queries with *common subexpressions*. We evaluate performance of the query cache implementation by varying data and query parameters and show that it decreases query execution time by up to 93%, with a potential for even higher savings with increasing query complexity.

CCS CONCEPTS

• **Information systems** → **Database management system engines**; **Database query processing**; **Query optimization**;

KEYWORDS

array databases, query caching, datacubes

ACM Reference format:
Veranika Liaukevich, Dimitar Mišev, Peter Baumann, and Vlad Merticariu. 2017. Location and Processing Aware Datacube Caching. In *Proceedings of 29th International Conference on Scientific and Statistical Database Management, Chicago, Illinois, USA, June 27 - June 29 2017 (SSDBM2017),* 6 pages.
https://doi.org/10.475/123_4

1 INTRODUCTION

Large multi-dimensional arrays of data appear in many areas of science and engineering as natural representation for sensor data, images and image time-series, statistics data and simulation results. The quest for achieving flexible, scalable query support on this information category has led to the development of a new type of database management system (DBMS), so-called array databases. Based on query languages offering declarative array operators such systems allow to store and process array data efficiently while allowing for effective optimizations as well as parallel query evaluation. The field was pioneered by rasdaman [4], with more systems following in the meantime: Oracle GeoRaster [7], SciQL [14], SciDB [22], PostGIS Raster [18], EXTASCID [6], etc.

A Web map service where users retrieve map images of a specified geographic region is a common application for array DBMS. Requests are typically generated through a point-and-click interface and internally transformed into queries sent to the server. After a map image is retrieved, users typically zoom in/out or pan to adjacent areas. In the latter case, the server evaluates the same query over different regions. This presents an optimization potential: when the user does not completely leave the previous area, results of previous query executions could be reused in evaluation of subsequent *spatially overlapping queries*. Queries with *common subexpressions* present another possibility for intelligent reuse of previous results. Would the evaluation result of such subexpressions be stored then these could be reused for faster overal evaluation. These are all cases of caching, which is a popular and effective approach to avoid costly recomputations and improve database application performance.

In this work we propose a model of an in-memory array query cache, implement it in the rasdaman Array DBMS, and show evaluation results. We describe the cache capabilities, cache invalidation, and cache entries reuse. A relatively rare feature of our cache model is that it stores in main memory intermediate results of the array query execution, as it was proposed by Zanfaly et. al. in [9] for conventional databases. Our array cache component allows to reuse partial results of previous queries in the case when the current query spatially overlaps with them. To the best of our knowledge the proposed model is the first one to exploit this relation between queries for optimization purposes.

2 BACKGROUND AND RELATED WORK

The idea of caching is widely used in various ways. On a database server it can be a simple cache buffer for reducing the number of disk I/Os or a request cache which stores results of previous requests, thereby reducing computational costs [8]. If it is not possible or not desirable to alter the database server itself, a middle-tier proxy

*research done while at Jacobs University

SSDBM2017, June 27 - June 29 2017, Chicago, Illinois, USA

ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.1145/3085504.3085539

server can be introduced. which forwards incoming queries to the database server while caching results coming back [2, 16].

Caching is used by client-side applications as well, in this case it can significantly reduce network traffic between the client and the server. Some of these applications, which use client-side caching, can predict user behaviour and issue queries to the database in a background process before their results are explicitly requested by the user [15]; on the downside, all clients need to employ a cache as opposed to server-side caching where only one centrally maintained cache exists.

Query caching for relational databases is well investigated – see, e.g., [8]. In [21], full results of queries evaluation are kept in cache, but caching of intermediate results aiming to refine this brute-force approach of "materialize all" has been studied as well. Zanfaly et al. [9] propose to put into the cache results of each intermediate operation required for query evaluation so that queries containing common subexpressions can potentially benefit from the results of previous requests. In [11] the authors propose an automated cost-based selection of which intermediate results should be cached, with more fine-grained control possible through manually specifying in the configuration file the queries that should be cached. In addition, caching of query execution plans has been investigated in [3]

Research has also been published on caching for knowledge bases [1], XML [17], and graph databases [12]. To the best of our knowledge, however, no corresponding work exists for array databases as we propose it.

3 RASDAMAN QUERY PROCESSING

Before explaining the query caching architecture, we briefly describe the data and query model of rasdaman [5]. Multidimensional arrays are the central data type; each array has a domain that indicates its dimensionality, and the lower and upper bounds of each axis. The array elements are typical atomic values (boolean, integer, floating-point), or composite of several atomic components. On storage, the multi-dimensional arrays are partitioned into sub-arrays called *tiles* [13]. A tile is the unit of disk access during the query evaluation process [23]. In this way, the engine efficiently scales to support processing on arrays of unlimited sizes.

3.1 Operations Overview

Globally, the rasdaman query language (rasql) resembles SQL in that it operates on sets. A number of array operations is built into rasql which is based on Array Algebra, a minimal, orthogonal algebraic framework [4, 5]. An operation typically takes one or more arrays/scalars as its arguments and produces a new array or a scalar. Following is a brief overview of the operations important for our discussion [20]. Notably, all of these can be expressed by some combination of *MARRAY* and the *CONDENSE* operations.

Geometric operations reduce or change the array's domain without changing the array cell values. Each dimension can be *trimmed* to a smaller extent, or *sliced* (removed) at a certain index.
Example. `C[0:9,-1:5]`, `C[0,*:*]`.

Induced operations apply a function or an operation, to each cell of the argument arrays; the domain remains unchanged.
Example. `log(C+1)`, `(C+D)/2`, `-B`.

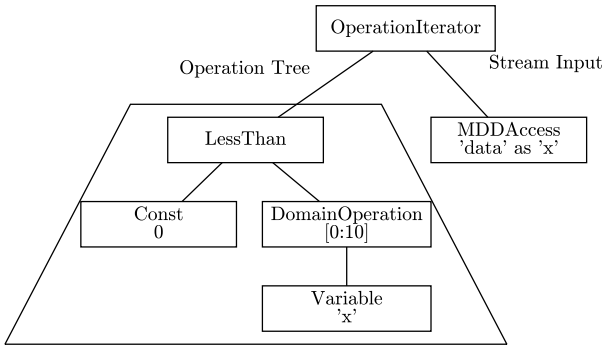

Figure 1: Sample rasdaman query tree.

Case statement allows conditional evaluation of each array cell.
Example. `CASE WHEN C > 128 THEN 0 ELSE C END`.

Array constructor creates an array with a given domain where the value of each cell is given by a coordinate-dependent expression:
Example. `MARRAY x IN [1:5,1:5] VALUES x[0]+1`.

Array condenser aggregate an array into a single scalar.
Example. `CONDENSE + OVER x in sdom(C) USING C[x[0],x[1]]`.

Condense functions are array condenser shorthands.
Example. `max_cells(C)`.

Format encoding and decoding allows to export and import data in common formats.
Example. `encode(C,"tiff")`.

3.2 Query Evaluation

An incoming query is parsed into an internal query tree structure, which is then checked for correctness, optimized and evaluated. Figure 1 shows the query tree corresponding to `SELECT x[0:10]>0 FROM data AS x`. The result of this query is a boolean array of domain `[0:10]`, such that each cell is `true` if the corresponding cell of the array `data` is greater than zero, or `false` otherwise.

Simplified, the query evaluation is carried out as follows: while the stream input is not empty, the `OperationIterator` gets the next array from the `MDDAccess` node, passes it to the operation tree for evaluation and returns the result. The operation tree consists of operation nodes which accept input arguments, possibly call their children nodes, process the data and return results. The caching component is able to cache evaluation results of any subtree of the query operation tree.

4 CACHE DESIGN

We base our cache work on Zanfaly et al. [9] who propose to cache all intermediate results obtained during query execution.

4.1 Common Subexpressions

Consider these (partial) queries:

A: `(x.nir-x.red) / (x.nir+x.red)`
B: `((x.nir-x.red) / (x.nir+x.red)) > 0.5`

Query *A* computes the Normalized Difference Vegetation Index (NDVI) values for each cell of `x`, *B* computes a boolean array with

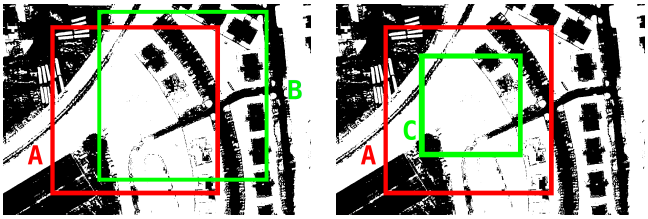

Figure 2: Sample spatially overlapping queries A, B, and C. (Images: http://standards.rasdaman.com).

`true` values where the NDVI is greater than 0.5, thereby filtering the cells that indicate dense vegetation. These two queries share the common subexpression `"(x.nir-x.red) / (x.nir+x.red)"` (modulo consistent variable renaming). One of the requirements to design for our cache component is that cached results of query A should be reused during the evaluation of the query B, so that NDVI values are not computed twice.

4.2 Tiling on Cached Results

Most often query processing is done on array subsets rather than whole arrays. E.g, a map application typically requests the map of a region of interest, not the whole globe. The two main actions in map applications, panning and zooming, usually lead to spatially overlapping queries. On Figure 2a the user computes NDVI over an area A and then moves to the partially overlapping area B. Figure 2b illustrates another case: first, the user requests NDVI over A, then zooms in over the area C such that $C \subset A$.

For such use-cases, it makes sense to tile cached results with the same tiling scheme as the query result, which can be predicted deterministically depending on the operation and tiling schemes of its arguments. For example, suppose we have 2D arrays A and B: A is tiled vertically, whereas B is tiled horizontally (Figures 3a and 3c). Unary induced operations preserve the tiling scheme in the result (Figure 3b). The tiles produced by binary induced operations are computed as intersections of the input tiles (Figure 3d).

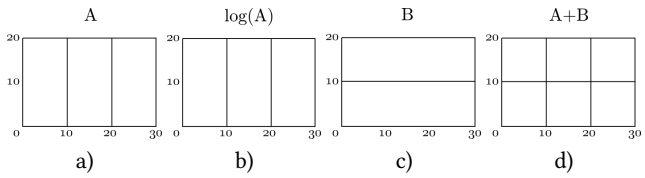

Figure 3: Tiling schemes of induced operations results

During the evaluation process each node in a operation tree needs to compute the result only for tiles, which were not found in the cache. However, it is sometimes not possible to introduce tiling for query evaluation results; the result is cached as a single tile or scalar value in such a case. Condensers, for example, produce scalar results, and the `MARRAY` operator constructs a brand new array.

4.3 Cache Record Identification

We discuss cache record identification using the following convention. R denotes a cache record; C_R is a set of cache records; $Q(R)$ is the query which yielded the record R; $A(R)$ is the list of arguments used by $Q(R)$; and $dom(R)$ is the domain of R. A cache record identifier $id(R)$ consists of three elements: a modified query string $id_Q(Q(R))$, variable bindings list $id_V(A(R))$, and the domain of the record.

$id(R)$ needs to uniquely identify cache records, as well as allow to find and recombine cache records with freshly computed data. Such properties allow to find cache records which are tiles of results of a given query. Method CACHE.GETTILES(Query, Domain, Args) returns a subset of cache records set C_R for a given query, arguments and domain of interest:

$$\begin{aligned} R_c = \;& \text{CACHE.GETTILES(Query, Domain, Args)} \\ = \;& \{R_i : R_i \in C_R \wedge id_Q(Q(R_i)) = id_Q(\text{Query}) \\ & \wedge\; id_V(Q(R_i)) = id_V(\text{Args}) \wedge dom(R_i) \cap \text{Domain} \neq \emptyset\} \end{aligned}$$

A *variable binding list* $id_V(A(R))$ is a list of varname/OID pairs, where *varname* is the name of a variable used in $Q(R)$ and *OID* is the object id substituted for the variable. It is needed in the cache record identifier in order to distinguish multiple results of the same query. For example, let Q be `C + D`, where C contains one array of OID 123, and D has two arrays having OIDs 78 and 90. Q will yield two results with the following variable bindings:

$$id_V(R_1) = \{C = 123, D = 78\},\; id_V(R_2) = \{C = 123, D = 90\}$$

The query string itself cannot be directly used for identification as it makes the identifiers "too" unique and does not work for spatially overlapping queries. For this reason we use a *modified query string* $id_Q(Q(R))$. Consider the two separate queries, A: `C[0:4]+2*C` and B: `C[0:9]+2`. The result of A is a part of B's result. However, with a naive approach to id_Q in this case $id(R_1) \neq id(R_2)$. Therefore, it is necessary to eliminate the information about the accessed domain from the query identifier while preserving its uniqueness, resulting in $id_Q(Q)$.

4.4 Cache Reuse

Cache reuse is possible for queries with tiled results, namely subsetting and induced operations. Furthermore, `marray` constructor results can be reused for queries with the same `values` clause, but operating on some intersecting domain. For many operations, however, the result depends on the whole array, so reuse is possible only for exactly matching queries. One example is format encoding.

4.5 Cache Invalidation

It is important to clear all records from the cache which are not valid after the corresponding base data have been updated in the database. This includes `DELETE` and `UPDATE` queries, whereas `INSERT` and `SELECT INTO` require no action as they create new arrays. Suppose there are several cache records resulting from a query. If some region of that array gets updated only some of the records become invalid. Unfortunately, it is not always possible to determine whether a certain region of an array was used to compute the cache record or not. For example, assume `data` and `index` being 2-D collections, and cache record R is the result of query Q:

```
MARRAY x IN sdom(index) VALUES data[index[x[0],x[1]]]
```

In Q, cell values of `index` arrays are used to index the `data` data. If a region of the array in `data` is updated, there is no way to determine statically (i.e. without inspection of `index` values) whether this update changes the validity of R or not. As a result, in presence of complicated addressing schemes we prophylactically invalidate cache records whenever one of its arguments has changed.

We call *invalidation policy* of a cache record the decision whether the record should be invalidated whenever its argument object was updated ("invalidation by object") or it should be updated whenever a certain region of the argument object was updated ("invalidation by region"). Invalidation policy is a property of a cache record and it depends on the query which has produced the record.

4.6 Cache Rules

Storing all intermediate evaluation results in the shared memory is expensive. Considering that in most use cases only a fraction of these stored results will be reused, it is useful to allow database administrators to specify patterns (*cache rules*) for query subexpressions which should be put into the cache. A cache rule consists of a query pattern and an arguments rule variables binding list. In *query patterns* an underscore '_' matches any expression. Examples:

- '_': Matches any query
- '`log(x)`': Matches only '`log(x)`'
- '`log(_)`': Matches '`log(x)`', '`log(x + log(y))`', but not '`(log(x) + log(y))`'
- '`(_ + _)`': Matches '`(x + (y * z))`', '`(log(x) + log(y))`', but not '`(x * (y + z))`'

The *arguments rule* restricts which results (tuples) should be cached. Suppose C has two arrays with OIDs 123 and 456 and collection D has two arrays with OIDs 78 and 90; the query `C+D` will yield four results, one for each pair of objects from C and D. Examples:

- arguments rule {} matches all four results of the query;
- {C=123} matches two results of the query;
- {C=123, D=123} does not match any of results.

5 PERFORMANCE EVALUATION

5.1 Fine-grained Evaluation

First, we compared query evaluation performance with enabled and disabled caching component. The experiment was done on a 10950x5475 RGB image of 171 MB, partitioned into 1095x1095 tiles of 3.6 MB each. The machine had 6 GB RAM, 128 GB SSD, and two 2 GHz dual-core CPUs. We ran the same set of queries with different cache memory limits to achieve different cache hit rates, from 10% to 100% hit rate (i.e. all data fits in cache).

hit rate	x	x.red	log(x.red+1)
0% (base)	85.74 ms	116.45 ms	262.94 ms
10%	(+14.8%)	(+0.6%)	(+5%)
50%	(−33.3%)	(−17.9%)	(−15.6%)
90%	(−73.8%)	(−65.9%)	(−66.5%)
100%	(−82.5%)	(−86.7%)	(−93.7%)

Table 1: Mean processing times with varying cache hit rate

In all these setups we tested three queries for induced operations. Each query requested a random 1000x1000 region of the input data, so that each query computation involves 1, 2 or 4 tiles.

- `x[...]`: simple data retrieval query.
- `x.red[...]`: extracting red channel.
- `log(x.red+1)[...]`: complex induced operation.

During the experiments, for each setup we issued 1000 requests and calculated the mean of server-side processing time. Table 1 and Figure 4 show these results.

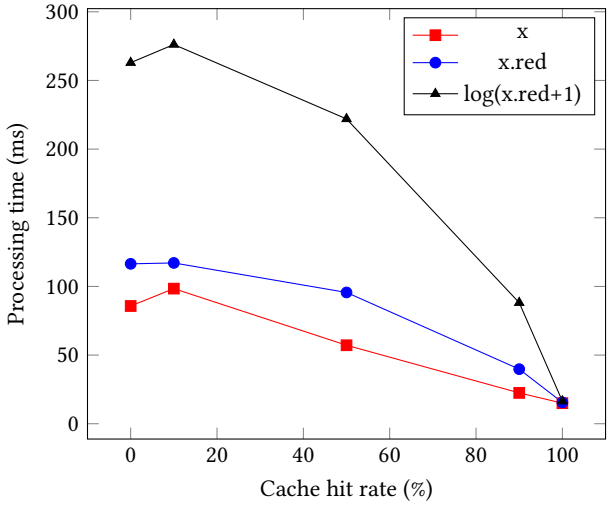

Figure 4: Mean processing times with varying cache hit rate

Caching has its own cost (computing the result tiling schemes as described in Section 4.4, shared memory management, creating cache index entries, etc.); we see this reflected in the performance drop for experiments with 10% hit rate. Query evaluation speedup decreases as cache hit rate increases, which is also as expected. We also observe that the more complex query is cached, the greater query evaluation speedup is achieved: for simple data retrieval queries we got ≈ 82.5% speedup, but up to ≈ 93.7% for time-consuming logarithm computation.

5.2 System Evaluation

We have then evaluated the cache in a more real-life setting, comparing the design in this paper which was implemented in rasdaman, with SciDB, another major array DBMS. The systems were installed with their default configurations on a single machine (Table 2 shows the specs), and only the caching parameters were modified as needed for the benchmark.

OS	Ubuntu 14.04.4 Trusty
CPU	2 x Intel Xeon E5-2609 v3 (12 cores @ 1.90 GHz)
RAM	4 x 16 GB 2133 MHz
Disk	3TB 7200 RPM, read 193 MB/s, write 162 MB/s

Table 2: Benchmark machine specification.

SciDB offers two parameters to control caching [19]:

mem-array-threshold Maximum size in MB of temporary data to cache in memory before writing to temporary disk files. Default: 1024 MB.

smgr-cache-size Size of memory in MB allocated to the shared cache of array chunks. The cache is used only for the chunks belonging to persistent arrays. Default: 256 MB.

It is not clear which one corresponds more to the caching solution outlined in this paper, so the cache size was equally divided to both in the benchmarks.

In rasdaman the maximum cache size can be set in the server configuration file `rasmgr.conf` with the command `define cache -size S`, where `S` can be expressed as percent of all available memory (e.g. 50%), or absolute amount in bytes, MB or GB.

It would be ideal to use real data (e.g. Landsat 8), however it proved very difficult to insert TIFF files into SciDB which natively only supports binary and CSV encoded data. For this reason we decided to generate a random 2D array of double values with total size of 1GB as test data. In both systems the data was inserted with tile sizes of 16MB (1414 x 1414 x 8 bytes).

The benchmark consisted of several sessions of different queries. Each session was started cold – services were restarted and the OS cache was cleared with `echo 3 > /proc/sys/vm/drop_caches`. However, the queries within a session were run hot in order to measure the impact of caching. Queries were executed serially, so that benchmark runs are deterministic and repeatable. In all queries the final result is an aggregation (typically the minimum value); we chose to do this as retrieving the whole array on client side was up to 10 times slower in SciDB than in rasdaman, which would greatly skew the benchmark results.

5.2.1 Web Map Service. This session emulates a Web Map Service that automatically generates queries to select, scale and encode the map that is currently in view at a certain zoom level. Usage of map services typically include zooming in an out over overlapping areas, and presents a good case for cache application.

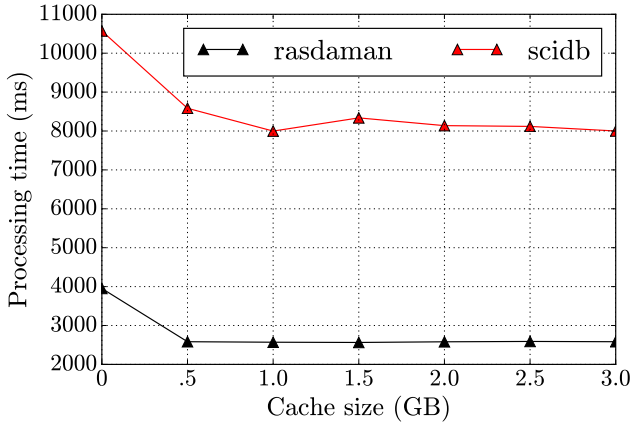

Figure 5: Sliding a subset window of size 4000x4000 from left to right. The execution time is a total of 10 queries with each successive query moving the window 500 pixels to the right.

We ran four type of sessions: panning from left to right, panning diagonally from lower left to upper right, zooming in (selecting a smaller and smaller region), and zooming out. Figure 5 shows the benchmark results for the left to right panning queries; the graphs for the others are very similar and have been left out. As can be noticed both systems benefit from caching up to a certain cache size, after which more cache size does not entail benefit (500MB for rasdaman, 1GB for SciDB).

5.2.2 Repeating Unary Operations. In this session a particular unary operation is chosen, and each successive query applies it one more time than the previous query. For example, for the cosine function the following queries are executed: $cos(A)$, $cos(cos(A))$, etc.

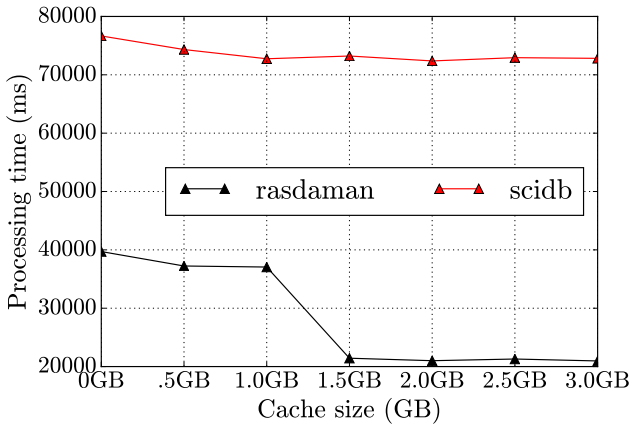

Figure 6: Repeatedly applying a cosine operation on the double values of a 1GB matrix. The graph shows an aggregate of 10 queries, where the first one just loads the array, and the last one applies cosine 9 times.

In theory the cache would save the cosine result of a query and the next query would only need to apply one cosine operation on the saved result. This is exactly what can be noticed in the case of rasdaman on Figure 6: as soon as the maximum cache size is greater than 1GB, the results can be fully cached and processing time stays linear from one query to the next.

5.2.3 Repeating Binary Operations. Similarly to the previous benchmark case, here the caching of multiple binary operations (comparison, arithmetic, logical) is covered. Figure 7 shows the aggregated processing times of 10 queries, where the first query has a single operation comparing every value of the array to a random number, i.e. $A = 1$, the second then adds one more comparison to the first case, e.g. $A = 1$ and $A = 8$, and so on. Similarly to the previous case with the unary operations, the cache shows significant positive impact on the performance.

Further, on Figure 8 the measured execution times of each query per varying cache sizes is shown. In this case the dotted lines represent the varying cache sizes. As can be expected, when the cache is disabled (0GB), the runtime of each query increases linearly. With .5 and 1GB cache sizes, the runtime of each query is constant,

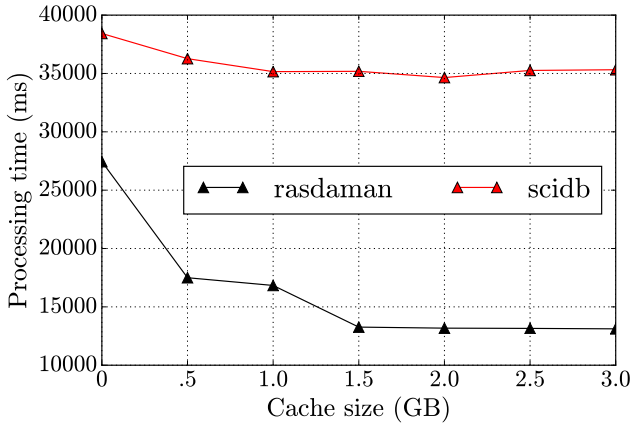

Figure 7: Aggregated execution times of 10 queries with multiple equality comparisons.

but finally the optimal performance is achieved with cache sizes above 1GB in this case.

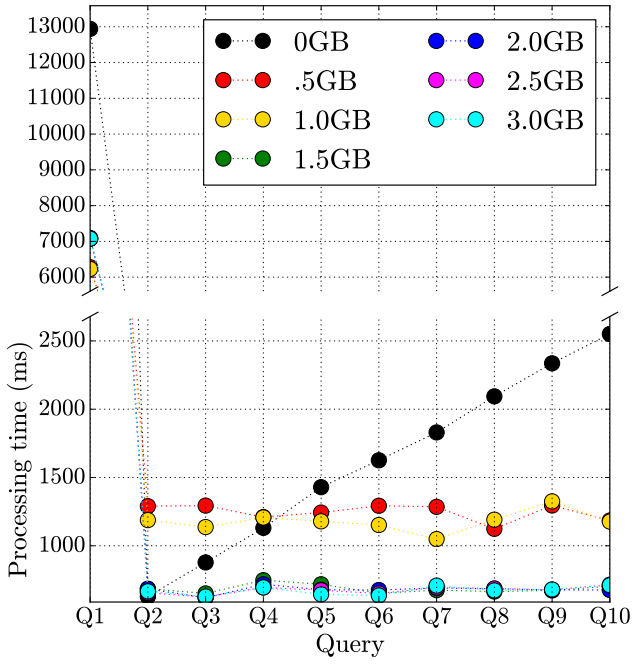

Figure 8: A break down of execution times per query with multiple equality comparisons.

6 CONCLUSION

We have proposed a query caching concept specifically crafted for array databases. It is based on maintaining intermediate query evaluation results and allows to reuse cached data for common subexpressions referring to the array area which may be identical or partially overlapping. An algorithm for integrating the proposed query cache into the query evaluation process has been described and details of cache record identification and invalidation have been covered. The algorithm has been implemented in rasdaman. Experimental results have revealed that the more large and/or complex query results are cached, the more relative speedup can be achieved. In our tests – which will be extended in a next step – we have seen savings of 93% corresponding to a performance speedup factor of 14. Further work includes application of the cache for complex processing and visualization on timeseries datacubes of sizes between 20 Terabytes and 1 Petabyte in the course of the EarthServer project [10]. Also, we plan to extend the main memory caching to include persistent storage for materializing intermediate query results.

ACKNOWLEDGEMENTS

Work has been supported by the European Commission under FP7 EarthServer and H2020 EarthServer-2.

REFERENCES

[1] Sibel Adali, K Selçuk Candan, Yannis Papakonstantinou, and VS Subrahmanian. 1996. Query Caching and Optimization in Distributed Mediator Systems. In *ACM SIGMOD Record*, Vol. 25. 137–146.
[2] Mehmet Altinel, Qiong Luo, Sailesh Krishnamurthy, C Mohan, Hamid Pirahesh, Bruce G Lindsay, Honguk Woo, and Larry Brown. 2002. DBcache: Database Caching for Web Application Servers. In *Proc. SIGMOD*. ACM, 612–612.
[3] Gopi Krishna Attaluri and David Joseph Wisneski. 2002. Method and System for Transparently Caching and Reusing Query Execution Plans Efficiently. (2002). US Patent 6466931.
[4] Peter Baumann. 1994. Management of Multidimensional Discrete Data. *VLDB Journal* 3, 4 (1994), 401–444.
[5] Peter Baumann. 1999. A Database Array Algebra for Spatio-Temporal Data and Beyond. In *Proc. NGITS*. Springer, 76–93.
[6] Yu Cheng and Florin Rusu. 2013. Astronomical Data Processing in EXTASCID. In *Proc. SSDBM*. ACM, Article 47, 4 pages.
[7] Oracle Corporation. 2008. *Oracle Spatial GeoRaster Developer's Guide, 11g.*
[8] Oracle Corporation. 2014. *Guide to Database Performance and Tuning, 11g.*
[9] Doan El Zanfaly, AS Eldean, and Ammar RA. 2003. Multilevel caching to speedup query processing in distributed databases. In *Proc. 3rd IEEE International Symposium on Signal Processing and Information Technology*. IEEE, 580–583.
[10] Baumann et al. 2015. Big Data Analytics for Earth Sciences: the EarthServer Approach. *International Journal of Digital Earth* (2015).
[11] Bornhovd Christof et al. 2004. Adaptive Database Caching with DBCache. *IEEE Data Eng. Bull.* 27.2 (2004), 11–18.
[12] Arash Fard, Satya Manda, Lakshmish Ramaswamy, and John A. Miller. 2014. Effective Caching Techniques for Accelerating Pattern Matching Queries. In *Proc. IEEE Intl. Conf. on Big Data*. IEEE.
[13] Paula Furtado and Peter Baumann. 1999. Storage of Multidimensional Arrays Based on Arbitrary Tiling. In *Proc. 15th Int. Conf. on Data Eng.* IEEE, 480–489.
[14] Martin Kersten, Ying Zhang, Milena Ivanova, and Niels Nes. 2011. SciQL, a Query Language for Science Applications. In *Proc. EDBT/ICDT 2011 Workshop on Array Databases*. ACM, 1–12.
[15] Chang Liu, Brendan Fruin, and Hanan Samet. 2013. SAC: Semantic Adaptive Caching for Spatial Mobile Applications. In *Proc. ACM SIGSPATIAL*. ACM.
[16] Qiong Luo, Sailesh Krishnamurthy, C Mohan, Hamid Pirahesh, Honguk Woo, Bruce G Lindsay, and Jeffrey F Naughton. 2002. Middle-Tier Database Caching for e-Business. In *Proc. SIGMOD*. ACM, 600–611.
[17] Bhushan Mandhani and Dan Suciu. 2005. Query Caching and View Selection for XML Databases. In *Proc. VLDB*. VLDB Endowment, 469–480.
[18] R. Obe and L. Hsu. 2011. *PostGIS in Action*. Manning Pubs.
[19] paradigm4. 2016. *SciDB Documentation 15.12.*
[20] rasdaman GmbH. 2016. *rasdaman Query Language Guide* (9.3 ed.).
[21] T. Sellis. 1988. Intelligent caching and indexing techniques for relational database systems. *Information Systems* 13.2 (1988), 175–185.
[22] Michael Stonebraker, Paul Brown, Alex Poliakov, and Suchi Raman. 2011. The Architecture of SciDB. In *Proc. SSDBM*. Springer-Verlag, 1–16.
[23] Norbert Widmann and Peter Baumann. 1998. Efficient Execution of Operations in a DBMS for Multidimensional Arrays. In *Proc. SSDBM*. IEEE, 155–165.