Incremental View Maintenance over Array Data

Weijie Zhao[1] Florin Rusu[1,2] Bin Dong[2] Kesheng Wu[2] Peter Nugent[2]
[1]UC Merced, [2] Lawrence Berkeley National Laboratory
{wzhao23, frusu}@ucmerced.edu, {dbin, kwu, penugent}@lbl.gov

ABSTRACT

Science applications are producing an ever-increasing volume of multi-dimensional data that are mainly processed with distributed array databases. These raw arrays are "cooked" into derived data products using complex pipelines that are time-consuming. As a result, derived data products are released infrequently and become stale soon thereafter. In this paper, we introduce materialized array views as a database construct for scientific data products. We model the "cooking" process as incremental view maintenance with batch updates and give a three-stage heuristic that finds effective update plans. Moreover, the heuristic repartitions the array and the view continuously based on a window of past updates as a side-effect of view maintenance without overhead. We design an analytical cost model for integrating materialized array views in queries. A thorough experimental evaluation confirms that the proposed techniques are able to incrementally maintain a real astronomical data product in a production environment.

1. INTRODUCTION

Scientific applications – from astronomy to high-energy physics and genomics – collect and process massive amounts of data at an unprecedented scale. For example, projects in astronomy such as the Sloan Digital Sky Survey[1] (SDSS) and the Palomar Transient Factory[2] (iPTF) record observations of stars and galaxies at nightly rates varying between 60 and 500 GB. The Large Synoptic Survey Telescope[3] (LSST) is projected to increase these volumes by two orders of magnitude—to 20 TB. A common feature of the datasets produced by these astronomy projects is that data are represented as *multi-dimensional arrays* rather than unordered sets—the case in the relational data model. Due to the inefficacy of traditional relational databases to handle ordered array data [9, 16], a series of specialized array processing systems [6, 4, 9, 55, 10, 11] have emerged. These array processing systems implement natively a distributed multi-dimensional array data model in which arrays are chunked across a *distributed shared-nothing cluster* and processed concurrently.

According to [49], queries on arrays fall in two categories. The first category consists of relational-style operations that can be executed efficiently by any traditional database. They include filtering, sub-sampling, join, and group-by aggregates. The second category – containing array-specific operations such as smoothing, regridding, clustering, and cross-matching which are implemented as User-Defined Functions (UDF) in specialized array processing systems [6, 9, 11] – applies a multi-dimensional shape to each cell of the array, grouping the cell with other neighboring cells. Subsequently, an entire range of statistical functions can be applied to the resulting groups to derive domain-specific properties. To better illustrate the array-specific operations, we provide a real example from astronomy.

Galaxy group catalogs. One of the ultimate challenges in astronomy is to understand how galaxies form and evolve into the large-scale distribution of matter throughout the universe. The state-of-the-art method to study galaxy evolution is based on galaxy group catalogs [53] which are derived data products from general astronomical catalogs, e.g., SDSS [3] and iPTF [21]. Since their computation is time-consuming and they are heavily used in subsequent analysis – such as galaxy correlation [51], Gamma-ray bursts [32], and M-dwarf flares [26] – galaxy group catalogs are always materialized. However, they are built statically and rarely updated. For example, a galaxy group catalog is built only for new SDSS data releases [53]. This is extremely problematic for the iPTF and LSST projects which acquire data continuously since the galaxy group catalog becomes outdated soon after construction. Thus, the problem we tackle in this paper is how to maintain these derived catalogs incrementally under updates to the base catalog.

An astronomical catalog contains objects extracted from images of the sky taken by a telescope—a collection of detections. A detection is characterized by three dimensions – equatorial coordinates `ra`, `dec` and `time` – and tens to a few hundred attributes such as brightness and magnitude. Consequently, a catalog is typically represented as a sparse 3-D array

```
catalog<bright,mag,...>[ra,dec,time]
```

in which a cell corresponds to a detection—identified by the index on each dimension. For example, in the iPTF catalog, an array cell corresponds to 1 arcsecond on `ra` and `dec`, and 1 minute on `time`. These values are enforced by the resolution and the exposure time of the telescope. The catalog array can be stored physically at a lower granularity by partitioning the dimension ranges into chunks and grouping detections. A galaxy group catalog contains a series of statistics for every detection in the astronomical catalog. The statistics are computed from detections that are nearby in space and/or time. This computation can be decomposed into two

[1]http://www.sdss.org/dr13/
[2]http://www.ptf.caltech.edu/iptf/
[3]http://dm.lsst.org

SIGMOD'17, May 14-19, 2017, Chicago, IL, USA

DOI: http://dx.doi.org/10.1145/3035918.3064041

steps—identify nearby or similar detections and statistics evaluation. Nearby detections are computed as a similarity self-join query over the entire catalog. Statistics such as the number of nearby detections and the density are computed for each individual detection, i.e., group-by aggregation over the similarity self-join result. While a galaxy group catalog is also modeled as a 3-D array `galaxy_catalog<cnt,density,...>[ra,dec,time]` the statistics evolve over time as new detections are added to the astronomical catalog. Thus, `galaxy_catalog` corresponds to a materialized view. Although `catalog` and `galaxy_catalog` have the same dimensionality in the example, this is not a requirement in general. Moreover, the base array(s) and the materialized view are not required to have identical chunking and partitioning.

Problem statement. In this paper, we introduce the concept of *materialized array views* defined over complex shape-based similarity join aggregate queries. Since shape-based array similarity join is a generalization of array equi-join and distance-based similarity join [56], materialized array views cover an extensive class of array algebra operations [38]. According to the literature [9, 16, 20, 44], these are the most prevalent operations in scientific applications. With regard to SQL, array views include the class of join views with standard aggregates [37]. We tackle incremental array view maintenance under batch updates to the base arrays. Batch updates are the only form of updates in many real applications, e.g., astronomy, and are essential for amortizing the cost of network communication and synchronization in a distributed deployment [41]—the case for array databases.

Challenges. There are two primary challenges posed by incremental array view maintenance under batch updates. The first challenge is identifying the cells in the base array(s) that are involved in the maintenance computation and the cells that require update in the array view. This is a problem because of the complex query in the view definition which involves a shape-based similarity join—enumerating all the cells in the shape array corresponding to an update can be expensive for large shapes and sparse arrays. The second challenge is due to the distributed nature of array databases. Given the current distribution of the array(s) and the view, the challenge is to find the optimal strategy – data transfer and computation balancing – to integrate the updates into the view. Direct application of distributed relational view maintenance algorithms – defined over equi-join queries and horizontally partitioned views [37] – to arrays suffers from excessive communication and load imbalance due to the static array partitioning strategies and the skewed distribution of the updates in scientific applications.

Approach. Since the granularity of I/O and computation in array databases is the chunk – a group of adjacent cells [44] – the first challenge requires only the identification of the chunks involved in view maintenance. This can be done efficiently as a preprocessing step over the metadata. Our approach for the second challenge is to model distributed array view maintenance as an optimization formulation that computes the optimal plan to update the view based on the chunks identified in preprocessing. Moreover, the optimization continuously repartitions the array and the view based on a window of past batch updates. In the long run, repartitioning improves view maintenance time by grouping relevant portions of the array and the view and by distributing join computation across the cluster. Meanwhile, repartitioning does not incur additional time because it takes advantage of the communication required in view maintenance. Since the optimization cannot be solved efficiently, we decompose the formulation into three separate stages – differential view computation, view chunk reassignment, and array chunk reassignment – that we solve independently.

Contributions. The technical contributions we make in this paper can be summarized as follows:

- We define formally array views (Section 3). As far as we know, the concept of views has not yet been adapted to array databases.
- We model incremental array view maintenance as an optimization formulation that integrates view updating and continuous array repartitioning based on a historical workload of batch updates (Section 4.2). To the best of our knowledge, this is the first solution that considers adaptive array reorganization in the context of incremental view maintenance.
- We design a three-stage heuristic that solves the incremental array view maintenance effectively (Section 4.3-4.5). This is necessary because the original optimization and each of the three stages are NP-hard problems.
- We introduce an analytical cost model for answering similarity join queries over arrays with materialized views (Section 5). This model identifies the best alternative between a complete similarity join and a differential query on the view.
- We perform an extensive set of experiments on real datasets and batch updates (Section 6). The results confirm the effectiveness of the heuristics and the quality of the maintenance plan for an individual update. The continuous array reorganization further reduces the view maintenance time by as much as a factor of 2 over a sequence of real updates.

2. PRELIMINARIES

In this section, we introduce multi-dimensional arrays, similarity join over arrays, and relational incremental view maintenance. These concepts are the foundation for defining views over arrays and formalizing incremental array view maintenance.

2.1 Multi-Dimensional Arrays

A multi-dimensional array [49, 44, 20] is defined by a set of *dimensions* $\mathcal{D} = \{D_1, D_2, \ldots, D_d\}$ and a set of *attributes* $\mathcal{A} = \{A_1, A_2, \ldots, A_m\}$. Each dimension D_i is a finite ordered set. We assume that $D_i, i \in [1, d]$ is represented by the continuous range of integer values $[1, N]$. Each combination of dimension values, or indices, $[i_1, i_2, \ldots, i_d]$, defines a *cell*. Cells have the same scalar type, given by the set of attributes $\mathcal{A}$. Dimensions and attributes define the schema of the array. Based on these concepts, an array can be thought of as a function defined over dimensions and taking value attribute tuples:

$$Array : [D_1, D_2, \ldots, D_d] \longmapsto \langle A_1, A_2, \ldots, A_m \rangle$$

A 2-D array `A<r:int,s:int>[i=1,6,2;j=1,8,2]` with dimensions `i` and `j` and two attributes `r` and `s` of integer type is depicted in the upper part of Figure 1 (a). This is the notation to define arrays in SciDB's AQL language [38, 35]. The range of `i` is $[1, 6]$, while the range of `j` is $[1, 8]$. The numbers in each non-empty cell are the values of `r` and `s`, e.g., $A[i = 1, j = 2] \mapsto \langle r = 2, s = 5 \rangle$.

Chunking. Array databases apply chunking for storing, processing, and communicating arrays. A *chunk* groups adjacent array cells into a single access and processing unit. While many strategies have been proposed in the literature – see [44] for a survey – *regular chunking* is the most popular in practice, e.g., SciDB. Chunks have the same dimensionality as the input array, are aligned with the dimensions, and have the same shape. The cells inside a chunk are sorted one dimension at a time, where dimensions are ordered according to the array schema. Array `A` in Figure 1 is

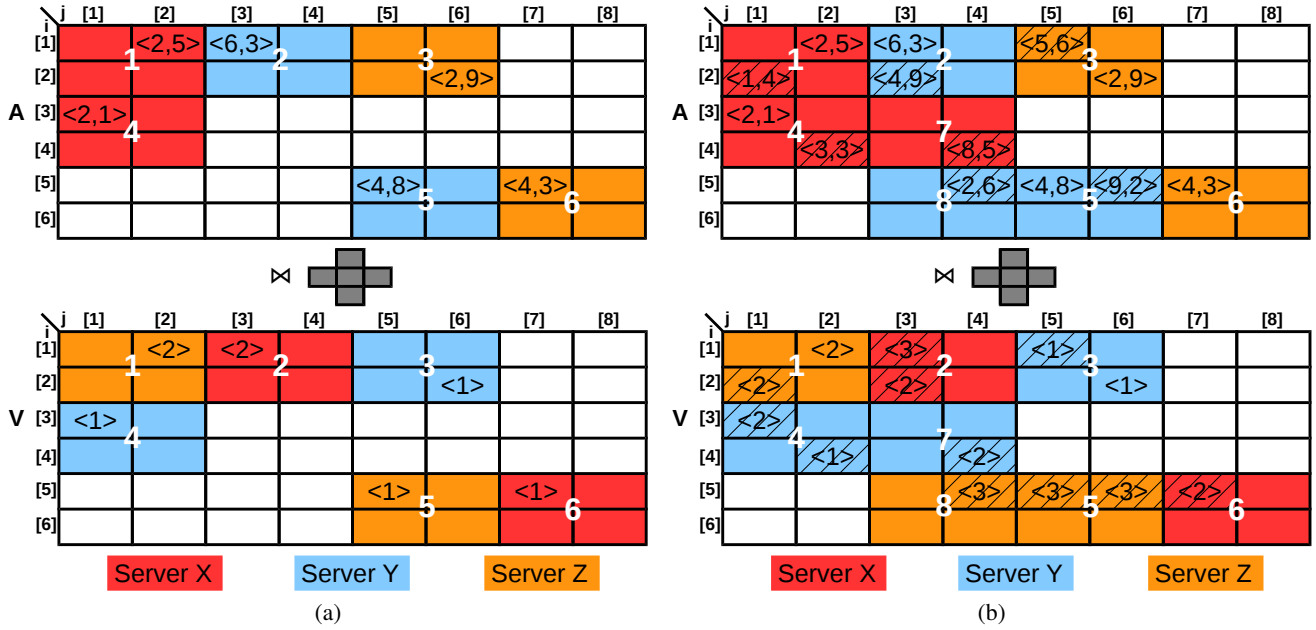


Figure 1: (a) AQL notation for array `A<r:int,s:int>[i=1,6,2;j=1,8,2]` and materialized array view `V` defined as the array similarity self-join query `CREATE ARRAY VIEW V AS SELECT COUNT(*) FROM A A1 SIMILARITY JOIN A A2 ON (A1.i = A2.i) AND (A1.j = A2.j) WITH SHAPE L1(1) GROUP BY A1.i,A1.j`. (b) View maintenance under insertions at indices [1,5], [2,1], [2,3], [4,2], [4,4], [5,4], and [5,6] into array `A`.

partitioned into 12 2×2 regular chunks grouping 2 indices on each dimension—the reason for 2 in the notation $[i = 1, 6, 2; j = 1, 8, 2]$. Only 6 of these chunks contain data—colored and numbered in Figure 1 (a).

Shared-nothing architecture. We assume a distributed array database having a *shared-nothing architecture* over a cluster of N *servers* or *nodes*, each hosting one instance of the query processing engine and having its local attached storage. The chunks of each array are stored across several servers in order to support parallel processing. All servers participate in query execution and share access to a centralized system catalog that maintains information about active servers, array schemas, and chunk distribution. A *coordinator* stores the system catalog and manages the nodes and their access to the catalog. The coordinator is the single query input point into the system. For example, in Figure 1 (a) there are 3 servers in the database. The chunks of array `A` are distributed round-robin in row-major order over the 3 servers. The color of the chunk corresponds to the color of the assigned server.

2.2 Array Similarity Join

Array similarity join is introduced in [56] as a generalization of array equi-joins [20] and the `APPLY` operator proposed in [39]. Formally, given two multi-dimensional arrays α and β

$$\alpha : \mathcal{D}^\alpha = [D_1^\alpha, \ldots, D_d^\alpha] \longmapsto \mathcal{A}^\alpha = \langle A_1^\alpha, \ldots, A_m^\alpha \rangle$$
$$\beta : \mathcal{D}^\beta = \left[D_1^\beta, \ldots, D_d^\beta\right] \longmapsto \mathcal{A}^\beta = \left\langle A_1^\beta, \ldots, A_m^\beta \right\rangle,$$

their similarity join over a mapping function $\mathcal{M} : \mathcal{D}^\alpha \longmapsto \mathcal{D}^\beta$, the shape $\sigma : \mathcal{D}^\beta \longmapsto <>$ – an array with the dimensionality of β and without any attributes – and a value function $f : \mathcal{A}^\alpha \cup \mathcal{A}^\beta \longmapsto \mathcal{A}^\tau$ is an array $\tau = \alpha \bowtie_{\sigma,f}^{\mathcal{M}} \beta$ having as dimensions the union of dimensions in α and β. Function $\mathcal{M}$ maps a cell $\Upsilon \in \alpha$ to a cell $\Psi \in \beta$, while function f is defined over the set of attributes in the two input arrays. For each cell $\Upsilon \in \alpha$, array τ contains those non-empty cells in β that are in shape σ centered on Ψ, i.e., $\sigma[\Psi]$. The dimension of the τ cell is given by the concatenation of the Υ and $\sigma[\Psi]$ dimensions, while the value is computed by function $f(\Upsilon, \sigma[\Psi])$. This can be written in AQL as:

```
SELECT f INTO τ
FROM α SIMILARITY JOIN β ON M WITH SHAPE σ
```

Array similarity join can encode any join conditions on dimensions and it can express the most common similarity distances, e.g., L^p norms, Earth Mover's Distance (EMD), and Hamming [56]. For example, the similarity self-join of array `A` in Figure 1 (a) with identity mapping $\mathcal{M}$ and value f, and shape $L^1(1)$ – the cross in the figure – is a 4-D array with 8 non-empty cells. There is a cell for each non-empty cell in `A`, e.g., $\tau[i = 1, j = 2, i' = 1, j' = 2] \mapsto \langle r = 2, s = 5, r' = 2, s' = 5 \rangle$. The two additional cells are generated by the only adjacent cells $[1, 2]$ and $[1, 3]$. They are $\tau[i = 1, j = 2, i' = 1, j' = 3] \mapsto \langle r = 2, s = 5, r' = 6, s' = 3 \rangle$ and $\tau[i = 1, j = 3, i' = 1, j' = 2] \mapsto \langle r = 6, s = 3, r' = 2, s' = 5 \rangle$.

2.3 Materialized Views

Materialized views are used by traditional database systems to answer queries faster. Data cubes [22] are a classical example of materialized views that aggregate data across all the possible combinations of a set of dimensions. Maintaining the content of a materialized view up to date in the presence of changes to the base tables is known as view maintenance. Complete recomputation is the simplest maintenance strategy, however, it is expensive for updates that reference only a small number of tuples in the view. In such cases, incremental maintenance that applies deltas only to the modified tuples is usually more efficient for large base tables.

Formally, let $(Q, M(\mathbf{D}))$ denote a materialized view [41], where Q is the definition query and $M(\mathbf{D})$ is the materialized query result

for an input dataset $\mathbf{D}$. When the input changes from $\mathbf{D}$ to $(\mathbf{D} + \Delta\mathbf{D})$, incremental view maintenance evaluates a delta query ΔQ that updates $M(\mathbf{D})$ accordingly:

$$M(\mathbf{D} + \Delta\mathbf{D}) = M(\mathbf{D}) + \Delta Q(\mathbf{D}, \Delta\mathbf{D})$$

In general, computing ΔQ and updating $M(\mathbf{D})$ accordingly is faster than re-evaluating Q from scratch. ΔQ has to be derived for each view and can have one delta query for each base table in the dataset $\mathbf{D}$. These delta queries are grouped into an associated view maintenance trigger. The frequency at which the trigger is executed – or, alternatively, the size of $\Delta\mathbf{D}$ – has a serious impact on the view maintenance time [41].

3. ARRAY VIEWS

A relational view is defined over any query [12]. Since the result of a query is a relation, the view can be used in queries as any other relation. Following the same principle, an array view is defined by a query over arrays. However, not all the queries on arrays produce an array as the result [38, 35]. This is problematic because it hinders composability—a fundamental property of relational algebra. Thus, in order to support composition, the queries on which an array view is defined have to be limited to those that output arrays. Nonetheless, the class of such queries has to be general. Array similarity join is known to be a generalization of array equi-join and distance-based similarity join [56]. Moreover, it is composable. Thus, we define an array view over multiple arrays starting from their shape-based similarity join followed by a series of unary array operators, e.g., sub-sampling, projection, group-by aggregate, etc.

DEFINITION 1. *Given n multi-dimensional arrays $\alpha_1, \ldots, \alpha_n$ and k unary array operators $\oplus_1, \ldots, \oplus_k$ that produce array output, we define an array view*
$V \leftarrow\!\shortmid \oplus_1 \left(\cdots \oplus_k \left(\alpha_1 \bowtie^{\mathcal{M}_1}_{\sigma_1, f_1} \alpha_2 \bowtie^{\mathcal{M}_2}_{\sigma_2, f_2} \ldots \bowtie^{\mathcal{M}_{n-1}}_{\sigma_{n-1}, f_{n-1}} \alpha_n \right) \cdots \right)$
as the result of applying the sequence of k operators to the chain of similarity joins among the n input arrays.

EXAMPLE 1. *Consider array view V defined by the AQL query*

```
CREATE ARRAY VIEW V AS
  SELECT COUNT(*) AS cnt
  FROM A A1 SIMILARITY JOIN A A2
      ON (A1.i = A2.i) AND (A1.j = A2.j)
      WITH SHAPE L1(1)
  GROUP BY A1.i, A1.j
```

depicted in Figure 1 (a). V consists of a single similarity self-join on A based on $L^1(1)$ similarity shape, i.e., a 5-cell cross centered on each cell, and identity mapping and value function. The resulting 4-D array is then projected on a single pair of dimensions `(i,j)` passed as arguments to the `GROUP BY` clause. The value of a cell in V is given by the number of non-empty neighbor cells of the corresponding cell in A. Thus, the single unary operator in V definition is a group-by aggregate. Following the similarity join example, there are only two cells with value 2—$V[i = 1, j = 2] \mapsto \langle cnt = 2\rangle$, $V[i = 1, j = 3] \mapsto \langle cnt = 2\rangle$.

To put this example in perspective, A corresponds to a simplified astronomical catalog without the `time` dimension, while V is the equivalent of a galaxy group catalog derived from A. Dimensionality reduction is necessary only for presentation purposes.

Materialized array views. We consider materialized array views that are evaluated eagerly at view definition and have their result stored as an array V. While the schema of V is well-defined by the query, its chunking can be either specified explicitly or it can be inferred from the chunking of the input arrays α_i [56]. Maintaining view V under modifications to the base arrays α_i becomes the primary challenge in this case.

EXAMPLE 2. *Figure 1 (b) depicts array A and materialized array view V after 7 new cells are added to A—they are hatched in the figure. The new cells result in the creation of two new chunks – 7 and 8 – in A and V, respectively, assigned to nodes according to the chunking strategy corresponding to each array. Since no chunking is specified in the view definition, the chunking of V is inherited from A. However, the assignment of V chunks to nodes is done by considering V as an independent array. The number of cells in view V that are impacted by the insertions to A – also hatched – is 11. These cells cover all the chunks in the view, thus, the entire view has to be updated. This is due to the shape in the similarity join—a cell in A can impact as many as 5 cells in V.*

Incremental array view maintenance is a complex problem that spans several axes:

- *Batch updates.* We consider batches of updates to the base array(s) because this is the standard use case in scientific applications, e.g., the iPTF pipeline ingests a series of images rather than one. Moreover, view maintenance is executed as part of a processing pipeline, not concurrently with queries. For relational views, batch updates are mostly meant to reduce the per-transaction overhead while deferring view maintenance [14, 33, 25, 30]. In the case of array views, we go one step further and share the delta computation across the batch. This is beneficial because similarity join increases the degree of sharing—unlike relational equi-join.

- *Update granularity.* Array databases access and process data at chunk granularity [9, 11]. Thus, cell updates degenerate into chunk-level operations. Metadata is also kept at chunk granularity. As a result, we group update operations, i.e., diffs [30], into chunks and perform view maintenance over chunks. This is different from relational view maintenance which operates at tuple granularity since no ordering is enforced. Chunk-level maintenance is suboptimal when the number of updated cells inside a chunk is small. However, updates are clustered by the acquisition process in many scientific applications [19]—including iPTF. Batching also helps. Performing maintenance at cell granularity has the potential to prune unnecessary joins between chunk pairs. This requires more detailed metadata, e.g., positional information on non-empty cells inside the chunk, and more time-consuming join pair identification. While this may be acceptable for relational equi-join, the cost is magnified for arbitrary shape-based array similarity join.

- *Aggregates.* We consider the standard SQL aggregate functions, e.g., `SUM`, `COUNT`, `AVG`, that can be maintained incrementally. These functions are also commutative and associative, thus the order in which updates are applied to the view is not important.

- *Distributed processing.* Distributed maintenance in data warehousing typically assumes that the base tables and the view are stored non-partitioned on a single server, albeit each of them on different servers [47, 48, 36]. When the base tables and the view are partitioned [5, 37], all the partitions require maintenance in the worst case. This can be alleviated by building indexes on each server. We consider distributed array views defined over distributed base arrays where updates are handled by a coordinator. Determining the communication among partitions involved

Variable	Description	Parameter	Description
x_{ikj}	transfer chunk i from node k to node j	U	set of historical batch updates
z_{pqk}	chunks p and q are joined at node k	W_l	weight of update batch U_l
y_{ij}	chunk i is assigned to node j	S_q	node storing chunk q at beginning of current updates
x'_{ijk}	transfer chunk i from node j to k given y	B_q	size of chunk q in bytes
z'_{pqk}	chunks p and q are joined at node k given y	T_{ntwk}	time to transfer a unit chunk between two nodes
		T_{cpu}	time to compute the join between two unit chunks
		λ	importance ratio between current and historical batches

Table 1: Binary variables and parameters in the MIP formulation.

in the maintenance becomes an important optimization parameter in this case because the number of potential join pairs for each chunk is magnified by the similarity join in the definition.

- *Recursive maintenance.* If the array view is defined over more than two arrays, updates to a single array require $n-1$ similarity joins with base arrays. This can be very expensive. A restricted form of recursive view maintenance [2, 41] can be applied to array views by materializing the result of each pair of join chains obtained by splitting the $n-1$ joins in the view definition. While these auxiliary views reduce maintenance time, they also require maintenance, i.e., recursive maintenance.

4. ARRAY VIEW MAINTENANCE

In this section, we present the first incremental array view maintenance algorithm for a distributed array database. We begin with a high-level description of the view maintenance architecture. Then, we provide an optimization formulation for the problem and derive efficient algorithms from it.

High-level approach. Given input arrays α and β and a materialized array view $V = \alpha \bowtie^{\mathcal{M}}_{\sigma,f} \beta$ defined as the similarity join between α and β, the goal is to maintain V efficiently under updates $\Delta\alpha$ and $\Delta\beta$ to the base arrays. We consider only two arrays in order to focus on the fundamental similarity join problem—the extension to multiple arrays is recursive. α, β, and V, respectively, are chunked over the database servers, while each of $\Delta\alpha$ and $\Delta\beta$ consists of a set of chunks located initially at the coordinator. The assignment of chunks to nodes for all the arrays in the database is stored in the catalog metadata—also managed by the coordinator. The updates $\Delta\alpha$ and $\Delta\beta$, respectively, come in batches at regular time intervals, i.e., cyclic batch updates. In an algebraic notation [23, 34, 12], the maintenance of V, i.e., computing $V + \Delta V$, requires computing the differential view $\Delta V = \left(\alpha \bowtie^{\mathcal{M}}_{\sigma,f} \Delta\beta\right) \cup \left(\Delta\alpha \bowtie^{\mathcal{M}}_{\sigma,f} \beta\right) \cup \left(\Delta\alpha \bowtie^{\mathcal{M}}_{\sigma,f} \Delta\beta\right)$ followed by merging into V. ΔV is the similarity join between the updates and the base arrays and the updates themselves, respectively. $V + \Delta V$ is computed according to the view definition.

4.1 Baseline View Maintenance

The baseline array view maintenance algorithm is based on the parallel relational view maintenance process proposed in [37]. Since the original process in [37] works only for single tuple updates, we extend it to batch updates. The algorithm works as follows. The new chunks $\Delta\alpha$ and $\Delta\beta$ – as well as new chunks in view V – are first assigned to nodes using the chunking strategy for each array. In order to compute the differential view ΔV, the coordinator identifies – from the catalog metadata – all the chunks in β that join with a chunk in $\Delta\alpha$ and sends the new chunk to the nodes that store these β chunks. If no β chunk joins with $\Delta\alpha$ – $\Delta\alpha$ is an irrelevant update [8] – no computation is required. The same process is applied to $\Delta\beta$ and α. Since the join $\Delta\alpha \bowtie^{\mathcal{M}}_{\sigma,f} \Delta\beta$ is included twice, we exclude $\Delta\alpha$ from α when we compute $\alpha \bowtie^{\mathcal{M}}_{\sigma,f} \Delta\beta$. The merging $V + \Delta V$ is realized by sending the chunks in the differential view to the nodes that store the corresponding view chunks. This is done asynchronously as the ΔV chunks are received. The coordinator provides the location of the view chunks and the number of ΔV chunks to all the nodes participating in the computation.

We exemplify how the baseline algorithm works based on the updates in Figure 1. The new chunks 7 and 8 are assigned to node X and Y in array A, while the corresponding view chunks are assigned to node Y and Z, respectively. We consider the differential view for chunk 7. Since chunk 7 joins with chunks 2, 4, and 8, it is also sent to node Y. Joins $7 \bowtie 2$ and $7 \bowtie 8$ are computed on Y, while $7 \bowtie 4$ on X, respectively. The view merging $V + \Delta V$ requires communication between the following pairs of nodes: $Y \longrightarrow X$ for view chunk 2, $X \longrightarrow Y$ for view chunk 4, and $Y \longrightarrow Z$ for view chunk 8, respectively.

This example illustrates one problem of the baseline algorithm—excessive communication. To update view chunk 2 on X, node X first sends chunk 7 to node Y to compute the join $7 \bowtie 2$ only to have the result returned to node X. If chunk 2 is sent from Y to X, the view can be updated locally, without additional communication. The second problem is excessive computation at a node, i.e., load imbalance. This happens for chunk 4 which is joined with all its neighbors locally at node X. The main cause for these problems is the static assignment of chunks to nodes which is completely independent from the updates. In the first case, the chunks are too spread over the nodes in the cluster, while in the second case, the chunks are clustered at a single node. These problems are magnified in scientific data processing by the chunking strategies [19] and the fact that updates are concentrated on a limited area of the base arrays, e.g., the iPTF telescope points to a relatively small area of the sky during each night. For hash-based chunking, each join computation is likely to require communication because adjacent chunks are assigned to different nodes. For space-partitioning strategies – space-filling curves, quadtree, k-d tree – most of the joins are concentrated on a single node, thus the load is imbalanced.

4.2 Optimal View Maintenance

We address the limitations of the baseline algorithm – excessive communication and load imbalance – by modeling the array view maintenance problem as a mixed-integer program (MIP) that identifies the optimal plan to compute the differential view ΔV and the merging $V + \Delta V$ and determines the optimal reassignment to nodes for both the array and the view chunks—new and existing. The optimal view update plan reduces the excessive communication/computation, while the reassignment guarantees that we do not get stuck with an unfavorable static chunking strategy. Even more, the incoming chunks are not first assigned to a node based on a pre-determined chunking strategy. The reason for considering these two seemingly disjoint objectives together is to piggyback on the chunk replication incurred by view maintenance when

$$\min\left\{\lambda\cdot\max\left\{\max_k\left\{\sum_{i,j}^{k\neq j}x_{ikj}\cdot B_iT_{ntwk}+\sum_{p,q,v,j}^{k\neq j,(p,q,v)\in U_0}z_{pqk}\cdot y_{vj}\cdot B_{pq}T_{ntwk}\right\},\max_k\left\{\sum_{p,q}z_{pqk}\cdot B_{pq}T_{cpu}\right\}\right\}\right.$$
$$\left.+(1-\lambda)\cdot\max\left\{\max_k\left\{\sum_{i,j}^{k\neq j}x'_{ikj}\cdot B_iT_{ntwk}+\sum_{p,q,v,j,l}^{k\neq j,(p,q,v)\in U_l}z'_{pqk}\cdot y_{vj}\cdot W_lB_{pq}T_{ntwk}\right\},\max_k\left\{\sum_{p,q,l}^{(p,q)\in U_l}z'_{pqk}\cdot W_lB_{pq}T_{cpu}\right\}\right\}\right\} \quad (1)$$

$$C_1:\sum_{j=1}^{N}y_{ij}=1\ \forall i\qquad C_2:2z_{pqk}\leq x_{pS_pk}+x_{qS_qk}\ \forall(p,q,k),(p,q)\in U_0\qquad C_3:\sum_{k=1}^{N}z_{pqk}=1\ \forall(p,q)\in U_0$$
$$C_4:2z'_{pqk}\leq x'_{pS_pk}+\sum_{j=1}^{N}x'_{qjk}\cdot y_{qj}\ \forall(p,q,k),(p,q)\in U\qquad C_5:\sum_{k=1}^{N}z'_{pqk}=1\ \forall(p,q)\in U \quad (2)$$

computing the reassignment. Otherwise, the benefit of the reassignment may be overlooked by the required communication and chunk replication. Moreover, the reassignment takes into account a window of past batch updates, rather than only the current batch. This is necessary in order to avoid frequent unstable reassignments. The main difference to the baseline algorithm is that we consider view maintenance and reassignment together, rather than first assigning chunks to nodes based on a pre-determined chunking and then solving the maintenance under that assignment.

MIP variables & parameters. The MIP variables and parameters are displayed in Table 1 included in the Appendix. There are two families of variables. x encodes the chunk communication between two nodes, while z determines the node where the join between two chunks is evaluated. Essentially, x corresponds to communication and z corresponds to computation. The y variables encode the chunk to node reassignment after the current batch update. While the chunk to node assignment at the beginning of the update is fixed, y corresponds to the optimal reassignment that is derived from the past batches of updates U. x' and z' have the same meaning as x and z, however, they are based on the reassignment y, not the constant input assignment s_q at the current batch update time. Moreover, x and z correspond to the communication/computation for the current batch, while x' and z' are defined over the window of historical batch updates U.

The constant parameters in the MIP formulation – also shown in Table 1 – include the historical batch of updates U and their weight W, the chunk to node assignment S_q when the current update batch is processed, the size of the chunk B_q, the time to transfer chunks between nodes T_{ntwk} and to join two chunks T_{cpu}, and the weighted sum ratio λ. A batch of updates U_l is a set of triples (p,q,v) where chunks p and q from base arrays $(\alpha\cup\Delta\alpha)$ and $(\beta\cup\Delta\beta)$ have to be joined and the result has to be merged into chunk v in view V. For each update batch, U_l is computed by the coordinator from the catalog metadata. A fixed-size window of past batch updates is collected, where U_0 corresponds to the current batch and U_l is for the l^{th} previous batch. A weight W_l is assigned to each batch U_l. While the values of W_l can follow any distribution, we use exponential decay in our implementation. The older a batch is, the smaller its corresponding weight W. S_q is the current node assignment of chunk q. It is always fixed for existing chunks and set to the coordinator for the new chunks. B_q is the size of chunk q and B_{pq} is the total size of chunks p and q. The values of T_{ntwk} and T_{cpu} are determined based on an empirical calibration process. λ is an importance weight between 0 and 1 that discriminates between the current and past batches of updates.

We illustrate how the variables and the parameters are instantiated based on the updates in Figure 1. The original assignments of chunks 4 and 7 when the batch is considered are $S_{A_4}=X$, $S_{V_4}=Y$, and $S_{A_7}=s_{V_7}=$ *Coordinator*, respectively. The triples in U_0 for chunk 4 are $\{(\Delta A_4,A_1,V_1)$, $(\Delta A_4,\Delta A_1,V_1)$, $(\Delta A_4,A_4,V_4)$, $(\Delta A_4,\Delta A_7,V_7)\}$.

Analytical cost model. The MIP formalization of the optimal view maintenance problem is given in Eq. (1) – the objective function – and Eq. (2)—the constraints. The cost model is the weighted λ summation of two terms corresponding to the view maintenance for the current update batch and the chunk reassignment based on the historical updates. The difference between the terms is that view maintenance considers only the triples in U_0, while the reassignment considers all the triples across the historical batches in U. Each of the terms is the maximum between the communication and the computation executed across all the nodes in the cluster. We consider the maximum between these two because we overlap communication and computation in our implementation setting. The communication involves two terms as well. The first term is for co-locating on the same node chunks that have to be joined for computing the differential view ΔV, while the second term is for merging the result into the view chunk $V+\Delta V$. By minimizing the maximum across nodes, load-balancing is achieved since none of the nodes is allowed to perform excessive communication and/or computation.

The cost model introduced in this paper differs significantly from the cost model for array similarity join in [56]. The cost for array similarity join considers only the current join and no historical queries. More importantly, we include the CPU cost and exclude the disk I/O cost—the reverse holds in the case of array similarity join. The reason for this is the much smaller number of referenced chunks in a batch of updates. While in array similarity join all the chunks have to be considered, only the chunk triples in U_0 are required for view maintenance. In practice, we observe that these chunks fit in memory and no further disk access is required beyond the initial loading. As a result, the CPU time becomes the dominant factor in computation at a node.

The most relevant constraints are given in Eq. (2). C_1 enforces that a chunk is stored at a single node and new chunks are assigned to a node. This is the standard in array databases which do not consider replication. C_2 forces chunk co-locating at node k where the join between chunks p and q is performed. C_3 and C_5 guarantee that all the join triples are computed. C_4 is the equivalent of C_2 for the historical updates. However, the chunk assignment is no longer constant. It has to be inferred from variable y.

Challenges. Solving directly the MIP formulation with an integer programming solver poses severe challenges for several reasons. First, both the objective and the constraints contain quadratic terms that correlate the chunk assignment with the join computa-

tion, e.g., $z_{pqk}y_{vj}$. While quadratic solvers exist, they are not as advanced as linear solvers such as CPLEX[4]. Linearizing the quadratic terms by introducing new variables and constraints is the standard solution to solve this type of optimizations. However, in our case, this results in the creation of variables and constraints with 5 indices, e.g., $pqkvj$. This number is huge even for a small number of triples in U. Second, the $\max$ functions in the objective increase the difficulty of convergence for barrier search branch & bound methods. While $\max$ can be also linearized, this adds even more constraints. In order to verify the practicality of directly solving the MIP formulation, we experimented with the linear reduction for a single batch of 1000 updates that generate less than 4000 triples—average batch update in iPTF. CPLEX cannot find any solution beyond a feasible starting point in an hour on a massive 56-thread server.

Solution. We decompose the complete view maintenance formulation into three separate stages that solve the optimization for a subset of the variables. The variables set at a stage are used as input for the subsequent stages. The three stages are *differential view computation*, *view chunk reassignment*, and *array chunk reassignment*. The differential view computation determines the nodes where each pair of chunks (p, q) is joined and the chunk communication plan for each node. This corresponds to solving the first line in the objective Eq. (1) for variables z and x with the constraints C_2 and C_3. The chunk assignment y is fixed as S at this stage. In view chunk reassignment, the node where to store each view chunk is determined based on the join computation plan. We solve the same objective as in the first stage with constraint C_1 for variables y_{vj} corresponding to the view chunks by taking the values of z and x as input. Given the join computation plan and the view chunk reassignment, in array chunk reassignment, the base array chunks are relocated to nodes such that the view maintenance cost across the historical batch updates is minimized. This corresponds to solving the complete optimization formulation for variables y_{aj}, x', and z' with constraints C_1, C_4, and C_5. Notice that only the y variables for base array chunks are considered. We discuss the three stages of the proposed solution in the following sections.

Algorithm 1 Differential View Computation

Input: triples $U_0 = \{(p, q, v)\}$; chunk location S and size B
Output: x_{ikj}; z_{pqk}
1. *ntwk*$[1..N]$, *cpu*$[1..N] \leftarrow 0$; $T[q] \leftarrow \{S_q\}, \forall q$
2. **for each** $(p, q, *) \in U_0$ in random order **do**
3. *opt* $\leftarrow \infty$; *dest* $\leftarrow \emptyset$
4. **for** $j \leftarrow 1$ **to** N **do**
5. *ntwk'*$[1..N]$, *cpu'*$[1..N] \leftarrow 0$
6. **if** $j \notin T[q]$ **then** *ntwk'*$[S_q] \leftarrow$ *ntwk'*$[S_q] + B_q T_{ntwk}$
7. *cpu'*$[j] \leftarrow$ *cpu'*$[j] + B_{pq} T_{cpu}$
8. *opt_now* $\leftarrow \max_k\{$*ntwk*$[k] +$ *ntwk'*$[k],$ *cpu*$[k] +$ *cpu'*$[k]\}$
9. **if** *opt* $>$ *opt_now* **then** *opt* $\leftarrow$ *opt_now*; *dest* $\leftarrow j$
10. **end for**
11. $T[p] \leftarrow T[p] \cup \{$*dest*$\}$; $T[q] \leftarrow T[q] \cup \{$*dest*$\}$
12. update *ntwk* and *cpu*
13. $x_{p,S_p,dest} \leftarrow 1$; $z_{p,q,dest} \leftarrow 1$
14. **end for**

4.3 Differential View Computation

The first stage aims to find the optimal join plan for computing the differential view $\Delta V = \left(\alpha \bowtie^{\mathcal{M}}_{\sigma,f} \Delta\beta\right) \cup \left(\Delta\alpha \bowtie^{\mathcal{M}}_{\sigma,f} \beta\right) \cup \left(\Delta\alpha \bowtie^{\mathcal{M}}_{\sigma,f} \Delta\beta\right)$ given the current chunk assignment of the base arrays. Δ chunks are initially stored at the coordinator. However, the coordinator does not participate in the join computation. Although ΔV is the union of three array similarity join queries, it can be evaluated as a single array similarity join query across the union of the chunk join pairs in these three queries. While a solution to array similarity join is proposed in [56], it is not feasible in this context because it restricts the evaluation of a join between two chunks to the node storing the chunk in the base array—Δ chunks cannot be joined at the coordinator. Our approach is to consider all the nodes as candidates for performing the join for all the chunk pairs. However, this is an NP-hard problem—the reduction is given in Appendix A.1. We propose an efficient randomized local search heuristic for computing the differential view join plan depicted in Algorithm 1. At high-level, the algorithm iterates randomly over the chunk join pairs (p, q) in the input triples U_0 corresponding to the current batch update and chooses the node to perform the join such that the objective function in the MIP formulation is minimized. The input of the algorithm is represented by the triples U_0 and the current array chunk assignment S. Line (1) initializes the components of the objective function and the nodes where each chunk q is stored—initially set to S_q. The remaining part is the main loop of the algorithm which iterates over the chunk join pairs, while the inner loop at lines (4)-(10) iterates over the nodes. The cost of evaluating the join is computed for each node j and the node that provides the minimum cost for the $\max$ objective function is selected in line (9). Line (11) updates the chunk replication location, while line (13) sets the output variables x and z. $x_{p,S_p,dest}$ set to 1 means that chunk p is sent from its original location to the selected node for performing the join. $z_{p,q,dest}$ set to 1 signals that chunk pair (p, q) is joined on the selected node. Line (12) updates the objective function based on the selected node. We show how the algorithm works for the array in Figure 1 in Appendix B.1.

The algorithm gives priority to nodes that already store the involved chunks because they do not require additional communication. However, if these nodes are doing more work than other nodes not having the chunk, sending the chunk somewhere else becomes the preferred choice. This guarantees that none of the nodes are the bottleneck because of the chunk assignment strategy—a fundamental limitation of the baseline algorithm. The complexity of the algorithm is $\mathcal{O}(|U_0|N \log N)$ since the $\max_k$ on line (8) can be computed in logarithmic time with a binary heap. In the case of a large cluster with thousands of nodes N, solutions to accelerate this algorithm include the parallel processing of the inner loop over the nodes and node partitioning strategies. We plan to investigate these directions in future work.

4.4 View Chunk Reassignment

The second stage identifies the optimal node to compute the merging $V + \Delta V$ for all the chunks in the view. The location where the differential view ΔV is evaluated in the first stage is taken as a pre-condition instead of blindly sending ΔV to the node that contains the corresponding view chunk—the case in the baseline algorithm. This corresponds to determining the variables y_{vj} given values for x and z and entails the reassignment of the view chunks. We show that this problem is NP-hard in Appendix A.2. We design an efficient randomized heuristic that follows the same ideas from Algorithm 1. This heuristic – depicted in Algorithm 2 – iterates randomly over all the view chunks v that appear among the triples U_0 and selects the optimal node where to reassign v from all the nodes in the cluster. The cost in the MIP formulation is used to discriminate between nodes. The iteration over the triples U_0 is split into two sections because all the updates to a view chunk v

[4]http://www-01.ibm.com/software/commerce/optimization/cplexoptimizer/

have to be considered together when computing the cost. Beyond this slight difference, the details of the view chunk reassignment algorithm follow from Algorithm 1 and we do not repeat them here. We provide an example showing how the algorithm works in Appendix B.2.

Algorithm 2 View Chunk Reassignment

Input: triples $U_0 = \{(p, q, v)\}$; chunk location S and size B; x_{ikj}; z_{pqk}
Output: y_{vj}
1. initialize *ntwk*$[1..N]$, *cpu*$[1..N]$ from x_{ikj}, z_{pqk}
2. **for each** $(*, *, v) \in U_0$ in random order **do**
3. *opt* $\leftarrow \infty$; *dest* $\leftarrow \emptyset$
4. **for** $j' \leftarrow 1$ **to** N **do**
5. *ntwk'*$[1..N]$, *cpu'*$[1..N] \leftarrow 0$
6. **for each** $(p, q, v) \in U_0$ **do**
7. $j \leftarrow \arg\max_{j \in \{1,..,N\}} z_{pqj}$
8. **if** $j \neq j'$ **then** *ntwk'*$[j] \leftarrow$ *ntwk'*$[j] + B_{pq} T_{ntwk}$
9. *cpu'*$[j'] \leftarrow$ *cpu'*$[j'] + B_{pq} T_{cpu}$
10. **end for**
11. *opt_now* $\leftarrow \max_k\{$*ntwk*$[k] +$ *ntwk'*$[k]$, *cpu*$[k] +$ *cpu'*$[k]\}$
12. **if** *opt* $>$ *opt_now* **then** *opt* $\leftarrow$ *opt_now*; *dest* $\leftarrow j'$
13. **end for**
14. update *ntwk* and *cpu*
15. $y_{v,dest} \leftarrow 1$
16. **end for**

If we consider a view chunk v in isolation, the reassignment is biased towards the nodes that compute more differential view chunks relevant to v. However, since there is interaction between view chunks, the reassignment avoids the nodes that are communication or computation bottlenecks. The complexity of the algorithm follows from that of Algorithm 1.

4.5 Array Chunk Reassignment

The idea in array chunk reassignment is to reuse the replication required to evaluate the differential view in order to reorganize the array chunks. Replication is induced through variables x_{ikj} which send a chunk i from its origin k to several nodes j. Since chunk i is replicated across all the nodes j, there is no communication overhead in performing the array chunk reassignment. Only the storage across nodes is redistributed. The goal of the reassignment is to reduce the communication cost of merging the differential view $V + \Delta V$ for future batches of updates. This is accomplished by co-locating the array chunks in the differential view with the corresponding view chunk. Since the location of the differential view computation z_{pqk} and the assignment of the view chunks y_{vj} are known at this stage, only the variables y_{aj} for the array chunks have to be determined. One solution to compute y_{aj} considers only the current update batch. However, this has the potential to generate highly-unstable reassignments that are overreacting to changes in the update workload.

Our solution is to consider a window of past update batches when computing y_{aj}. Thus, rather than considering only the triples U_0, we also include in the optimization the triples U_l for the past updates. For U_l to provide any useful information, though, we have to know their associated variables x and z for the reassignment configuration—these are variables x' and z'. While computing these variables looks similar to computing x and z in the first stage, it is actually more difficult because the assignment of the array chunks S_a is unknown—these are the variables y_{aj} we have to compute in this stage. Notice, though, that the values of variables x' and z' do not have to be explicitly determined because we do not have to reevaluate the past update batches. What matters is the benefit a given assignment brings to past updates. We quantify this benefit with a frequency-based score associated to every chunk pair (q, v) – q is an array chunk, v is a view chunk – that appears across the triples U_l. The more frequent a pair (q, v) is, the higher its score can be. The history is taken into account by biasing towards recent updates based on the weight W_l associated with the batch. The score takes into consideration only the communication cost in the objective function. It ignores the computation cost. This can lead to skewed reassignments in which certain nodes have to perform almost all the computation. We handle these cases by limiting the number of pairs that are assigned to each node. With score reformulation and CPU limitation, array chunk reassignment is NP-hard (Appendix A.3).

Algorithm 3 Array Chunk Reassignment

Input: sets of triples $U_l = \{(p, q, v)\}$ and associated weights W_l; chunk location S and size B; x_{ikj}; z_{pqk}; y_{vj}
Output: y_{aj}
1. initialize *cpu_thr*$[1..N]$; *done* $\leftarrow \emptyset$
2. *score*$[a, v] \leftarrow 0, \forall(a, *, v), (*, a, v) \in U_l, \forall l$
3. **for each** $(a, *, v), (*, a, v) \in U_l$ **do**
4. *score*$[a, v] \leftarrow$ *score*$[a, v] + W_l B_a$
5. **for each** (a, v) in descending order of *score*$[a, v]$ **do**
6. **if** $a \notin$ *done* **then**
7. $j \leftarrow \arg\max_{k \in \{1,..,N\}} y_{vk}$
8. **if** $(x_{aS_aj} = 1)$ **and** (*cpu_thr*$[j] \geq B_a$) **then**
9. *cpu_thr*$[j] \leftarrow$ *cpu_thr*$[j] - B_a$
10. $y_{aj} \leftarrow 1$; *done* $\leftarrow$ *done* $\cup \{a\}$
11. **end if**
12. **end if**
13. **end for**
14. $y_{aS_a} \leftarrow 1, \forall a \notin$ *done*

We design a greedy algorithm for chunk reassignment (Algorithm 3). The algorithm iterates over the chunk pairs (a, v) in decreasing order of their score and assigns array chunk a to the node where v is assigned, as long as the computation threshold *cpu_thr* is not exceeded—lines (5)-(13). This guarantees that a is grouped together with the view chunk it is most correlated with. The chunks that cannot be assigned – due to the CPU limitation at a node – stay at their previous location S_a (line (14)). *cpu_thr* is initialized as the average join cost per node across all the triples in U_l. It is important to notice that the assignment of Δ chunks is also handled by this algorithm. However, if a Δ chunk cannot be assigned to any node due to tight CPU limitations, we assign it to the node containing the v chunk with the highest score. The example in Appendix B.3 illustrates how the algorithm works.

5. QUERY INTEGRATION

In this section, we present a succinct discussion on how to integrate array views in shape-based similarity join queries. This is orthogonal to using views in relational databases [24, 18]. We focus on the difficult case when the shape of the query is different from the shape used in the view definition. Differences in the aggregate function are treated similar to relational views [13]. Given a query and a view, our goal is to optimally answer the query using the view. We show that this can be derived from the MIP formulation for view maintenance (Section 4.2). However, it may not be more efficient than evaluating the query from scratch. Thus, we devise a cost model that allows us to choose the best alternative.

$$\min\left\{\max\left\{\max_k\left\{\sum_{i,j}^{k\neq j} x_{ikj}\cdot B_i T_{ntwk} + \sum_{p,q,v,j}^{k\neq j,(p,q,v)\in U_0} z_{pqk}\cdot y_{vj}\cdot B_{pq}T_{ntwk}\right\}, \max_k\left\{\sum_{p,q} z_{pqk}\cdot B_{pq}T_{cpu}\right\}\right\}\right\} \text{ (with view)}$$
$$\min\left\{\max\left\{\max_k\left\{\sum_{i,j}^{k\neq j} x_{ikj}\cdot B_i T_{ntwk}\right\}, \max_k\left\{\sum_{p,q} z_{pqk}\cdot B_{pq}T_{cpu}\right\}\right\}\right\} \text{ (with complete similarity join)} \quad (3)$$

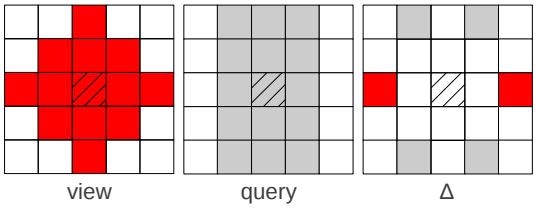


Figure 2: View, query, and corresponding Δ shapes.

Δ shape. Consider the shape arrays in Figure 2. An array view is defined over shape *view*. The query to evaluate is an array similarity join over the shape *query*. We define Δ shape as the positional symmetric set difference between *view* and *query*, i.e., $\Delta = (view \setminus query) \cup (query \setminus view)$. In Figure 2, the Δ shape contains the 6 cells—2 for $(view \setminus query)$ and 4 for $(query \setminus view)$.

Differential query evaluation. In order to evaluate the query using the view, the result of the similarity join with the Δ shape has to be merged with the view. This process can be mapped into the view maintenance problem by generating the update triples in U_0 for all the chunks in the base array with shape Δ. However, the merging with the view creates the result array rather than updating the view. This reduction allows us to apply the MIP optimization framework and the derived algorithms for differential view computation in Section 4.3. The alternative to answer the query is to compute the similarity join with shape *query* over the base arrays.

Analytical cost model. We present an analytical cost model that allows us to identify the better solution for a given view and query. In both cases, the model is a subset of the MIP formulation for view maintenance in Eq. (1). The cost for each alternative is depicted in Eq. (3). The only difference is the additional term corresponding to the interaction with the view. The common parts correspond to the shape-based similarity join. From the two costs, it appears that it is always better to compute the join from scratch since the cost of the view solution contains an additional term. We remind the reader that the input to these costs is different. The triples in the view cost are generated from the Δ shape, while the chunk join pairs in similarity join are extracted from the *query* shape. The main factor that determines the relationship between the costs is the relative ratio between the size of Δ and *query*. Intuitively, if the ratio is larger than 1, the full similarity join is more efficient. By solving the two optimization formulations and finding their minimum cost, we can decide which alternative to pursue. Nonetheless, the cost model may not reflect the reality accurately—as with any query optimizer.

6. EXPERIMENTAL EVALUATION

The objective of the experimental evaluation is to investigate the performance of the proposed heuristics on incrementally maintaining the PTF "association table" under the nightly batch updates to the base catalog. The "association table" is a derived data product that clusters raw candidates that are within a specified distance of each other over a given time horizon, i.e., FoF clustering. We use the real data and batch updates from the PTF pipeline. The LinkedGeoData dataset is used to confirm the generality of the incremental array view maintenance framework. Specifically, the experiments are targeted to answer the following questions:

- Does the proposed differential view computation improve upon the baseline algorithm for a single update batch?
- Does the chunk and view reassignment improve the view maintenance time across a series of update batches? How sensitive is the reassignment to correlations between batches?
- What is the execution time of the incremental array view maintenance heuristic (Appendix C)? How is the time split between differential view maintenance and chunk reassignment?
- When is querying with an array view better than complete similarity join computation?
- How sensitive is the proposed method to batch size and chunk spread in the updates? (Appendix C)

6.1 Setup

Implementation. We implement incremental view maintenance as a layer on top of the array similarity join operator proposed in [56]. Similarity join is implemented as a `C++11` distributed multi-thread prototype that uses an enhanced storage manager derived from ArrayStore [49]. The catalog is stored at the coordinator and replicated to all the nodes in the cluster at runtime. The incremental view maintenance heuristic is executed at the coordinator and the resulting plans containing information on chunk transfer and reassignment, and chunk join pair evaluation are distributed to the nodes. The similarity join operator runs as a server on each node in the cluster. It manages a pool of worker threads equal to the number of CPU cores in each node. A worker thread is invoked with a pair of chunks that have to be joined and the node where to send the result for view merging. Requests are made to the local and remote array storage managers to retrieve the chunks to join. This happens concurrently across all the workers. View merging is also executed by worker threads from the pool. Whenever a join result is received, a worker is assigned to merge it to the view. The code contains special functions to harness detailed profiling data.

System. We execute the experiments on a 9-node cluster. The coordinator runs on one node while the other 8 nodes are workers. Each node has 2 AMD Opteron 6128 series 8-core processors (64 bit) – 16 cores – 28 GB of memory, and 4 TB of HDD storage. The number of worker threads is set to 16—the number of cores. Ubuntu 14.04.5 SMP 64-bit with Linux kernel 3.13.0-43 is the operating system. The nodes are mounted inside the same rack and are inter-connected through a Gigabit Ethernet switch. The measured network bandwidth on a link is 125 MB/second. Since the disk bandwidth is in the same range, there is not a significant difference between the network and disk I/O.

Methodology. We include in the evaluation three methods—baseline, differential, and reassign. Baseline (Section 4.1) is the parallel relational view maintenance procedure adapted to array

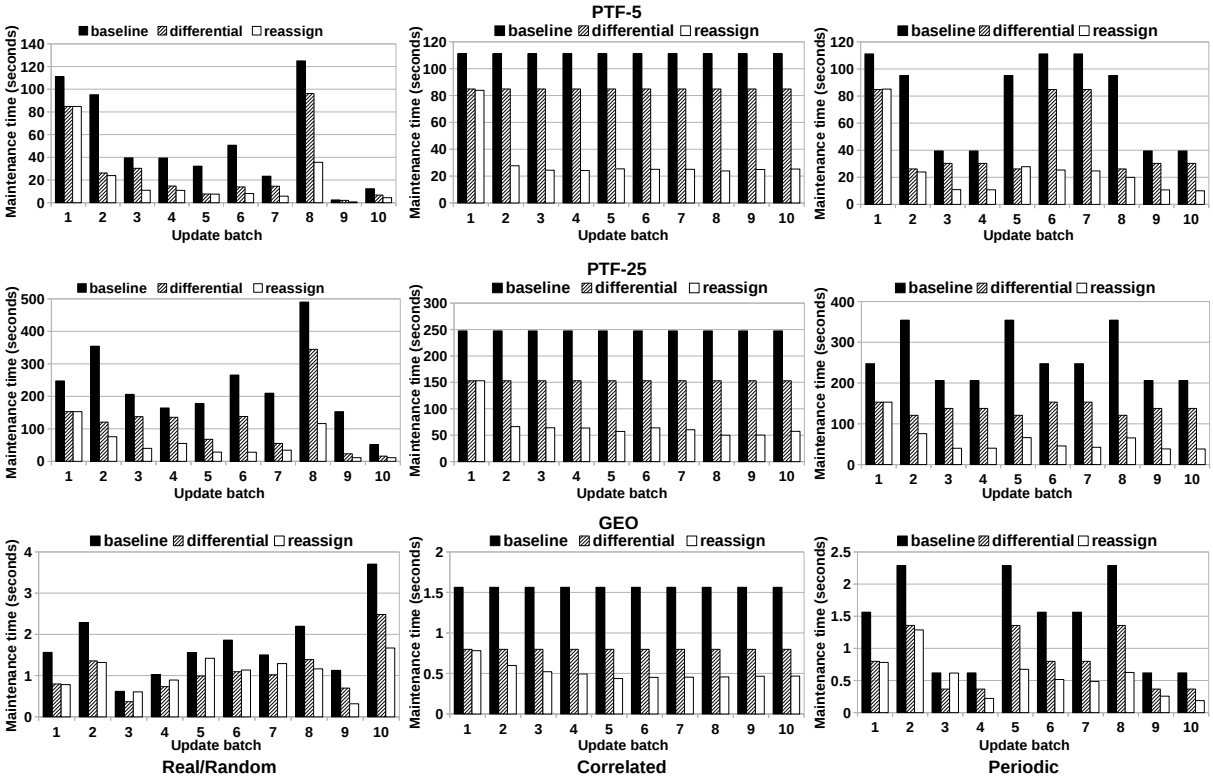


Figure 3: View maintenance time on all the datasets and all the update batch configurations.

data. Differential corresponds to the first stage of the proposed heuristic in which only the join plan is optimized (Section 4.3). Reassign is the complete heuristic that adds view and array chunk reassignment to differential (Section 4.5). By separating the heuristic into differential and reassign, we can study the impact of reassignment separately. We measure wall-clock time.

Data. We use the same two real datasets as in [56] for experiments. The *PTF catalog* consists of 1 billion time-stamped objects represented in the equatorial coordinate system (ra, dec). The range of the time coordinate spans over 153,064 distinct values, while for *ra* and *dec* we use ranges of 100,000 and 50,000, respectively. In array format, this corresponds to:

`PTF[time=1,153064;ra=1,100000;dec=1,50000]`

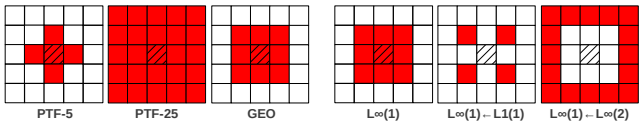
which is a sparse array with density less than 10^{-6}. Objects are not uniformly distributed over this array. They are heavily skewed around the physical location of the acquiring telescope—latitude corresponds to *dec*. After experimenting with several chunk sizes, we found that $(112, 100, 50)$ provides the best results. The size of the PTF catalog is 343 GB.

LinkedGeoData[5] stores geo-spatial data used in OpenStreetMap. We use the "Place" dataset which contains location information on roughly 3 million 2-D (long, lat) points-of-interest (POI). Since this is a too small dataset, we synthetically generate a larger dataset by adding 9 synthetic points with coordinates derived from each original point using a Gaussian distribution with $\mu = 0$ and $\sigma = 10$ miles [46]. In array format, this corresponds to:

`GEO[long=1,100000;lat=1,50000]`

[5]http://linkedgeodata.org

having chunk size of $(100, 50)$. Even with this replication, the size of GEO is still less than 1 GB.



(a) (b)

Figure 4: (a) Query shapes. (b) Δ shapes.

Views. We create three materialized array views – two on the PTF dataset and one on LinkedGeoData – that count the number of similar neighbors for each cell in the base arrays. The shapes in the similarity join from the view definition are depicted in Figure 4a. PTF-5 defines similarity as the L^1-norm of size 1 on the (ra, dec) dimensions across the previous 200 days. This corresponds to clustering objects within 10 arcseconds of each other. In PTF-25, similarity is defined as the L^∞-norm of size 2 on (ra, dec) which corresponds to 400 arcminutes. All the objects that appear in the catalog within this distance are considered similar—independent of the time. The size of the PTF-5 and PTF-25 views is identical – 32 GB – since they have the same schema. PTF-5 is a real "association table" used in the production PTF pipeline to follow-up interesting transient candidates. Given the massive similarity shape, PTF-25 is used to test the scalability of the proposed solution. The GEO view clusters POIs that are within 1 mile of each other. The correspond-

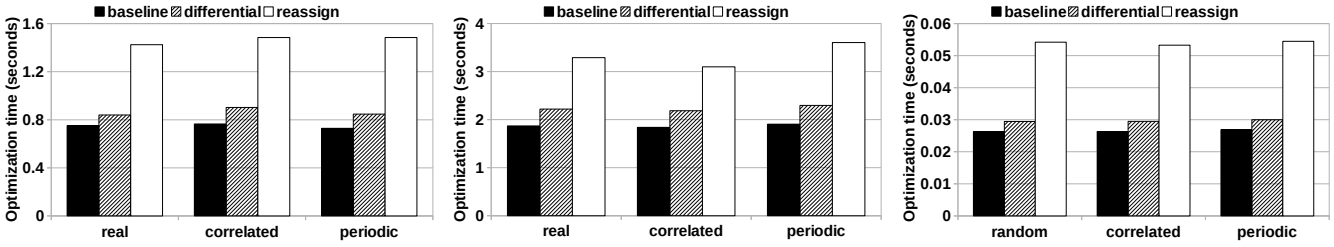


Figure 5: Average optimization time per update batch: (a) PTF-5, (b) PTF-25, and (c) GEO.

ing shape array is the L^∞-norm equal to 1. Since the GEO dataset is small, the corresponding materialized view is only 720 MB.

Batch updates. We extract 10 update batches from the original datasets using four methods—real, random, correlated, and periodic. Real takes the latest time-stamped batches from PTF. For random, the batches are randomly sampled out of the entire dataset. We do this only for GEO data which is not time-stamped and synthetically generated. Correlated batches are generated by repeating one of the real/random batches 10 times. Periodic batches are created by repeating the first 3 real/random batches while alternating their order, e.g., $1, 2, 3, 3, 2, 1, 1, 2, 3, 3$. This order preserves correlation only for some of the batches and allows us to better evaluate the impact of chunk reassignment. We extract batches from the PTF based on the time of the observation. This is exactly the procedure updates are applied in the production pipeline. The number of chunks we select in a batch varies between 600 and 2000 which corresponds to a two-week period of updates. A typical nightly update has less than 100 chunks—too small for a meaningful evaluation. Since GEO does not include a time dimension, extracting meaningful batches is more complicated. We select 1% of the entire dataset in a batch.

6.2 View Maintenance Per Update Batch

The results for view maintenance time are depicted in Figure 3 for each individual batch—they do not include the optimization time. Reassignment considers windows of 5 previous queries having weights with exponential decay and impacts subsequent queries.

PTF-5. The maintenance time for real updates exhibits large variations among batches. This is mostly due to the difference in batch size—in some nights the PTF telescope takes more images than in others. The proposed heuristics always outperform the baseline algorithm. The difference varies across batches and is larger for large update batches—by as much as a factor of 4 for batch 2, 6, and 8. Reassign improves upon differential in all batches except 1—when it cannot even be applied. In the case of correlated batches, as expected, baseline and differential have the same maintenance time across all the batches. While reassign starts at the same level with differential, it continuously improves until it reaches the best partitioning for the given batch of updates. This happens at batch 4 where the gaps to the baseline and differential are $5X$ and around $4X$, respectively. For periodic batches, the maintenance time for the same batch in the sequence – (1,6,7), (2,5,8), and (3,4,9,10) – is similar for baseline and differential. The behavior of reassign is interesting when the same batch appears consecutively, e.g., (3,4), (6,7), and (9,10). In all these cases, the second batch is processed slightly faster than the first, thus, reassignment has an effect, however, it is much smaller than for correlated batches.

PTF-25. The maintenance time for view PTF-25 exhibits higher variance – especially for baseline – even though we use the same update batches as for PTF-5. This is due to the much larger join shape in the view definition. The difference between reassign and baseline is larger – $6X$ for batch 6 – because the number of update triples is larger and this increases the optimization space. In the case of batch 8 which is very different from the previous ones, baseline and differential take a significant hit. Reassign benefits from view and array chunk colocation and processes the join pairs faster.

GEO. While similar trends to PTF are observed for GEO, the interesting fact is that there are quite a few batches in the random setting where differential outperforms reassign. This is normal when future updates are unpredictable and there is no relationship between batches. However, this effect is also magnified by the small update times specific to the GEO dataset. The impact of reassign is clear, though, in the periodic workload for batches 3 and 4 which contain the same set of chunks. While for batch 3 reassign is slower than differential, at batch 4 the situation is reversed. Reassign considers previous updates and generates a more efficient partitioning. As a result, the maintenance time is halved.

6.3 Optimization Time

The time to compute the view maintenance plan – and repartitioning for reassign – is depicted in Figure 5. We present the average optimization time across the 10 update batches. The measurement for baseline corresponds to generating the triples (p, q, v) in which chunk p is joined with chunk q and the result is merged into view chunk v. These triples have to be computed for all the methods. Differential adds the execution time of Algorithm 1, while reassign adds the times for Algorithm 2 and Algorithm 3 on top of that. There is a clear trend across all the datasets. Differential incurs a minimal overhead over baseline, while reassign takes at most double the time of baseline. In absolute terms, the optimization time per batch is at most 3.5 seconds which, we believe, is an acceptable value considering the significant reduction in maintenance time it brings—as much as 200 seconds or more (for PTF-25).

6.4 Query Integration

We evaluate answering similarity join queries with materialized views on several shapes over the PTF dataset. The baseline is computing the query from scratch. The results are depicted in Figure 6. On the x-axis, the arrow points from the available view to the query, e.g., $L^\infty(1) \leftarrow L^1(1)$ means that we answer query with shape $L^\infty(1)$ using a view defined with shape $L^1(1)$. We observe that in

some cases the view is better, while in other cases it is not. It all depends on the relative size of Δ shape compared to the size of the query shape. We depict these shapes for queries $L^\infty(1) \leftarrow L^1(1)$ and $L^\infty(1) \leftarrow L^\infty(2)$ in Figure 4b. Since the ratio for the first query is 4/9, it is more efficient to use the view. The ratio for the second query is 16/9 and the complete similarity join outperforms the view. The analytical cost model we introduce in Section 5 is able to identify the optimal solution.

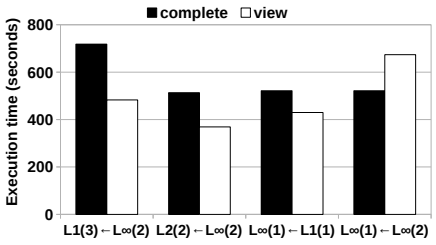


Figure 6: Differential query vs. similarity join on PTF.

6.5 Discussion

The experimental results show that the proposed heuristic provides considerable improvement over the baseline algorithm both for a single update batch as well as for a sequence of batches. The maintenance plan generated by the heuristic always outperforms the baseline—by as much as a factor of 3. The repartitioning increases this factor to almost $5X$ for periodic updates. The time taken by the heuristic is a small fraction of the view maintenance time, more so when compared to the reduction it generates. To put these results in perspective, the proposed solution is able to incrementally maintain the production PTF "association table" under a batch of updates for a month in less than 15 minutes. Currently, update batches are produced every 45 minutes.

7. RELATED WORK

Array databases. While many array databases have been proposed over the years, none of them supports views. In the following, we focus only on how these systems handle the computation of derived array products. We point the interested reader to [44] for a comprehensive discussion on array database systems in general. RasDaMan [6] is a general middleware for array processing with chunks stored as BLOBs in a back-end relational database. RAM [4] and SRAM [15] provide support for array processing on top of the MonetDB [27] columnar database. They do not provide native support for arrays since arrays are represented as relations and array operations are mapped over relational algebra operators. While these systems do not explicitly include array views, it is conceivable that the support for relational views in the back-end systems can be reused. RIOT [54] is a prototype system for linear algebra operations over large vectors and matrices mapped into a standard relational database representation. Linear algebra operations are rewritten into SQL views and evaluated lazily—these are not materialized views. SciDB [9] is the most advanced shared-nothing parallel database system designed specifically for dense array processing. It supports multi-dimensional nested arrays with cells containing records, which in turn can contain multi-dimensional arrays. Although SciDB supports a large set of array operations, it lacks support for views. SciHadoop [10] implements array processing on top of the popular Hadoop Map-Reduce framework which lacks view support. ArrayStore [49] and TrajStore [17] are storage managers optimized for multi-dimensional arrays and trajectories, respectively. They do not provide a query execution strategy to implement it nor views.

Array joins. Positional array equi-joins are introduced in the first releases of SciDB [9]. They are evaluated in the context of different chunking strategies in [49], where structural join is introduced. A complete formalization of array equi-joins and the shuffle join algorithm are given in [20], while a graph formulation is introduced in [7]. Array similarity join is introduced in [56] as a generalization of array equi-joins and distance-based similarity join. While it bears similarities to shuffle join in allocating join units to nodes based on an optimization process, array similarity join encompasses a larger variety of join predicates and is composable.

Incremental view maintenance. Materialized views are a classical concept in databases, with several surveys [23, 24] written on the topic—the most recent by Chirkova and Yang [12]. Based on the discussion in Section 3, array view maintenance falls in the category of deferred maintenance [57, 14, 33, 30] with batch updates [43, 25, 41]; theta- and other complex joins [34, 28]; standard SQL aggregates [52, 42]; parallel/distributed processing [45, 37, 5, 1, 47, 48, 36, 41]; and recursive handling of many joins [2, 41]. The emphasis of query integration is on how to use the view in query optimization [18, 40, 13]—not data integration [24]. We focus the following discussion on parallel incremental view maintenance – introduced in relational databases by [37] – because it considers a similar setting to ours—both the tables and the view are partitioned across servers. The objective of [37] is to maintain materialized views partitioned on a different key than the join keys in the view definition. Communication is minimized by building indexes on the join attributes at each node. Since arrays are partitioned on dimensions, these techniques are not applicable. [5] proposes an incremental view maintenance method that replicates the updates to all the nodes that contain relevant data. This is similar to the baseline algorithm we improve upon. [1] introduces materialized views in Hadoop and designs replication methods for efficient maintenance. [30] employs key-foreign key constraints to identify diffs and avoid join with the base tables. These works consider only equi-joins—a subclass of array similarity join. In the distributed setting considered by [47, 48, 36], the base tables and the view are not partitioned across servers. The goal is to find the optimal join ordering in which to apply the updates. Batch updates – chunks in our case – are shown to be more efficient in a distributed environment by [41]. In the context of array databases, incremental array repartitioning [19] comes the closest to incremental materialized view maintenance. However, the focus is exclusively on repartitioning, not on view maintenance. In [28], algorithms for maintaining k-nn results and spatial joins on continuously moving points are proposed.

8. CONCLUSIONS

In this paper, we introduce materialized array views as a database construct for derived data products in science. We model incremental array view maintenance with batch updates as an MIP optimization and give a three-stage heuristic that finds effective update plans. Moreover, the heuristic repartitions the array and the view continuously based on a window of past updates as a side-effect of view maintenance. We also design an analytical cost model for integrating materialized array views in queries. Experimental results confirm the effectiveness of the heuristics and the quality of the maintenance plan both for a batch as well as for a sequence of update batches. Concretely, the proposed solution is able to incrementally maintain the production iPTF "association table" under a batch of updates for a month in less time than update batches are currently generated.

Acknowledgments. This work is supported in part by the Director, Office of Laboratory Policy and Infrastructure Management of the U.S. Department of Energy under contract No. DE-AC02-05CH11231 and by a U.S. Department of Energy Early Career Award (DOE Career). Weijie Zhao has done part of this work while at Lawrence Berkeley National Laboratory (LBNL). We want to thank the anonymous reviewers for their insightful comments that improved the quality of the paper significantly.

9. REFERENCES

[1] P. Agrawal, A. Silberstein, B. F. Cooper, U. Srivastava, and R. Ramakrishnan. Asynchronous view maintenance for vlsd databases. In *SIGMOD 2009*.

[2] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic. DBToaster: Higher-Order Delta Processing for Dynamic, Frequently Fresh Views. *PVLDB*, 5, 2012.

[3] F. D. Albareti et al. The Thirteenth Data Release of the Sloan Digital Sky Survey: First Spectroscopic Data from the SDSS-IV Survey Mapping Nearby Galaxies at Apache Point Observatory. http://arxiv.org/abs/1608.02013, 2016.

[4] A. R. van Ballegooij. RAM: A Multidimensional Array DBMS. In *EDBT 2004*.

[5] M. Bamha, F. Bentayeb, and G. Hains. An efficient scalable parallel view maintenance algorithm for shared nothing multi-processor machines. In *DEXA 1999*.

[6] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann. The Multidimensional Database System RasDaMan. In *SIGMOD 1998*.

[7] P. Baumann and V. Merticariu. On the Efficient Evaluation of Array Joins. geo-bigdata.github.io/2015/peter.pdf.

[8] J. A. Blakeley, P.-A. Larson, and F. W. Tompa. Efficiently Updating Materialized Views. In *SIGMOD 1986*.

[9] P. Brown et al. Overview of SciDB: Large Scale Array Storage, Processing and Analysis. In *SIGMOD 2010*.

[10] J. B. Buck, N. Watkins, J. LeFevre, K. Ioannidou, C. Maltzahn, N. Polyzotis, and S. Brandt. SciHadoop: Array-based Query Processing in Hadoop. In *SC 2011*.

[11] Y. Cheng and F. Rusu. Formal Representation of the SS-DB Benchmark and Experimental Evaluation in EXTASCID. *Distrib. and Parallel Databases*, 2014.

[12] R. Chirkova and J. Yang. Materialized Views. *Foundations and Trends in Databases*, 4(4):295–405, 2011.

[13] S. Cohen, W. Nutt, and Y. Sagiv. Rewriting Queries with Arbitrary Aggregation Functions using Views. *ACM Transactions on Database Systems (TODS)*, 2006.

[14] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for Deferred View Maintenance. In *SIGMOD 1996*.

[15] R. Cornacchia, S. Héman, M. Zukowski, A. P. de Vries, and P. Boncz. Flexible and Efficient IR using Array Databases. *VLDB Journal (VLDBJ)*, 17, 2008.

[16] P. Cudre-Mauroux, H. Kimura, K.-T. Lim, J. Rogers, S. Madden, M. Stonebraker, S. B. Zdonik, and P. G. Brown. SS-DB: A Standard Science DBMS Benchmark. http://www.xldb.org/science-benchmark/.

[17] P. Cudre-Mauroux, E. Wu, and S. Madden. TrajStore: An Adaptive Storage System for Very Large Trajectory Data Sets. In *ICDE 2010*.

[18] D. DeHaan, P.-A. Larson, and J. Zhou. Stacked Indexed Views in Microsoft SQL Server. In *SIGMOD 2005*.

[19] J. Duggan and M. Stonebraker. Incremental Elasticity For Array Databases. In *SIGMOD 2014*.

[20] J. Duggan, O. Papaemmanouil et al. Skew-Aware Join Optimization for Array Databases. In *SIGMOD 2015*.

[21] A. Gal-Yam et al. Real-Time Detection and Rapid Multiwavelength Follow-Up Observations of a Highly Subluminous Type II-P Supernova from the Palomar Transient Factory Survey. *Astrophysical Journal*, 736(2), 2011.

[22] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[23] A. Gupta and I. S. Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Eng. Bull.*, 18(2), 1995.

[24] A. Y. Halevy. Answering Queries using Views: A Survey. *VLDB Journal*, 10(4):270–294, 2001.

[25] H. He, J. Xie, J. Yang, and H. Yu. Asymmetric Batch Incremental View Maintenance. In *ICDE 2005*.

[26] E. J. Hilton, A. A. West, S. L. Hawley, and A. F. Kowalski. M Dwarf Flares from Time-resolved Sloan Digital Sky Survey Spectra. *The Astronomical Journal*, 140(5), 2010.

[27] S. Idreos et al. MonetDB: Two Decades of Research in Column-Oriented Database Architectures. *IEEE Data Eng. Bull.*, 35(1), 2012.

[28] G. S. Iwerks, H. Samet, and K. P. Smith. Maintenance of K-NN and Spatial Join Queries on Continuously Moving Points. *ACM Transactions on Database Systems (TODS)*, 31(2), 2006.

[29] B. Jansen. Constrained Bipartite Vertex Cover: The Easy Kernel Is Essentially Tight. In *LIPIcs 2016*.

[30] Y. Katsis, K. W. Ong, Y. Papakonstantinou, and K. K. Zhao. Utilizing IDs to Accelerate Incremental View Maintenance. In *SIGMOD 2015*.

[31] H. Kellerer, U. Pferschy, and D. Pisinger. *Introduction to NP-Completeness of Knapsack Problems*. Springer, 2004.

[32] L. J. Kewley, W. R. Brown, M. J. Geller, S. J. Kenyon, and M. J. Kurtz. SDSS 0809+ 1729: Connections Between Extremely Metal-Poor Galaxies and Gamma-Ray Burst Hosts. *The Astronomical Journal*, 133(3), 2007.

[33] H. A. Kuno and G. Graefe. Deferred Maintenance of Indexes and of Materialized Views. In *DNIS 2011*.

[34] P.-A. Larson and J. Zhou. Efficient Maintenance of Materialized Outer-Join Views. In *ICDE 2007*.

[35] K.-T. Lim, D. Maier, J. Becla, M. Kersten, Y. Zhang, and M. Stonebraker. ArrayQL Syntax. http://www.xldb.org/wp-content/uploads/2012/09/ArrayQL-Draft-4.pdf. [Online; February 2017].

[36] B. Liu and E. A. Rundensteiner. Cost-driven General Join View Maintenance over Distributed Data Sources. In *ICDE 2005*.

[37] G. Luo, J. F. Naughton, C. J. Ellmann, and M. W. Watzke. A Comparison of Three Methods for Join View Maintenance in Parallel RDBMS. In *ICDE 2003*.

[38] D. Maier. ArrayQL Algebra: version 3. http://www.xldb.org/wp-content/uploads/2012/09/ArrayQL_Algebra_v3+.pdf. [Online; February 2017].

[39] A. P. Marathe and K. Salem. Query Processing Techniques for Arrays. *VLDB Journal*, 11(1):68–91, 2002.

[40] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham. Materialized View Selection and Maintenance Using Multi-Query Optimization. In *SIGMOD 2001*.

[41] M. Nikolic et al. How to Win a Hot Dog Eating Contest: Distributed Incremental View Maintenance with Batch Updates. In *SIGMOD 2016*.

[42] T. Palpanas, R. Sidle, R. Cochrane, and H. Pirahesh. Incremental Maintenance for Non-Distributive Aggregate Functions. In *VLDB 2002*.

[43] D. Quass and J. Widom. Online View Maintenance. In *SIGMOD 1997*.

[44] F. Rusu and Y. Cheng. A Survey on Array Storage, Query Languages, and Systems. *CoRR*, abs/1302.0103, 2013.

[45] K. Salem, K. Beyer, B. Lindsay, and R. Cochrane. How to Roll a Join: Asynchronous Incremental View Maintenance. In *SIGMOD 2000*.

[46] A. D. Sarma, Y. He, and S. Chaudhuri. ClusterJoin: A Similarity Joins Framework using MapReduce. *PVLDB*, 7, 2014.

[47] A. Segev and J. Park. Maintaining Materialized Views in Distributed Databases. In *ICDE 1989*.

[48] A. Segev and J. Park. Updating Distributed Materialized Views. *TKDE*, 1989.

[49] E. Soroush, M. Balazinska, and D. Wang. ArrayStore: A Storage Manager for Complex Parallel Array Processing. In *SIGMOD 2011*.

[50] J. D. Ullman. NP-Complete Scheduling Problems. *Journal of Computer and System Sciences*, 10(3), 1975.

[51] Y. Wang, X. Yang, H. Mo, F. C. Van den Bosch, S. M. Weinmann, and Y. Chu. The Clustering of SDSS Galaxy Groups: Mass and Color Dependence. *The Astrophysical Journal*, 687(2), 2008.

[52] J. Yang and J. Widom. Incremental Computation and Maintenance of Temporal Aggregates. In *ICDE 2001*.

[53] X. Yang, H. Mo, F. C. Van den Bosch, A. Pasquali, C. Li, and M. Barden. Galaxy Groups in the SDSS DR4. *The Astrophysical Journal*, 671(1), 2007.

[54] Y. Zhang, H. Herodotos, and J. Yang. RIOT: I/O-Efficient Numerical Computing without SQL. In *CIDR 2009*.

[55] Y. Zhang, M. Kersten, M. Ivanova, and N. Nes. SciQL: Bridging the Gap between Science and Relational DBMS. In *IDEAS 2011*.

[56] W. Zhao, F. Rusu, B. Dong, and K. Wu. Similarity Join over Array Data. In *SIGMOD 2016*.

[57] J. Zhou, P.-A. Larson, and H. G. Elmongui. Lazy Maintenance of Materialized Views. In *VLDB 2007*.

APPENDIX

A. NP-HARD PROOFS

We provide reductions from known NP-hard problems to our formulations. Formal complete equivalence proofs are immediate once a reduction is established.

A.1 Differential View Computation

DEFINITION 2. *Given update triples $U_0 = \{(p, q, *)\}$ consisting of array chunks p, q having size B_p, B_q and being located on server S_p and S_q, respectively, the differential view computation has to determine how to replicate these chunks such that there exists a server j that contains both p and q, $\forall (p, q) \in U_0$, and guarantees that $ntwk[j] \leq K$, $\forall j \in \{1..N\}$, where K is an arbitrary constant. Replicating a chunk q incurs a cost of B_q to server S_q—same for p.*

This simplified decision problem ignores the CPU cost in the optimization by setting $T_{cpu} = 0$. Moreover, it is polynomially equivalent to the min over max objective by setting max to K.

DEFINITION 3. *Given a bipartite graph G with the vertex partitions L and R, the constrained bipartite vertex cover (CBVC) problem has to determine whether there exists a vertex cover containing at most K_L vertices from L and K_R vertices from R, where K_L and K_R are arbitrary constants. CBVC is NP-hard [29].*

REDUCTION 1. *For each vertex $l \in L$, create a chunk l located on server 1 ($S_l = 1$). B_l – the size of chunk l – is $1/K_L$. For each vertex $r \in R$, create a chunk r located on server 2 ($S_r = 2$). For each edge (l, r), create an update triple $(l, r, *)$ in U_0. B_r – the size of chunk r – is $1/K_R$. Constant K is set as 1. If there exists a solution for differential view computation, a solution for CBVC exists—and vice-versa.*

A.2 View Chunk Reassignment

DEFINITION 4. *Given update triples $U_0 = \{(p, q, v)\}$ consisting of array chunks p and q already joined at server k ($z_{pqk} = 1$), and view chunk v located originally at server S_v, view chunk reassignment has to determine the server S'_v such that the largest view merge time across the N servers is minimized. Triples $(*, *, v)$ have to be moved to the same server for merging.*

DEFINITION 5. *Given a machine with n processors and m jobs, job i taking J_i time to be processed, in multiprocessor scheduling we have to distribute the m jobs to the multiprocessors such that the latest processing time across multiprocessors is minimized. Multiprocessor scheduling is NP-hard [50].*

REDUCTION 2. *For each job, create a differential view with size J_i corresponding to the join between array chunk p and q. Set the number of servers to n. This multiprocessor scheduling corresponds to a simplified view chunk reassignment that does not even consider the correlations imposed by the update triples in U_0—all the triples are independent.*

A.3 Array Chunk Reassignment

DEFINITION 6. *Given N servers and a list of quadruples $L = \{p, q, v, s\}$ meaning that if array chunks p and q are both on server v generates a score s, in array chunk reassignment we have to maximize the overall score across quadruples by assigning p and q to servers. The total size of the chunks assigned to a server j can be at most cpu_thr_j.*

DEFINITION 7. *Consider a knapsack with capacity W and n items. Each item has size w_i and value v_i. We are also given a list of triples $Q = (i, j, k)$. If item i and item j are both packed in the knapsack, we get an additional value k. In the quadratic knapsack problem, we have to pack the items such that the overall value is maximized. This is an NP-hard problem [31].*

REDUCTION 3. *Set the number of servers N as 2, i.e., we have two servers, 0 and 1. Server 1 corresponds to packing an item, while server 0 to not packing it. For each item i, create an array chunk i with size w_i. Create an additional dummy array chunk 0 with size 0. For each triple in Q, create a quadruple $(i, j, 1, k)$ in L. For each item i, create a quadruple $(i, 0, 1, v_i)$ in L. Set cpu_thr_1 as W.*

B. ALGORITHM EXAMPLES

In order to illustrate how the proposed heuristic view maintenance procedure works, we provide examples for each stage based on the input in Figure 1.

B.1 Differential View Computation

The set of update triples U_0 is given by the insertions in Figure 1 (b). In this example, we consider the following 4 triples $(\Delta A_4, A_1, *)$, $(\Delta A_2, A_1, *)$, $(\Delta A_2, \Delta A_3, *)$, and $(\Delta A_7, A_2, *)$ resulted after the random ordering. The size of a chunk B_p is given by the number of non-empty cells, while its location S_p is as in Figure 1. T_{ntwk} is set to 4 and T_{cpu} to 1, respectively. The state of the algorithm when the 4$^{\text{th}}$ triple $(\Delta A_7, A_2, *)$ is processed is shown on the top part of Figure 7. For example, server X stores chunks A_1 and A_4; a replica of chunk ΔA_2; does not transfer any chunk, thus *ntwk* $= 0$; and processes two joins, $\Delta A_4 \bowtie A_1$ and $\Delta A_2 \bowtie A_1$, thus *cpu* $= 4$. The algorithm considers the cost of assigning the join $\Delta A_7 \bowtie A_2$ to each of the servers and selects the one having minimum value for *opt_now*—computed as the maximum between the network and cpu costs across the servers. If the join is assigned to server Y, ΔA_7 has to be transferred from X to Y for a network cost of $T_{ntwk} \times B_{\Delta A_7} = 4 \times 1 = 4$ (incurred by X) and a cpu cost $T_{cpu} \times (B_{\Delta A_7} + B_{A_2}) = 1 \times (1 + 1) = 2$ on Y. These costs are combined with the existing cost at X and Y to generate a maximum of 4 for *opt_now*. Since this value is the minimum across the three servers – *opt_now* $= 8$ on X and Z – $\Delta A_7 \bowtie A_2$ is assigned to Y.

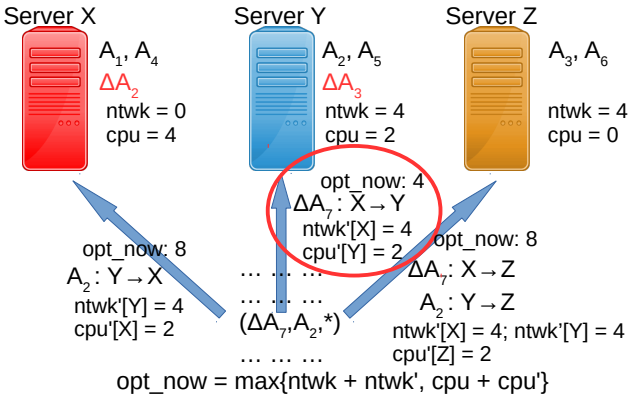


Figure 7: Example for Algorithm 1.

B.2 View Chunk Reassignment

The output of differential view computation – which is the input to view chunk reassignment – is depicted in the top two tables of Table 2. It consists of the network and cpu costs for each server and the assignment of joins to servers. In this example, we include only the joins relevant to V_1—the first view chunk considered. As the tables show, both the communication and computation are balanced across the servers. The algorithm tries to assign V_1 to each of the servers and selects the one incurring the minimum cost—computed as in Algorithm 1. The table at the bottom of Table 2 depicts the cost *opt_now* corresponding to each server. In the optimal assignment, V_1 is moved to server Y together with joins J_1 and J_2—computed on X. The reason for this assignment is the availability of computation resources on Y. This is not the case for X. While Z has network resources, it does not contain any of the join results required by V_1.

server	*ntwk*	*cpu*
X	32	36
Y	36	30
Z	30	35

join result	server
$J_1 : \Delta A_1 \bowtie A_1$	X
$J_2 : \Delta A_4 \bowtie A_1$	X
$J_3 : \Delta A_2 \bowtie A_1$	Y

$V_1 \rightarrow$	transfers	*ntwk'*	*cpu'*	*opt_now*
X	J_3	$Y = 4$	$X = 6$	42
Y	J_1, J_2	$X = 8$	$Y = 6$	**40**
Z	J_1, J_2, J_3	$X = 8, Y = 4$	$Z = 6$	41

Table 2: Example for Algorithm 2.

B.3 Array Chunk Reassignment

The input to Algorithm 3 is represented by historical update triples rather than the current update batch. Each of the (array chunk, view chunk) pair appearing in the triples is assigned a score based on their frequency. These are depicted in the left table of Figure 8. The other inputs to the algorithm are the size and location of the array chunks as computed by Algorithm 1 (bottom-right table in Figure 8) and the assignment of the view chunks computed by Algorithm 2 (top-right table in Figure 8). The (array chunk, view chunk) pairs are considered in descending order of their score and the array chunk is assigned to one of the replicas that also contains the view chunk capped by its cpu processing quota *cpu_thr*. Chunk A_2 in pair (A_2, V_1) is assigned to Y because it contains V_1 and has sufficient capacity. A_1 in (A_1, V_1) is ignored because it is not replicated on Y and later assigned to X when considered in (A_1, V_2). A_3 is finally assigned to Z which cannot be assigned any other chunks further because it is at capacity—*cpu_thr* $= 0$.

score	
A_2, V_1	8
A_1, V_1	6
A_1, V_2	4
A_2, V_3	4
A_3, V_3	2
...	...

server	views	cpu_thr
X	V_2, V_6	4
Y	V_1, V_4, V_7	3
Z	V_3, V_5, V_8	1

	A_1	A_2	A_3	...
size	1	1	1	...
replica	X, Z	Y, Z	Z, Y	

Figure 8: Example for Algorithm 3.

C. ADDITIONAL EXPERIMENTS

C.1 Overall View Maintenance Time

The overall time incurred by optimization and view maintenance across the entire batch of updates is depicted in Figure 9. As expected, the benefit of repartitioning is maximized for correlated batches. In this case, reassign is faster than baseline by more than $3X$ on PTF-25. Reassign always outperforms differential, even in the case of GEO random batches and with the optimization time included. The optimization overhead is marginal in the overall maintenance time compared to the reduction it brings. In the worst case – PTF-5 with real batches – reassign incurs an overhead of 6 seconds in optimization, while achieving a reduction of 300 seconds in overall maintenance time—100 seconds over differential.

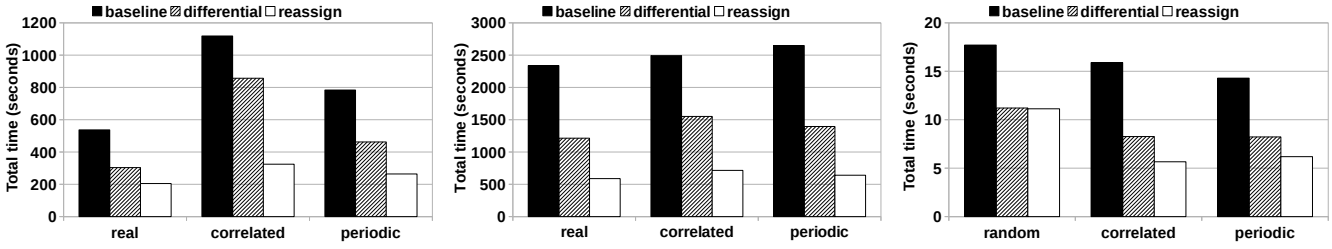


Figure 9: Overall execution time (optimization + view maintenance): (a) PTF-5, (b) PTF-25, and (c) GEO.

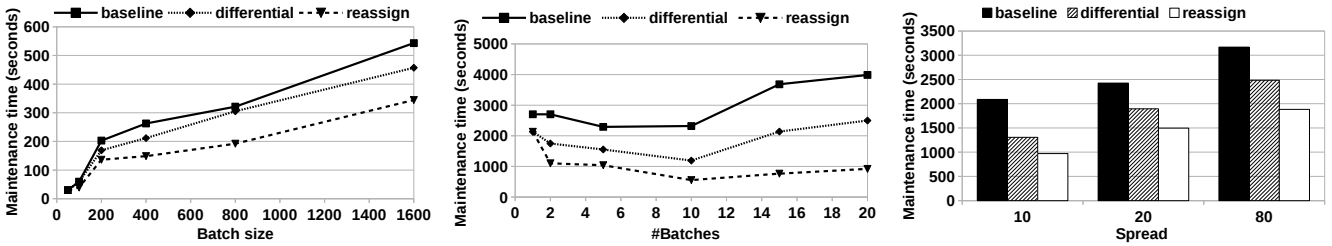


Figure 10: Sensitivity analysis on PTF-25 with real updates: (a) increasing batch size, (b) number of batches, (c) update spread.

C.2 Sensitivity to Batch Size

The sensitivity experiments depicted in Figure 10 are executed over the PTF-25 view because this has the most complicated shape in the definition. The larger maintenance time permits a clearer investigation of the scalability of the proposed algorithms. Figure 10a depicts the maintenance time for update batches consisting of increasing number of chunks. The real update PTF workload is partitioned into batches with exponentially increasing number of chunks—50, 100, 200, 400, 800, and 1600. The batches are fed into the view maintenance algorithms in this order. As expected, larger batches incur a linear increase in maintenance time. When the number of chunks in a batch is below 200, the difference between the three algorithms is minimal because the number of update triples is small. However, as the size of the batch increases – and the number of update triples – the gap between reassign and the other algorithms increases—it is 200 seconds for a batch with 1600 chunks. While the optimization time also increases linearly with the batch size, it represents an insignificant fraction of the maintenance time – less than 1% – with an absolute value below 3 seconds for 1600 chunks.

C.3 Sensitivity to Number of Batches

Figure 10b depicts the sensitivity of the view maintenance algorithms as a function of the number of batches for a fixed update workload. In this case, the real PTF workload is divided into batches with the same number of chunks. The goal is to identify the optimal batch size. For a single batch, reassign and differential are identical and superior to baseline due to reduced communication and better load balancing. Reassign exhibits the smallest variance with the number of batches—if one batch is excluded. All the algorithms achieve the smallest maintenance time for 10 batches which is in the middle of the considered range. Baseline and differential suffer a significant increase if the number of batches grows beyond this point. This clearly shows that many small batches are not optimal because of the overhead they incur. However, reassign is able to use a larger number of batches to find a better chunk assignment that compensates for the increased overhead.

C.4 Sensitivity to Update Spread

Figure 10c depicts maintenance time as a function of the spread of updates over the range of (ra, dec) with a fixed number of batches (10) and sampled chunks per batch (500) that overlap with the range. Spread value 10 corresponds to a rectangle of 10 chunks on *ra* and *dec*—100 chunks overall. 20 doubles the range of 10 on both dimensions while guaranteeing inclusion—similar for 80. The larger the spread, the least concentrated the updates are. Thus, less sharing is possible which results in longer maintenance time. However, the increase for reassign is smaller in absolute value – 900 compared to more than 1000 – than for the alternatives.