Similarity Join over Array Data

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

ABSTRACT

Scientific applications are generating an ever-increasing volume of multi-dimensional data that are largely processed inside distributed array databases and frameworks. Similarity join is a fundamental operation across scientific workloads that requires complex processing over an unbounded number of pairs of multi-dimensional points. In this paper, we introduce a novel distributed similarity join operator for multi-dimensional arrays. Unlike immediate extensions to array join and relational similarity join, the proposed operator minimizes the overall data transfer and network congestion while providing load-balancing, without completely repartitioning and replicating the input arrays. We define formally array similarity join and present the design, optimization strategies, and evaluation of the first array similarity join operator.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*query processing*

1. INTRODUCTION

In the era of data deluge, scientific applications 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] (PTF) record observations of stars and galaxies at nightly rates varying between 60 and 500 GB. The Large Synoptic Survey Telescope[3] (LSST), which is under construction, is expected to increase these volumes by two orders of magnitude—to 20 TB. Other scientific domains such as high-energy physics and genomics produce even larger datasets.

A common characteristic of these and many other scientific applications 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 [10], numerous array processing solutions that implement a distributed multi-dimensional array data model have emerged [2, 43, 30, 47, 6, 8]. To cope with the massive data volumes, these systems partition the arrays across a *distributed shared-nothing cluster* and process the partitions concurrently.

Queries on arrays fall in two categories [41]. 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 aggregate. The second category contains array-specific operations such as smoothing, regridding, clustering, and cross-matching, which are not built-in operators in relational databases. These are supported as User-Defined Functions (UDF) in specialized array processing systems [2, 30, 8]. A common feature of the array-specific operations is that they apply a multi-dimensional shape to each cell of an array, essentially pairing the cell with other cells that are in its vicinity in the multi-dimensional array space. A subsequent function can be applied to the resulting group to generate a single aggregate value. To better illustrate this operation, we provide an example from astronomy.

Cross-matching astronomical catalogs. A set of observed celestial objects characterized by their position and the time of their observation is stored in a catalog, i.e., database. Given a newly extracted set of objects – from a new image – the goal in catalog matching [1] is to determine if they have appeared before in the catalog – they are matches of existing objects – or they are entirely new. Astronomical applications use the matches differently. For example, in the PTF project, matches are used to find transient objects [35], i.e., objects that are not permanent in the sky, such as supernovae and variable stars. Computing accurate matches is the most critical step in the PTF real-time transient detection pipeline because of the large number of potential candidates, i.e., $1 - 1.5$ million per night, and the limited time window for follow-up observations—several hours to a few days. Therefore, there has been considerable interest in developing efficient algorithms for the catalog matching problem [28, 45, 34]. Even with these algorithms, matching in PTF considers only a single catalog – the goal is to include several catalogs, e.g., SDSS, BOSS[4], NED[5] – and applies severe pruning and ranking to limit the number of matches. As a generalization, consider two distinct catalogs, e.g., PTF and SDSS, that store different properties of celestial objects. The cross-matching problem aims to create a single catalog that contains a unique instance of the objects in the two input catalogs that merges all their properties. In the PTF context, such a unified catalog has the potential to enhance the effectiveness of the real-time transient detection pipeline because it includes a larger number of properties. It is, therefore, a central piece of the PTF project.

In database jargon, catalog cross-matching corresponds to `FULL OUTER JOIN`. However, due to the uncertainty of astronomical

[1]www.sdss.org/dr12/
[2]www.astro.caltech.edu/ptf/
[3]www.lsst.org

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA
© 2016 ACM. ISBN 978-1-4503-3531-7/16/06...$15.00
DOI: http://dx.doi.org/10.1145/2882903.2915247

[4]https://www.sdss3.org/instruments/boss_spectrograph.php
[5]https://ned.ipac.caltech.edu/

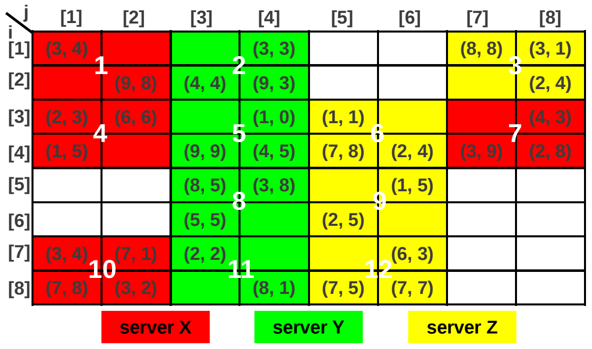


Figure 1: Array `A<v:int,u:int>[i=1,8,2;j=1,8,2]`.

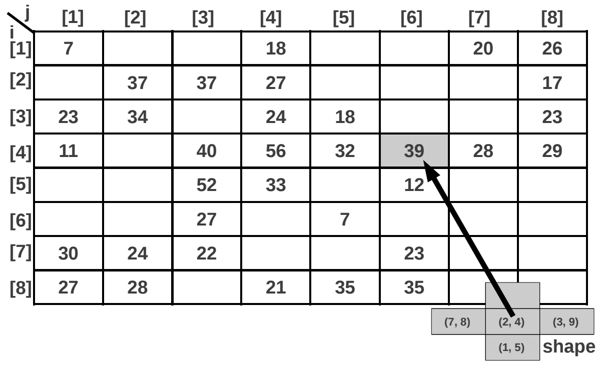


Figure 2: Result array of `APPLY(A, SUM(v+u), shape)`.

observations – the exact position of an object in the sky varies from one catalog to another – the matching is not exact, but rather approximate. For example, in the PTF transient detection pipeline, objects that are within 3 arcseconds of each other over the past hour are considered instances of the same celestial body. Thus, the join condition becomes a distance function inequality. This type of condition corresponds to similarity join [21].

An astronomical catalog can be represented as an array having dimensions the celestial coordinates *ra* and *dec*, and the *time* when the image is taken—a 3-D array `catalog[ra,dec,time]`. The range of each dimension is set according to the accuracy of the telescope. A cell in the array corresponds to an observed object, which is entirely identified by its index values in the array. As a result, distance-based similarity maps to a positional neighborhood relationship among array cells. The neighborhood of a cell can be expressed concisely as a shape array, defined as a collection of offsets in each dimension around the cell. Moreover, mapping from distance to shape array is a required step in similarity join whenever the input is array data. For example, in the PTF data, a cell corresponds to 1 arcsecond on *ra* and *dec*, and 1 minute on *time*. The shape array corresponding to identical objects defined above, e.g., within 3 arcseconds of each other over the past 60 minutes, is a $7 \times 7 \times 60$ cuboid with offsets $[-3:3, -3:3, -60:-1]$ on *ra*, *dec*, and *time*, respectively.

Problem statement & approach. In this paper, we investigate the design and implementation of an array similarity join operator for a distributed array database. Unlike previous work on relational data, we define similarity based on a shape array instead of a distance function. This novel formulation takes into consideration the discrete nature of array data and supports asymmetric similarity measures. We introduce a novel operator that builds upon existing array join algorithms [19, 3] by minimizing the overall data transfer and network congestion while providing load-balancing across the nodes that store data, but without completely repartitioning and replicating the arrays. In the query optimization phase, the operator computes an optimal execution plan for each of the worker nodes. The plan consists of three components—transfer graph, transfer schedule, and data access plan. Finding the optimal plan is challenging because it involves solving a complex non-linear optimization problem. Our solution is to decompose the original optimization problem into three separate sub-problems – one for each plan component – and solve them independently. The solution at each stage is taken as a pre-condition in the following stage. Even with these simplifications, finding the optimal solution at a stage cannot be done efficiently. We design graph-based heuristic algorithms that find competitive solutions much faster. At query execution, the array similarity join operator overlaps I/O – disk and network – with join computation which is heavily parallelized using a dynamic thread pool. Thus, several joins can be executed concurrently. This adds additional pressure on the I/O components – network, in particular – and makes their optimization critical.

Contributions. The main contributions of this work are:

- We define formally array similarity join with a shape array. As far as we know, this is the first array similarity join formulation that generalizes distance-based similarity. (Section 3)
- We introduce the first array similarity join operator that minimizes the overall data transfer and network congestion while providing load-balancing, but without completely repartitioning and replicating the arrays. Moreover, the array operator overlaps disk, network, and join computation in a multi-thread pipelined architecture. (Section 4.1-4.2)
- We model the query optimization of array similarity join as a vertex cover problem and introduce efficient algorithms to find optimal execution plans. (Section 4.3-4.5)
- We evaluate experimentally the proposed array similarity join operator and compare it against existing solutions on the PTF catalog consisting of 1 billion celestial objects. The results confirm the efficacy of the optimizations in reducing the overall data transfer as well as the execution time. (Section 5)

2. PRELIMINARIES

In this section, we introduce the Array Data Model (ADM), array joins, and similarity join over multi-dimensional data. These concepts are the foundation for array similarity join.

2.1 Array Data Model

A multi-dimensional array [41, 16, 19] is defined by a set of *dimensions* $\mathcal{D} = \{D_1, D_2, \ldots, D_d\}$ and a set of *attributes* $\mathcal{A} = \{A_1, A_2, \ldots, A_m\}$. These define the schema of the array. Each dimension D_i is a finite totally ordered discrete set. Without loss of generality, 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 $i_1, i_2, \ldots, i_d$, i.e., coordinates, defines a *cell*. All cells in a given array have the same type, given by the set of attributes $\mathcal{A}$. Each attribute has a scalar type, such as an integer or float. This is identical to the relational model. Based on these concepts, an array can be thought of as a function defined over dimensions and taking values attribute tuples:

$$Array : D_1 \times D_2 \times \cdots \times D_d \longmapsto (A_1, A_2, \ldots, A_m) \quad (1)$$

The cells for which the function is defined – occupied cells – contain data. The other cells are empty.

Chunking. Array databases apply chunking for storing, processing, and communicating arrays. A *chunk* groups several array cells into a single access – memory, I/O, network – and processing unit. To some extent, chunks are the equivalent of pages in relational databases. The main difference is the size—a page is in the order of kilobytes and a chunk is in the order of megabytes. While many strategies have been proposed in the literature – see [16] for a survey – *regular chunking* is the most popular in practice, e.g., SciDB. We adopt it in this paper. Chunks have the same dimensionality as the input array, are aligned with the dimensions, and have the same shape. There are many strategies to layout the array cells inside a chunk [16]. Without loss of generality, we adopt the *standard C-style ordering on the dimensions*, where the cells are sorted one dimension at a time, traversing the innermost dimension completely before increasing the outer dimension value. Dimensions are ordered according to the array schema. Array chunks are *vertically partitioned*, i.e., each attribute is stored in a separate set of pages, similar to column-oriented databases.

Shared-nothing architecture. We assume a distributed array database having a *shared-nothing architecture* over a cluster of *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, i.e., nodes, in order to support parallel processing. Unlike distributed systems such as Hadoop Map-Reduce, chunk replication is not a major concern. 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.

EXAMPLE 1 (ARRAY CHUNKING). *Figure 1 depicts a 2-D array with schema* `A <v:int,u:int> [i=1,8,2;j=1,8,2]` *given in AQL [25, 24]—the analogous of SQL for array databases. Array* `A` *has two dimensions,* `i` *and* `j`*, with values in* $[1 \ldots 8]$. *Each dimension has a chunk interval of* 2*, for a total of* 16 *chunks. The array has two attributes,* `v` *and* `u`*, of integer type. Out of the* 16 *chunks, only* 12 *contain data—colored and numbered in Figure 1. These chunks are distributed round-robin in row-major order over the* 3 *servers on which the array database runs. Only the cells that contain data are materialized on disk.*

2.2 Array Joins

Consider two d-dimensional arrays α and β given in the functional representation in Eq. (1):

$$\alpha : \{\mathcal{D}^{\alpha} = D_1^{\alpha} \times \cdots \times D_d^{\alpha}\} \longmapsto \{\mathcal{A}^{\alpha} = (A_1^{\alpha}, \ldots, A_m^{\alpha})\}$$
$$\beta : \left\{\mathcal{D}^{\beta} = D_1^{\beta} \times \cdots \times D_d^{\beta}\right\} \longmapsto \left\{\mathcal{A}^{\beta} = (A_1^{\beta}, \ldots, A_m^{\beta})\right\}$$

A join between arrays α and β, $\tau = \alpha \bowtie_P \beta$, is written in AQL as: `SELECT expression INTO` τ `FROM` α `JOIN` β `ON P`, where P is the join predicate which consists of pairs of attributes and/or dimensions from the two source arrays. The output is a new array $\tau : \mathcal{D}^{\tau} \longmapsto \mathcal{A}^{\tau}$ having the default schema:

$$\mathcal{D}^{\tau} = \mathcal{D}^{\alpha} \cup \mathcal{D}^{\beta} \qquad \mathcal{A}^{\tau} = \mathcal{A}^{\alpha} \cup \mathcal{A}^{\beta} \tag{2}$$

in which both the dimensions and the attributes from the input schemas are merged. Essentially, the result array has dimensionality equal to the sum of the dimensionality of the input arrays and each cell contains the union of the attributes. It is important to notice that the non-empty cells are given exclusively by the combination of non-empty cells in the input arrays. For example, cell [1,1,4,2] in the join result $A \bowtie A$ in Figure 1 is empty since cell [4,2] in `A` is empty. As is the case with the relational cross product, the default array join is not of particular practical importance since it requires the replication of array β for every chunk of α.

Duggan et al. [19] introduce a series of array equi-joins – *dimension:dimension*, *attribute:attribute*, and *attribute:dimension* – for the case in which predicate P contains only equality conditions. Out of the three types of array equi-join, dimension:dimension join is the most relevant for array databases since these do not provide any benefit for the other two types, compared to their relational counterparts.

2.3 Similarity Join

We consider the ϵ-join version [21] of similarity join. The input to ϵ-join is given by two d-dimensional datasets $A, B \in \mathbb{R}^d$. The goal is to find all the pairs of points $(\vec{a}, \vec{b})$, where $\vec{a} \in A$ and $\vec{b} \in B$, such that $||\vec{a} - \vec{b}|| < \epsilon$. This can be written formally as:

$$A \bowtie_{\epsilon} B = \left\{(\vec{a}, \vec{b}) : ||\vec{a} - \vec{b}|| < \epsilon; \vec{a} \in A, \vec{b} \in B\right\} \tag{3}$$

where $\vec{a} = (a_1, \ldots, a_d)$ and $\vec{b} = (b_1, \ldots, b_d)$ are d-dimensional points and $||\vec{a} - \vec{b}||$ is the distance between $\vec{a}$ and $\vec{b}$ measured using the generic L^p norm:

$$||\vec{a} - \vec{b}||^p = \left[\sum_{i=1}^{d} (a_i - b_i)^p\right]^{\frac{1}{p}} \tag{4}$$

with $p = 1, 2, \ldots, \infty$. L^2 corresponds to the familiar Euclidean distance, L^1 to the Manhattan distance, and L^{∞} to the maximum distance in any dimension. Abstractly, similarity join can be represented as an array join between two d-dimensional arrays. These arrays are obtained by enforcing that each cell corresponds to a single point in the original dataset—ADM does not support attributes having container data type. As long as the original dataset does not contain duplicates, this can be achieved by increasing the resolution on dimensions.

3. ARRAY SIMILARITY JOIN

In this section, we define formally array similarity join with a shape array. To the best of our knowledge, we are the first to propose this novel operation. We show how standard similarity measures map into corresponding shape arrays and discuss the relationship of array similarity join with relational similarity join and array joins, respectively.

While ϵ-join is well-defined over relational tuples, it is not entirely extensible to arrays due to the discrete nature of the dimensions. Similarity join over arrays has to take into account the ordered structure imposed by dimensions. We define formally such an operation starting from the `APPLY` operators in the ArrayQL algebra [25] and AML [26], respectively. In the simplest form, `APPLY` applies a function given as argument to the attributes of each non-empty cell of an array. The output is an array with the same dimensionality, containing the result of the function in each cell. In a more general `APPLY` operator, the argument function is defined over an array shape and generates as output another shape. The function is applied to the shape centered on each cell in the array. The value of the output cell is a function of multiple adjacent cells in the input array. The most common case is when the output shape is a single array cell. In this case, there is a direct correspondence between the origin cell in the input array and the output cell. Figure 2 depicts the result of `APPLY(A, SUM(v+u), shape)` over array `A` in Figure 1. The sum of $v + u$ across all the cells

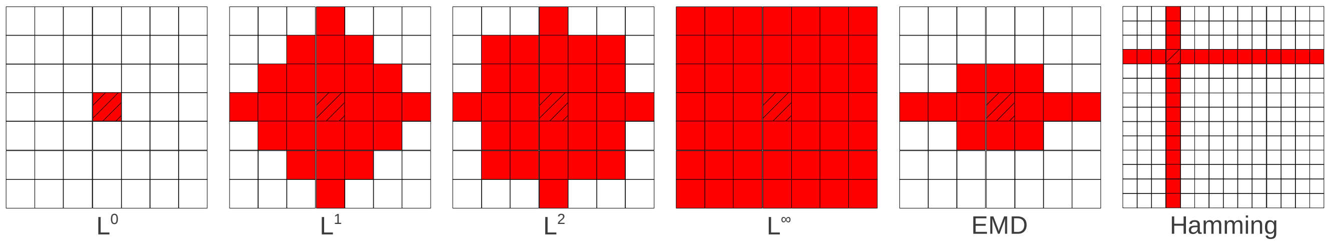


Figure 3: Mapping of similarity distance into shape array for $\epsilon = 3$ (L^0, L^1, L^2, L^∞, Earth Mover's Distance (EMD)) and $\epsilon = 1$ (Hamming).

within Manhattan distance of 1 – the shape is a symmetric cross – is computed for every non-empty cell. We show explicitly how the output for cell $[i = 4; j = 6]$ is generated in the figure.

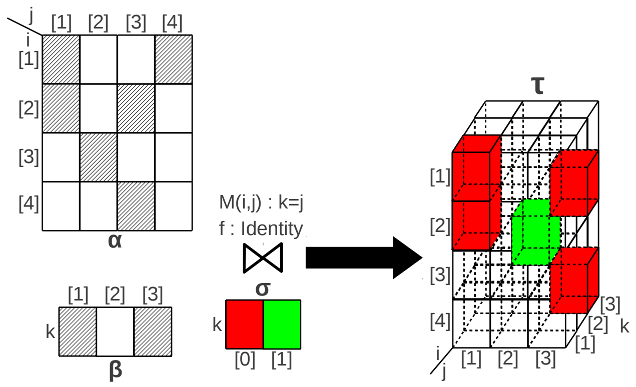


Figure 4: Array similarity join example:
SELECT $\star$ INTO τ FROM α SIMILARITY JOIN β
ON $(k = j)$ WITH SHAPE σ

Definition 1 (Array Similarity Join). *Given two multi-dimensional arrays α and β in functional representation:*

$$\alpha : \{\mathcal{D}^\alpha = D_1^\alpha \times \cdots \times D_d^\alpha\} \longmapsto \{\mathcal{A}^\alpha = (A_1^\alpha, \ldots, A_m^\alpha)\}$$

$$\beta : \left\{\mathcal{D}^\beta = D_1^\beta \times \cdots \times D_d^\beta\right\} \longmapsto \left\{\mathcal{A}^\beta = (A_1^\beta, \ldots, A_m^\beta)\right\},$$

a mapping function $\mathcal{M} : \mathcal{D}^\alpha \longmapsto \mathcal{D}^\beta$, a shape array $\sigma : \mathcal{D}^\beta \longmapsto ()$, and a value function $f : \mathcal{A}^\alpha \cup \mathcal{A}^\beta \longmapsto \mathcal{A}^\tau$, let $\tau : \mathcal{D}^\alpha \cup \mathcal{D}^\beta \longmapsto \mathcal{A}^\tau$ be the result of the similarity join between α and β under $\mathcal{M}$, σ, and f, i.e., $\alpha \bowtie_{\sigma,f}^{\mathcal{M}} \beta$, written in AQL as:

SELECT f INTO τ
FROM α SIMILARITY JOIN β ON $\mathcal{M}$ WITH SHAPE σ

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 $f(\Upsilon, \sigma(\Psi))$.

According to the definition, PTF catalog cross-matching corresponds to array similarity join between two 3-D arrays. While identity over *ra* and *dec* is an immediate choice for the mapping function, the mapping on *time* depends on the timestamps of the two catalogs. The shape array is centered on the mapped cell, e.g., same index for *ra* and *dec*, and the closest index on *time*. The actual shape is defined in the query, e.g., the $7 \times 7 \times 60$ cuboid with offsets $[-3 : 3, -3 : 3, -60 : -1]$ given in the introduction. Since the result is a 6-D array we cannot depict it graphically. Instead, we provide an illustrative example between a 2-D and a 1-D array corresponding to `[ra, dec]` and `[dec]`, respectively.

Example 2. *Figure 4 depicts the result of the array similarity join between a 2-D array $\alpha[i = 1,4; j = 1,4]$ and a 1-D array $\beta[k = 1,3]$. The mapping function $\mathcal{M}$ maps a cell $(i,j) \in \alpha$ to the cell in β which has index $k = j$. This is the Ψ cell to which shape $\sigma[k = 0,1]$ is applied. The range of k in $\sigma[k = 0,1]$ determines the relationship between j and k in the join. In this case, a cell with index j from α is similar with cells having index $k = j$ or $k = j+1$ from β. The value function f concatenates the attributes of α and β into a single tuple. The schema of the result array is $\tau[i = 1,4; j = 1,3; k = 1,3]$. Since there is no valid mapping for $j = 4$, the range of j in τ is reduced to $[j = 1,3]$. There are 5 non-empty cells in τ—the non-empty cells in α and β are hatched. 4 of them – red in Figure 4 – correspond to the condition $k = j$. The only cell corresponding to condition $k = j + 1$ is $[i = 3; j = 2; k = 3]$—green in Figure 4.*

Mapping of similarity distance into shape array. Figure 3 depicts the shape array corresponding to several common similarity distances, e.g., L^p norms, Earth Mover's Distance (EMD), and Hamming, for a 2-D array. Other similarity measures, such as cosine and Jaccard similarity, do not have a suitable representation for array data. In all the figures, the Ψ cell is hatched. For the L^p norms, the size of the shape array increases as p increases. Since L^0 and L^∞ are defined as limits, we introduce special shapes for them. The shape array corresponding to L^0 is cell Ψ, while the shape array of L^∞ covers a square with side $(2\epsilon + 1)$. Although EMD and total variation distance (TVD) are defined for probability distributions, we can adapt them to arrays. The main characteristic of EMD is that dimensions are not symmetric. This can be seen in the corresponding figure, where the vertical dimension has priority. TVD is L^1 with distance 2ϵ. The shape for Hamming distance keeps the index of a dimension constant, while it covers the full range of the other dimension. It makes sense only for ϵ values smaller than the array dimensionality, otherwise it covers the entire array. The derived shapes can be generalized to higher dimensionality than 2-D.

Comparison with array joins. The array similarity join operator is a generalization of the dimension:dimension join [19] to more complex join predicates. dimension:dimension join is defined as a natural join with equality predicates. This corresponds to a mapping function $\mathcal{M} : (D_1^\beta = D_1^\alpha, \ldots, D_d^\beta = D_d^\alpha)$ and a similarity shape $\sigma\left[D_1^\beta = 0; \ldots; D_d^\beta = 0\right]$—the most primitive shape in a similarity join. The shape in Example 2 corresponds to the predicate `k=j OR k=j+1`, or, equivalently, `k≥j AND k<j+2`. None of these can be expressed as dimension:dimension equi-join.

Instead of expressing these predicates in a relational form, we generalize the array APPLY operator since it allows for complex predicates to be expressed concisely—and more intuitively. There are two modifications we make to the original APPLY which has a single array argument. First, we introduce a mapping function that assigns a unique cell Ψ in β to every cell Υ in α. Second, the shape σ is applied to Ψ rather than Υ—the case in APPLY.

Comparison with similarity join. Similarity join is defined only for sets of points having the same dimensionality. The proposed operator can handle arrays with any dimensionality. The relationship between points is expressed based on a distance function in the case of similarity join. While applicable to arrays – see the examples in Figure 3 – a distance function does not encode neighborhood relationships directly. This can introduce problems due to the discrete nature of the array dimensions and also requires mapping to array indices. Moreover, some complex shapes, e.g., arrays with empty cells or non-symmetric arrays, cannot always be expressed as a norm. The mapping function and similarity shape array in Definition 1 – on the other hand – can encode virtually any relationship between array cells.

4. ARRAY SIMILARITY JOIN OPERATOR

In this section, we present the first array similarity join operator for a distributed array database proposed in the literature. We begin with a high-level description of the operator's inner-workings that identifies the main processing stages. Then, we delve into the details of each stage and introduce our technical contributions.

High-level approach. Given two input arrays α and β chunked over the database servers, the goal is to compute optimally the result of their similarity join $\tau = \alpha \bowtie_{\sigma,f}^{\mathcal{M}} \beta$ as defined by the mapping function $\mathcal{M}$, the shape σ, and the value function f. Abstractly, the computation of τ requires three stages. First, for each chunk $\Upsilon \in \alpha$, the chunks $\Psi \in \beta$ that are in $\sigma\left(\mathcal{M}(\Upsilon)\right)$ have to be identified. They can be located on one or multiple nodes in the cluster. This requires access only to the catalog metadata—considerably smaller in size. We assume the catalog can be stored in the memory of the coordinator and is properly indexed to efficiently identify the chunks in $\Psi \in \sigma\left(\mathcal{M}(\Upsilon)\right)$. Second, chunks Υ and Ψ have to be transferred to the same node. In this case, network bandwidth is the resource to optimize. A special situation arises when the join is performed at a node that stores one of the chunks. In the third stage, function f is applied to compute the value of the output chunk in τ. This stage is computationally intensive, but fully parallelizable across chunks. Optimality, i.e., minimum processing time, is achieved when all the nodes perform the same amount of work or, equivalently, they operate on data having similar size—the workload is balanced. The execution of stage 2 and 3 can be overlapped across different chunks, reducing the overall optimization problem to minimizing the most expensive stage. We argue that network bandwidth – as the single common resource in a shared-nothing distributed architecture – has to be the primary optimization target, while load-balanced execution is secondary. Moreover, optimizing disk access to local chunks is paramount when the join is processed where chunks are stored.

4.1 Query Execution

The array similarity join operator functions as follows. The user submits a query to the coordinator node in charge of managing the workers and the metadata catalog. The coordinator computes an optimal execution plan for the query and sends it to the worker nodes. The workers process their share of chunks concurrently and asynchronously, informing the coordinator only when they finish.

The execution of the array similarity join operator follows closely the structural join [41]. The most significant difference is that the node which computes the join between two chunks is determined in the query optimization process—it is not the node that stores the chunk in array α. As a result, although the schema of the output array τ is well-defined at query time, the chunking of τ is known only at query execution. In general, τ is chunked along the array α dimensions—not necessarily regular. The actual shape of the chunks is determined by the shape array σ in the query. A chunk covers the full range of the dimensions in array β. In order to impose a specific chunking on array τ, schema alignment from shuffle join [19] is required. This is a costly operation that incurs complete data repartitioning and full replication of arrays α and β for every chunk in τ—the dimensionality of τ is the sum of the dimensionality of α and β, respectively. Due to the high cost, by default, there is no schema alignment in array similarity join.

Join algorithm. The most general algorithm to join two chunks is *nested-loop join*. It considers all the cell pairs and keeps only those that are similar. Since the cells $\Psi \in \beta$ that join with a cell $\Upsilon \in \alpha$ are structurally connected by the shape array σ and their number is typically much smaller than the number of cells in a chunk, more efficient join algorithms can be applied to identify similar cells. When the shape σ is convex, i.e., does not have holes, a *multi-dimensional index*, e.g., kd-tree or generalizations of quad-tree, can be built over the chunk in β. To find the similar cells corresponding to a cell $\Upsilon \in \alpha$, a range query with the predicate $\sigma\left(\mathcal{M}(\Upsilon)\right)$ is executed over the index. As long as the index is queried a sufficiently large number of times, the construction cost is amortized. For the most general shape σ, *hash join* is likely the most efficient algorithm. A hash table is built over the chunk in β and is probed for each cell in $\sigma\left(\mathcal{M}(\Upsilon)\right)$ corresponding to Υ. The proposed array similarity join operator chooses the join algorithm for each chunk pair at runtime. If the size of the chunks are below a specified threshold, nested-loop join is applied. Otherwise, index join is chosen for convex shapes, while hash join for non-convex.

Overlap I/O and processing through multi-threading. The array similarity join operator overlaps I/O – disk and network – with join computation at chunk granularity. Network transfer and local disk I/O are each handled by a separate thread. The join between two chunks is executed in a separate worker thread. The operator is configured with a pool of worker threads, allocated based on the number of CPU cores available in the system. Thus, several joins between pairs of chunks can be executed concurrently. All the threads – I/O and workers – execute asynchronously and coordinate through message exchange. The extensive degree of parallelism puts additional pressure on the I/O components – network, in particular – and makes their optimization even more critical. Thus, minimizing data transfer and network congestion are the primary objectives in the optimization of the array similarity join operator.

4.2 Query Optimization

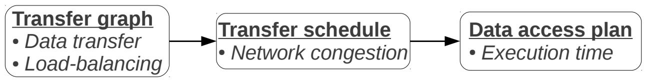
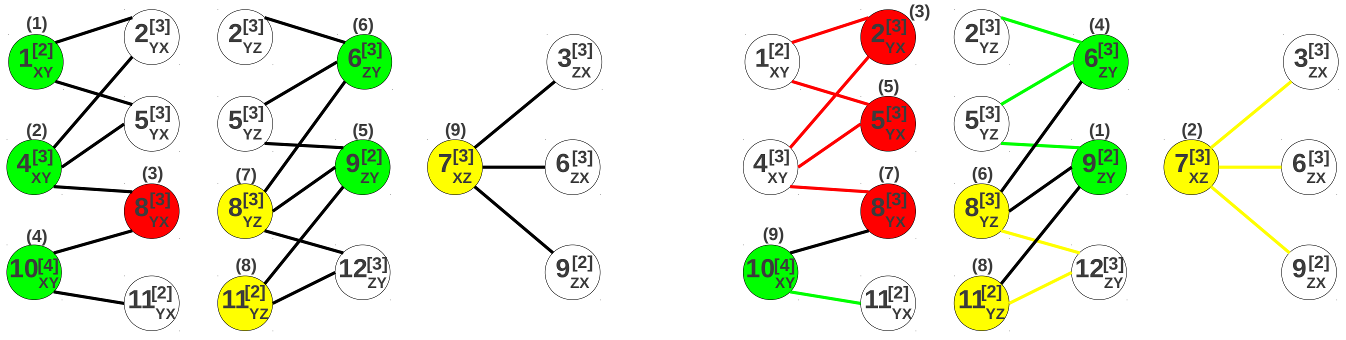


Figure 6: Query execution sub-plan for a node.

The coordinator is responsible for computing the optimal execution plan that minimizes the overall query processing time. The optimal plan is computed exclusively from the array similarity join query – the mapping function $\mathcal{M}$ and the shape array σ – and the catalog metadata which stores the location of the chunks and rele-



(a) (b)

Figure 5: Optimal transfer graph: (a) HVC and (b) BHVC.

vant chunk statistics, such as the number of non-empty cells. The global plan consists of a detailed execution sub-plan (Figure 6) for each of the worker nodes.

The *transfer graph* specifies the chunks and the nodes where to send them, and the chunks to be received and from which nodes, respectively. This can be modeled through a series of binary variables x_{ijk} that specify that chunk i is sent from node j to node k. i ranges over the total number of chunks in α and β. However, x_{ijk} is set for a single j – the node where the chunk resides – and for the values of k on which there are chunks that join with chunk i.

The *transfer schedule* gives the order in which to send/receive chunks. It has two components: 1) the order of the nodes and 2) the order of the chunks assigned to a node. The optimal transfer schedule minimizes network congestion by enforcing only pairs of nodes to exchange data at any instant in time—a node receives data from a single other node. The objective function for the network transfer time is:

$$\min\left\{\max_{ijk}\left\{y_{ijkt}\cdot t\cdot T_{ntwk}\right\}\right\} \tag{5}$$

where y_{ijkt} specifies that chunk i is sent from node j to node k at time instant t. t takes values from 1 to the total number of chunks. Time-based permutations of the transfer graph variables x_{ijk} are encoded in the constraints on y_{ijkt}, i.e., $\sum_t y_{ijkt} = x_{ijk}, \forall i,j,k$. T_{ntwk} is the time to transfer a chunk between two nodes. It can be determined experimentally from the characteristics of the network.

The *data access plan* specifies the order in which to access local chunks when joining them with remote chunks. Since a local chunk joins with several remote chunks and a remote chunk joins with several local chunks, reducing I/O traffic – the assumption is that not all the local chunks fit in memory – plays an important role in minimizing query processing time. The objective function for disk access time is:

$$\min\left\{\max_{k}\left\{\sum_{tij} y_{ijk(t-1)}\sum_{i'\in\Psi_i}\left(1-z_{i'j(t-1)}\right)\cdot T_{disk}\right\}\right\} \tag{6}$$

where z_{ijt} corresponds to local chunk i on node j being cached in memory at time instant t and T_{disk} is the time to load a chunk into memory. T_{disk} can be determined empirically.

Analytical cost model. The cost of an array similarity join query is the maximum between the network transfer time and the disk access time, i.e., $\max\{Eq.\ (5), Eq.\ (6)\}$, because these are overlapped. The cost does not include the join time. Since both disk and network are shared resources, while joins are processed in parallel across multiple threads, the computation is not the bottleneck, as long as sufficient threads are available in the system, e.g., we use 16 threads in the experiments. The cost also does not consider disk access time in network transfer time explicitly—before a chunk can be transferred from node j to node k, it has to be read from disk. This cost is accounted for in T_{ntwk}. In our model, data caching is employed only on the client side and exclusively for local chunks—it does not make sense to cache remote chunks since they are accessed once. Caching on the server side has to consider the access pattern from all the clients. Due to asynchronous execution, this is non-deterministic, thus, the benefits of optimization beyond standard LRU are unclear. We defer such an exploration to future work.

Challenges. The complete optimization formulation is given in Appendix A. Although it can be written as a linear integer program, finding the optimal network transfer schedule, i.e., Eq. (5), is known to be a combinatorially hard problem [33]. The disk access time objective, i.e., Eq. (6), is a quadratic integer optimization—an identically difficult problem. The number of variables in both these objectives is quadratic in the total number of chunks in the two arrays. When this number is in the order of thousands – the number of chunks in PTF is 200K – it is unfeasible to solve these optimizations in reasonable time since they contain billions of binary variables.

Solution. In these conditions, *our solution is to decompose the original optimization problem into a sequence of three separate sub-problems and solve them independently* (Figure 6). Notice, though, that solving each of the individual problems separately still poses significant challenges. First, we compute the optimal transfer graph, without considering the transfer schedule or the local data access plan. The graph guarantees minimal overall data transfer with a certain level of load-balancing. Second, the optimal transfer schedule is determined for the graph computed in the first stage. This schedule minimizes congestion over the network. The graph and the schedule target the network transfer time. Instead of computing which chunk goes to which node at which time instant, we first compute the assignment of chunks to nodes (transfer graph) and then the order only for those chunks assigned to a node (transfer schedule). Essentially, we introduce a new optimization problem over the x_{ijk} variables and take the result as a pre-conditioning for y_{ijkt}. Third, the local data access plan at each node is computed from the order in which remote chunks are received from other nodes. The data access plan targets the disk access time. By pre-conditioning Eq. (6) on y_{ijkt}, it becomes the well-known optimal cache replacement problem with a standard solution.

4.3 Transfer Graph Optimization

Consider x_{ijk} to be the variable obtained from y_{ijkt} by dropping index t. x_{ijk} specifies that chunk i is sent from node j to node k. With this simplification, minimizing the network transfer time becomes minimizing the amount of data each node has to send/receive. This can be written formally as:

$$\min \left\{ \max_j \sum_i \sum_k x_{ijk} \cdot s_i, \max_k \sum_i \sum_j x_{ijk} \cdot s_i \right\} \quad (7)$$

where s_i is the size of chunk i. While simpler, it turns out that solving this problem for a large number of chunks – while feasible – still takes minutes to hours. For example, CPLEX[6] takes more than 10 hours to compute the solution for joining two 3-D PTF arrays with 16 threads. This is not an acceptable optimization time since the actual execution takes less than 30 minutes (Section 5).

Our solution to solve this problem practically is to modify the objective to the total amount of data transfered over the network, i.e., $\min \sum_i \sum_j \sum_k x_{ijk} \cdot s_i$. This modified formulation can be mapped into a graph theoretical problem that can be solved efficiently. The main issue, though, is that it does not consider load-balancing—implicit in the original formulation in Eq. (7). We propose a novel graph-based algorithm for the modified formulation and show experimentally that is both close to optimal and provides load-balancing. Moreover, we show that the solutions to the original formulation and the proposed algorithm are similar.

The optimal transfer graph minimizes the overall data communicated over the network while guaranteeing a certain level of load-balancing. The graph encodes what chunks have to be communicated between nodes. Two chunks that have similar cells stored on separate nodes require network transfer. In order to determine these *chunk pairs* – in the worst case – the query optimizer has to compute $\sigma(\mathcal{M}(\Upsilon))$ for every cell in the α chunk. However, if the mapping function $\mathcal{M}$ preserves the order between cells along each of the dimensions, the chunks with potentially similar cells can be determined by computing $\sigma(\mathcal{M}(\Upsilon))$ only for the chunk corners.

Given potential chunk pairs and their node location, we build the transfer graph as follows. For each pair $\{\upsilon_i, \psi_j\}$, where υ is a chunk in α stored on node i and ψ a chunk in β stored on node j, we create two vertices υ_{ij} and ψ_{ji} connected by an edge. We assign a cost c proportional to the number of non-empty cells in the chunk to each vertex. c represents the amount of data that have to be transferred in order to evaluate similarity between cells in chunks υ_i and ψ_j. It is important to notice that multiple vertices can be created for a single chunk if it is paired with chunks stored on different nodes. For chunks stored on the same node, multiple edges are created. An assumption embedded in the transfer graph is that chunk pairs can be joined only on the node that contains one of the chunks. This assumption is valid in array databases, where all the nodes store a relatively equal number of chunks [14].

EXAMPLE 3. *Figure 5 depicts the transfer graph for the array similarity self-join query* `SELECT * INTO τ FROM A` A_1 `SIMILARITY JOIN A` A_2 `ON (`$A_1.i = A_2.i$ `AND` $A_1.j = A_2.j$`) WITH SHAPE σ`, *where A is the array in Figure 1 and* σ *is a* (3×3) *square centered on the cell. The notation for a vertex contains the chunk id – in white font in Figure 1 – the source and destination nodes, and the cost—in square brackets []. Node* $1_{XY}^{[2]}$ *corresponds to chunk 1 with 2 non-empty cells on node X being paired with chunks 2 and 5 on Y. Although the array has only 12 non-empty chunks, there are 18 vertices in the graph. This is because some chunks, e.g., chunk 2, are paired with chunks stored on multiple nodes, e.g., chunk 1 on X and chunk 6 on Z. There are 3 connected components in the graph—one for each pair of nodes.*

[6]http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/

4.3.1 *Vertex Cover*

The *minimum weighted vertex cover of the transfer graph* corresponds to the minimum amount of data that have to be transferred for a given similarity join query modeled as a transfer graph. The minimum weighted vertex cover of a graph[7] is defined as the set of vertices with minimal total weight, i.e., cost, that cover all the edges. An edge is covered if at least one of its vertices is part of the solution set. In the case of the transfer graph, a selected vertex υ_{ij} indicates that the corresponding chunk υ is transfered from node i to node j. For general graphs, this is an NP-complete problem. However, there exist heuristics and approximation algorithms that provide constant-factor approximation ratio solutions.

Clarkson's heuristic for vertex cover (HVC) [9]. In this greedy algorithm, vertices are sorted according to the ratio $c/\textit{degree}$, where *degree* is the number of incidental edges. This ratio represents the weight w of a vertex. At each iteration, the vertex with the minimum weight is selected and is eliminated from the graph together with all its incidental edges. This results in a reduction of the degree for all the adjacent vertices. Moreover, the weight of the selected vertex is subtracted from the cost of each adjacent vertex. HVC has a 2-approximation ratio. Figure 5a shows how HVC works for the transfer graph in Example 3. Vertices are selected independent of edges in this case. An important property of HVC is that it can be applied to each connected component separately.

4.3.2 *Load-Balanced Vertex Cover*

HVC does not consider load-balancing. In our case, this is a stringent requirement because, otherwise, a single or a few nodes have to do all the computation. This results in an increased execution time which, although not the primary optimization criterion, is still important. To cope with this problem, we introduce a new algorithm for minimum vertex cover that includes load-balancing in the vertex selection process. We start with the HVC algorithm and include penalties $p(\upsilon, \textit{recv}) = \textit{recv}(\upsilon.j) \cdot \upsilon.c \cdot |\textit{nodes}|$ and $q(\upsilon, \textit{send}) = \textit{send}(\upsilon.j) \cdot \upsilon.c \cdot |\textit{nodes}|$ in the weight w of each vertex $\upsilon_{ij}^{[c,w]}$. These penalties quantify the amount of data already assigned to the node corresponding to the destination (recv) and source (send) of the vertex, respectively. Constant r controls the penalty fraction assigned to each of these penalties. The modified objective gives higher priority to vertices on nodes that are assigned less data, even if their weight is larger. The pseudo-code for this new balanced heuristic vertex cover (BHVC) algorithm is presented in Algorithm 1. Line 3 selects the vertex with the minimum modified weight. The *for* loop in lines 4-8 updates the adjacent vertices, while lines 9 and 10 update the load of the destination/source node—initialized with 0.

Figure 5b depicts the output of BHVC on the transfer graph in Example 3. The main difference compared to HVC (Figure 5a) is that connected components are not considered independently. This is because each vertex assignment modifies the *recv* and *send* of the destination/source node. As a result, the assignment is done across components such that nodes always receive/send a similar amount of data. While HVC assigns 5 chunks to node Y and only 1 to X, with BHVC each node is assigned 3 chunks. However, BHVC

[7]https://en.wikipedia.org/wiki/Vertex_cover

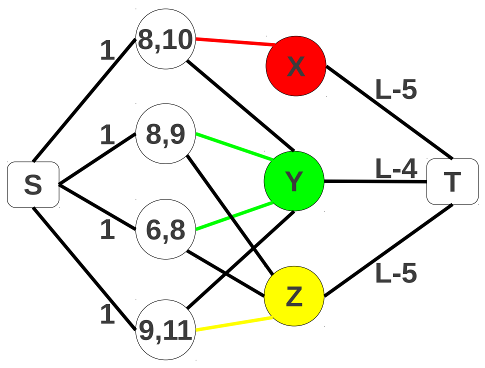


Figure 7: Assignment of chunk pairs to nodes. The non-colored vertices correspond to unassigned chunk pairs.

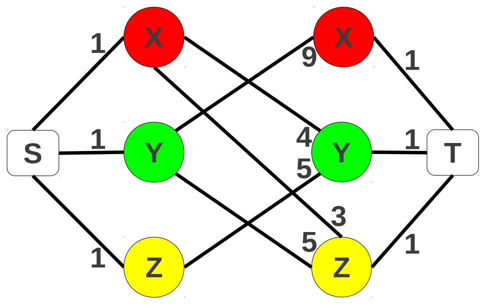


Figure 8: Transfer schedule network.

Algorithm 1 Balanced Heuristic Vertex Cover (BHVC)

```
Input: transfer graph $G(V, E)$
Output: balanced vertex cover $V' \subset V$
 1. $load(n) \leftarrow 0, \forall n \in nodes$
 2. while $E \neq \emptyset$ do
 3.     select vertex $v_{ij}^{[c,w]}$ s.t.
            $argmin_{v \in V} \left\{ \frac{v.w + r \cdot p(v, recv) + (1-r) \cdot q(v, send)}{degree(v)} \right\}$
 4.     for $(v, \psi) \in E$ do
 5.         $E \leftarrow E \setminus (v, \psi)$
 6.         $\psi.w \leftarrow \psi.w - \frac{v.w + r \cdot p(v, recv) + (1-r) \cdot q(v, send)}{degree(v)}$
 7.         $degree(\psi) \leftarrow degree(\psi) - 1$
 8.     end for
 9.     $recv(v.j) \leftarrow recv(v.j) + v.c$
10.     $send(v.j) \leftarrow send(v.j) + v.c$
11.     $V' \leftarrow V' \cup v$
12.     $V \leftarrow V \setminus v$
13. end while
```

increases the total amount of data transferred by 1 array cell—from 25 to 26. r is set to 0.5 in this example.

4.3.3 *Computation-to-Node Assignment*

The minimum vertex cover of the transfer graph determines only the data transfer path, i.e., what nodes each chunk is sent to. It does not include information on which node two chunks are joined on. However, in the case of edges covered by a single vertex, the node on which the join is executed can be easily determined—it is the destination node of the covered vertex. For example, in Figure 5b chunks 1 and 2 are joined on node X. All the edges for which the assignment is immediate are colored in the color of the assigned node. The join corresponding to edges covered by two vertices can be computed on either of the two nodes storing the involved chunks.

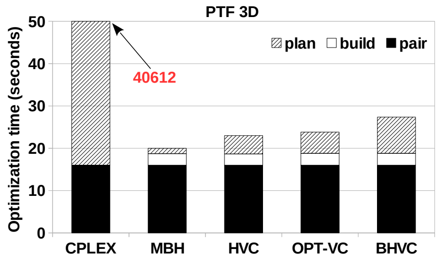
In order to maintain load-balancing, a clever assignment strategy has to be devised. For this, we resort to the max-flow algorithm. We build a network consisting of a vertex for each unassigned chunk pair in the transfer graph and vertices for each of the processing nodes. Network edges connect only unassigned chunk pairs to nodes. These edges are assigned infinite capacity. A source vertex S has edges with capacity 1 to each chunk pair. A sink vertex T receives edges from each of the processing nodes. The capacity of these edges is to be determined by the assignment policy and it includes the already assigned chunk pairs. For example, the network in Figure 7 corresponds to the BHVC graph in Figure 5b in which 5 chunk pairs are already assigned to node X, 4 to node Y, and 5 to node Z, respectively. We want to assign the remaining 4 pairs, e.g., $(8, 10)$, $(8, 9)$, $(6, 8)$, and $(9, 11)$, such that the workload remains balanced. Let L be the maximum number of pairs assigned to a node in the final assignment. The minimum value of L for which a flow of size equal to the number of unassigned chunk pairs in this network exists, corresponds to a balanced assignment. This value can be found with a simple binary search over the interval $[1$, total number of chunk pairs$]$. For the network in Figure 7, L takes value 6 and the assignment of chunk pairs to nodes is color-coded. 6 chunk pairs have to be joined on each node.

4.4 Transfer Schedule Optimization

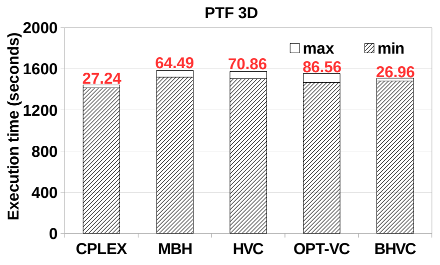
Given the solution to the transfer graph optimization, i.e., the chunks that have to be transferred between each two nodes x_{ijk}, the transfer schedule computes the order in which these chunks are sent/received, i.e., y_{ijkt}. For each node, we divide the transfer schedule into two components: 1) the order of the nodes and 2) the order of the chunks assigned to a node. We enforce that there is only node-to-node communication at any instant in time in order to minimizes network congestion.

Node order. We build a network in which there are two vertex instances corresponding to each node—one for sending and one for receiving. For each node pair that requires data transfer, an edge with capacity given by the total number of cells in the chunks to be transferred between them is added. The source and sink vertices are connected with each sending/receiving node instance by edges with capacity 1 (Figure 8). A max-flow in this network guarantees only node-to-node communication. At each step, the flown-through edge between nodes with the lowest capacity is removed and the process is repeated for the remaining graph. We stop when no edges exist between vertices corresponding to nodes. Figure 8 depicts the network for our running example. There is no edge between nodes Z and X since no chunks are sent from Z to X. A possible transfer schedule is: $\{(X \rightarrow Z, Y \rightarrow X, Z \rightarrow Y) : 3; (Y \rightarrow X, Z \rightarrow Y) : 2; (X \rightarrow Y, Y \rightarrow X) : 4; (Y \rightarrow Z) : 5\}$.

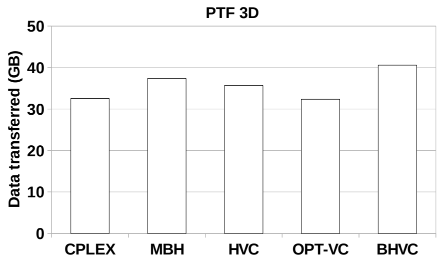
Chunk order. Given a set of chunks that have to be sent between two nodes, the question that arises immediately is in what order to do this? Consider the chunks sent from node Y to node X in Figure 5b. Chunk 2 and 5 have to be joined with chunk 1 and 4, while chunk 8 has to be joined with chunk 4 and 10, respectively. Since chunks on X have to be retrieved from disk, in the optimal case, each chunk is read only once, independent of the number of remote chunks it has to be joined with. Given that only a limited number of chunks can be cached in memory, it is impor-



(a)



(b)



(c)

Figure 9: Transfer graph evaluation on PTF 3D query: (a) optimization time, (b) execution time, and (c) transferred data.

tant to use a chunk entirely once it is read. This can be achieved only if all the remote chunks that require the same local chunk are received (almost) contiguously. Our optimization framework does not guarantee this because chunks from different nodes are treated separately. However, we devise a solution that reorders the remote chunks from a node such that the number of local chunks they share is maximized. For a given remote chunk, the main idea is to choose another chunk from the same node that shares the largest number of local chunks. This can be found from the transfer graph. For example, the order 2, 5, 8 requires fewer disk accesses than 2, 8, 5 if only two local chunks can be stored in memory.

4.5 Data Access Plan Optimization

The last stage in the optimization process is to determine what local chunk to expel from memory. This problem arises whenever a remote chunk is joined with a local chunk that is not cached and the memory is full. The solution to this problem can be computed optimally since the transfer schedule specifies the incoming chunk order. Essentially, the chunk that will be used in the furthest future has to be expelled from memory. This minimizes the data access time from secondary storage, thus, the overall execution time. Following our running example, when chunk 8 is received at node X, local chunk 1 has to be removed from memory since chunk 4 has to be joined with 8.

5. EXPERIMENTAL EVALUATION

The objective of the experimental evaluation is to investigate the overall performance – as well as the impact of optimizations – of the proposed operator for two types of similarity queries over the PTF catalog. These are real queries executed in the PTF pipeline for transient detection. We use the LinkedGeoData[8] dataset in order to confirm the behavior of the proposed operator. Specifically, the experiments are targeted to answer the following questions:

- What is the effect of query optimization on execution time, amount of data transferred, network congestion, and data access plan at a node?
- What overhead is incurred by the optimization?
- How does the proposed BHVC algorithm compare against the original optimization formulation and HVC?
- What is the time distribution for a processing thread?
- How scalable is the array similarity join operator and how does it compare against state-of-the-art solutions?

Implementation. We implement the array similarity join operator as a `C++11` distributed multi-thread prototype on top of an array storage manager derived from ArrayStore [41]. The catalog is replicated at all the nodes in the cluster. The operator manages a pool of worker threads equal to the number of CPU cores in the system. Each worker is invoked with a pair of chunks that it has to join. The worker makes requests to the local and remote array storage managers. Once the chunks are received, the join is processed. This happens concurrently across all the workers. The optimization is executed at the coordinator as a separate process and the plans are distributed to the nodes. The code contains special function calls 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 four 1 TB 7200 RPM SAS HDDs accessed independently. In the experiments, we use a single disk supporting buffered read rates of 120 MB/second on average per node. The number of worker threads is set to 16—the number of cores. Ubuntu 14.04.3 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 network bandwidth on a link is similar to the disk bandwidth.

Methodology. We perform all experiments at least 3 times and report the average value as the result. We always enforce data to be read from disk, i.e., cold caches.

Data. We use two real datasets in our experiments. The *PTF catalog* consists of 1 billion time-stamped objects represented in the celestial 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 50,0000, 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. Due to the skew, there is considerable variance between the number of non-empty cells across chunks. However, the optimization algorithm takes this into account when assigning chunk pairs to nodes. Depending on how many attributes are stored for an object, the size of the catalog varies between 12 and 70 GB.

LinkedGeoData stores geo-spatial data used in OpenStreetMap[9]. 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

[8]http://linkedgeodata.org

[9]www.openstreetmap.org

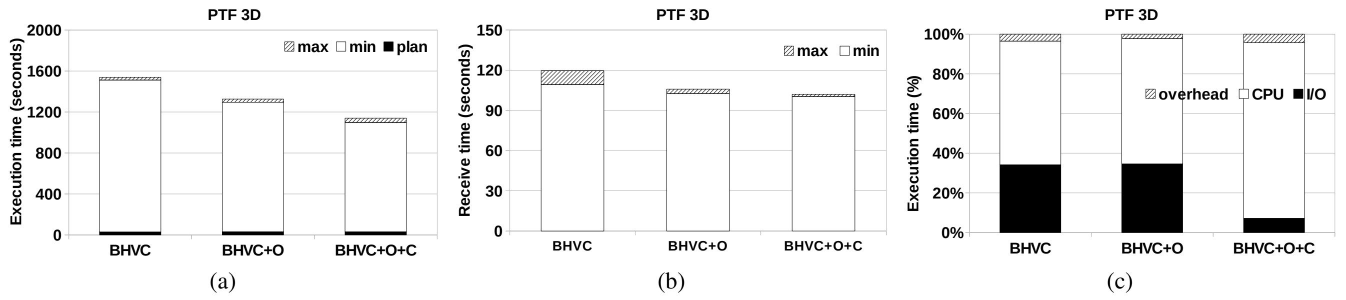


Figure 10: Query optimization evaluation on PTF 3D query: (a) execution time, (b) receiving time, and (c) thread time distribution.

original point using a Gaussian distribution with $\mu = 0$ and $\sigma = 10$ miles [36]. In array format, this corresponds to:

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

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

Queries. In order to execute a real similarity join query, we create two separate instances of the datasets. They are treated as two independent arrays, even though special optimizations specific to self-join queries can be applied. We execute two types of queries – 2-D and 3-D – on PTF and 2-D on GEO. In PTF 2-D queries, the similarity shape is defined only over the (ra, dec) dimensions. The time is considered to be 0. This corresponds to finding objects within 10 arcseconds of each other in the same image. The similarity shape considers the 4 cells having L^1-norm of 1 from the origin cell. In 3-D queries, the time is part of the similarity shape, which includes the 26 cells having L^∞-norm of 1 from the center. The GEO 2-D query finds POIs within 1 mile of each other. The corresponding shape array is the L^2-norm equal to 4 (Figure 3). Since the results for all the queries follow similar patterns, we include below only the results for the PTF 3-D query. The results for the other two queries are in Appendix C.

5.1 Transfer Graph Evaluation

We evaluate the impact of the proposed BHVC algorithm on the overall execution time and amount of data transferred. We take as baseline the solution to the original network transfer time optimization formulation in Eq. (5). We solve this formulation using CPLEX with 16 threads and compare its optimization time to that of BHVC. In order to verify the ability of BHVC to provide load-balancing, we include in the comparison three algorithms for the vertex cover problem that minimize only the total amount of data transferred. The first algorithm is HVC (see Section 4.3.1). The minimum-bandwidth heuristic (MBH) [19] is a greedy algorithm that always transfers the smaller chunk in a pair to the larger one. This is done iteratively until all the edges are covered. Recall that vertices correspond to chunks and edges to chunk pairs in the transfer graph. The third algorithm is OPT-VC [18]. It computes the minimum weighted vertex cover of a bipartite graph using network max-flow. The transfer graph is bipartite by construction.

Optimization time. The time to compute the transfer graph is depicted in Figure 9a. The time to determine the chunk pairs is common across all the methods. In CPLEX, these are used to construct the linear program. The graph structure is built from the pairs in all the other methods. Even though it uses 16 threads, CPLEX cannot solve the linear program in less than 10 hours. This is because the number of chunk pairs in the PTF 3-D query is in the order of tens of thousands. The simple heuristics MBH and HVC are the fastest. BHVC takes a few seconds more than these methods – and OPT-VC – because it has to handle load-balancing, in addition to only minimizing the transferred data. The same trend is observed for the other two queries (Figure 14a and 15a).

Execution time. Figure 9b depicts query completion time both for the fastest as well as the slowest node in the cluster. The difference between the two is included in the figure. The optimization time is not included—only the transfer graph optimization is activated. As expected, the CPLEX solution has the fastest execution time. BHVC comes in a close second, while the other algorithms perform almost identical. When it comes to load-balancing, BHVC clearly shows its improvement over the algorithms that consider only the amount of data transferred. BHVC performs even better than CPLEX since minimizing the maximum data sent/received does not translate immediately into load-balancing.

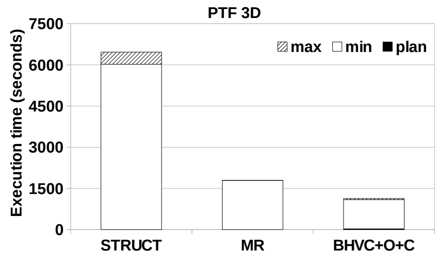
Data transferred. OPT-VC is guaranteed to minimize the overall data transferred over the network. Figure 9c confirms this. The reason BHVC transfers the most data is because its goal is to obtain similar execution times across all the nodes. This requires additional data movement from the slower nodes to the faster ones in order to balance the overall load. We observe similar trends for the other two queries (Figure 14c and 15c).

5.2 Query Optimization Evaluation

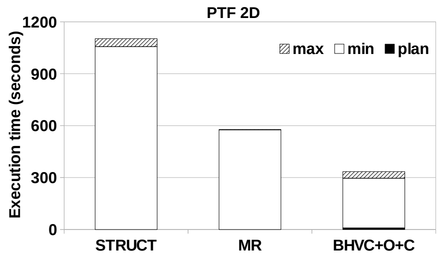
We measure the effect each stage in the query optimization process has on execution time, receiving time as a surrogate for network congestion, and thread execution time distribution. We take as baseline the transfer graph generated by the BHVC algorithm and quantify the additional improvements brought by node and chunk reordering in the transfer schedule (O) and caching in local chunk accessing (C).

Execution time. Figure 10a depicts the impact on execution time of each stage in the optimization process. While the optimization time stays constant, schedule reordering and caching improve the execution time considerably. Schedule reordering enforces that only pairs of nodes communicate at every time instant. This reduces the traffic at each node and streamlines the communication. Intelligent caching at client reduces the number of requests made to the local storage manager. We also observe that these do not have a negative effect on load-balancing for any of the queries.

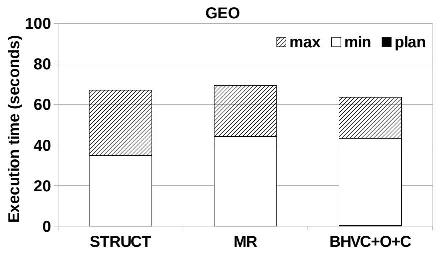
Receive time. The effect of schedule reordering on network congestion is depicted in Figure 10b. Since we cannot measure congestion directly, we measure the receive time at each node. We observe a significant reduction in receiving time when reordering is enabled across all the queries. Caching does not influence receiving time in any meaningful way.



(a)

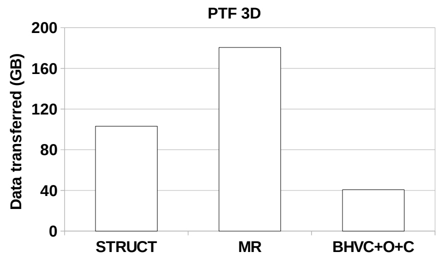


(b)

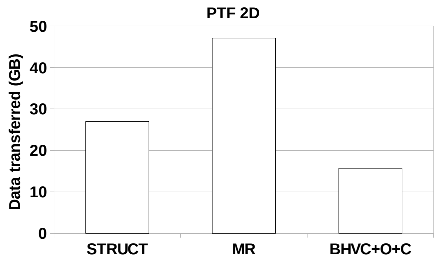


(c)

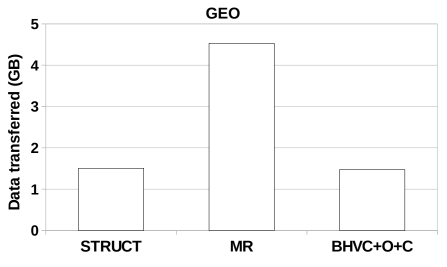
Figure 11: Algorithm execution time comparison: (a) PTF 3-D, (b) PTF 2-D, (c) GEO.



(a)



(b)



(c)

Figure 12: Algorithm data transferred comparison: (a) PTF 3-D, (b) PTF 2-D, (c) GEO.

Thread time distribution. Figure 10c shows the relative time distribution inside a worker thread. This is a good indicator of where the time is spent and where the bottlenecks are. Ideally, all the time should be spent in CPU processing. This is closely realized only when caching is active. The network traffic – not included in the figure – remains the bottleneck even in this situation. Schedule reordering plays a minimal role on thread time distribution.

5.3 Comparison with Related Algorithms

Since we are the first to introduce array similarity join, it is difficult to find appropriate candidate algorithms for direct comparison. In this situation, we adapt two classes of distributed algorithms to array similarity join. The first class is array join, while the second is relational similarity join. A complete description of these algorithms is given in Appendix B.

Structural join (STRUCT) [41] is the standard algorithm to implement dimension:dimension array join. It iterates over chunks of the outer array α. For each chunk, it looks-up the corresponding chunks in the inner array β, retrieves them all, and joins the outer chunk with each of the inner chunks in turn. The algorithm is applicable to array similarity join if the mapping function $\mathcal{M}$ preserves the adjacency relationship between cells. We implement structural join by having each of the nodes executing this algorithm concurrently on their share of the data. Moreover, we process multiple chunk pairs in parallel by assigning each of them to a separate worker from the thread pool.

MapReduce (MR) similarity join algorithms over relational multi-dimensional data [40, 37, 17, 36] are immediate instantiations of MapReduce join [4]. The most important challenge faced by these algorithms is how to choose the join units such that the amount of data that have to be replicated – thus, transferred over the network – is minimized. In the case of arrays, data are already chunked, thus, the join units are determined. The mapping of join units to reducers is done with a random hash function that partitions the join units uniformly. In our implementation, each reducer, i.e., worker from the thread pool, transfers its allocated join units and executes the join concurrently with the other reducers.

Figure 11 depicts the execution time of the three algorithms for the three queries considered. Including optimization time, BHVC is always the fastest—by as much as a factor of 2 over MR in the case of PTF 2-D. Even though it moves considerably more data, MR is faster than STRUCT. The reason is the better distribution of the workload across nodes. BHVC also transfers the least amount of data—a factor of 2.5 less than STRUCT in the case of PTF 3-D. These results prove that the proposed optimizations provide a significant improvement over state-of-the-art relational algorithms modified to work on array data.

5.4 Scalability

The speedup of the array similarity join operator is depicted in Figure 13 for the PTF 3-D query when executed over 2, 4, 6, and 8 nodes. The reason for the sub-linear speedup is that the bottleneck of the entire process is data access – disk or network – not the computation. However, a speedup of 3.64 out of 4 is obtained for 8 nodes. Due to less computation, the speedup for the other queries is worse – only around 2.6 – and we do not include it here.

5.5 Discussion

Based on the results presented above, we can answer the questions driving the experimental evaluation. The transfer graph optimization formulation cannot be solved by CPLEX in acceptable time—it takes more than the overall query execution. All the solutions that minimize the data transferred are scalable. However, only BHVC achieves load-balancing—at a small increase in op-

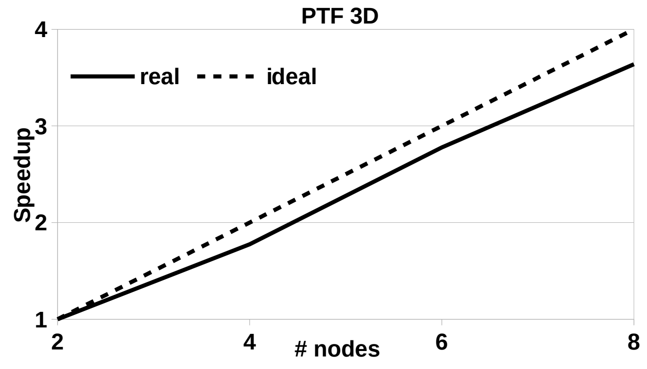


Figure 13: Speedup for the PTF 3-D query.

timization time and data transferred. The transfer schedule and caching reduce network congestion and increase the CPU utilization, respectively. They also improve the overall execution time by a significant margin. The proposed array similarity join operator is clearly more efficient – by at least a factor of 2 – than relational similarity operators modified to work with array data. As long as the query requires significant computation, the operator achieves close-to-linear speedup.

6. RELATED WORK

In this section, we discuss relevant work from array joins, relational similarity join, parallel similarity join, and spatial joins.

Array joins. The `APPLY` operator is part of original array algebra specifications [2, 26]. We define array similarity join as a generalization of the `APPLY` operator to two arrays. Positional array equi-joins are introduced in the first releases of the array database SciDB [30]. They are evaluated in the context of different chunking strategies in [41], where structural join is introduced. A complete formalization of array equi-joins and the shuffle join algorithm are given in [19], while a graph formulation is introduced in [3]. Array similarity join is not considered in any of the previous work. The proposed operator bears similarities to shuffle join [19] in allocating join units/chunk pairs to nodes based on an optimization process. However, the allocation is considerably more difficult in the case of similarity join. Moreover, our cost model considers disk access and overlapped network, I/O, and processing.

Relational similarity join. There is a significant amount of work on similarity join processing in the relational database literature [23, 39, 5, 38, 22, 21, 13, 20]. These algorithms can be separated into two categories—index-based and all-pairs [46]. ϵ-kdB tree [39] is the first high-dimensional index targeted at similarity join. It modifies the splitting criterion of kdB tree to take ϵ into account. In general, a new index has to be built for every similarity query. A parallel version of ϵ-kdB tree that requires complete data repartitioning is introduced in [38]. The ϵ Grid Ordering (EGO) family [5, 22, 21] of algorithms is a divide and conquer grid-based method. Points are mapped into grid cells and sorted before a recursive join is applied. This is done for every query. The effectiveness of the EGO-family of algorithms is heavily dependent upon the heuristic to prune the non-joinable sequences [22, 21, 13]. The Quickjoin algorithm [20] recursively partitions the data until each partition contains a few objects, at which point a nested-loop join is used. The data can be partitioned using a variety of techniques, such as ball partitioning or generalized hyperplane partitioning. Data at the border are replicated. While Quickjoin is perfect for self-join similarity queries, it encounters problems when handling join queries. The solution given in [20] is to put the points from the two sets together, execute Quickjoin on the merged dataset, but return only the results in which there is a point from each dataset. This is hardly efficient. Space filling curves algorithms such as ZC and MSJ [23], although simple, require space transformations and heavy preprocessing. These make them hard to parallelize.

Parallel similarity join. There has been considerable interest in parallelizing similarity join algorithms using the MapReduce framework [44, 29, 15, 27, 12]. MRSimJoin [40] is an extension of QuickJoin [20] to MapReduce. It is a divide-and-conquer algorithm that partitions the points iteratively into clusters until an in-memory similarity join algorithm can be executed on pair clusters. It uses pivot-based partitioning with random pivots extracted from one of the relations. While repartitioning is part of the MapReduce job, there is an unbounded number of such jobs because the exact partitioning cannot be controlled. ClusterJoin [36] introduces a set of bisector-based distance filters [7, 46] into MRSimJoin to generate more accurate partitions. MR-DSJ [37] and PHiDJ [17] are similarity self-join grid-based algorithms that extend EGO [5] to MapReduce. As is the case with all MapReduce solutions, complete data repartitioning is automatic. This is not necessary for arrays which are already chunked. The proposed array similarity join operator takes advantage of chunking to schedule join processing and to reduce data transfer. It is not clear how set [44], multiset [27], social image [48], and string [12] data can be represented as arrays in order to apply the array similarity join operator. These use different metrics and have special similarity join operators. We plan to explore these topics in future work.

Spatial join. In spatial join, pairs of overlapping objects have to be computed. This corresponds to cell overlapping in array similarity, i.e., L^0 norm. A standard approach is to decompose the space into grids and assign each object to all the grids it overlaps with [31]. The join is computed for each grid and duplicates are eliminated. Thermal-Join [42] introduces a non-uniform space partitioning that allows for overlapped objects to be identified without any computation. This approach does not work for array data unless a chunk contains a single cell. In general, while non-similar cells can be pruned away, similar cells require computation.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we introduce the first array similarity join operator for a distributed array database. Unlike previous work on relational data, we define similarity based on a shape array instead of a distance function. This novel formulation takes into consideration the discrete nature of array data and supports asymmetric similarity measures. The proposed operator minimizes the overall data transfer and network congestion while providing load-balancing across the nodes that store data, but without completely repartitioning and replicating the arrays. Moreover, the array operator overlaps disk, network, and join computation in a multi-thread pipelined architecture. We model the query optimization of array similarity join as a vertex cover problem and introduce efficient algorithms to find optimal execution plans. We evaluate experimentally the proposed array similarity join operator and compare it against existing solutions on the PTF catalog consisting of 1 billion celestial objects. The results confirm the efficacy of the optimizations in reducing the overall data transfer as well as the execution time—by a factor of 2 or more. In future work, we plan to consider plans that assign computation to nodes that do not store the chunks in a join unit. While this makes the optimization harder, solutions similar to TrackJoin [32] can be devised. We also plan to consider other measures than L norms, e.g., Jaccard coefficient and cosine similarity.

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 and Florin Rusu have done part of this work while at Lawrence Berkeley Laboratory. We want to thank Peter Nugent from LBNL for providing us the PTF data used in the experiments. We also want to thank the anonymous reviewers for their insightful comments that improved the quality of the paper significantly.

8. REFERENCES

[1] 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.

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

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

[4] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A Comparison of Join Algorithms for Log Processing in MaPreduce. In *SIGMOD 2010*.

[5] C. Bohm, B. Braunmuller, F. Krebs, and H.-P. Kriegel. Epsilon Grid Order: An Algorithm for the Similarity Join on Massive High-Dimensional Data. In *SIGMOD 2001*.

[6] 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*.

[7] S. Chaudhuri, V. Ganti, and R. Kaushik. A Primitive Operator for Similarity Joins in Data Cleaning. In *ICDE 2006*.

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

[9] K. Clarkson. A Modification of the Greedy Algorithm for Vertex Cover. *IPL*, 16(1), 1983.

[10] 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/.

[11] D. J. DeWitt et al. The Gamma Database Machine Project. *IEEE Data Eng. Bull.*, 2(1):44–62, 1990.

[12] D. Deng, G. Li, S. Hao, J. Wang, and J. Feng. MassJoin: A MapReduce-based Algorithm for String Similarity Joins. In *ICDE 2013*.

[13] J.-P. Dittrich and B. Seeger. GESS: A Scalable Similarity-Join Algorithm for Mining Large Data Sets in High Dimensional Spaces. In *KDD 2001*.

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

[15] F. Afrati et al. Fuzzy Joins using MapReduce. In *ICDE 2012*.

[16] F. Rusu and Y. Cheng. A Survey on Array Storage, Query Languages, and Systems. http://arxiv.org/abs/1302.0103, 2013.

[17] S. Fries, B. Boden, G. Stepien, and T. Seidl. PHiDJ: Parallel Similarity Self-Join for High-Dimensional Vector Data with MapReduce. In *ICDE 2014*.

[18] L. S. Goddard. *Mathematical Techniques of Operational Research*. 2014.

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

[20] E. H. Jacox and H. Samet. Metric Space Similarity Joins. *ACM Transactions on Database Systems (TODS)*, 2007.

[21] D. V. Kalashnikov. Super-EGO: Fast Multi-dimensional Similarity Join. *VLDB Journal*, 4(2):561–585, 2013.

[22] D. V. Kalashnikov and S. Prabhakar. Fast Similarity Join for Multi-Dimensional Data. *Information Systems*, 32:160–177, 2007.

[23] N. Koudas and C. Sevcik. High Dimensional Similarity Joins: Algorithms and Performance Evaluation. In *ICDE 1998*.

[24] K.-T. Lim, D. Maier, J. Becla, M. Kersten, Y. Zhang, and M. Stonebraker. Array QL Syntax. http://www.xldb.org/wp-content/uploads/2012/09/ArrayQL-Draft-4.pdf. [Online; January 2013].

[25] D. Maier. ArrayQL Algebra: version 3. http://www.xldb.org/wp-content/uploads/2012/09/ArrayQL_Algebra_v3+.pdf. [Online; January 2013].

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

[27] A. Metwally and C. Faloutsos. V-SMART-Join: A Scalable MapReduce Framework for All-Pair Similarity Joins of Multisets and Vectors. *PVLDB*, 5, 2012.

[28] M. A. Nieto-Santisteban, A. R. Thakar, and A. S. Szalay. Cross-Matching Very Large Datasets. In *National Science and Technology Council (NSTC) NASA 2006*.

[29] A. Okcan and M. Riedewald. Processing Theta-Joins using MapReduce. In *SIGMOD 2011*.

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

[31] J. Patel and D. DeWitt. Partition Based Spatial-Merge Join. In *SIGMOD 1996*.

[32] O. Polychroniou, R. Sen, and K. A. Ross. Track Join: Distributed Joins with Minimal Network Traffic. In *SIGMOD 2014*.

[33] W. Rodiger, T. Muhlbauer, P. Unterbrunner, A. Reiser, A. Kemper, and T. Neumann. Locality-Sensitive Operators for Parallel Main-Memory Database Clusters. In *ICDE 2014*.

[34] D. Rohde, M. Gallagher, and M. Drinkwater. Astronomical Catalog Matching as a Mixture-Model Problem. In *AIP 2012*.

[35] F. Rusu, P. Nugent, and J. K. Wu. Implementing the Palomar Transient Factory Real-Time Detection Pipeline in GLADE: Results and Observations. In *DNIS 2014*.

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

[37] T. Seidl, S. Fries, and B. Boden. MR-DSJ: Distance-based Self-Join for Large-Scale Vector Data Analysis with MapReduce. In *BTW 2013*.

[38] J. C. Shafer and R. Agrawal. Parallel Algorithms for High-dimensional Proximity Joins. In *VLDB 1997*.

[39] K. Shim, R. Srikant, and R. Agrawal. High-Dimensional Similarity Joins. In *ICDE 1997*.

[40] Y. N. Silva, J. M. Reed, and L. M. Tsosie. Mapreduce-based Similarity Join for Metric Spaces. In *Cloud-I 2012*.

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

[42] F. Tauheedy, T. Heinis, and A. Ailamaki. THERMAL-JOIN: A Scalable Spatial Join for Dynamic Workloads. In *SIGMOD 2015*.

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

[44] R. Vernica, M. Carey, and C. Li. Efficient Parallel Set-Similarity Joins using MapReduce. In *SIGMOD 2010*.

[45] S. Wang, Y. Zhao, Q. Luo, C. Wu, and Y. Xu. Accelerating In-Memory Cross Match of Astronomical Catalogs. In *eScience 2013*.

[46] Y. Wang, A. Metwally, and S. Parthasarathy. Scalable All-Pairs Similarity Search in Metric Spaces. In *KDD 2013*.

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

[48] Y. Zhuang, N. Jiang, Z.-A. Wu, J. Cao, and C. Ju. Efficient Batch Similarity Join Processing of Social Images based on Arbitrary Features. In *WWW 2015*.

APPENDIX

A. OPTIMIZATION FORMULATION

In this section, we provide the complete formulation for array similarity join optimization. We express the optimization as a constrained mixed integer program (MIP). In addition to the objective function presented in the cost model (Section 4), we include the relevant constraints. The binary variables are depicted in Table 1.

Variable	Significance
x_{ijk}	chunk i is sent from node j to node k
y_{ijkt}	chunk i is sent from node j to node k at time t
z_{ijt}	chunk i on node j is cached at time t

Table 1: MIP binary variables.

The cost of array similarity join has to be minimized:

$$\min \Big\{ \max_{ijk} \{y_{ijkt} \cdot t \cdot T_{\mathit{ntwk}}\}, \quad \max_{k} \left\{ \sum_{tij} y_{ijk(t-1)} \sum_{i' \in \Psi_i} \left(1 - z_{i'j(t-1)}\right) \cdot T_{\mathit{disk}} \right\} \Big\} \tag{8}$$

under the following set of constraints:

$$\begin{aligned}
C_1 &: \; x_{ijk} + x_{i'kj} = 1; \forall i, \forall i' \in \Psi_i, p(i) = j, p(i') = k \\
C_2 &: \; \sum_{t} y_{ijkt} = x_{ijk}; \forall i, \forall j, \forall k \\
C_3 &: \; \sum_{ijk} y_{ijkt} = 1; \forall t \\
C_4 &: \; \sum_{ik} y_{ijkt} = 1; \forall t, \forall j \\
C_5 &: \; \sum_{ij} y_{ijkt} = 1; \forall t, \forall k \\
C_6 &: \; \sum_{i} z_{ijt} \leq B; \forall t, \forall j \\
C_7 &: \; z_{i'jt} \leq z_{i'j(t-1)} + y_{ikjt}; \forall t, \forall k, \forall j, \forall i' \in \Psi_i
\end{aligned} \tag{9}$$

C_1 guarantees that every join unit is processed at a node that stores one of the chunks. C_2 connects variables x and y such that only chunks that are transferred are assigned a permutation position. C_3 ensures that a chunk is transferred to a remote node only once. C_4 guarantees that a node can send at most one chunk per time instant, while C_5 corresponds to receiving at most a chunk per time instant. C_6 is a bounding constraint limiting the size of the cache at a node. Finally, C_7 enforces that a chunk is cached only when it is required in a join at time t.

There are three parameters in this MIP optimization. T_{ntwk} is the time to transfer a chunk between two nodes. T_{disk} is the time to load a chunk into memory. B is the cache size. Their values can be determined based on an empirical calibration process.

B. BASELINE ALGORITHMS

In this section, we provide a detailed description of the state-of-the-art algorithms used for comparison with the proposed array similarity join operator. They are immediate extensions of algorithms for array join and similarity join, respectively. We discuss the advantages/disadvantages of each algorithm and provide a thorough comparison along several dimensions.

B.1 Structural Join

The standard algorithm to implement dimension:dimension, or structural, equi-join is a special form of nested-loop join operating at chunk level (see Algorithm 3 in [41]). The join iterates over chunks of the outer array α. For each chunk, it looks-up the corresponding chunks in the inner array β, retrieves them all, and joins the outer chunk with each of the inner chunks in turn. The join between two chunks is itself implemented as nested-loops iterating over chunk cells. If the cells in the chunks are sorted according to dimensions, the optimal merge join algorithm can be executed instead. The algorithm is immediately applicable to the array similarity join operator if the mapping function $\mathcal{M}$ preserves the adjacency relationship between cells. Identity function is one such example. Otherwise, cells have to be treated independently.

The structural join algorithm can be readily implemented in a distributed array database. Once the cells in $\sigma\left(\mathcal{M}(\Upsilon)\right)$ are determined for an entire chunk – or all the chunks – the node identifies their location by querying the catalog on the coordinator. A message is sent to the corresponding node for every chunk and, when the chunk is received, the output cells in τ are computed. While the degree of parallelism across nodes is maximized, there are several problems with such an asynchronous decentralized approach. Although each node aims to minimize the amount of transferred data – it behaves locally optimal – there is no guarantee that the overall data are minimized. In fact, this is very unlikely since nodes do not coordinate at all. As a consequence, the actual data transfer can be severely imbalanced due to the contention for network bandwidth. In the extreme case, all the nodes in the cluster send/receive data to/from the same node. Load-balancing beyond what is achievable with a uniform chunk distribution to nodes is not considered at all in structural join.

B.2 MapReduce Similarity Join

In the structural join algorithm, the computation is executed exclusively at nodes storing chunks from the outer array α. Unless these chunks are distributed across the entire cluster, there will be nodes that do not participate in join processing. Moreover, if chunk distribution is not even, there will be load imbalance. MapReduce join [4] – as a direct extension of distributed Grace hash join [11] – guarantees that all the nodes in the cluster participate in join processing. Load-balancing is typically enforced at runtime through dynamic assignment of work – chunks to be joined – to nodes. The tradeoff to achieve these two goals is network traffic. MapReduce join is far from network-optimal because it transfers almost the full size of both arrays over the network [32]. Using pre-determined hash functions limits the probability that a hashed tuple will not be transferred over the network to $1/N$ on N nodes.

MapReduce join works as follows. The result array is divided into logical non-overlapping chunks, i.e., join units. These are computed from the schema of the result array—specified by the user, or inferred by the system in some restricted situations. Join units are computed at the coordinator and sent to all the nodes storing data from arrays α and β, or they are encoded directly into the Map hash function. Each node partitions the cells it stores over the join units. This is done concurrently across all the nodes. Cells Υ in α are assigned to a single join unit. Cells Ψ in β are assigned either to a single join unit – for equi-join – or they can be replicated in several units—for similarity join. The data alignment phase, i.e., shuffling, transmits all the partitions belonging to the same join unit to a single node for the computation of function f. In the case of distributed hash join, the assignment of join units to nodes is static and uniform. In MapReduce, tasks are assigned dynamically at runtime to better adapt to the processing capacity of the nodes, resulting in more adaptive load-balancing. The computation of f can start only after all the partitions are received.

MapReduce similarity join algorithms over multi-dimensional data [40, 37, 17, 36] are immediate instantiations of MapReduce join. They are readily applicable to arrays in which chunking is ignored. The most important challenge faced by these algorithms

	Structural Join	MapReduce Join	Shuffle Join	**Array Similarity Join**
Targeted join	general	equi-join	equi-join	**similarity join**
Data transfer	locally optimal	suboptimal	globally optimal	**globally optimal**
Network congestion	ignored	ignored	runtime global synchronization	**optimal scheduling**
Load-balancing	ignored	runtime reactive	static optimized	**static optimized**
Processing nodes	store one of α or β	all	all (store α or β preferred)	**store α or β**
Repartitioning	not required	complete	complete	**not required**
Replication	minimally required	suboptimal	suboptimal	**minimally required**

Table 2: Comparison of array similarity join algorithms.

is how to choose the join units such that the amount of data that have to be replicated – thus, transferred over the network – is minimized. MRSimJoin [40] uses pivot-based partitioning with random pivots extracted from array α. The points at the join unit border are replicated across all the neighboring units. This has the potential to degenerate into an unbounded number of data alignment rounds. ClusterJoin [36] introduces a series of filters to reduce the number of replicated points. Both these algorithms are extensions of QuickJoin [20] to MapReduce. MR-DSJ [37] uses grid-based partitioning, i.e., regular chunking across all dimensions, to compute join units. A join unit consists of a chunk and its neighbors—there are 2^d such neighbor chunks, where d is the number of dimensions. This results in a replication factor of 2^d for every cell. PHiDJ [17] reduces the replication factor in MR-DSJ by splitting the dimensions into several groups and executing MR-DSJ independently on each group. For k dimension groups, data are replicated only $k \cdot 2^{\frac{d}{k}}$ times. This reduction in replication – and amount of data transferred – translates into a corresponding exponential increase in computation—due to executing MR-DSJ k times.

B.3 Shuffle Join

None of the above algorithms optimize for the overall data transfer. In structural join, each node minimizes its local data receiving, while MapReduce join ignores communication completely. Shuffle join [19] aims to minimize the overall data transfer imposed by the execution of an array equi-join, while guaranteeing some form of load-balancing. It extends upon the track join minimal network traffic distributed hash algorithms introduced in [32]. The main idea is to consider the assignment of join units to nodes as a global optimization problem and solve it after all the nodes finish their local partitioning. The amount of data each node has in a join unit is the principal decision variable. Several algorithms are considered, including a simple minimum bandwidth greedy heuristic that assigns a join unit to the node storing the largest portion of cells in the unit; a tabu search algorithm that incorporates load-balancing into the minimum bandwidth heuristic; and an integer programming formulation that optimizes the end-to-end execution time. The proposed analytical cost model has the inherent limitation that communication and computation cannot be overlapped across the join units assigned to the same node. Moreover, the order in which a node has to send its partitions is not computed—it is arbitrary. A global synchronization mechanism that enforces a single node to transmit data to a destination at any given time instant is deployed in order to prevent network congestion. However, this can have the negative effect of stalling nodes.

B.4 Discussion

Table 2 summarizes the properties of the baseline algorithms for array similarity join in a distributed array database. It also includes the specific operator proposed in this work—in bold font. Structural join is the most general of these algorithms. MapReduce and shuffle join are in the same family of equi-join algorithms. Extensions to array similarity join and other types of join are possible, however, they incur costly modifications. Shuffle join is the only algorithm that aims to minimize the overall data transfer. MapReduce join incurs heavy all-to-all communication, while structural join targets only local optimizations at each node. Network congestion is addressed only by shuffle join through a runtime global synchronization mechanism that gives writing access on a link to a single sender. Load-balancing is supported as a runtime reactive process in MapReduce join. The approach in shuffle join is to embed load-balancing into the data transfer scheduling. Thus, only MapReduce and shuffle join include all the nodes – not only the ones storing the join arguments – in the processing. Since they are not specific to array similarity join, MapReduce and shuffle join require complete data repartitioning and mapping to the output array space. As a result, repartitioning incurs unnecessary data replication due to the output array having higher dimensionality than the input arrays. As shown in Table 2, the proposed array similarity join operator aims inherits the benefits of the baseline algorithms by minimizing the overall transfer and network congestion while providing load-balancing across the nodes that store data, but without completely repartitioning and replicating the arrays.

C. ADDITIONAL EXPERIMENTS

In this section, we include experimental results for the queries PTF 2-D and GEO. While the trend is similar to PTF 3-D, these results provide another reference point on the characteristics of the proposed array similarity join operator. They prove the benefits of the proposed operator over non-array similarity join operators applied to array data even when the number of join pairs is not extremely large (PTF 2-D) and the dataset size is rather small (GEO)—no parallel processing is required in this case.

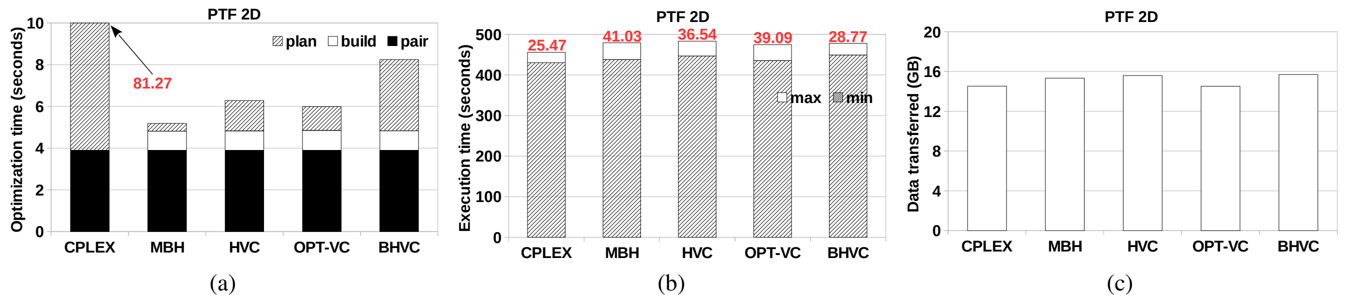


Figure 14: Transfer graph evaluation on PTF 2D query: (a) optimization time, (b) execution time, and (c) transferred data.

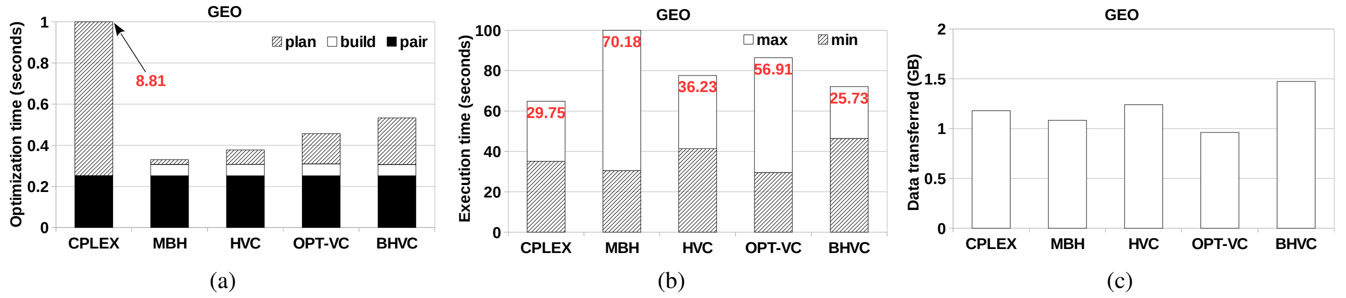


Figure 15: Transfer graph evaluation on GEO query: (a) optimization time, (b) execution time, and (c) transferred data.

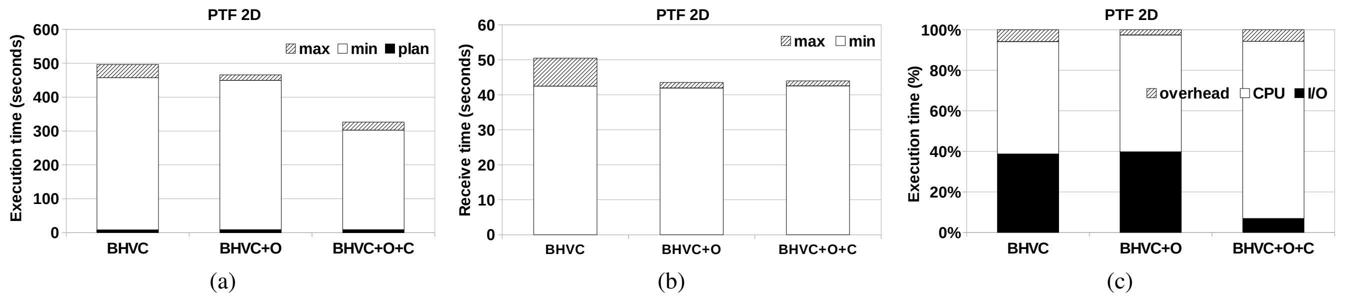


Figure 16: Query optimization evaluation on PTF 2D query: (a) execution time, (b) receiving time, and (c) thread time distribution.

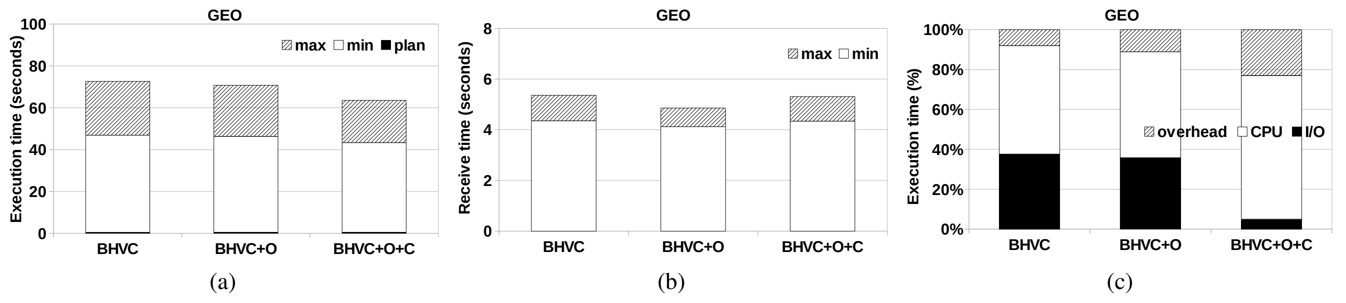


Figure 17: Query optimization evaluation on GEO query: (a) execution time, (b) receiving time, and (c) thread time distribution.