Advanced Database Techniques
for Processing Scientific Multi-Dimensional Data

by

Weijie Zhao

A dissertation submitted in partial satisfaction of the requirements for the degree of
Doctor of Philosophy

in

Electrical Engineering and Computer Science

Committee in charge:
Professor Florin Rusu, Chair
Professor Sungjin Im
Professor Dong Li

Summer 2018
The dissertation of Weijie Zhao is approved:

Florin Rusu, Chair
Date

Sungjin Im
Date

Dong Li
Date

University of California, Merced

©Summer 2018
To my family
Acknowledgments

First of all, I would like to sincerely thank my advisor Professor Florin Rusu. Without his constant guidance over the years, this project would not have been possible. I was inspired and enlightened by his profound knowledge, acute insights, deep understanding of the philosophy of research, sense of humor and most importantly, meticulous attention to details.

I would like to thank my dissertation committee, Professor Sungjin Im and Professor Dong Li for their time, insightful comments and constructive guidance that significantly improved this dissertation.

I have spent all of my three summers during my PhD years in Berkeley. I would like to thank my colleagues in Lawrence Berkeley National Lab, Dr. Kesheng Wu, Dr. Bin Dong and Professor Peter Nugent for providing me with helpful advice, interesting and challenging problems. I also want to thank Dr. Kai Zeng for his insightful research collaboration with me.

I also would like to express my appreciation to my previous labmates and colleagues at UC Merced. Dr. Yu Cheng, Dr. Chengjie Qin and Dr. Tao Ren offered me valuable advice when I began my research and life at Merced. The wonderful two years with them are certainly memorable. I also want to thank my current and previous labmates, Mr Zixuan Zhuang, Mr Martin Torres, Mr. Zhiyi Huang, Dr. Ye Zhu and Mr. Jun Hyung Shin and for all the collaborations and helps. Especially, I want to thank my current labmate Ms Yujing Ma for her supports.

For the past few years back in China, I have been so lucky to work with great scholars both at East China Normal University. As my bachelor advisor, Professor Yugen Du, Professor Zhenbing Zeng and Professor Wensheng Yu, they inspired me with their sense of responsibility, rigorous scientific spirit; they led me to the journey of scientific research. Special thanks goes to Professor Linhua Jiang, without his guidance, I would not begin my journey in academia.

Last but not least I want to express my gratitude to my family. My parents sacrificed a lot to raise me with a sense of responsibility, a deep understanding the meaning of hard work and a optimal attitude for appreciation of life. They are always being a constant source of love and encouragement for me. And again, I would like to thank my fiancee, Yujing Ma, for her support and understanding to me.
Curriculum Vitae

Education


Publications

1. Zhao, W., Rusu, F., Dong, B., Wu, K., Ho, A., & Nugent, P. Distributed Caching for Processing Raw Arrays. SSDBM 2018


5. Cheng, Y., Zhao, W., & Rusu, F. Bi-Level Online Aggregation on Raw Data. SSDBM 2017


7. Zhao, W., Rusu, F., Dong, B., & Wu, K. Similarity Join over Array Data. SIGMOD 2016


Honors

- Graduate Dean’s Dissertation Fellowship 2018
- SIGMOD Student Travel Awards 2017
- SIGMOD Student Travel Awards 2016
- Silver Medal The 2013 ACM-ICPC Asia Chengdu Regional Contest 2013
- Silver Medal The 2012 ACM-ICPC Asia Chengdu Regional Contest 2012
- National Scholarship 2012 (Top 2% students) 2012
# Contents

## List of Figures

- Contents

## List of Tables

- List of Figures x
- List of Tables xiii

## 1 Introduction

- 1.1 Scientific Data Processing ................................................. 2
- 1.1.1 Multi-dimensional Data Processing ................................... 2
- 1.1.2 Raw Data Exploration .................................................... 3
- 1.2 Primary Contributions ..................................................... 4
- 1.3 Outline and Previously Published Work .................................. 6

## 2 Similarity Join over Array Data

- 2.1 Background ................................................................. 7
- 2.2 Preliminaries ............................................................... 10
- 2.2.1 Array Data Model ....................................................... 10
- 2.2.2 Array Joins ............................................................... 11
- 2.2.3 Similarity Join ........................................................... 11
- 2.3 Array Similarity Join ...................................................... 12
- 2.4 Array Similarity Join Operator .......................................... 15
- 2.4.1 Query Execution ......................................................... 15
- 2.4.2 Query Optimization ..................................................... 16
- 2.4.3 Transfer Graph Optimization ......................................... 18
- 2.4.4 Transfer Schedule Optimization ..................................... 22
- 2.4.5 Data Access Plan Optimization ..................................... 23
- 2.5 Experimental Evaluation .................................................. 23
- 2.5.1 Transfer Graph Evaluation ........................................... 25
- 2.5.2 Query Optimization Evaluation ..................................... 26
- 2.5.3 Comparison with Related Algorithms ................................. 27
- 2.5.4 Scalability ............................................................... 27
- 2.5.5 Discussion ............................................................... 28
- 2.6 Related Work ............................................................... 28
- 2.7 Conclusions and Future Work ........................................... 29
- 2.8 Optimization Formulation .................................................. 30
2.9 Baseline Algorithms ................................................. 31
  2.9.1 Structural Join ............................................... 31
  2.9.2 MapReduce Similarity Join ................................. 32
  2.9.3 Shuffle Join ................................................ 33
  2.9.4 Discussion .................................................. 34
2.10 Additional Experiments ........................................... 34

3 Incremental View Maintenance over Array Data ................................................. 36
  3.1 Background ........................................................ 36
  3.2 Preliminaries ..................................................... 39
    3.2.1 Materialized Views ........................................... 40
  3.3 Array Views ....................................................... 40
  3.4 Array View Maintenance .......................................... 43
    3.4.1 Baseline View Maintenance ................................. 43
    3.4.2 Optimal View Maintenance .................................. 44
    3.4.3 Differential View Computation .............................. 47
    3.4.4 View Chunk Reassignment .................................... 48
    3.4.5 Array Chunk Reassignment ................................... 48
  3.5 Query Integration ................................................ 50
  3.6 Experimental Evaluation .......................................... 52
    3.6.1 Setup ........................................................ 53
    3.6.2 View Maintenance Per Update Batch ......................... 55
    3.6.3 Optimization Time ............................................ 56
    3.6.4 Query Integration ............................................. 56
    3.6.5 Discussion .................................................. 57
  3.7 Related Work .................................................... 57
  3.8 Conclusions ...................................................... 58
  3.9 NP-Hard Proofs .................................................. 58
    3.9.1 Differential View Computation .............................. 58
    3.9.2 View Chunk Reassignment ................................... 59
    3.9.3 Array Chunk Reassignment ................................... 59
  3.10 Algorithm Examples ............................................... 60
    3.10.1 Differential View Computation .............................. 60
    3.10.2 View Chunk Reassignment ................................... 61
    3.10.3 Array Chunk Reassignment ................................... 61
  3.11 Additional Experiments ........................................... 62
    3.11.1 Overall View Maintenance Time ............................. 62
    3.11.2 Sensitivity to Batch Size .................................... 62
    3.11.3 Sensitivity to Number of Batches ............................ 63
    3.11.4 Sensitivity to Update Spread ............................... 63
4 User-Defined Scientific Data Analysis on Arrays

4.1 Background ................................................................. 64
   4.1.1 Motivating Examples .............................................. 66
   4.1.2 Research Challenges ............................................. 68
4.2 ArrayUDF Approach ..................................................... 68
   4.2.1 STENCIL Design Considerations ................................. 70
   4.2.2 ARRAY Design Considerations ................................. 71
   4.2.3 Analytical Model for ArrayUDF ................................. 76
4.3 Performance Evaluation ................................................ 78
   4.3.1 Synthetic Data and Poisson Equation Solver .................. 79
   4.3.2 ArrayUDF for Real Scientific Data Analysis .................. 84
4.4 Related Work ............................................................. 85
4.5 Conclusions and Future Work ......................................... 86

5 Distributed Caching for Processing Raw Arrays .......................... 87

5.1 Background ................................................................. 87
5.2 Preliminaries ............................................................. 90
   5.2.1 In-Situ Processing over Raw Data ............................... 90
   5.2.2 Caching ............................................................... 91
5.3 Raw Array Distributed Caching ........................................ 91
   5.3.1 Raw Array Chunking .............................................. 93
   5.3.2 Cost-Based Chunk Caching ...................................... 95
   5.3.3 Cache Eviction .................................................... 96
   5.3.4 Cache Placement .................................................. 97
5.4 Experimental Evaluation ............................................... 98
   5.4.1 Setup ............................................................... 100
   5.4.2 Results ........................................................... 101
   5.4.3 Discussion ........................................................ 103
5.5 Related Work ............................................................ 104
5.6 Conclusions and Future Work ......................................... 106

6 Conclusions ........................................................................... 107

6.1 Array Data Organization .................................................. 107
6.2 Limitations and Future Work ............................................. 108
6.3 Closing Thoughts ........................................................ 108

Bibliography ........................................................................... 110
List of Figures

2.1 Array $A_{<r:int,s:int>[i=1,6,2;j=1,8,2]}$ .................................................. 8
2.2 Result array of $\text{APPLY}(A, \text{SUM}(v+u), \text{shape})$. ........................ 8
2.3 Mapping of similarity distance into shape array for $\epsilon = 3$ ($L^0$, $L^1$, $L^2$, $L^\infty$, Earth Mover's Distance (EMD)) and $\epsilon = 1$ (Hamming). ........................................... 12
2.4 Array similarity join example: $\text{SELECT} \ast \text{ INTO } \tau \text{ FROM } \alpha \text{ SIMILARITY JOIN } \beta$
\text{ ON } (k = j) \text{ WITH SHAPE } \sigma ................................. 13
2.5 Optimal transfer graph: (a) HVC and (b) BHVC. .......................... 16
2.6 Query execution sub-plan for a node. .............................. 17
2.7 Assignment of chunk pairs to nodes. The non-colored vertices correspond to unassigned chunk pairs. ................. 20
2.8 Transfer schedule network. ........................................... 20
2.9 Transfer graph evaluation on PTF 3D query: (a) optimization time, (b) execution time, and (c) transferred data. ......................................................... 22
2.10 Query optimization evaluation on PTF 3D query: (a) execution time, (b) receiving time, and (c) thread time distribution. ......................................................... 23
2.11 Algorithm execution time comparison: (a) PTF 3-D, (b) PTF 2-D, (c) GEO. ... 26
2.12 Algorithm data transferred comparison: (a) PTF 3-D, (b) PTF 2-D, (c) GEO. ... 26
2.13 Speedup for the PTF 3-D query. ........................................... 28
2.14 Transfer graph evaluation on PTF 2D query: (a) optimization time, (b) execution time, and (c) transferred data. ......................................................... 34
2.15 Transfer graph evaluation on GEO query: (a) optimization time, (b) execution time, and (c) transferred data. ......................................................... 35
2.16 Query optimization evaluation on PTF 2D query: (a) execution time, (b) receiving time, and (c) thread time distribution. ......................................................... 35
2.17 Query optimization evaluation on GEO query: (a) execution time, (b) receiving time, and (c) thread time distribution. ......................................................... 35
3.1 (a) AQL notation for array $A_{<r:int,s:int>[i=1,6,2;j=1,8,2]}$ and materialized array view $V$ defined as the array similarity self-join query $\text{CREATE ARRAY VIEW V AS SELECT COUNT(*) FROM A A}$ $\text{SIMILARITY JOIN A A}$ $\text{ON (A1.i = A2.i) AND (A1.j = A2.j) WITH SHAPE L^1(1) GROUP BY A1.i,A1.j}$. (b) View maintenance under insertions at indices [1,5], [2,1], [2,3], [4,2], [4,4], [5,4], and [5,6] into array $A$. ......................................................... 39
3.2 View, query, and corresponding $\Delta$ shapes.
3.3 View maintenance time on all the datasets and all the update batch configurations.
3.4 (a) Query shapes. (b) $\Delta$ shapes.
3.5 Average optimization time per update batch: (a) PTF-5, (b) PTF-25, and (c) GEO.
3.6 Differential query vs. similarity join on PTF.
3.7 Example for Algorithm 2.
3.8 Example for Algorithm 4.
3.9 Overall execution time (optimization + view maintenance): (a) PTF-5, (b) PTF-25, and (c) GEO.
3.10 Sensitivity analysis on PTF-25 with real updates: (a) increasing batch size, (b) number of batches, (c) update spread.

4.1 An example using ArrayUDF to compute user-defined average, i.e., “MyAvg”, on a 2D array $A$, stored in a HDF5 file, named “d2d.h5”. The results are stored in array $B$, which has the same dimensions as that of $A$ and is stored in file “d2davg.h5”. Given different UDF operators, $B$ can have different dimensions from $A$. The user-specified chunk size ($cs$) and ghost zone size ($os$) are used to support parallel processing. Both $cs$ and $os$ are optional and can be determined by ArrayUDF automatically.
4.2 Example of chunks and ghost zones in a 2D array.
4.3 The method for selecting chunking parameters.
4.4 An example showing the number of ghost cells for different chunk shapes.
4.5 An example showing the layout-aware chunking for a $4 \times 9$ array. On the left, a chunk occupies more than one row. On the right, a chunk occupies a partial row.
4.6 Apply algorithm in ArrayUDF. We use “$\rightarrow$” symbol to denote the components of $A$. For example, $A \rightarrow Apply$ is the Apply method of $A$ and $A \rightarrow cs$ is the chunk size metadata of $A$.
4.7 Linear relationship between the chunk size and the cost of reading a chunk.
4.8 Cost of reading ghost cells for different chunk shapes and for different layouts.
4.9 Performance of reading ghost zone in row-major layout.
4.10 Comparing two methods to handle ghost zone.
4.11 Comparison of ArrayUDF with the state-of-the-art data processing systems.
4.12 Evaluation with analyzing real-world scientific datasets. ArrayUDF is up to 2070X faster than Spark. For the CoRTAD dataset, where we show the performance only for ArrayUDF, Spark crashes due to out-of-memory (OOM) error.

5.1 Array $A<\text{n:char, f:int}[^{i=1,6; j=1,8}]$. This is a raw array consisting of 7 files distributed over 3 nodes. Attributes $n$ and $f$ correspond to the node and the file id on the node, e.g., $<X,2>$ is a cell in the second file on node X. The cell color also shows the server on which the file is stored. The dashed rectangle specifies a query.
5.2 Architecture of distributed caching for raw arrays.
5.3 Raw array query-driven chunking. (a) depicts a raw array and its bounding box. (b), (c), and (d) show how the chunking evolves based on a workload of three queries. (e) depicts the resulting four chunks into which the initial array is partitioned.
5.4 High-level view of raw array distributed caching. The chunks in Figure 5.1 are shown together with the corresponding bounding box separately for each server. The queried subarray is depicted as a dashed rectangle.

5.5 Query execution time as a function of the cache memory budget. Three caching strategies – file_LRU, chunk_LRU, and cost-based caching (evict) – are compared across three workloads and three file formats. The time for cost-based caching includes raw file chunking, cache eviction, and cache placement. Chunk_LRU includes only raw file chunking, while file_LRU is the time for caching complete raw files using LRU.

5.6 Execution time improvement over file_LRU on PTF in HDF5 with 160 GB cache budget.

5.7 Optimization time on LinkedGeoData.

5.8 Reduction in similarity join execution time with cache placement (40 GB cache): (a) PTF-1, (b) PTF-2, and (c) GEO.
List of Tables

2.1 MIP binary variables. ......................................................... 30
2.2 Comparison of array similarity join algorithms. .......................... 32

3.1 Binary variables and parameters in the MIP formulation. ................. 41
3.2 Example for Algorithm 3. .................................................. 61

4.1 Notations used in the chunking strategy model ............................ 76
4.2 Overhead of the trial run (microsecond). .................................. 81
4.3 Real-world scientific datasets and operations ............................... 84
Abstract

Scientific applications are generating an ever-increasing volume of multi-dimensional data that are largely processed inside distributed array databases and frameworks. Traditional databases are not equipped with the adequate functionality to handle the volume and variety of “Big Data”. Scientific data have dual structure. Raw data are preponderantly ordered multi-dimensional arrays or sequences while metadata and derived data are best represented as unordered relations. Scientific data processing requires complex operations over arrays and relations. These operations cannot be expressed using only standard linear and relational algebra operators, respectively. In this dissertation, we investigate array data analysis, in order to help scientists manage and analyze multi-dimensional array data in a more efficient way. We propose an array database system that supports complex array similarity join and incremental array view maintenance over sparse arrays. We build a user-defined function query execution framework on dense arrays. We also develop a distributed caching for raw array framework to enable the system to directly query on unorganized arrays in raw formats. The resulting prototype systems show order-of-magnitude speedups compared to their traditional solutions on a variety of tasks.
Chapter 1
Introduction

With the rapid development of science and technology, huge volumes of data are generated from scientific researches. The needs for large scale scientific data analysis are rising gradually. 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. Traditional databases are not equipped with the adequate functionality to handle the volume and variety of “Big Data”. Strict schema definition and data loading are prerequisites even for the most primitive query session. Scientific data have dual structure. Raw data are preponderantly ordered multi-dimensional arrays or sequences while metadata and derived data are best represented as unordered relations. Scientific data processing requires complex operations over arrays and relations. These operations cannot be expressed using only standard linear and relational algebra operators, respectively. Existing scientific data processing systems are designed for a single data model and handle complex processing at the application level. This dissertation focuses on multi-dimensional array data in scientific data analysis, in order to help scientists manage and analyze data in a more efficient way.

1.1 Scientific Data Processing

The major features of scientific data processing are: a) data is multidimensional and queries are complex, e.g., convolution, stencil, etc.; b) large data volumes in raw formats generated by experimental devices.

1.1.1 Multi-dimensional Data Processing

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.

\(^1\)http://www.sdss.org/dr12/
\(^2\)http://www.astro.caltech.edu/ptf/
\(^3\)http://www.lsst.org
Due to the inefficacy of traditional relational databases to handle ordered array data [43], numerous array processing solutions that implement a distributed multi-dimensional array data model have emerged [16, 24, 34, 122, 154, 170]. 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 [145]. 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 [16, 34, 122]. 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. Palomar Transient Factory (PTF) project is another example. The PTF project aims to identify and automatically classify transient astrophysical objects such as variable stars and supernovae in real-time. As a secondary objective, a catalog containing the identified transients and other celestial objects is constructed for subsequent querying and analysis. Between 2000 and 4000 high-resolution (2048 × 4096 pixels) images are taken each night and fed into pipelines through high-speed communication links. The total amount of raw data generated varies between 60 and 100 GB per night. We have a strict time constraints in this real-time processing situation. Designing a distribution system to analysis these data efficiently on supercomputer is a challenging task.

Complicated Array Query. 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. So that astronomers can study the property of celestial object across time dimension by an array similarity join. The array contains more than 100 million cells, we have to store the array in a distributed system to accelerate query processing. Network communication among computing nodes is likely to become the bottleneck when we have tens of CPU cores on each node. We have to design an array query processing system to optimize the communication.

Data Product Maintenance The array similarity join query results on PTF data are considered as a derived data product. For example, “select all the celestial objects that are only detected in one night” corresponds to a collection of candidates for Gamma Ray Burst. PTF pipeline generates new detections per night. If a new detection is found, we have to maintain the previous query result – remove the object from the candidates. Re-compute the similarity join query from scratch is time-consuming. We have to develop a novel system that is able to incrementally update the data product efficiently over rapid incoming new experimental data.

1.1.2 Raw Data Exploration

Scientific data are generated at an unprecedented volume by scientific instruments observing the macrocosm [3, 74, 114] and the microcosm [151, 165]. These data come in a variety of formats,
ranging from delimited text to semi-structured JSON and multi-dimensional binaries such as FITS.

The volume and variety of “Big Data” pose serious problems to traditional database systems. Before it is even possible to execute queries over a dataset, a relational schema has to be defined and data have to be loaded inside the database [33, 35, 36]. Schema definition imposes a strict structure on the data, which is expected to remain stable. However, this is rarely the case for rapidly evolving datasets represented using key-value and other semi-structured data formats, e.g., JSON. Data loading is a schema-driven process in which data are duplicated in the internal database representation to allow for efficient processing. Even though storage is relatively cheap, generating and storing multiple copies of the same data can easily become a bottleneck for massive datasets. Moreover, it is quite often the case that many of the attributes in the schema are never used in queries.

For example, Sloan Digital Sky Survey (SDSS) is a decade-long astronomy project having the goal to build a catalog of all the astrophysical objects in the observable Universe. Images of the sky are taken by a high-resolution telescope, typically in binary FITS format. The catalog data summarize quantities measured from the images for every detected object. The catalog is stored as binary FITS tables. Additionally, the catalog data are loaded into an RDBMS and made available through standard SQL queries. The loading process replicates multi-terabyte data three times — in ASCII CSV and internal database representation — and it can take several days — if not weeks. Unlike conventional dataset, tables of SDSS usually contain a large number of attributes, i.e. photoPrimary has 509 attributes. However, out of which only 74 are referenced in queries. This means that 435 attributes are replicated three times without ever being used — a significantly sub-optimal storage utilization.

When a data analyst/scientist tries to execute a query workload on SDSS FITS data to do some data exploration, she has to first convert the FITS format data into a format which is supported by conventional databases, e.g. CSV. Then she loads the entire dataset into an RDBMS. Even without the slow format converting — database supports the corresponding format natively, data loading are also time-consuming. Although the query processing is fast after loading the whole dataset, data loading time becomes the most expensive stage. Since scientific datasets contains hundreds of attributes in one single table — unlike conventional relational dataset only has tens of attributes — data loading stage takes most of the time to parse and load the unused attributes. We need to design a novel data storage/processing architecture to store and manage data efficiently.

1.2 Primary Contributions

In this section, we summarize the primary contributions of this work.

Similarity Join over Array Data. 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 [17, 76] 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 optimiza-
tion 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. The technical details are in Chapter 2.

Incremental view maintenance over array data. We introduce the concept of materialized array views defined over complex shape-based similarity join aggregate queries. Since shape-based array similarity join is a generalization of array equi-join and distance-based similarity join [172], materialized array views cover an extensive class of array algebra operations [105]. According to the literature [43, 76, 122, 133], these are the most prevalent operations in scientific applications. With regard to SQL, array views include the class of join views with standard aggregates [100]. We tackle incremental array view maintenance under batch updates to the base arrays. Batch updates are the only form of updates in many real applications, e.g., astronomy, and are essential for amortizing the cost of network communication and synchronization in a distributed deployment [103]—the case for array databases. Our solution is discussed in Chapter 3.

User-Defined Functions on Dense Arrays. We propose a UDF system named ArrayUDF. With ArrayUDF, we extend the expressiveness of structural locality to allow users to easily define operations on adjacent cells of an array and to perform data management tasks efficiently for supporting these user-defined operations. ArrayUDF is also capable of identifying the minimum portion of an array accessed on each process (e.g., CPU core) and operate on that portion of the array data stored in files, without loading the entire array into the system. This optimization is possible because the array syntax used by ArrayUDF to describe the operations provide a clear mechanism to identify the relevant cells and the optimal data partition can be determined analytically.

Distributed caching over raw arrays. We tackle the problem of distributed caching for raw arrays with the goal to accelerate queries over frequently accessed ranges. Given a set of raw files that contain portions of an array and an online dynamic query workload, we have to determine which cells to cache in the distributed memory of an array database system. Rather than having each node manage its memory buffer to cache local cells, we aim for a global caching infrastructure that automatically identifies both the cells to cache and the instance where to cache them. Our ultimate goal is to provide both instant access to distributed multi-dimensional raw arrays and optimal query performance through caching. We design a distributed caching framework that computes an effective caching plan in two stages. First, the plan identifies the cells to be cached locally from each of the input files by continuously refining an evolving R-tree index. Each query range generates a
finer granularity bounding box that allows advanced pruning of the raw files that require inspection. This guarantees that – after a sufficiently large number of queries – only relevant files are scanned. In the second stage, an optimal assignment of cells to nodes that collocates dependent cells in order to minimize the overall data transfer is determined. We model cache eviction and placement as cost-based heuristics that generate an effective cache eviction plan and reorganize the cached data based on a window of historical queries. We design efficient algorithms for each of these stages. In the long run, the reorganization improves cache data co-locality by grouping relevant portions of the array and by balancing the computation across nodes. The details are described in Chapter 5.

In all, these results highlight a widespread potential for scientific database system that supports in-situ analysis, complex query and view maintenance for multidimensional data processing. Our resulting database system prototypes and their regular order-of-magnitude speedups compared to conventional solutions show the power of this latent potential for integrating in-situ processing and multidimensional data optimization into the distributed database system.

1.3 Outline and Previously Published Work

The remainder of this dissertation proceeds as follows. Chapter 2 and Chapter 3 investigate the similarity join over array data and array view maintenance for array similarity join results. Chapter 4 examines the user-defined function executions on dense arrays. Chapter 5 presents the distributed caching mechanism that makes direct array queries on raw data possible. Chapter 6 concludes with a discussion of lessons learned, topics for future work, and closing thoughts.

Chapter 2 revises [172]. Chapter 3 includes material from [174]. Chapter 4 revises [53]. Chapter 5 includes material from [173].
Chapter 2

Similarity Join over Array Data

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 chapter, 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.

2.1 Background

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 \textit{multi-dimensional arrays} rather than unordered sets—the case in the relational data model. Due to the ineffectiveness of traditional relational databases to handle ordered array data [43], numerous array processing solutions that implement a distributed multi-dimensional array data model have emerged [16, 24, 34, 122, 154, 170]. To cope with the massive data volumes, these systems partition the arrays across a \textit{distributed shared-nothing cluster} and process the partitions concurrently.

Queries on arrays fall in two categories [145]. The first category consists of relational-style

\(^1\)www.sdss.org/dr12/
\(^2\)www.astro.caltech.edu/ptf/
\(^3\)www.lsst.org
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 [16, 34, 122]. 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.

**Figure 2.1:** Array \( A < v : \text{int}, u : \text{int}> [i = 1, 8; j = 1, 8, 2]. \)

**Figure 2.2:** Result array of \( \text{APPLY}(A, \text{SUM}(v+u), \text{shape}). \)

**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 [2] 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 [134], 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., \( 1 \sim 1.5 \)
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 [116, 131, 156]. 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.

\(^4\)https://www.sdss3.org/instruments/boss_spectrograph.php
\(^5\)https://ned.ipac.caltech.edu/
In database jargon, catalog cross-matching corresponds to FULL OUTER JOIN. However, due to the uncertainty of astronomical 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 [80].

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 $\text{catalog}[ra, dec, time]$. 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.** 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 [17,76] 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 2.3)
- 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 2.4.1-2.4.2)
- We model the query optimization of array similarity join as a vertex cover problem and
introduce efficient algorithms to find optimal execution plans. (Section 2.4.3-2.4.5)

- 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 2.5)

2.2 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.

2.2.1 Array Data Model

A multi-dimensional array \([76,133,145]\) is defined by a set of dimensions \(D = \{D_1, D_2, \ldots, D_d\}\) and a set of attributes \(A = \{A_1, A_2, \ldots, A_m\}\). 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 \(A\). 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:

\[
\text{Array} : D_1 \times D_2 \times \cdots \times D_d \mapsto (A_1, A_2, \ldots, A_m)
\]

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 [133] for a survey – regular chunking is the most popular in practice, e.g., SciDB. We adopt it in this work. 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 [133]. 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 2.1 depicts a 2-D array with schema \( A < v: \text{int}, u: \text{int} > \) \([i=1, 8, 2; j=1, 8, 2]\) given in AQL [96, 105]—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 2.1. 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.

2.2.2 Array Joins

Consider two \( d \)-dimensional arrays \( \alpha \) and \( \beta \) given in the functional representation in Eq. (2.1):

\[
\alpha : \{D^\alpha = D^\alpha_1 \times \cdots \times D^\alpha_d\} \mapsto \{A^\alpha = (A^\alpha_1, \ldots, A^\alpha_m)\} \\
\beta : \{D^\beta = D^\beta_1 \times \cdots \times D^\beta_d\} \mapsto \{A^\beta = (A^\beta_1, \ldots, A^\beta_m)\}
\]

A join between arrays \( \alpha \) and \( \beta \), \( \tau = \alpha \bowtie \beta \), is written in AQL as: SELECT expression INTO \( \tau \) FROM \( \alpha \) JOIN \( \beta \) 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 \( \tau : D^\tau \mapsto A^\tau \) having the default schema:

\[
D^\tau = D^\alpha \cup D^\beta \\
A^\tau = A^\alpha \cup A^\beta
\]  

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 2.1 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 \( \beta \) for every chunk of \( \alpha \).

Duggan et al. [76] introduce a series of array equi-joins – \( \text{dimension:dimension}, \text{attribute:attribute}, \) and \( \text{attribute:dimension} \) – 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.

2.2.3 Similarity Join

We consider the \( \epsilon \)-join version [80] of similarity join. The input to \( \epsilon \)-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:

\[
A \bowtie \epsilon B = \{(\vec{a}, \vec{b}) : ||\vec{a} - \vec{b}|| < \epsilon; \vec{a} \in A, \vec{b} \in B\}
\]  

(2.3)
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:

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

with \( p = 1, 2, \ldots, \infty \). \( L^2 \) corresponds to the familiar Euclidean distance, \( L^1 \) to the Manhattan distance, and \( L^\infty \) 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.

\[
\begin{align*}
L^0 & \quad L^1 & \quad L^2 & \quad L^\infty & \quad \text{EMD} & \quad \text{Hamming}
\end{align*}
\]

Figure 2.3: Mapping of similarity distance into shape array for \( \epsilon = 3 \) (\( L^0, L^1, L^2, L^\infty, \text{Earth Mover’s Distance (EMD)} \)) and \( \epsilon = 1 \) (Hamming).

### 2.3 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 \( \epsilon \)-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 \texttt{APPLY} operators in the ArrayQL algebra \cite{105} and AML \cite{107}, respectively. In the simplest form, \texttt{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 \texttt{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 2.2 depicts the result of \texttt{APPLY(A, SUM(v+u), shape)} over array \( A \) in Figure 2.1. The sum of \( v + u \) across all
the cells 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.

**Definition 1 (Array Similarity Join)**

Given two multi-dimensional arrays \(\alpha\) and \(\beta\) in functional representation:

\[
\alpha : \{D^\alpha = D^\alpha_1 \times \cdots \times D^\alpha_d\} \longrightarrow \{A^\alpha = (A^\alpha_1, \ldots, A^\alpha_m)\}
\]

\[
\beta : \{D^\beta = D^\beta_1 \times \cdots \times D^\beta_d\} \longrightarrow \{A^\beta = (A^\beta_1, \ldots, A^\beta_m)\},
\]

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

**SELECT** \(f\) **INTO** \(\tau\) **FROM** \(\alpha\) **SIMILARITY JOIN** \(\beta\)

**ON** \((k = j)\) **WITH SHAPE** \(\sigma\)

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 \([\text{dec}]\), respectively.
Example 2  Figure 2.4 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 \( M \) maps a cell \((i, j) \in \alpha\) to the cell in \( \beta \) which has index \( k = j \). This is the \( \Psi \) 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 \( \alpha \) is similar with cells having index \( k = j \) or \( k = j + 1 \) from \( \beta \). The value function \( f \) concatenates the attributes of \( \alpha \) and \( \beta \) 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 \( \tau \) is reduced to \( [j = 1, 3] \). There are 5 non-empty cells in \( \tau \)—the non-empty cells in \( \alpha \) and \( \beta \) are hatched. 4 of them—red in Figure 2.4—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 2.4.

Mapping of similarity distance into shape array. Figure 2.3 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 \( \Psi \) cell is hatched. For the \( L^p \) norms, the size of the shape array increases as \( p \) increases. Since \( L^0 \) and \( L^\infty \) are defined as limits, we introduce special shapes for them. The shape array corresponding to \( L^0 \) is cell \( \Psi \), while the shape array of \( L^\infty \) covers a square with side \((2e + 1)\). 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 \( 2e \). 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 \( e \) 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 \( [76] \) to more complex join predicates. dimension:dimension join is defined as a natural join with equality predicates. This corresponds to a mapping function \( M : (D_1^\beta = D_1^\alpha, \ldots, D_d^\beta = D_d^\alpha) \) and a similarity shape \( \sigma[D_1^\beta = 0; \ldots; D_d^\beta = 0] \)—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, \( k \geq j \) \( AND \) \( k < j + 2 \). 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 \( \Psi \) in \( \beta \) to every cell \( \Gamma \) in \( \alpha \). Second, the shape \( \sigma \) is applied to \( \Psi \) rather than \( \Gamma \)—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 2.3—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.
2.4 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 $\alpha$ and $\beta$ chunked over the database servers, the goal is to compute optimally the result of their similarity join $\tau = \alpha \bowtie_M^{\sigma, f} \beta$ as defined by the mapping function $M$, the shape $\sigma$, and the value function $f$. Abstractly, the computation of $\tau$ requires three stages. First, for each chunk $\Upsilon \in \alpha$, the chunks $\Psi \in \beta$ that are in $\sigma(M(\Upsilon))$ 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 $\Psi \in \sigma(M(\Upsilon))$. Second, chunks $\Upsilon$ and $\Psi$ 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 $\tau$. 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.

2.4.1 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 [145]. 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 $\alpha$. As a result, although the schema of the output array $\tau$ is well-defined at query time, the chunking of $\tau$ is known only at query execution. In general, $\tau$ is chunked along the array $\alpha$ dimensions—not necessarily regular. The actual shape of the chunks is determined by the shape array $\sigma$ in the query. A chunk covers the full range of the dimensions in array $\beta$. In order to impose a specific chunking on array $\tau$, schema alignment from shuffle join [76] is required. This is a costly operation that incurs complete data repartitioning and full replication of arrays $\alpha$ and $\beta$ for every chunk in $\tau$—the dimensionality of $\tau$ is the sum of the dimensionality of $\alpha$ and $\beta$, 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 $\sigma$ 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 $\sigma$ 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 $\beta$. To find the similar cells corresponding to a cell $\Upsilon \in \alpha$, a range query with the predicate $\sigma(M(\Upsilon))$ 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 $\sigma$, hash join is likely the most efficient algorithm. A hash table is built over the chunk in $\beta$ and is probed for each cell in $\sigma(M(\Upsilon))$ corresponding to $\Upsilon$. 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.

![Figure 2.5: Optimal transfer graph: (a) HVC and (b) BHVC.](image)

2.4.2 Query Optimization

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 $M$ and the shape array $\sigma$ – and the catalog metadata which stores...
Figure 2.6: Query execution sub-plan for a node.

the location of the chunks and relevant chunk statistics, such as the number of non-empty cells. The global plan consists of a detailed execution sub-plan (Figure 2.6) 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 \( \alpha \) and \( \beta \). 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:

\[
\min \left\{ \max_{ijk} \left\{ y_{ijk} \cdot t \cdot T_{ntwk} \right\} \right\} 
\]

(2.5)

where \( y_{ijk} \) 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 \( y_{ijk} \), i.e., \( \sum_t y_{ijk} = x_{ijk}, \forall i, j, k \). \( T_{ntwk} \) 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:

\[
\min \left\{ \max_k \left\{ \sum_{ij} y_{ijk(t-1)} \sum_{i' \in \Psi_i} (1 - z_{ij}(t-1)) \cdot T_{disk} \right\} \right\} 
\]

(2.6)

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., \( \max \{\text{Eq. (2.5)}, \text{Eq. (2.6)}\} \), 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 asym-
chronous 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 Section 2.8. Although it can be written as a linear integer program, finding the optimal network transfer schedule, i.e., Eq. (2.5), is known to be a combinatorially hard problem [130]. The disk access time objective, i.e., Eq. (2.6), 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 2.6). 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. (2.6) on $y_{ijkt}$, it becomes the well-known optimal cache replacement problem with a standard solution.

### 2.4.3 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:

$$
\min \left\{ \max_j \sum_i \sum_k x_{ijk} \cdot s_i, \max_k \sum_i \sum_j x_{ijk} \cdot s_i \right\} \quad (2.7)
$$

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 2.5).

Our solution to solve this problem practically is to modify the objective to the total amount of data transferred 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. (2.7). We propose a novel graph-based algorithm for the modified formulation and show experimentally that

---

\(^6\)http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/
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 \( \sigma(M(\Upsilon)) \) for every cell in the \( \alpha \) chunk. However, if the mapping function \( M \) preserves the order between cells along each of the dimensions, the chunks with potentially similar cells can be determined by computing \( \sigma(M(\Upsilon)) \) 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 \( \upsilon \) is a chunk in \( \alpha \) stored on node \( i \) and \( \psi \) a chunk in \( \beta \) stored on node \( j \), we create two vertices \( \upsilon_{ij} \) and \( \psi_{ji} \) 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 \( \upsilon_i \) and \( \psi_j \). 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 [54].

**Example 3** Figure 2.5 depicts the transfer graph for the array similarity self-join query:

\[
\text{SELECT * INTO } \tau \text{ FROM } \alpha_1 \text{ SIMILARITY JOIN } \alpha_2 \text{ ON } (\alpha_1.i = \alpha_2.i \text{ AND } \alpha_1.j = \alpha_2.j) \text{ WITH SHAPE } \sigma,
\]

where \( \alpha \) is the array in Figure 2.1 and \( \sigma \) is a \((3 \times 3)\) square centered on the cell. The notation for a vertex contains the chunk id – in white font in Figure 2.1 – 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.

**2.4.3.1 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 modeled as a transfer graph. The minimum weighted vertex cover of a graph 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 \( \upsilon_{ij} \) indicates that the corresponding chunk \( \upsilon \) is transferred 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)** [39]. In this greedy algorithm, vertices are sorted according to the ratio \( c/\text{degree} \), where \( \text{degree} \) is the number of incidental edges. This ratio

---

7https://en.wikipedia.org/wiki/Vertex_cover
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 2.5a 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.

### 2.4.3.2 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

\[
p(v, \text{recv}) = \text{recv}(v, j) \cdot v.c \cdot |\text{nodes}|
\]

and

\[
q(v, \text{send}) = \text{send}(v, j) \cdot v.c \cdot |\text{nodes}|
\]

in the weight \( w \) of each vertex \( v \). 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 2.5b depicts the output of BHVC on the transfer graph in Example 3. The main difference compared to HVC (Figure 2.5a) 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 increases the total amount of data transferred by...
Algorithm 1 Balanced Heuristic Vertex Cover (BHVC)

Input: transfer graph $G(V,E)$
Output: balanced vertex cover $V' \subset V$

1. $\text{load}(n) \leftarrow 0, \forall n \in \text{nodes}$
2. while $E \neq \emptyset$ do
3. select vertex $\nu_{ij}^{[c,w]}$ s.t.
$$\arg\min_{\nu \in V} \left\{ \frac{\nu.w + r.\nu.\text{recv} \cdot (1-r)\cdot \nu.\text{send}}{\text{degree}(\nu)} \right\}$$
4. for $(\nu, \psi) \in E$ do
5. $E \leftarrow E \setminus (\nu, \psi)$
6. $\psi.w \leftarrow \psi.w - \frac{\nu.w + r.\nu.\text{recv} \cdot (1-r)\cdot \nu.\text{send}}{\text{degree}(\nu)}$
7. $\text{degree}(\psi) \leftarrow \text{degree}(\psi) - 1$
8. end for
9. $\text{recv}(\nu.j) \leftarrow \text{recv}(\nu.j) + \nu.c$
10. $\text{send}(\nu.j) \leftarrow \text{send}(\nu.j) + \nu.c$
11. $V' \leftarrow V' \cup \nu$
12. $V \leftarrow V \setminus \nu$
13. end while

1 array cell—from 25 to 26. $r$ is set to 0.5 in this example.

2.4.3.3 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 2.5b 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 2.7 corresponds to the BHVC graph in Figure 2.5b 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 $[1, \text{total number of chunk pairs}]$. For the
network in Figure 2.7, \( L \) takes value 6 and the assignment of chunk pairs to nodes is color-coded. 6 chunk pairs have to be joined on each node.

Figure 2.9: Transfer graph evaluation on PTF 3D query: (a) optimization time, (b) execution time, and (c) transferred data.

### 2.4.4 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 minimize 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 2.8). 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 2.8 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: \( \{(X \to Z, Y \to X, Z \to Y) : 3; (Y \to X, Z \to Y) : 2; (X \to Y, Y \to X) : 4; (Y \to Z) : 5\} \).

**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 2.5b. 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 important 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.

2.4.5 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.

2.5 Experimental Evaluation

![Figure 2.10: Query optimization evaluation on PTF 3D query: (a) execution time, (b) receiving time, and (c) thread time distribution.](image)

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?

\(^8\)http://linkedgeodata.org
• 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 [145]. 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 50,000, respectively. In array format, this corresponds to:

\[
\text{PTF[time}=1,153064;ra=1,100000;dec=1,50000]}
\]

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 original point using a Gaussian distribution with $\mu = 0$ and $\sigma = 10$ miles [137]. In array format, this corresponds to:

\[
\text{GEO[long}=1,100000;1at=1,50000]}
\]

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

\(^9\)www.openstreetmap.org
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 \(L^\infty\)-norm of \(1\) from the center. The GEO 2-D query finds POIIs within \(1\) mile of each other. The corresponding shape array is the \(L^2\)-norm equal to \(4\) (Figure 2.3). Since the results for all the queries follow similar patterns, we include below only the results for the PTF 3-D query.

### 2.5.1 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. (2.5). 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 2.4.3.1). The minimum-bandwidth heuristic (MBH) [76] 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 [61]. 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 2.9a. 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 2.14a and 2.15a).

#### Execution time.

Figure 2.9b 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 2.9c 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 2.14c and 2.15c).
2.5.2 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 2.10a 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 2.10b. 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.

**Thread time distribution.** Figure 2.10c 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.

2.5.3 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 Section 2.9.

**Structural join (STRUCT)** [145] 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 $M$ 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 [60, 137, 140, 144] are immediate instantiations of MapReduce join [20]. 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 2.11 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.

2.5.4 Scalability

The speedup of the array similarity join operator is depicted in Figure 2.13 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.
### 2.5.5 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 optimization 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.

### 2.6 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 [16, 107]. 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 [122]. They are evaluated in the context of different chunking strategies in [145], where structural join is introduced. A complete formalization of array equi-joins and the shuffle join algorithm are given in [76], while a graph formulation is introduced in [17]. Array similarity join is not considered in any of the previous work. The proposed operator bears similarities to shuffle join [76] 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 [22, 50, 77, 80, 81, 91, 141, 142]. These algorithms can be separated into two categories—index-based and all-pairs [159]. $\epsilon$-kB tree [142] is the first high-dimensional index targeted at similarity join. It modifies the splitting criterion of kB tree to take $\epsilon$ into account. In general, a new index has to be built for every similarity query. A parallel version
of $\epsilon$-kDB tree that requires complete data repartitioning is introduced in [141]. The $\epsilon$ Grid Ordering (EGO) family [22, 80, 81] 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 [50, 80, 81]. The Quickjoin algorithm [77] 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 [77] 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 [91], 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 [49, 56, 110, 117, 155]. MRSimJoin [144] is an extension of QuickJoin [77] 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 [137] introduces a set of bisector-based distance filters [28, 159] into MRSimJoin to generate more accurate partitions. MR-DSJ [140] and PHiDJ [60] are similarity self-join grid-based algorithms that extend EGO [22] 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 [155], multiset [110], social image [176], and string [49] 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 [125]. The join is computed for each grid and duplicates are eliminated. Thermal-Join [150] 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.

### 2.7 Conclusions and Future Work

In this chapter, 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 [127] can be devised. We also plan to consider other measures than $L$ norms, e.g., Jaccard coefficient and cosine similarity.

### 2.8 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 2.4), we include the relevant constraints. The binary variables are depicted in Table 2.1.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_{ijk}$</td>
<td>chunk $i$ is sent from node $j$ to node $k$</td>
</tr>
<tr>
<td>$y_{ijk}$</td>
<td>chunk $i$ is sent from node $j$ to node $k$ at time $t$</td>
</tr>
<tr>
<td>$z_{ij}$</td>
<td>chunk $i$ on node $j$ is cached at time $t$</td>
</tr>
</tbody>
</table>

**Table 2.1: MIP binary variables.**

The cost of array similarity join has to be minimized:

$$
\min \left\{ \max_{ijk} \left\{ y_{ijk} \cdot t \cdot T_{\text{ntwk}} \right\}, \right. \\
\max_k \left\{ \sum_{ij} y_{ijk(t-1)} \sum_{i' \in \Psi_i} \left(1 - z_{i'j(t-1)} \right) \cdot T_{\text{disk}} \right\} \right. 
$$

(2.8)
under the following set of constraints:

\[ C_1 : \ x_{ijk} + x_{i'kj} = 1; \ \forall i, i' \in \Psi_j, p(i) = j, p(i') = k \]
\[ C_2 : \ \sum_{ij} y_{ijkt} = x_{ijk}; \ \forall i, j, k \]
\[ C_3 : \ \sum_{ijk} y_{ijkt} = 1; \ \forall t \]
\[ C_4 : \ \sum_{ik} y_{ijkt} = 1; \ \forall t, j \]
\[ C_5 : \ \sum_{lj} y_{ijkt} = 1; \ \forall t, k \]
\[ C_6 : \ \sum_{i} z_{ijt} \leq B; \ \forall t, j \]
\[ C_7 : \ z_{i'jt} \leq z_{ij(t-1)} + y_{ijkt}; \ \forall t, j, k, i, i' \in \Psi_i \]

\[ (2.9) \]

\( C_1 \) 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.

### 2.9 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.

#### 2.9.1 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 [145]). The join iterates over chunks of the outer array \( \alpha \). For each chunk, it looks-up the corresponding chunks in the inner array \( \beta \), 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 \( M \) 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 \( \sigma (M(\Upsilon)) \) 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 \( \tau \) 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.

<table>
<thead>
<tr>
<th>Targeted join</th>
<th>Structural Join</th>
<th>MapReduce Join</th>
<th>Shuffle Join</th>
<th>Array Similarity Join</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data transfer</td>
<td>generally optimal</td>
<td>locally optimal</td>
<td>globally optimal</td>
<td>globally optimal</td>
</tr>
<tr>
<td>Network congestion</td>
<td>ignored</td>
<td>ignored</td>
<td>runtime global synchronization</td>
<td>optimal scheduling</td>
</tr>
<tr>
<td>Load-balancing</td>
<td>ignored</td>
<td>runtime reactive</td>
<td>static optimized</td>
<td>static optimized</td>
</tr>
<tr>
<td>Processing nodes</td>
<td>store one of ( \alpha ) or ( \beta )</td>
<td>all</td>
<td>all (store ( \alpha ) or ( \beta ) preferred)</td>
<td>store ( \alpha ) or ( \beta )</td>
</tr>
<tr>
<td>Repartitioning</td>
<td>not required</td>
<td>complete</td>
<td>complete</td>
<td>not required</td>
</tr>
<tr>
<td>Replication</td>
<td>minimally required</td>
<td>suboptimal</td>
<td>suboptimal</td>
<td>minimally required</td>
</tr>
</tbody>
</table>

Table 2.2: Comparison of array similarity join algorithms.

### 2.9.2 MapReduce Similarity Join

In the structural join algorithm, the computation is executed exclusively at nodes storing chunks from the outer array \( \alpha \). 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 [20] – as a direct extension of distributed Grace hash join [45] – 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 [127]. 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 \( \alpha \) and \( \beta \), 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 \( \Upsilon \) in \( \alpha \) are assigned to a single join unit. Cells \( \Psi \) in \( \beta \) 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 [60, 137, 140, 144] 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 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 [144] uses pivot-based partitioning with random pivots extracted from array \( \alpha \). 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 [137] introduces a series of filters to reduce the number of replicated points. Both these algorithms are extensions of QuickJoin [77] to MapReduce. MR-DSJ [140] 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 [60] 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 \( k \cdot 2^d \) 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.

### 2.9.3 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 [76] 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 [127]. 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.
2.9.4 Discussion

Table 2.2 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 2.2, the proposed array similarity join operator 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.

2.10 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 2.14: Transfer graph evaluation on PTF 2D query: (a) optimization time, (b) execution time, and (c) transferred data.
Figure 2.15: Transfer graph evaluation on GEO query: (a) optimization time, (b) execution time, and (c) transferred data.

Figure 2.16: Query optimization evaluation on PTF 2D query: (a) execution time, (b) receiving time, and (c) thread time distribution.

Figure 2.17: Query optimization evaluation on GEO query: (a) execution time, (b) receiving time, and (c) thread time distribution.
Chapter 3

Incremental View Maintenance over Array Data

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

3.1 Background

Scientific applications – from astronomy to high-energy physics and genomics – collect and process massive amounts of data at an unprecedented scale. For example, projects in astronomy such as the Sloan Digital Sky Survey\(^1\) (SDSS) and the Palomar Transient Factory\(^2\) (iPTF) record observations of stars and galaxies at nightly rates varying between 60 and 500 GB. The Large Synoptic Survey Telescope\(^3\) (LSST) is projected to increase these volumes by two orders of magnitude—to 20 TB. A common feature of the datasets produced by these astronomy projects is that data are represented as *multi-dimensional arrays* rather than unordered sets—the case in the relational data model. Due to the inefficacy of traditional relational databases to handle ordered array data [43,122], a series of specialized array processing systems [16,24,34,122,154,170] have emerged. These array processing systems implement natively a distributed multi-dimensional array data model in which

\(^1\)http://www.sdss.org/dr13/
\(^2\)http://www.ptf.caltech.edu/iptf/
\(^3\)http://dm.lsst.org
arrays are chunked across a *distributed shared-nothing cluster* and processed concurrently.

According to [145], queries on arrays fall in two categories. The first category consists of relational-style operations that can be executed efficiently by any traditional database. They include filtering, sub-sampling, join, and group-by aggregates. The second category – containing array-specific operations such as smoothing, regridding, clustering, and cross-matching which are implemented as User-Defined Functions (UDF) in specialized array processing systems [16,34,122] – applies a multi-dimensional shape to each cell of the array, grouping the cell with other neighboring cells. Subsequently, an entire range of statistical functions can be applied to the resulting groups to derive domain-specific properties. To better illustrate the array-specific operations, we provide a real example from astronomy.

**Galaxy group catalogs.** One of the ultimate challenges in astronomy is to understand how galaxies form and evolve into the large-scale distribution of matter throughout the universe. The state-of-the-art method to study galaxy evolution is based on galaxy group catalogs [168] which are derived data products from general astronomical catalogs, e.g., SDSS [57] and iPTF [2]. Since their computation is time-consuming and they are heavily used in subsequent analysis – such as galaxy correlation [161], Gamma-ray bursts [89], and M-dwarf flares [68] – galaxy group catalogs are always materialized. However, they are built statically and rarely updated. For example, a galaxy group catalog is built only for new SDSS data releases [168]. This is extremely problematic for the iPTF and LSST projects which acquire data continuously since the galaxy group catalog becomes outdated soon after construction. Thus, the problem we tackle in this chapter is how to maintain these derived catalogs incrementally under updates to the base catalog.

An astronomical catalog contains objects extracted from images of the sky taken by a telescope—a collection of detections. A detection is characterized by three dimensions—equatorial coordinates $\text{ra}$, $\text{dec}$ and $\text{time}$—and tens to a few hundred attributes such as brightness and magnitude. Consequently, a catalog is typically represented as a sparse 3-D array
\[
\text{catalog}_{\text{bright,mag,}...}[\text{ra,dec,}\text{time}]
\]
in which a cell corresponds to a detection—identified by the index on each dimension. For example, in the iPTF catalog, an array cell corresponds to 1 arcsecond on $\text{ra}$ and $\text{dec}$, and 1 minute on $\text{time}$. These values are enforced by the resolution and the exposure time of the telescope. The catalog array can be stored physically at a lower granularity by partitioning the dimension ranges into chunks and grouping detections. A galaxy group catalog contains a series of statistics for every detection in the astronomical catalog. The statistics are computed from detections that are nearby in space and/or time. This computation can be decomposed into two steps—identify nearby or similar detections and statistics evaluation. Nearby detections are computed as a similarity self-join query over the entire catalog. Statistics such as the number of nearby detections and the density are computed for each individual detection, i.e., group-by aggregation over the similarity self-join result. While a galaxy group catalog is also modeled as a 3-D array
\[
\text{galaxy\_catalog}_{\text{cnt,density,}...}[\text{ra,dec,}\text{time}]
\]
the statistics evolve over time as new detections are added to the astronomical catalog. Thus, $\text{galaxy\_catalog}$ corresponds to a materialized view. Although catalog and $\text{galaxy\_catalog}$ have the same dimensionality in the example, this is not a requirement in general. Moreover, the base array(s) and the materialized view are not required to have identical chunking and partitioning.

**Problem statement.** We introduce the concept of *materialized array views* defined over
complex shape-based similarity join aggregate queries. Since shape-based array similarity join is a
generalization of array equi-join and distance-based similarity join [172], materialized array views
cover an extensive class of array algebra operations [105]. According to the literature [43, 76,
122, 133], these are the most prevalent operations in scientific applications. With regard to SQL,
array views include the class of join views with standard aggregates [100]. We tackle incremental
array view maintenance under batch updates to the base arrays. Batch updates are the only form
of updates in many real applications, e.g., astronomy, and are essential for amortizing the cost of
network communication and synchronization in a distributed deployment [103]—the case for array
databases.

**Challenges.** There are two primary challenges posed by incremental array view maintenance
under batch updates. The first challenge is identifying the cells in the base array(s) that are involved
in the maintenance computation and the cells that require update in the array view. This is a problem
because of the complex query in the view definition which involves a shape-based similarity join—
enumerating all the cells in the shape array corresponding to an update can be expensive for large
shapes and sparse arrays. The second challenge is due to the distributed nature of array databases. Given
the current distribution of the array(s) and the view, the challenge is to find the optimal
strategy – data transfer and computation balancing – to integrate the updates into the view. Direct
application of distributed relational view maintenance algorithms – defined over equi-join queries
and horizontally partitioned views [100] – to arrays suffers from excessive communication and load
imbalance due to the static array partitioning strategies and the skewed distribution of the updates
in scientific applications.

**Approach.** Since the granularity of I/O and computation in array databases is the chunk – a
group of adjacent cells [133] – the first challenge requires only the identification of the chunks in-
volved in view maintenance. This can be done efficiently as a preprocessing step over the metadata.
Our approach for the second challenge is to model distributed array view maintenance as an optim-
ization formulation that computes the optimal plan to update the view based on the chunks identi-
fied in preprocessing. Moreover, the optimization continuously repartitions the array and the view
based on a window of past batch updates. In the long run, repartitioning improves view maintenance
time by grouping relevant portions of the array and the view and by distributing join computation
across the cluster. Meanwhile, repartitioning does not incur additional time because it takes advan-
tage of the communication required in view maintenance. Since the optimization cannot be solved
efficiently, we decompose the formulation into three separate stages – differential view computation,
view chunk reassignment, and array chunk reassignment – that we solve independently.

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

- We define formally array views (Section 3.3). As far as we know, the concept of views has
  not yet been adapted to array databases.

- We model incremental array view maintenance as an optimization formulation that integrates
  view updating and continuous array repartitioning based on a historical workload of batch
  updates (Section 3.4.2). To the best of our knowledge, this is the first solution that considers
  adaptive array reorganization in the context of incremental view maintenance.

38
Figure 3.1: (a) AQL notation for array $A<r:int,s:int>[i=1,6,2;j=1,8,2]$ and materialized array view $V$ defined as the array similarity self-join query

```sql
CREATE ARRAY VIEW V AS
SELECT COUNT(*) FROM A A
1
SIMILARITY JOIN A A
2
ON
3
(A1.i = A2.i) AND (A1.j = A2.j)
4
WITH SHAPE L1(1) GROUP BY A1.i, A1.j.
```

(b) View maintenance under insertions at indices [1,5], [2,1], [2,3], [4,2], [4,4], [5,4], and [5,6] into array $A$.

- We design a three-stage heuristic that solves the incremental array view maintenance effectively (Section 3.4.3-3.4.5). This is necessary because the original optimization and each of the three stages are NP-hard problems.
- We introduce an analytical cost model for answering similarity join queries over arrays with materialized views (Section 3.5). This model identifies the best alternative between a complete similarity join and a differential query on the view.
- We perform an extensive set of experiments on real datasets and batch updates (Section 3.6). The results confirm the effectiveness of the heuristics and the quality of the maintenance plan for an individual update. The continuous array reorganization further reduces the view maintenance time by as much as a factor of 2 over a sequence of real updates.

### 3.2 Preliminaries

In this section, we introduce materialized views and relational incremental view maintenance. These concepts are the foundation for defining views over arrays and formalizing incremental array view maintenance.
3.2.1 Materialized Views

Materialized views are used by traditional database systems to answer queries faster. Data cubes [62] are a classical example of materialized views that aggregate data across all the possible combinations of a set of dimensions. Maintaining the content of a materialized view up to date in the presence of changes to the base tables is known as view maintenance. Complete recomputation is the simplest maintenance strategy, however, it is expensive for updates that reference only a small number of tuples in the view. In such cases, incremental maintenance that applies deltas only to the modified tuples is usually more efficient for large base tables.

Formally, let \((Q, M(D))\) denote a materialized view [103], where \(Q\) is the definition query and \(M(D)\) is the materialized query result for an input dataset \(D\). When the input changes from \(D\) to \((D + \Delta D)\), incremental view maintenance evaluates a delta query \(\Delta Q\) that updates \(M(D)\) accordingly:

\[
M(D + \Delta D) = M(D) + \Delta Q(D, \Delta D)
\]

In general, computing \(\Delta Q\) and updating \(M(D)\) accordingly is faster than re-evaluating \(Q\) from scratch. \(\Delta Q\) has to be derived for each view and can have one delta query for each base table in the dataset \(D\). These delta queries are grouped into an associated view maintenance trigger. The frequency at which the trigger is executed – or, alternatively, the size of \(\Delta D\) – has a serious impact on the view maintenance time [103].

3.3 Array Views

A relational view is defined over any query [37]. Since the result of a query is a relation, the view can be used in queries as any other relation. Following the same principle, an array view is defined by a query over arrays. However, not all the queries on arrays produce an array as the result [96, 105]. This is problematic because it hinders composability—a fundamental property of relational algebra. Thus, in order to support composition, the queries on which an array view is defined have to be limited to those that output arrays. Nonetheless, the class of such queries has to be general. Array similarity join is known to be a generalization of array equi-join and distance-based similarity join [172]. Moreover, it is composable. Thus, we define an array view over multiple arrays starting from their shape-based similarity join followed by a series of unary array operators, e.g., sub-sampling, projection, group-by aggregate, etc.

**Definition 2** Given \(n\) multi-dimensional arrays \(\alpha_1, \ldots, \alpha_n\) and \(k\) unary array operators \(\oplus_1, \ldots, \oplus_k\) that produce array output, we define an array view

\[
V \leftarrow \oplus_1 \left( \cdots \oplus_k \left( \alpha_1 \oplus_{\sigma_1} M_1 \alpha_2 \oplus_{\sigma_2} M_2 \cdots \oplus_{\sigma_{n-1}, f_{n-1}} M_{n-1} \alpha_n \right) \right)
\]

as the result of applying the sequence of \(k\) operators to the chain of similarity joins among the \(n\) input arrays.

**Example 4** Consider array view \(V\) defined by the AQL query

CREATE ARRAY VIEW V AS
   SELECT COUNT(*) AS cnt
   FROM A_1 SIMILARITY JOIN A A_2
Table 3.1: Binary variables and parameters in the MIP formulation.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_{ikj}$</td>
<td>transfer chunk $i$ from node $k$ to node $j$</td>
<td>$U$</td>
<td>set of historical batch updates</td>
</tr>
<tr>
<td>$z_{pqk}$</td>
<td>chunks $p$ and $q$ are joined at node $k$</td>
<td>$W_i$</td>
<td>weight of update batch $U_i$</td>
</tr>
<tr>
<td>$y_{ij}$</td>
<td>chunk $i$ is assigned to node $j$</td>
<td>$S_q$</td>
<td>node storing chunk $q$ at beginning of current updates</td>
</tr>
<tr>
<td>$x'_{ijk}$</td>
<td>transfer chunk $i$ from node $j$ to $k$ given $y$</td>
<td>$B_q$</td>
<td>size of chunk $q$ in bytes</td>
</tr>
<tr>
<td>$z'_{pqk}$</td>
<td>chunks $p$ and $q$ are joined at node $k$ given $y$</td>
<td>$T_{ntwk}$</td>
<td>time to transfer a unit chunk between two nodes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$T_{cpu}$</td>
<td>time to compute the join between two unit chunks</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\lambda$</td>
<td>importance ratio between current and historical batches</td>
</tr>
</tbody>
</table>

ON $(A_1.i = A_2.i)$ AND $(A_1.j = A_2.j)$
WITH SHAPE $L^1(1)$
GROUP BY $A_1.i$, $A_1.j$

depicted in Figure 3.1 (a). $V$ consists of a single similarity self-join on $A$ based on $L^1(1)$ similarity shape, i.e., a 5-cell cross centered on each cell, and identity mapping and value function. The resulting 4-D array is then projected on a single pair of dimensions $(i, j)$ passed as arguments to the GROUP BY clause. The value of a cell in $V$ is given by the number of non-empty neighbor cells of the corresponding cell in $A$. Thus, the single unary operator in $V$ definition is a group-by aggregate. Following the similarity join example, there are only two cells with value 2—$V[i=1, j=2] \mapsto \langle \text{cnt} = 2 \rangle$, $V[i=1, j=3] \mapsto \langle \text{cnt} = 2 \rangle$.

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

Materialized array views. We consider materialized array views that are evaluated eagerly at view definition and have their result stored as an array $V$. While the schema of $V$ is well-defined by the query, its chunking can be either specified explicitly or it can be inferred from the chunking of the input arrays $A_i$ [172]. Maintaining view $V$ under modifications to the base arrays $A_i$ becomes the primary challenge in this case.

Example 5 Figure 3.1 (b) depicts array $A$ and materialized array view $V$ after 7 new cells are added to $A$—they are hatched in the figure. The new cells result in the creation of two new chunks—7 and 8—in $A$ and $V$, respectively, assigned to nodes according to the chunking strategy corresponding to each array. Since no chunking is specified in the view definition, the chunking of $V$ is inherited from $A$. However, the assignment of $V$ chunks to nodes is done by considering $V$ as an independent array. The number of cells in view $V$ that are impacted by the insertions to $A$—also hatched—is 11. These cells cover all the chunks in the view, thus, the entire view has to be updated. This is due to the shape in the similarity join—a cell in $A$ can impact as many as 5 cells in $V$.

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

- **Batch updates.** We consider batches of updates to the base array(s) because this is the standard use case in scientific applications, e.g., the iPTF pipeline ingests a series of images
rather than one. Moreover, view maintenance is executed as part of a processing pipeline, not concurrently with queries. For relational views, batch updates are mostly meant to reduce the per-transaction overhead while deferring view maintenance [41, 67, 86, 92]. In the case of array views, we go one step further and share the delta computation across the batch. This is beneficial because similarity join increases the degree of sharing—unlike relational equi-join.

- **Update granularity.** Array databases access and process data at chunk granularity [34, 122]. Thus, cell updates degenerate into chunk-level operations. Metadata is also kept at chunk granularity. As a result, we group update operations, i.e., diffs [86], into chunks and perform view maintenance over chunks. This is different from relational view maintenance which operates at tuple granularity since no ordering is enforced. Chunk-level maintenance is sub-optimal when the number of updated cells inside a chunk is small. However, updates are clustered by the acquisition process in many scientific applications [54]—including iPTF. Batching also helps. Performing maintenance at cell granularity has the potential to prune unnecessary joins between chunk pairs. This requires more detailed metadata, e.g., positional information on non-empty cells inside the chunk, and more time-consuming join pair identification. While this may be acceptable for relational equi-join, the cost is magnified for arbitrary shape-based array similarity join.

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

- **Distributed processing.** Distributed maintenance in data warehousing typically assumes that the base tables and the view are stored non-partitioned on a single server, albeit each of them on different servers [97, 138, 139]. When the base tables and the view are partitioned [14, 100], all the partitions require maintenance in the worst case. This can be alleviated by building indexes on each server. We consider distributed array views defined over distributed base arrays where updates are handled by a coordinator. Determining the communication among partitions involved in the maintenance becomes an important optimization parameter in this case because the number of potential join pairs for each chunk is magnified by the similarity join in the definition.

- **Recursive maintenance.** If the array view is defined over more than two arrays, updates to a single array require $n-1$ similarity joins with base arrays. This can be very expensive. A restricted form of recursive view maintenance [8, 103] can be applied to array views by materializing the result of each pair of join chains obtained by splitting the $n-1$ joins in the view definition. While these auxiliary views reduce maintenance time, they also require maintenance, i.e., recursive maintenance.
from it.

3.4 Array View Maintenance

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

High-level approach. Given input arrays $\alpha$ and $\beta$ and a materialized array view $V = \alpha \bowtie_{\sigma,f}^M \beta$ defined as the similarity join between $\alpha$ and $\beta$, the goal is to maintain $V$ efficiently under updates $\Delta\alpha$ and $\Delta\beta$ to the base arrays. We consider only two arrays in order to focus on the fundamental similarity join problem—the extension to multiple arrays is recursive. $\alpha$, $\beta$, and $V$, respectively, are chunked over the database servers, while each of $\Delta\alpha$ and $\Delta\beta$ consists of a set of chunks located initially at the coordinator. The assignment of chunks to nodes for all the arrays in the database are chunked over the database servers, while each of $\Delta\alpha$ and $\Delta\beta$ is stored in the catalog metadata—also managed by the coordinator. The assignment of chunks to nodes for all the arrays in the database are chunked over the database servers, while each of $\Delta\alpha$ and $\Delta\beta$ is stored in the catalog metadata—also managed by the coordinator. The updates $\Delta\alpha$ and $\Delta\beta$, respectively, come in batches at regular time intervals, i.e., cyclic batch updates. In an algebraic notation [37, 63, 93], the maintenance of $V$, i.e., computing $V + \Delta V$, requires computing the differential view $\Delta V = (\alpha \bowtie_{\sigma,f}^M, \Delta\beta) \cup (\Delta\alpha \bowtie_{\sigma,f}^M, \beta) \cup (\Delta\alpha \bowtie_{\sigma,f}^M, \Delta\beta)$ followed by merging into $V$. $\Delta V$ is the similarity join between the updates and the base arrays and the updates themselves, respectively. $V + \Delta V$ is computed according to the view definition.

3.4.1 Baseline View Maintenance

The baseline array view maintenance algorithm is based on the parallel relational view maintenance process proposed in [100]. Since the original process in [100] works only for single tuple updates, we extend it to batch updates. The algorithm works as follows. The new chunks $\Delta\alpha$ and $\Delta\beta$—as well as new chunks in view $V$—are first assigned to nodes using the chunking strategy for each array. In order to compute the differential view $\Delta V$, the coordinator identifies — from the catalog...
metadata – all the chunks in $\beta$ that join with a chunk in $\Delta \alpha$ and sends the new chunk to the nodes that store these $\beta$ chunks. If no $\beta$ chunk joins with $\Delta \alpha - \Delta \alpha$ is an irrelevant update [19] – no computation is required. The same process is applied to $\Delta \alpha$ and $\alpha$. Since the join $\Delta \alpha \bowtie M \Delta \beta$ is included twice, we exclude $\Delta \alpha$ from $\alpha$ when we compute $\alpha \bowtie M \Delta \beta$. The merging $V + \Delta V$ is realized by sending the chunks in the differential view to the nodes that store the corresponding view chunks. This is done asynchronously as the $\Delta V$ chunks are received. The coordinator provides the location of the view chunks and the number of $\Delta V$ chunks to all the nodes participating in the computation.

We exemplify how the baseline algorithm works based on the updates in Figure 3.1. The new chunks 7 and 8 are assigned to node $X$ and $Y$ in array $A$, while the corresponding view chunks are assigned to node $Y$ and $Z$, respectively. We consider the differential view for chunk 7. Since chunk 7 joins with chunks 2, 4, and 8, it is also sent to node $Y$. Joins $7 \bowtie 2$ and $7 \bowtie 8$ are computed on $Y$, while $7 \bowtie 4$ on $X$, respectively. The view merging $V + \Delta V$ requires communication between the following pairs of nodes: $Y \rightarrow X$ for view chunk 2, $X \rightarrow Y$ for view chunk 4, and $Y \rightarrow Z$ for view chunk 8, respectively.

This example illustrates one problem of the baseline algorithm—excessive communication. To update view chunk 2 on $X$, node $X$ first sends chunk 7 to node $Y$ to compute the join $7 \bowtie 2$ only to have the result returned to node $X$. If chunk 2 is sent from $Y$ to $X$, the view can be updated locally, without additional communication. The second problem is excessive computation at a node, i.e., load imbalance. This happens for chunk 4 which is joined with all its neighbors locally at node $X$. The main cause for these problems is the static assignment of chunks to nodes which is completely independent from the updates. In the first case, the chunks are too spread over the nodes in the cluster, while in the second case, the chunks are clustered at a single node. These problems are magnified in scientific data processing by the chunking strategies [54] and the fact that updates are concentrated on a limited area of the base arrays, e.g., the iPTF telescope points to a relatively small area of the sky during each night. For hash-based chunking, each join computation is likely to require communication because adjacent chunks are assigned to different nodes. For space-partitioning strategies – space-filling curves, quadtree, k-d tree – most of the joins are concentrated on a single node, thus the load is imbalanced.

3.4.2 Optimal View Maintenance

We address the limitations of the baseline algorithm – excessive communication and load imbalance – by modeling the array view maintenance problem as a mixed-integer program (MIP) that identifies the optimal plan to compute the differential view $\Delta V$ and the merging $V + \Delta V$ and determines the optimal reassignment to nodes for both the array and the view chunks—new and existing. The optimal view update plan reduces the excessive communication/computation, while the reassignment guarantees that we do not get stuck with an unfavorable static chunking strategy. Even more, the incoming chunks are not first assigned to a node based on a pre-determined chunking strategy. The reason for considering these two seemingly disjoint objectives together is to piggyback on the chunk replication incurred by view maintenance when computing the reassignment. Otherwise, the benefit of the reassignment may be overlooked by the required communication and chunk replication. Moreover, the reassignment takes into account a window of past batch updates, rather than only the current batch. This is necessary in order to avoid frequent unstable reassignments. The
main difference to the baseline algorithm is that we consider view maintenance and reassignment together, rather than first assigning chunks to nodes based on a pre-determined chunking and then solving the maintenance under that assignment.

**MIP variables & parameters.** The MIP variables and parameters are displayed in Table 3.1. There are two families of variables. $x$ encodes the chunk communication between two nodes, while $z$ determines the node where the join between two chunks is evaluated. Essentially, $x$ corresponds to communication and $z$ corresponds to computation. The $y$ variables encode the chunk to node reassignment after the current batch update. While the chunk to node assignment at the beginning of the update is fixed, $y$ corresponds to the optimal reassignment that is derived from the past batches of updates $U$. $x'$ and $z'$ have the same meaning as $x$ and $z$, however, they are based on the reassignment $y$, not the constant input assignment $s_q$ at the current batch update time. Moreover, $x$ and $z$ correspond to the communication/computation for the current batch, while $x'$ and $z'$ are defined over the window of historical batch updates $U$.

The constant parameters in the MIP formulation – also shown in Table 3.1 – include the historical batch of updates $U$ and their weight $W$, the chunk to node assignment $S_q$ when the current update batch is processed, the size of the chunk $B_q$, the time to transfer chunks between nodes $T_{ntwk}$ and to join two chunks $T_{cpu}$, and the weighted sum ratio $\lambda$. A batch of updates $U_i$ is a set of triples $(p, q, v)$ where chunks $p$ and $q$ from base arrays $(\alpha \cup \Delta \alpha)$ and $(\beta \cup \Delta \beta)$ have to be joined and the result has to be merged into chunk $v$ in view $V$. For each update batch, $U_i$ is computed by the coordinator from the catalog metadata. A fixed-size window of past batch updates is collected, where $U_0$ corresponds to the current batch and $U_i$ is for the $i^{th}$ previous batch. A weight $W_i$ is assigned to each batch $U_i$. While the values of $W_i$ can follow any distribution, we use exponential decay in our implementation. The older a batch is, the smaller its corresponding weight $W_i$. $S_q$ is the current node assignment of chunk $q$. It is always fixed for existing chunks and set to the coordinator for the new chunks. $B_q$ is the size of chunk $q$ and $B_{pq}$ is the total size of chunks $p$ and $q$. The values of $T_{ntwk}$ and $T_{cpu}$ are determined based on an empirical calibration process. $\lambda$ is an importance weight between 0 and 1 that discriminates between the current and past batches of updates.

We illustrate how the variables and the parameters are instantiated based on the updates in Figure 3.1. The original assignments of chunks 4 and 7 when the batch is considered are $S_{A_i} = X$, $S_{V_4} = Y$, and $S_{A_7} = s_{V_7} = Coordinator$, respectively. The triples in $U_0$ for chunk 4 are $\{\Delta A_4, A_1, V_1\}$, $(\Delta A_4, \Delta A_1, V_1)$, $(\Delta A_4, A_4, V_4)$, $(\Delta A_4, \Delta A_7, V_7\}$.

**Analytical cost model.** The MIP formalization of the optimal view maintenance problem is given in Eq. (3.1) – the objective function – and Eq. (3.2)—the constraints. The cost model is the weighted $\lambda$ summation of two terms corresponding to the view maintenance for the current update batch and the chunk reassignment based on the historical updates. The difference between the terms is that view maintenance considers only the triples in $U_0$, while the reassignment considers all the triples across the historical batches in $U$. Each of the terms is the maximum between the communication and the computation executed across all the nodes in the cluster. We consider the maximum between these two because we overlap communication and computation in our implementation setting. The communication involves two terms as well. The first term is for co-locating on the same node chunks that have to be joined for computing the differential view $\Delta V$, while the second term is for merging the result into the view chunk $V + \Delta V$. By minimizing the maximum across nodes, load-balancing is achieved since none of the nodes is allowed to perform excessive communication.
and/or computation.

The cost model introduced in this chapter differs significantly from the cost model for array similarity join in [172]. The cost for array similarity join considers only the current join and no historical queries. More importantly, we include the CPU cost and exclude the disk I/O cost—the reverse holds in the case of array similarity join. The reason for this is the much smaller number of referenced chunks in a batch of updates. While in array similarity join all the chunks have to be considered, only the chunk triples in $U_0$ are required for view maintenance. In practice, we observe that these chunks fit in memory and no further disk access is required beyond the initial loading. As a result, the CPU time becomes the dominant factor in computation at a node.

The most relevant constraints are given in Eq. (3.2). $C_1$ enforces that a chunk is stored at a single node and new chunks are assigned to a node. This is the standard in array databases which do not consider replication. $C_2$ forces chunk co-locating at node $k$ where the join between chunks $p$ and $q$ is performed. $C_3$ and $C_5$ guarantee that all the join triples are computed. $C_4$ is the equivalent of $C_2$ for the historical updates. However, the chunk assignment is no longer constant. It has to be inferred from variable $y$.

**Challenges.** Solving directly the MIP formulation with an integer programming solver poses severe challenges for several reasons. First, both the objective and the constraints contain quadratic terms that correlate the chunk assignment with the join computation, e.g., $z_{pqk}y_{ij}$. While quadratic solvers exist, they are not as advanced as linear solvers such as CPLEX⁴. Linearizing the quadratic terms by introducing new variables and constraints is the standard solution to solve this type of optimizations. However, in our case, this results in the creation of variables and constraints with 5 indices, e.g., $pqkvij$. This number is huge even for a small number of triples in $U$. Second, the max functions in the objective increase the difficulty of convergence for barrier search branch & bound methods. While max can be also linearized, this adds even more constraints. In order to verify the practicality of directly solving the MIP formulation, we experimented with the linear reduction for a single batch of 1000 updates that generate less than 4000 triples—average batch update in iPTF. CPLEX cannot find any solution beyond a feasible starting point in an hour on a massive 56-thread server.

**Solution.** We decompose the complete view maintenance formulation into three separate stages that solve the optimization for a subset of the variables. The variables set at a stage are used as input for the subsequent stages. The three stages are differential view computation, view chunk reassignment, and array chunk reassignment. The differential view computation determines the nodes where each pair of chunks $(p, q)$ is joined and the chunk communication plan for each node. This corresponds to solving the first line in the objective Eq. (3.1) for variables $z$ and $x$ with the constraints $C_2$ and $C_3$. The chunk assignment $y$ is fixed as $S$ at this stage. In view chunk reassignment, the node where to store each view chunk is determined based on the join computation plan. We solve the same objective as in the first stage with constraint $C_1$ for variables $y_{vij}$ corresponding to the view chunks by taking the values of $z$ and $x$ as input. Given the join computation plan and the view chunk reassignment, in array chunk reassignment, the base array chunks are relocated to nodes such that the view maintenance cost across the historical batch updates is minimized. This corresponds to solving the complete optimization formulation for variables $y_{vij}$, $x'$, and $z'$ with constraints $C_1$, $C_4$, and $C_5$. Notice that only the $y$ variables for base array chunks are considered. We discuss the

⁴http://www-01.ibm.com/software/commerce/optimization/cplexoptimizer/
three stages of the proposed solution in the following sections.

Algorithm 2 Differential View Computation

**Input:** triples $U_0 = \{(p, q, v)\};$ chunk location $S$ and size $B$

**Output:** $x_{ikj}, z_{pqk}$

1. $ntwk[1..N], cpu[1..N] \leftarrow 0; T[q] \leftarrow \{S[q]\}, \forall q$
2. for each $(p, q, \ast) \in U_0$ in random order do
3.   $opt \leftarrow \infty$
4.   dest $\leftarrow \emptyset$
5.   for $j \leftarrow 1$ to $N$ do
6.     $ntwk'[1..N], cpu'[1..N] \leftarrow 0$
7.     if $j \notin T[q]$ then $ntwk'[S[q]] \leftarrow ntwk'[S[q]] + B_q T_{ntwk}$
8.     $cpu'[j] \leftarrow cpu'[j] + B_{pq} T_{cpu}$
9.     $opt_{now} \leftarrow \max_k \{ntwk[k] + ntwk'[k], cpu[k] + cpu'[k]\}$
10. if $opt > opt_{now}$ then $opt \leftarrow opt_{now}; dest \leftarrow j$
11. $T[p] \leftarrow T[p] \cup \{dest\}; T[q] \leftarrow T[q] \cup \{dest\}$
12. update $ntwk$ and $cpu$
13. $x_{p,S,p,dest} \leftarrow 1; z_{p,q,dest} \leftarrow 1$
14. end for

3.4.3 Differential View Computation

The first stage aims to find the optimal join plan for computing the differential view $\Delta V = (\alpha \bowtie_{\sigma,f}^{M} \Delta \beta) \cup (\Delta \alpha \bowtie_{\sigma,f}^{M} \beta) \cup (\Delta \alpha \bowtie_{\sigma,f}^{M} \Delta \beta)$ given the current chunk assignment of the base arrays. $\Delta$ chunks are initially stored at the coordinator. However, the coordinator does not participate in the join computation. Although $\Delta V$ is the union of three array similarity join queries, it can be evaluated as a single array similarity join query across the union of the chunk join pairs in these three queries. While a solution to array similarity join is proposed in [172], it is not feasible in this context because it restricts the evaluation of a join between two chunks to the node storing the chunk in the base array—$\Delta$ chunks cannot be joined at the coordinator. Our approach is to consider all the nodes as candidates for performing the join for all the chunk pairs. However, this is an NP-hard problem—the reduction is given in Section 3.9.1. We propose an efficient randomized local search heuristic for computing the differential view join plan depicted in Algorithm 2. At high-level, the algorithm iterates randomly over the chunk join pairs $(p, q)$ in the input triples $U_0$ corresponding to the current batch update and chooses the node to perform the join such that the objective function in the MIP formulation is minimized. The input of the algorithm is represented by the triples $U_0$ and the current array chunk assignment $S$. Line (1) initializes the components of the objective function and the nodes where each chunk $q$ is stored—initially set to $S[q]$. The remaining part is the main loop of the algorithm which iterates over the chunk join pairs, while the inner loop at lines (4)-(10) iterates over the nodes. The cost of evaluating the join is computed for each node $j$ and the node that provides the minimum cost for the max objective function is selected in line (9). Line (11) updates the chunk replication location, while line (13) sets the output variables $x$ and $z$. $x_{p,S,p,dest}$ set to 1.
means that chunk \( p \) is sent from its original location to the selected node for performing the join. \( z_{pq,dest} \) set to 1 signals that chunk pair \((p, q)\) is joined on the selected node. Line (12) updates the objective function based on the selected node. We show how the algorithm works for the array in Figure 3.1 in Section 3.10.1.

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

### 3.4.4 View Chunk Reassignment

The second stage identifies the optimal node to compute the merging \( V + \Delta V \) for all the chunks in the view. The location where the differential view \( \Delta V \) is evaluated in the first stage is taken as a pre-condition instead of blindly sending \( \Delta V \) to the node that contains the corresponding view chunk—the case in the baseline algorithm. This corresponds to determining the variables \( y_{v,j} \) given values for \( x \) and \( z \) and entails the reassignment of the view chunks. We show that this problem is NP-hard in Section 3.9.2. We design an efficient randomized heuristic that follows the same ideas from Algorithm 2. This heuristic – depicted in Algorithm 3 – iterates randomly over all the view chunks \( v \) that appear among the triples \( U_0 \) and selects the optimal node where to reassign \( v \) from all the nodes in the cluster. The cost in the MIP formulation is used to discriminate between nodes. The iteration over the triples \( U_0 \) is split into two sections because all the updates to a view chunk \( v \) have to be considered together when computing the cost. Beyond this slight difference, the details of the view chunk reassignment algorithm follow from Algorithm 2 and we do not repeat them here. We provide an example showing how the algorithm works in Section 3.10.2.

If we consider a view chunk \( v \) in isolation, the reassignment is biased towards the nodes that compute more differential view chunks relevant to \( v \). However, since there is interaction between view chunks, the reassignment avoids the nodes that are communication or computation bottlenecks. The complexity of the algorithm follows from that of Algorithm 2.

### 3.4.5 Array Chunk Reassignment

The idea in array chunk reassignment is to reuse the replication required to evaluate the differential view in order to reorganize the array chunks. Replication is induced through variables \( x_{ik,j} \) which send a chunk \( i \) from its origin \( k \) to several nodes \( j \). Since chunk \( i \) is replicated across all the nodes \( j \), there is no communication overhead in performing the array chunk reassignment. Only the storage across nodes is redistributed. The goal of the reassignment is to reduce the communication cost of merging the differential view \( V + \Delta V \) for future batches of updates. This is accomplished by co-locating the array chunks in the differential view with the corresponding view chunk. Since the location of the differential view computation \( z_{pq,k} \) and the assignment of the view chunks \( y_{v,j} \) are
Our solution is to consider a window of past update batches when computing \( y_{aj} \). Thus, rather than considering only the triples \( U_0 \), we also include in the optimization the triples \( U_l \) for the past updates. For \( U_l \) to provide any useful information, though, we have to know their associated variables \( x \) and \( z \) for the reassignment configuration—these are variables \( x' \) and \( z' \). While computing these variables looks similar to computing \( x \) and \( z \) in the first stage, it is actually more difficult because the assignment of the array chunks \( S_a \) is unknown—these are the variables \( y_{aj} \) we have to compute in this stage. Notice, though, that the values of variables \( x' \) and \( z' \) do not have to be explicitly determined because we do not have to reevaluate the past update batches. What matters is the benefit a given assignment brings to past updates. We quantify this benefit with a frequency-based score associated to every chunk pair \((q, v)\) – \( q \) is an array chunk, \( v \) is a view chunk – that

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

\[
\begin{align*}
\min \left\{ \max_{k} \left\{ \sum_{i,j} x_{ikj} \cdot B_{i,T_{ntwk}} + \sum_{p,q,v} z_{pqk} \cdot y_{aj} \cdot B_{pq} T_{ntwk} \right\}, \max_{k} \left\{ \sum_{p,q} z_{pqk} \cdot B_{pq} T_{cpu} \right\} \right\} \text{ (with view)} \\
\min \left\{ \max_{k} \left\{ \sum_{i,j} x_{ikj} \cdot B_{i,T_{ntwk}} \right\}, \max_{k} \left\{ \sum_{p,q} z_{pqk} \cdot B_{pq} T_{cpu} \right\} \right\} \text{ (with complete similarity join)} \\
\end{align*}
\]  

(3.3)

known at this stage, only the variables \( y_{aj} \) for the array chunks have to be determined. One solution to compute \( y_{aj} \) considers only the current update batch. However, this has the potential to generate highly-unstable reassignments that are overreacting to changes in the update workload.
appears across the triples $U_l$. The more frequent a pair $(q,v)$ is, the higher its score can be. The history is taken into account by biasing towards recent updates based on the weight $W_l$ associated with the batch. The score takes into consideration only the communication cost in the objective function. It ignores the computation cost. This can lead to skewed reassignments in which certain nodes have to perform almost all the computation. We handle these cases by limiting the number of pairs that are assigned to each node. With score reformulation and CPU limitation, array chunk reassignment is NP-hard (Section 3.9.3).

Algorithm 4 Array Chunk Reassignment

**Input:** sets of triples $U_l = \{(p, q, v)\}$ and associated weights $W_l$;
chunk location $S$ and size $B$; $x_{ikj}$; $z_{pqk}$; $y_{aj}$

**Output:** $y_{aj}$

1. initialize $cpu\_thr[1..N]$; done $\leftarrow \emptyset$
2. $score[a,v] \leftarrow 0, \forall (a,*\!,v), (*,a,v) \in U_l, \forall l$
3. for each $(a,*\!,v), (*,a,v) \in U_l$ do
4.    $score[a,v] \leftarrow score[a,v] + W_l B_a$
5. for each $(a,v)$ in descending order of $score[a,v]$ do
6.    if $a \notin done$ then
7.        $j \leftarrow \arg \max_{k \in [1,..,N]} y_{vk}$
8.        if $(x_{aS,a} = 1)$ and $(cpu\_thr[j] \geq B_a)$ then
9.            $cpu\_thr[j] \leftarrow cpu\_thr[j] - B_a$
10.       $y_{aj} \leftarrow 1$; done $\leftarrow$ done $\cup \{a\}$
11.    end if
12. end if
13. end for
14. $y_{aS_a} \leftarrow 1, \forall a \notin done$

We design a greedy algorithm for chunk reassignment (Algorithm 4). The algorithm iterates over the chunk pairs $(a,v)$ in decreasing order of their score and assigns array chunk $a$ to the node where $v$ is assigned, as long as the computation threshold $cpu\_thr$ is not exceeded—lines (5)-(13). This guarantees that $a$ is grouped together with the view chunk it is most correlated with. The chunks that cannot be assigned – due to the CPU limitation at a node – stay at their previous location $S_a$ (line (14)). $cpu\_thr$ is initialized as the average join cost per node across all the triples in $U_l$. It is important to notice that the assignment of $\Delta$ chunks is also handled by this algorithm. However, if a $\Delta$ chunk cannot be assigned to any node due to tight CPU limitations, we assign it to the node containing the $v$ chunk with the highest score. The example in Section 3.10.3 illustrates how the algorithm works.

### 3.5 Query Integration

In this section, we present a succinct discussion on how to integrate array views in shape-based similarity join queries. This is orthogonal to using views in relational databases [48, 65]. We
focus on the difficult case when the shape of the query is different from the shape used in the view definition. Differences in the aggregate function are treated similar to relational views [40]. Given a query and a view, our goal is to optimally answer the query using the view. We show that this can be derived from the MIP formulation for view maintenance (Section 3.4.2). However, it may not be more efficient than evaluating the query from scratch. Thus, we devise a cost model that allows us to choose the best alternative.

Figure 3.2: View, query, and corresponding Δ shapes.

Δ shape. Consider the shape arrays in Figure 3.2. An array view is defined over shape view. The query to evaluate is an array similarity join over the shape query. We define Δ shape as the positional symmetric set difference between view and query, i.e., $Δ = (view \setminus query) \cup (query \setminus view)$. In Figure 3.2, the Δ shape contains the 6 cells—2 for $(view \setminus query)$ and 4 for $(query \setminus view)$.

Differential query evaluation. In order to evaluate the query using the view, the result of the similarity join with the Δ shape has to be merged with the view. This process can be mapped into the view maintenance problem by generating the update triples in $U_0$ for all the chunks in the base array with shape Δ. However, the merging with the view creates the result array rather than updating the view. This reduction allows us to apply the MIP optimization framework and the derived algorithms for differential view computation in Section 3.4.3. The alternative to answer the query is to compute the similarity join with shape query over the base arrays.

Analytical cost model. We present an analytical cost model that allows us to identify the better solution for a given view and query. In both cases, the model is a subset of the MIP formulation for view maintenance in Eq. (3.1). The cost for each alternative is depicted in Eq. (3.3). The only difference is the additional term corresponding to the interaction with the view. The common parts correspond to the shape-based similarity join. From the two costs, it appears that it is always better to compute the join from scratch since the cost of the view solution contains an additional term. We remind the reader that the input to these costs is different. The triples in the view cost are generated from the Δ shape, while the chunk join pairs in similarity join are extracted from the query shape.

The main factor that determines the relationship between the costs is the relative ratio between the size of Δ and query. Intuitively, if the ratio is larger than 1, the full similarity join is more efficient. By solving the two optimization formulations and finding their minimum cost, we can decide which alternative to pursue. Nonetheless, the cost model may not reflect the reality accurately—as with any query optimizer.
3.6 Experimental Evaluation

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

- Does the proposed differential view computation improve upon the baseline algorithm for a single update batch?
- Does the chunk and view reassignment improve the view maintenance time across a series of update batches? How sensitive is the reassignment to correlations between batches?
- What is the execution time of the incremental array view maintenance heuristic (Section 3.11)? How is the time split between differential view maintenance and chunk reassignment?
- When is querying with an array view better than complete similarity join computation?

Figure 3.3: View maintenance time on all the datasets and all the update batch configurations.
How sensitive is the proposed method to batch size and chunk spread in the updates? (Section 3.11)

3.6.1 Setup

**Implementation.** We implement incremental view maintenance as a layer on top of the array similarity join operator proposed in [172]. Similarity join is implemented as a C++11 distributed multi-thread prototype that uses an enhanced storage manager derived from ArrayStore [145]. The catalog is stored at the coordinator and replicated to all the nodes in the cluster at runtime. The incremental view maintenance heuristic is executed at the coordinator and the resulting plans containing information on chunk transfer and reassignment, and chunk join pair evaluation are distributed to the nodes. The similarity join operator runs as a server on each node in the cluster. It manages a pool of worker threads equal to the number of CPU cores in each node. A worker thread is invoked with a pair of chunks that have to be joined and the node where to send the result for view merging. Requests are made to the local and remote array storage managers to retrieve the chunks to join. This happens concurrently across all the workers. View merging is also executed by worker threads from the pool. Whenever a join result is received, a worker is assigned to merge it to the view. The code contains special functions to harness detailed profiling data.

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

**Methodology.** We include in the evaluation three methods—baseline, differential, and reassign. Baseline (Section 3.4.1) is the parallel relational view maintenance procedure adapted to array data. Differential corresponds to the first stage of the proposed heuristic in which only the join plan is optimized (Section 3.4.3). Reassign is the complete heuristic that adds view and array chunk reassignment to differential (Section 3.4.5). By separating the heuristic into differential and reassign, we can study the impact of reassignment separately. We measure wall-clock time.

**Data.** We use the same two real datasets as in [172] for experiments. The **PTF catalog** consists of 1 billion time-stamped objects represented in the equatorial coordinate system (ra, dec). The range of the time coordinate spans over 153,064 distinct values, while for ra and dec we use ranges of 100,000 and 50,000, respectively. In array format, this corresponds to:

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

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

**LinkedGeoData** stores geo-spatial data used in OpenStreetMap. We use the “Place” dataset

http://linkedgeodata.org

53
which contains location information on roughly 3 million 2-D (long, lat) points-of-interest (POI). Since this is a too small dataset, we synthetically generate a larger dataset by adding 9 synthetic points with coordinates derived from each original point using a Gaussian distribution with \( \mu = 0 \) and \( \sigma = 10 \) miles [137]. In array format, this corresponds to:

\[
\text{GEO} [\text{long}=1,000,000; \text{lat}=1,500,000]
\]

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

**Figure 3.4:** (a) Query shapes. (b) \( \Delta \) shapes.

**Figure 3.5:** Average optimization time per update batch: (a) PTF-5, (b) PTF-25, and (c) GEO.

**Views.** We create three materialized array views – two on the PTF dataset and one on LinkedGeoData – that count the number of similar neighbors for each cell in the base arrays. The shapes in the similarity join from the view definition are depicted in Figure 3.4a. PTF-5 defines similarity as the \( L^1 \)-norm of size 1 on the \((ra, dec)\) dimensions across the previous 200 days. This corresponds to clustering objects within 10 arcseconds of each other. In PTF-25, similarity is defined as the \( L^\infty \)-norm of size 2 on \((ra, dec)\) which corresponds to 400 arcminutes. All the objects that appear in the catalog within this distance are considered similar—independent of the time. The size of the PTF-5 and PTF-25 views is identical – 32 GB – since they have the same schema. PTF-5 is a real “association table” used in the production PTF pipeline to follow-up interesting transient candidates. Given the massive similarity shape, PTF-25 is used to test the scalability of the proposed solution. The GEO view clusters POIs that are within 1 mile of each other. The corresponding shape array is the \( L^\infty \)-norm equal to 1. Since the GEO dataset is small, the corresponding materialized view is only 720 MB.
**Batch updates.** We extract 10 update batches from the original datasets using four methods—real, random, correlated, and periodic. Real takes the latest time-stamped batches from PTF. For random, the batches are randomly sampled out of the entire dataset. We do this only for GEO data which is not time-stamped and synthetically generated. Correlated batches are generated by repeating one of the real/random batches 10 times. Periodic batches are created by repeating the first 3 real/random batches while alternating their order, e.g., 1, 2, 3, 3, 2, 1, 1, 2, 3, 3. This order preserves correlation only for some of the batches and allows us to better evaluate the impact of chunk reassignment. We extract batches from the PTF based on the time of the observation. This is exactly the procedure updates are applied in the production pipeline. The number of chunks we select in a batch varies between 600 and 2000 which corresponds to a two-week period of updates. A typical nightly update has less than 100 chunks—too small for a meaningful evaluation. Since GEO does not include a time dimension, extracting meaningful batches is more complicated. We select 1% of the entire dataset in a batch.

### 3.6.2 View Maintenance Per Update Batch

The results for view maintenance time are depicted in Figure 3.3 for each individual batch—they do not include the optimization time. Reassignment considers windows of 5 previous queries having weights with exponential decay and impacts subsequent queries.

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

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

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

3.6.3 Optimization Time

The time to compute the view maintenance plan – and repartitioning for reassign – is depicted in Figure 3.5. We present the average optimization time across the 10 update batches. The measurement for baseline corresponds to generating the triples \((p, q, v)\) in which chunk \(p\) is joined with chunk \(q\) and the result is merged into view chunk \(v\). These triples have to be computed for all the methods. Differential adds the execution time of Algorithm 2, while reassign adds the times for Algorithm 3 and Algorithm 4 on top of that. There is a clear trend across all the datasets. Differential incurs a minimal overhead over baseline, while reassign takes at most double the time of baseline. In absolute terms, the optimization time per batch is at most 3.5 seconds which, we believe, is an acceptable value considering the significant reduction in maintenance time it brings—as much as 200 seconds or more (for PTF-25).

3.6.4 Query Integration

We evaluate answering similarity join queries with materialized views on several shapes over the PTF dataset. The baseline is computing the query from scratch. The results are depicted in Figure 3.6. On the x-axis, the arrow points from the available view to the query, e.g., \(L_\infty(1) \leftarrow L_1(1)\) means that we answer query with shape \(L_\infty(1)\) using a view defined with shape \(L_1(1)\). We observe that in some cases the view is better, while in other cases it is not. It all depends on the relative size of \(\Delta\) shape compared to the size of the query shape. We depict these shapes for queries \(L_\infty(1) \leftarrow L_1(1)\) and \(L_\infty(1) \leftarrow L_\infty(2)\) in Figure 3.4b. Since the ratio for the first query is \(4/9\), it is more efficient to use the view. The ratio for the second query is \(16/9\) and the complete similarity join outperforms the view. The analytical cost model we introduce in Section 3.5 is able to identify the optimal solution.

![Figure 3.6: Differential query vs. similarity join on PTF.](image-url)
3.6.5 Discussion

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

3.7 Related Work

Array databases. While many array databases have been proposed over the years, none of them supports views. In the following, we focus only on how these systems handle the computation of derived array products. We point the interested reader to [133] for a comprehensive discussion on array database systems in general. RasDaMan [16] is a general middleware for array processing with chunks stored as BLOBs in a back-end relational database. RAM [154] and SRAM [42] provide support for array processing on top of the MonetDB [135] columnar database. They do not provide native support for arrays since arrays are represented as relations and array operations are mapped over relational algebra operators. While these systems do not explicitly include array views, it is conceivable that the support for relational views in the back-end systems can be reused. RIOT [169] is a prototype system for linear algebra operations over large vectors and matrices mapped into a standard relational database representation. Linear algebra operations are rewritten into SQL views and evaluated lazily—these are not materialized views. SciDB [122] is the most advanced shared-nothing parallel database system designed specifically for dense array processing. It supports multi-dimensional nested arrays with cells containing records, which in turn can contain multi-dimensional arrays. Although SciDB supports a large set of array operations, it lacks support for views. SciHadoop [24] implements array processing on top of the popular Hadoop Map-Reduce framework which lacks view support. ArrayStore [145] and TrajStore [44] are storage managers optimized for multi-dimensional arrays and trajectories, respectively. They do not provide a query execution strategy to implement it nor views.

Array joins. Positional array equi-joins are introduced in the first releases of SciDB [122]. They are evaluated in the context of different chunking strategies in [145], where structural join is introduced. A complete formalization of array equi-joins and the shuffle join algorithm are given in [76], while a graph formulation is introduced in [17]. Array similarity join is introduced in [172] as a generalization of array equi-joins and distance-based similarity join. While it bears similarities to shuffle join in allocating join units to nodes based on an optimization process, array similarity join encompasses a larger variety of join predicates and is composable.

Incremental view maintenance. Materialized views are a classical concept in databases, with several surveys [63, 65] written on the topic—the most recent by Chirkova and Yang [37]. Based on the discussion in Section 3.3, array view maintenance falls in the category of deferred maintenance [41, 86, 92, 175] with batch updates [67, 103, 128]; theta- and other complex joins [75,
standard SQL aggregates [123, 167]; parallel/distributed processing [7, 14, 97, 100, 103, 136, 138, 139]; and recursive handling of many joins [8, 103]. The emphasis of query integration is on how to use the view in query optimization [40, 48, 111]—not data integration [65]. We focus the following discussion on parallel incremental view maintenance—introduced in relational databases by [100]—because it considers a similar setting to ours—both the tables and the view are partitioned across servers. The objective of [100] is to maintain materialized views partitioned on a different key than the join keys in the view definition. Communication is minimized by building indexes on the join attributes at each node. Since arrays are partitioned on dimensions, these techniques are not applicable. [14] proposes an incremental view maintenance method that replicates the updates to all the nodes that contain relevant data. This is similar to the baseline algorithm we improve upon. [7] introduces materialized views in Hadoop and designs replication methods for efficient maintenance. [86] employs key-foreign key constraints to identify diffs and avoid join with the base tables. These works consider only equi-joins—a subclass of array similarity join. The distributed setting considered by [97, 138, 139], the base tables and the view are not partitioned across servers. The goal is to find the optimal join ordering in which to apply the updates. Batch updates—chunks in our case—are shown to be more efficient in a distributed environment by [103]. In the context of array databases, incremental array repartitioning [54] comes the closest to incremental materialized view maintenance. However, the focus is exclusively on repartitioning, not on view maintenance. In [75], algorithms for maintaining k-nn results and spatial joins on continuously moving points are proposed.

### 3.8 Conclusions

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

### 3.9 NP-Hard Proofs

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

#### 3.9.1 Differential View Computation

**Definition 3** Given update triples $U_0 = \{(p, q, \ast)\}$ consisting of array chunks $p, q$ having size $B_p, B_q$ and being located on server $S_p$ and $S_q$, respectively, the differential view computation has to determine how to replicate these chunks such that there exists a server $j$ that contains both $p$ and
\( q, \forall (p, q) \in U_0, \) and guarantees that \( \text{ntwk}[j] \leq K, \forall j \in \{1..N\}, \) where \( K \) is an arbitrary constant. Replicating a chunk \( q \) incurs a cost of \( B_q \) to server \( S_q \)—same for \( p \).

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

**Definition 4** Given a bipartite graph \( G \) with the vertex partitions \( L \) and \( R, \) the constrained bipartite vertex cover (CBVC) problem has to determine whether there exists a vertex cover containing at most \( K_L \) vertices from \( L \) and \( K_R \) vertices from \( R, \) where \( K_L \) and \( K_R \) are arbitrary constants. CBVC is NP-hard [78].

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

### 3.9.2 View Chunk Reassignment

**Definition 5** Given update triples \( U_0 = \{(p, q, v)\} \) consisting of array chunks \( p \) and \( q \) already joined at server \( k (z_{pqk} = 1), \) and view chunk \( v \) located originally at server \( S_v, \) view chunk reassignment has to determine the server \( S'_v \) such that the largest view merge time across the \( N \) servers is minimized. Triples \( (*, *, v) \) have to be moved to the same server for merging.

**Definition 6** Given a machine with \( n \) processors and \( m \) jobs, job \( i \) taking \( J_i \) time to be processed, in multiprocessor scheduling we have to distribute the \( m \) jobs to the multiprocessors such that the latest processing time across multiprocessors is minimized. Multiprocessor scheduling is NP-hard [153].

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

### 3.9.3 Array Chunk Reassignment

**Definition 7** Given \( N \) servers and a list of quadruples \( L = \{p, q, v, s\} \) meaning that if array chunks \( p \) and \( q \) are both on server \( v \) generates a score \( s, \) in array chunk reassignment we have to maximize the overall score across quadruples by assigning \( p \) and \( q \) to servers. The total size of the chunks assigned to a server \( j \) can be at most \( \text{cpu}_{\text{thr}}. \)

**Definition 8** Consider a knapsack with capacity \( W \) and \( n \) items. Each item has size \( w_i \) and value \( v_i. \) We are also given a list of triples \( Q = (i, j, k). \) If item \( i \) and item \( j \) are both packed in the knapsack, we get an additional value \( k. \) In the quadratic knapsack problem, we have to pack the items such that the overall value is maximized. This is an NP-hard problem [87].

59
Reduction 3 Set the number of servers \( N \) as 2, i.e., we have two servers, 0 and 1. Server 1 corresponds to packing an item, while server 0 to not packing it. For each item \( i \), create an array chunk \( i \) with size \( w_i \). Create an additional dummy array chunk 0 with size 0. For each triple in \( Q \), create a quadruple \((i, j, k)\) in \( L \). For each item \( i \), create a quadruple \((i, 0, 1, v_i)\) in \( L \). Set \( cpu_{thr1} \) as \( W \).

3.10 Algorithm Examples

In order to illustrate how the proposed heuristic view maintenance procedure works, we provide examples for each stage based on the input in Figure 3.1.

3.10.1 Differential View Computation

The set of update triples \( U_0 \) is given by the insertions in Figure 3.1 (b). In this example, we consider the following 4 triples \((\Delta A_4, A_1, \ast), (\Delta A_2, A_1, \ast), (\Delta A_2, \Delta A_3, \ast), \) and \((\Delta A_7, A_2, \ast)\) resulting after the random ordering. The size of a chunk \( B_p \) is given by the number of non-empty cells, while its location \( S_p \) is as in Figure 3.1. \( T_{ntwk} \) is set to 4 and \( T_{cpu} \) to 1, respectively. The state of the algorithm when the 4th triple \((\Delta A_7, A_2, \ast)\) is processed is shown on the top part of Figure 3.7. For example, server \( X \) stores chunks \( A_1 \) and \( A_4 \); a replica of chunk \( \Delta A_2 \); does not transfer any chunk, thus \( ntwk = 0 \); and processes two joins, \( \Delta A_4 \bowtie A_1 \) and \( \Delta A_2 \bowtie A_1 \), thus \( cpu = 4 \). The algorithm considers the cost of assigning the join \( \Delta A_7 \bowtie A_2 \) to each of the servers and selects the one having minimum value for \( opt_{now} \)—computed as the maximum between the network and cpu costs across the servers. If the join is assigned to server \( Y \), \( \Delta A_7 \) has to be transferred from \( X \) to \( Y \) for a network cost of \( T_{ntwk} \times B_{\Delta A_7} = 4 \times 1 = 4 \) (incurred by \( X \)) and a cpu cost \( T_{cpu} \times (B_{\Delta A_7} + B_A) = 1 \times (1 + 1) = 2 \) on \( Y \). These costs are combined with the existing cost at \( X \) and \( Y \) to generate a maximum of 4 for \( opt_{now} \). Since this value is the minimum across the three servers – \( opt_{now} = 8 \) on \( X \) and \( Z - \Delta A_7 \bowtie A_2 \) is assigned to \( Y \).

![Figure 3.7: Example for Algorithm 2.](image-url)
### 3.10.2 View Chunk Reassignment

The output of differential view computation – which is the input to view chunk reassignment – is depicted in the top two tables of Table 3.2. It consists of the network and cpu costs for each server and the assignment of joins to servers. In this example, we include only the joins relevant to $V_1$—the first view chunk considered. As the tables show, both the communication and computation are balanced across the servers. The algorithm tries to assign $V_1$ to each of the servers and selects the one incurring the minimum cost—computed as in Algorithm 2. The table at the bottom of Table 3.2 depicts the cost $opt\_now$ corresponding to each server. In the optimal assignment, $V_1$ is moved to server $Y$ together with joins $J_1$ and $J_2$—computed on $X$. The reason for this assignment is the availability of computation resources on $Y$. This is not the case for $X$. While $Z$ has network resources, it does not contain any of the join results required by $V_1$.

<table>
<thead>
<tr>
<th>server</th>
<th>ntwk</th>
<th>cpu</th>
<th>join result</th>
<th>server</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X$</td>
<td>32</td>
<td>36</td>
<td>$J_1 : \Delta A_1 \bowtie A_1$</td>
<td>$X$</td>
</tr>
<tr>
<td>$Y$</td>
<td>36</td>
<td>30</td>
<td>$J_2 : \Delta A_4 \bowtie A_1$</td>
<td>$X$</td>
</tr>
<tr>
<td>$Z$</td>
<td>30</td>
<td>35</td>
<td>$J_3 : \Delta A_2 \bowtie A_1$</td>
<td>$Y$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$V_1 \rightarrow$ transfers</th>
<th>$ntwk'$</th>
<th>$cpu'$</th>
<th>$opt_now$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X$</td>
<td>$J_3$</td>
<td>$Y = 4$</td>
<td>$X = 6$</td>
</tr>
<tr>
<td>$Y$</td>
<td>$J_1, J_2$</td>
<td>$X = 8$</td>
<td>$Y = 6$</td>
</tr>
<tr>
<td>$Z$</td>
<td>$J_1, J_2, J_3$</td>
<td>$X = 8, Y = 4$</td>
<td>$Z = 6$</td>
</tr>
</tbody>
</table>

Table 3.2: Example for Algorithm 3.

### 3.10.3 Array Chunk Reassignment

The input to Algorithm 4 is represented by historical update triples rather than the current update batch. Each of the (array chunk, view chunk) pair appearing in the triples is assigned a score based on their frequency. These are depicted in the left table of Figure 3.8. The other inputs to the algorithm are the size and location of the array chunks as computed by Algorithm 2 (bottom-right table in Figure 3.8) and the assignment of the view chunks computed by Algorithm 3 (top-right table in Figure 3.8). The (array chunk, view chunk) pairs are considered in descending order of their score and the array chunk is assigned to one of the replicas that also contains the view chunk capped by its cpu processing quota $cpu\_thr$. Chunk $A_2$ in pair $(A_2, V_1)$ is assigned to $Y$ because it contains $V_1$ and has sufficient capacity. $A_1$ in $(A_1, V_1)$ is ignored because it is not replicated on $Y$ and later assigned to $X$ when considered in $(A_1, V_2)$. $A_3$ is finally assigned to $Z$ which cannot be assigned any other chunks further because it is at capacity—$cpu\_thr = 0$. 

61
3.11 Additional Experiments

3.11.1 Overall View Maintenance Time

The overall time incurred by optimization and view maintenance across the entire batch of updates is depicted in Figure 3.9. As expected, the benefit of repartitioning is maximized for correlated batches. In this case, reassign is faster than baseline by more than 3X on PTF-25. Reassign always outperforms differential, even in the case of GEO random batches and with the optimization time included. The optimization overhead is marginal in the overall maintenance time compared to the reduction it brings. In the worst case – PTF-5 with real batches – reassign incurs an overhead of 6 seconds in optimization, while achieving a reduction of 300 seconds in overall maintenance time—100 seconds over differential.

3.11.2 Sensitivity to Batch Size

The sensitivity experiments depicted in Figure 3.10 are executed over the PTF-25 view because this has the most complicated shape in the definition. The larger maintenance time permits a clearer investigation of the scalability of the proposed algorithms. Figure 3.10a depicts the maintenance time for update batches consisting of increasing number of chunks. The real update PTF workload is partitioned into batches with exponentially increasing number of chunks—50, 100, 200,
3.11.1 Sensitivity to Batch Size

400, 800, and 1600. The batches are fed into the view maintenance algorithms in this order. As expected, larger batches incur a linear increase in maintenance time. When the number of chunks in a batch is below 200, the difference between the three algorithms is minimal because the number of update triples is small. However, as the size of the batch increases – and the number of update triples – the gap between reassign and the other algorithms increases—it is 200 seconds for a batch with 1600 chunks. While the optimization time also increases linearly with the batch size, it represents an insignificant fraction of the maintenance time – less than 1% – with an absolute value below 3 seconds for 1600 chunks.

3.11.3 Sensitivity to Number of Batches

Figure 3.10b depicts the sensitivity of the view maintenance algorithms as a function of the number of batches for a fixed update workload. In this case, the real PTF workload is divided into batches with the same number of chunks. The goal is to identify the optimal batch size. For a single batch, reassign and differential are identical and superior to baseline due to reduced communication and better load balancing. Reassign exhibits the smallest variance with the number of batches—if one batch is excluded. All the algorithms achieve the smallest maintenance time for 10 batches which is in the middle of the considered range. Baseline and differential suffer a significant increase if the number of batches grows beyond this point. This clearly shows that many small batches are not optimal because of the overhead they incur. However, reassign is able to use a larger number of batches to find a better chunk assignment that compensates for the increased overhead.

3.11.4 Sensitivity to Update Spread

Figure 3.10c depicts maintenance time as a function of the spread of updates over the range of \((ra, dec)\) with a fixed number of batches (10) and sampled chunks per batch (500) that overlap with the range. Spread value 10 corresponds to a rectangle of 10 chunks on \(ra\) and \(dec\)—100 chunks overall. 20 doubles the range of 10 on both dimensions while guaranteeing inclusion—similar for 80. The larger the spread, the least concentrated the updates are. Thus, less sharing is possible which results in longer maintenance time. However, the increase for reassign is smaller in absolute value – 900 compared to more than 1000 – than for the alternatives.
Chapter 4

User-Defined Scientific Data Analysis on Arrays

User-Defined Functions (UDF) allow application programmers to specify analysis operations on data, while leaving the data management tasks to the system. This general approach enables numerous custom analysis functions and is at the heart of the modern Big Data systems. Even though the UDF mechanism can theoretically support arbitrary operations, a wide variety of common operations – such as computing the moving average of a time series, the vorticity of a fluid flow, etc., – are hard to express and slow to execute. Since these operations are traditionally performed on multi-dimensional arrays, we propose to extend the expressiveness of structural locality for supporting UDF operations on arrays. We further propose an in situ UDF mechanism, called ArrayUDF, to implement the structural locality. ArrayUDF allows users to define computations on adjacent array cells without the use of join operations and executes the UDF directly on arrays stored in data files without requiring to load their content into a data management system. Additionally, we present a thorough theoretical analysis of the data access cost to exploit the structural locality, which enables ArrayUDF to automatically select the best array partitioning strategy for a given UDF operation. In a series of performance evaluations on large scientific datasets, we have observed that – using the generic UDF interface – ArrayUDF consistently outperforms Spark, SciDB, and RasDaMan.

4.1 Background

As technology advancements in large-scale scientific experiments, observations, and simulations are generating unprecedented amounts of data, scientists are in need of novel techniques to process the data. Scientific datasets typically contain multi-dimensional arrays and are stored as files in shared disk-based storage systems [143]. Analysis of these large datasets has to be conducted directly on the raw data files, in an in situ manner ¹, because loading the data into database systems that typically assume shared-nothing architecture is a very expensive and cumbersome operation. Furthermore, analysis operations on datasets are different from one run to another. Therefore, it is

¹The ability to process data directly in their native file formats has been referred to as “in situ processing” in data management literature [9, 21].
necessary to allow application programmers to customize their analysis operations through the User-Defined Function (UDF) mechanism. To provide such an extensible analysis capability, we propose a novel UDF abstraction for multi-dimensional arrays and present an in situ system that executes UDF operations over large datasets dramatically faster than the state-of-the-art data management systems.

UDFs have been explored extensively in data management literature [11, 129] and are widely implemented in database servers [29, 112, 129, 177], recent parallel data processing systems [31, 47, 98, 108], as well as specialized scientific data processing systems [15, 16, 23, 34]. The assumption behind UDFs is that most data analysis operations and their input data have a relatively simple relationship. Typically, the relationship is that a single operation is applied to each element of a dataset, e.g., a tuple of a relational table or a cell of an array. In these cases, a user only needs to specify the operation on a single data element, while the underlying system automatically executes the operation over all the elements. To the user, the operation on a single data element is the application logic, while the task to manage the dataset is a support function. However, this support function is often much more complex, especially for large datasets that tend to be processed in parallel. The UDF mechanism allows users to concentrate on the application logic and leave the data management task to the system, which significantly improves productivity of the users.

A critical limitation of most existing UDF implementations is that they typically allow users to define an operation only on a single element. However, most real-world data analysis tasks, such as computing the moving average of a time series or the vorticity of a flow field [30] (see detailed examples in Section 4.1), require the values of not just a single element, but also many of its neighbors. This dependency on adjacent elements is referred as structural locality in the literature [106]. To alleviate this limitation and to support a flexible UDF mechanism, some DBMS systems allow UDFs on an entire data table [112, 162, 177], while MapReduce systems [47] allow users to define an operation on a set of related elements in the reduce stage. However, this flexibility comes at the expense of tedious aggregation that is required to build the input of UDF properly. Moreover, since a data element is required by multiple neighbors in real applications, the data management system often replicates each element multiple times during execution, which degrades the performance significantly. Furthermore, the reduction operations are performed uniformly on all neighboring elements involved, while real applications typically need to perform different operations on different neighboring elements. In short, there are many real-world operations that are not well-supported by the existing UDF mechanisms, i.e., these operations are hard to express and slow to execute.

In this chapter, we present the design and implementation of ArrayUDF, a novel in situ UDF abstraction optimized for multi-dimensional arrays. ArrayUDF supports generalized structural locality by allowing users to define operations not only on the local value of an array cell, but also on the values of its neighbors. Meanwhile, using ArrayUDF to define an operation for a given cell, users can define a specific operation for each neighbor of this cell in the UDF. ArrayUDF automatically identifies the optimal array chunking strategy that guarantees the efficient execution of each operation. In summary, this chapter includes:

- Introduction of ArrayUDF, the first UDF mechanism for multi-dimensional arrays with a generalized structural locality support. By providing a novel operator to express the relative position of a neighbor, ArrayUDF allows users to define complex analysis operations directly on arrays. Compared with the computing model of MapReduce, ArrayUDF uses a single step
to complete the task for both Map and Reduce operations.

- Implementation of ArrayUDF and its processing system using the SDS framework [51], a database-like system for high-performance computing (HPC). ArrayUDF executes in an in situ manner [9, 21] for efficiency. ArrayUDF works directly on raw scientific file formats, e.g., HDF5 [152], where the files are stored in parallel file systems.

- Algorithms to dynamically identify the optimal chunking strategy and build a “ghost zone” for a given UDF based on the data access cost. Compared with the chunking strategies for shared-nothing array databases, these unique features of ArrayUDF enable it to work efficiently on dynamically-scheduled resources in an HPC environment as well as on the large scientific datasets stored on shared-storage systems as files.

- An analytical performance model for providing theoretical support to justify the chunking strategies of ArrayUDF and also for tuning ArrayUDF to different array organizations on disk.

- Evaluation of ArrayUDF using both synthetic and real scientific datasets on a Cray XC30 supercomputer and also a commodity Linux server. We have compared ArrayUDF with SciDB [23] and RasDaMan [16] – specialized systems for multi-dimensional array processing – and Spark, the state-of-the-art MapReduce system that supports generic UDFs. Our evaluations show that ArrayUDF is considerably faster than existing alternatives. For instance, using the generic UDF interface, ArrayUDF is up to 2070 times faster than Spark to complete a real-world data analysis.

In Section 4.1, we present several motivating examples from real scientific applications. In Section 4.2, we introduce ArrayUDF and its design and implementation. We present performance evaluation of ArrayUDF in Section 4.3. In Section 4.4, we discuss related research efforts. We conclude the chapter in Section 4.5.

Here we introduce several real-world applications that motivate this research. Our work can be the building blocks for advanced data mining algorithms [126] and systems, e.g., TensorFlow [4].

### 4.1.1 Motivating Examples

**Example 1: Moving average based smoothing for time series.** A variety of applications produce 1D time series data. Examples include collection of temperature periodically on a flux tower in climate observations and daily stock prices in finance industry. These time series datasets usually contain two parts: a meaningful pattern (e.g., seasonal trend) and a superimposed noise with limited scientific meaning. Moving average based smoothing is widely used to extract the meaningful patterns. Specifically, at a time $t$, moving average based smoothing has to determine the average of observed values that are close to this particular time, as shown in the following equation:

$$V_t' = \frac{w_{t-k} V_{t-k} + \ldots + w_{t-1} V_{t-1} + w_0 V_t + \ldots + w_{t+m} V_{t+m}}{k + m + 1},$$  

(4.1)

where $V_t$ is the observed value, $w_t$ the weight and $V_t'$ the smoothed value. $k$ and $m$ are the steps before and after $t$.  

66
Example 2: Vorticity computation. S3D is a high-fidelity direct numerical simulation (DNS) designed to capture key turbulence-chemistry interactions in a combustion engine [30]. A key variable related to the turbulent motion is vorticity. It defines the local spinning motion around a given location. To simplify the description, we give the z component of the vorticity at a point \((i,j)\):

\[
\zeta_{i,j} = \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y} = \frac{u_{i,j+1}-u_{i,j-1}}{2\Delta x} + \frac{v_{i+1,j}-v_{i-1,j}}{2\Delta y},
\]

where \(u\) and \(v\) are the flow velocity (i.e., 2D arrays) on the \(x\) and \(y\) axes, respectively, and \(\Delta x\) and \(\Delta y\) are the constant differences. Note four neighbors per cell are required in this computation.

Example 3: Peak detection. Mass-spectrometry imaging (MSI) is an essential technology required by biological sciences to understand metabolism [132]. In MSI, the mass-to-charge ratio (m/z) is a variable of interest which is usually a 3D array for a single object (e.g., brain tissue sample). A key data analysis task in MSI is to find peaks of m/z via calculating the gradient of each point, typically through the Laplacian operator [119]. The Laplacian for a given point \((i, j, k)\) is defined as:

\[
g_{i,j,k} = 6 \times v_{i,j,k} - (v_{i,j,k+1} + v_{i,j,k-1} + v_{i,j+1,k} + v_{i,j-1,k} + v_{i+1,j,k} + v_{i-1,j,k}),
\]

where \(v\) is the m/z value and \(g\) denotes the gradient value. Note that seven values of \(v\) are needed for each value of \(g\).

Example 4: Trilinear interpolation. Plasma physics simulations, such as VPIC, are used to study magnetic reconnection [25]. During a simulation, magnetic field values are computed at mesh points. However, data analysis requires to find the magnetic field at the location of each particle. Generally, the magnetic field at the location of a particle is interpolated from the nearest eight adjacent magnetic values.

Key observations from the examples above:

- The computation follows a stencil [18, 109] pattern. In general, a new array \(B\) is computed from an existing array \(A\), where the value of \(B\) at location \((i, j, k)\) is determined not only by \(A(i, j, k)\), but also by its neighbors. This is called structural locality in a previous work [106].

- The neighbors of \(A(i, j, k)\) do not go through the same operation. Previous work – such as MapReduce [47] and GLADE [31] – generally assume that a uniform reduction or aggregation operation is sufficient.

- There is a variety of analysis operations for different applications. The best option for implementing all of them is to follow the UDF approach to support the common data management operations, while allowing users to define custom operations on data.
4.1.2 Research Challenges

SciDB, RasDaMan, and AML [106] implement the “window” operator, which supports some form of structural locality. However, the operations on the values within a window are generally a reduction operation, while the above examples show a variety of operations. Furthermore, the “window” typically has to be a rectangular subdomain, e.g., a $2 \times 2$ square, while the above examples contain more complex definitions of neighborhood. Additional flexibility is needed in specifying both the operations and the neighborhood. While UDFs provide a general framework to express custom analysis operations, they have to address the following challenges in order to be applicable to structural operations defined over arrays:

- How to define neighborhood cells involved in a UDF operation? These definitions have to be compact, easy to construct, and – at the same time – efficient to evaluate.

- How to develop UDFs that support in situ array data analysis? As most scientific datasets are stored in file formats such as HDF5 and netCDF [143], a new UDF mechanism has to work directly on these files without loading the data into a separate DBMS.

- How to effectively partition the data and computation on parallel computing systems? As each array cell is needed at different computation steps, the data partition may require overlapping. Maintaining load-balance and reducing overlap is critical to achieve good overall performance.

4.2 ArrayUDF Approach

To address the identified challenges, we propose a UDF system named ArrayUDF. With ArrayUDF, we extend the expressiveness of structural locality to allow users to easily define operations on adjacent cells of an array and to perform data management tasks efficiently for supporting these user-defined operations. ArrayUDF is also capable of identifying the minimum portion of an array accessed on each process (e.g., CPU core) and operate on that portion of the array data stored in files, without loading the entire array into the system. This optimization is possible because the array syntax used by ArrayUDF to describe the operations provide a clear mechanism to identify the relevant cells and the optimal data partition can be determined analytically.

**ArrayUDF computational model.** For comparison, we first introduce the computational models of relational databases and MapReduce systems. We use $f$ to denote a user-defined function.

For relational tables $T$ and $T'$, the generic UDF model is:

$$t' \leftarrow f(t)$$

(4.5)

where $f$ is applied to each tuple $t \in T$ and $t'$ represents the tuple in the output table $T'$. There is a one-to-one mapping between input and output tuples. Aggregate UDFs allow for more input tuples to determine an output tuple. However, it is done by grouping on the values of some attributes, i.e., the SQL GROUP-BY operator.
In the MapReduce paradigm, the input of a UDF is a \((key, value)\) pair, or \((k,v)\) for short. MapReduce has two components – \(Map\) and \(Reduce\) – which can be formally expressed as:

\[
\begin{align*}
Map &: 
\{ (k_i, v_i) \mid i \in [1, m] \} \leftarrow f_1((k, v)) \\
Reduce &: 
\{ (k'_i, v'_i) \mid i \in [1, p] \} \leftarrow f_2((k', v_i) \mid i \in [1, n])
\end{align*}
\]

(4.6)

where \(m, n, p \in \mathbb{N}\), \(\mathbb{N}\) is the natural number set, \(f_1\) is an enhanced UDF that implements a one-to-many mapping from the input pair to an intermediate set of key-value pairs and \(f_2\) is a SQL GROUP-BY AGGREGATE identical to the relational aggregate UDF. Key \(k'\) is the grouping parameter in \(f_2\). Value \(v'\) is generated by \(f_2\) through uniformly applying a single operator – such as \(SUM\) – to each \(v_i\).

The computational model of the ArrayUDF is defined on two \(d\)-dimensional arrays, \(A\) and \(A'\) \((d \in \mathbb{N})\). The cell \(c'\) at coordinate \((i_1, i_2, \ldots, i_d)\) in \(A'\) is computed by a stencil \(S\) of the cell \(c\) at the same coordinate in \(A\). Theoretically, the stencil \(S\) is a set of array cells which have structural locality. Specifically, for the cell \(c\) at coordinate \((i_1, i_2, \ldots, i_d)\), \(S = \{c_{i_1 + \delta_1, i_2 + \delta_2, \ldots, i_d + \delta_d} \mid \forall j \in [1, d], \delta_j \in [L_j, R_j]\}\), where \(L_j \in [-i_j, 0]\), \(R_j \in [0, N_j - i_j]\), and the \(N_j\) is the size of the \(j\)th dimension. Obviously, the \(\delta_1, \delta_2, \ldots, \delta_d\) are relative distances from the cell \(c\). For simplicity, each cell in \(S\) is expressed as \(s_{\delta_1, \delta_2, \ldots, \delta_d}\), which is \(c_{i_1 + \delta_1, i_2 + \delta_2, \ldots, i_d + \delta_d}\). With these notations, the formal computational model of ArrayUDF is:

\[
c'_{i_1, \ldots, i_d} \leftarrow f\left(\{s_{\delta_1, \ldots, \delta_d} \mid \forall j \in [1, d], \delta_j \in [L_j, R_j]\}\right).
\]

(4.7)

Distinct from the UDF models of relational databases and of MapReduce, in the model of ArrayUDF, the function \(f\) has a stencil \(S\) as input. A stencil \(S\) allows ArrayUDF to express any neighborhood shape implicitly via relative distance. This is also different from AML [106], where a shape parameter is required to express neighbor cells. Meanwhile, users can specify different operators on different cells of \(S\) within the UDF. This distinguishes ArrayUDF from most existing aggregate UDFs [31], where a single aggregate operator is applied onto all values. In the relational model, this functionality requires a chain of self-joins having cardinality equal to the number of cells in the stencil. In MapReduce, the self-joins are substituted by the one-to-many replications in the \(Map\) stage. In general, when the size of \(S\) is equal to one, ArrayUDF is identical to the relational and Map UDFs. Otherwise, ArrayUDF is similar to relational aggregates and the \(Reduce\) function. In summary, ArrayUDF eliminates the shape operators for computing a stencil \((S)\) and allows a more concise definition of UDFs for arrays.

**System overview.** Towards implementing the computational model of ArrayUDF, we introduce its key software components:

- **ARRAY** is a data structure that encapsulates the multi-dimensional array stored in files. This is the primary object a user interacts with. **ARRAY** implements the function \(Apply\) to execute the UDF defined by the user. On a parallel computing system, each process creates its own instance of the **ARRAY** object with the same arguments, and invokes the same UDF with the function **Apply**. Moreover, **ARRAY** has functions to partition the multi-dimensional array automatically, to build the necessary overlapping regions (known as ghost zones), and

69
to divide the computation among the processes. This partitioning method is guided by a theoretical analysis to be discussed in Section 4.2.3. More details on \texttt{ARRAY} are reported in Section 4.2.2.

- The data structure \texttt{STENCIL} represents an array cell and its neighborhood cells in relative coordinates. As defined earlier, \texttt{STENCIL} is a relative coordinate-based notation that allows users to describe the operations to be performed on each neighbor separately. Previously used notations for “shape” and “window” demand all neighbors to be described in a collective form, which limits aggregation operations used in scientific data analysis. In contrast, our relative coordinate-based notation is more flexible. Users define their UDFs with these \texttt{STENCILS}. In C++ syntax, this relative coordinate is expressed using the parenthesis operator of the \texttt{STENCIL} object. More details about \texttt{STENCIL} are in Section 4.2.1.

ArrayUDF is currently implemented as part of the Scientific Data Services (SDS) framework [51], which provides the basic I/O drivers for reading and writing the data from parallel file systems. We implement \texttt{ARRAY} and \texttt{STENCIL} as C++ classes. We show an example of using ArrayUDF in Fig. 4.1, where “MyAvg” is a UDF to compute the average value using four adjacent cells. “MyAvg” is executed by calling the \texttt{Apply} function of an \texttt{ARRAY} instance within the main function. The template feature is used to support different data types, e.g., float and double.

\subsection*{4.2.1 \texttt{STENCIL} Design Considerations}

The \texttt{STENCIL} data structure represents an array cell and its neighbors needed for a single invocation of the UDF. \texttt{STENCIL} plays a role similar to a tuple in a database or a key-value pair concept, working as the input to the UDF. It is more flexible than existing concepts such as “window” and “shape” in two ways. A \texttt{STENCIL} can be used to define more complex neighborhoods than “window” and “shape” and it allows the user to specify a different operation on each of the neighboring cells. This flexibility allows a much wider variety of analysis operations to be defined.

Since modern CPUs can carry out many arithmetic operations quickly, we anticipate that the complexity in arithmetic operations is less dominant in the overall performance of a UDF than the cost of data accesses. As with “window” and “shape,” we expect operations defined on our relatively compact stencils to access a small number of neighbors and, therefore, can be carried out efficiently, while operations involving a large number of neighbors, no matter in the form of a “window”, a “shape”, or a “stencil”, require more data accesses and take a longer time to complete. In short, we expect the flexibility to address individual neighboring cells in a UDF not to impose a significant cost on its own.

To understand the relative coordinates used in the definition of a \texttt{STENCIL}, it is useful to visualize the coordinates of a multi-dimensional array to be a set of mesh points in space and each array to be an attribute of a mesh point. For example, one set of attributes might be their location in space, (e.g., $x$, $y$, and $z$ dimensions), and another set might be temperature, pressure, and concentrations of some chemical species. The analysis examples given in earlier sections follow this basic schema and a typical analysis function computes $B(i, j, ...)$ from $A(i, j, ...)$ and its neighbors. Furthermore, the neighbors are of a fixed combination of offsets around $A(i, j, ...)$. In applied math, the pattern formed by these offsets (positions relative to a center $(i, j, ...)$) is known as a “stencil”. ArrayUDF
A UDF on a stencil containing four neighborhood cells

```cpp
float MyAvg(Stencil<float> &s) {
    return (s(1,0)+s(-1,0)+s(0,-1)+s(0,1))/4;
}
```

```cpp
int main() {
    vector<int> cs(2)= {2,2}; // Chunk size
    vector<int> gs(2)= {1,1}; // Ghost zone size
    Array<float> A("d2d.h5", cs, gs);
    Array<float> B("d2davg.h5");
    // Run UDF code using Apply function
    // Store the result in B
    A->Apply(MyAvg, B);
    ... // Other operations on B or A
}
```

Figure 4.1: An example using ArrayUDF to compute user-defined average, i.e., “MyAvg”, on a 2D array A, stored in a HDF5 file, named “d2d.h5”. The results are stored in array B, which has the same dimensions as that of A and is stored in file “d2davg.h5”. Given different UDF operators, B can have different dimensions from A. The user-specified chunk size (cs) and ghost zone size (os) are used to support parallel processing. Both cs and os are optional and can be determined by ArrayUDF automatically.

uses a syntax implemented as the parenthesis operator of C++ to allow each cell in the stencil to be explicitly named. For example, in a 2D case, $S(0,0)$ refers to the “center” of the stencil $A(i,j)$, and $S(1,0)$ refers to the neighbor $A(i+1,j)$.

The use of relative coordinates in the UDF allows the users to define the operations without mentioning the coordinates ($i, j,...$). When the function is actually evaluated, we need to convert these relative coordinates back to the absolute coordinates in order to access the specific cells. In this context, the use of the relative coordinates also allows each execution thread to work on its own portion of the global mesh as illustrated in Section 4.2.2 and Fig. 4.6.

4.2.2 Array Design Considerations

The Array class is a high-level abstraction and representation for multi-dimensional arrays that contains a group of functions for run-time tasks. Users employ Array to represent an array stored either in memory as std::array or on disk as an HDF5 dataset. For this work, the key function of Array is Apply, which executes a UDF. Behind this function, we are able to implement a number of data management techniques for parallelizing the execution of the UDF automatically through partitioning the global mesh into suitable sub-domains and overlapping the sub-domains through ghost zones.

4.2.2.1 Parallel Processing with Dynamic Chunking

A typical approach for parallel processing of a large problem is to divide the problem into smaller partitions or chunks. Following this general practice of data management systems, we also divide the evaluation of the UDF by partitioning data. We observe that an invocation of UDF requires access to values near ($i, j,...$). Therefore, it is essential for us to keep the neighbors close
to each other as much as possible. This basic requirement is similar to many parallel computing applications [90], which allows us to borrow a number of techniques for designing an efficient data partitioning algorithm. Our overall approach can be viewed as partitioning (chunking) the mesh defined by the multi-dimensional coordinates into chunks, and then map each chunk to a parallel processing element (PE). We assume that each chunk is small enough to fit into the available memory on a PE. A processing element is responsible for producing the output array belonging to the chunk. To complete this task, it not only needs to access the corresponding chunk of the input data array, but also some extra portion of the array, which we call ghost zone (as illustrated in Fig. 4.2). Since ArrayUDF is designed to process data directly on raw data files, there is no data pre-processing step (e.g., loading data into the system) for it to figure out an efficient chunking strategy in advance. We need to perform all the chunking related decisions (e.g., chunk size) dynamically. Moreover, for a given chunk, it is necessary to consider its logical view (i.e., shape) and its physical view (i.e., data layout on storage) as they affect the performance of reading the data.

Chunk management overview. In order to maintain load balance among processing elements, ArrayUDF attempts to keep the size and the shape of chunks similar. An example of chunking in ArrayUDF is shown in Fig. 4.2, where four chunks are created. Each chunk has a dynamically assigned ID (i.e., cid in the example). The ID is calculated using the row-major ordering from the coordinates of chunks, which allows ArrayUDF to identify a chunk quickly.

**FUNCTION ArrayUDFChunk** *(M, P, (N0,...,Nd))*

*M*: the available memory size per process;

*P*: the total number of processes;

(N0,...,Nd): the size of an array for *d* dimension

1: \( S = \min (M, (\prod_{i=1}^{d} N_i)/P) \)
2: if general chunking then
3: \( c_1 = w, c_2 = w, \ldots, c_d = w \)
4: end if
5: if layout-aware chunking then
6: for \( i \in (d, d-1, \ldots, 2) \) do
7: \( c_i = S \% N_i; S = S / N_i \)
8: end for
9: end if
10: \( c_1 = S \)
11: return \( (c_1, c_2, \ldots, c_d) \)

**Figure 4.3**: The method for selecting chunking parameters.

**Optimal chunking strategy selection.** A key challenge to support dynamic chunking is to decide the optimal chunking parameters, including chunk size, chunk shape, and chunk layout. The
chunk size is the number of cells within a chunk, the shape is its logical view, and the layout is its physical data layout on disk. To address this challenge, ArrayUDF provides two chunking strategies: a general chunking and a layout-aware chunking, as shown in the pseudocode in Fig. 4.3. We describe the theoretical reasoning for these strategies in Section 4.2.3. In the following, we describe the high-level idea and concrete applications.

- To find an optimal and general chunking, we assume that each array cell is accessed separately. Such an assumption guarantees that the chunk layout has no impact on the performance of accessing a chunk. In other words, we consider it as an average case for different chunk layouts. Users can choose this chunking strategy when the layout of the array on disk is unknown. As shown in line 1 in Fig. 4.3, the chunk size in this strategy is set to be as large as possible to fit in the memory of each processing element to reduce the startup overhead of I/O as well as to assign at least one chunk to each process to maximize parallelism. In terms of the chunk shape, ArrayUDF chooses a square shape (lines 3 and 4 in Fig. 4.3) to minimize the number of ghost cells for each chunk. A formal analysis of this strategy is given in Section 4.2.3 and is illustrated in Fig. 4.4, where the square chunk reads 12 ghost cells but the non-square one reads 20 ghost cells.

- To identify an optimal and layout-aware chunking, we take the row-major layout as an example because it is popular in most array data formats. A storage system typically organizes the elements of a multi-dimensional array in a linear order based on their coordinates. With the row-major layouts, the 1st dimension is the slowest varying dimension and the last dimension is the fastest varying dimension. In this case, ArrayUDF chooses the chunk whose layout on disk is as contiguous as possible by maximizing the fast-varying (or higher) dimensions for a chunk. In other words, ArrayUDF chooses the chunk size based on the linearized organization of an array in row-major order (lines 7 to 9 in Fig. 4.3). We also consider the memory limit and the parallelism (line 1 in Fig. 4.3). The impact of reading extra ghost cells is not considered here because the cells from the chunk and the ghost zones often form a single or few contiguous reads as illustrated in Fig. 4.5. More detailed analysis is given in Section 4.2.3.

![Figure 4.4: An example showing the number of ghost cells for different chunk shapes.](image)

### 4.2.2.2 Dynamic Ghost Zone Building Using a Trial Run

In most use cases, the programmer who develops the UDF can determine the thickness of the ghost zone and therefore can provide the information to the ARRAY object to help with dynamic data chunking. However, it is possible that the size of ghost zones are unknown a priori, for example
Figure 4.5: An example showing the layout-aware chunking for a $4 \times 9$ array. On the left, a chunk occupies more than one row. On the right, a chunk occupies a partial row.

**FUNCTION** $\mathbf{A} \rightarrow \text{Apply}(P_{udf}, B)$

$P_{udf}$ : pointer to user-defined function. $A$ : an array to Apply $P_{udf}$. $B$ : result array.

$A \rightarrow cs$ : chunk size. $A \rightarrow gs$ : ghost zone size. $A \rightarrow P$ : the total number of processes. $A \rightarrow R$ : the rank of current process.

1: if $A \rightarrow cs == \text{NULL}$ then
2: Determine the chunk size $A \rightarrow cs$, as discussed in Section 4.2.2.1 \hspace{1cm} //No user-defined chunk size
3: end if
4: if $A \rightarrow gs == \text{NULL}$ then
5: Issue a trial-run to get $A \rightarrow gs$, as discussed in Section 4.2.2.2 \hspace{1cm} //No user-defined ghost zone size
6: end if
7: $cid = A \rightarrow R$; \hspace{1cm} // the ID for the first chunk is equal to the rank of current process
8: while ($cbuf = \text{LoadNextChunk}(cid, A \rightarrow gs, A \rightarrow cs)$) != NULL do
9: for each cell $c$ within the chunk $cbuf$ do
10: $\text{cell}_{udf} = \text{Stencil}(c, A \rightarrow gs, A \rightarrow cs, cbuf)$ \hspace{1cm} //Initialize a Stencil to represent the real cell $c$
11: $B = P_{udf}(\text{cell}_{udf})$ \hspace{1cm} //Run the UDF function on the Stencil
12: end for
13: $cid = cid + A \rightarrow P$ \hspace{1cm} //Next chunk in round-robin manner
14: flush $B$ to disk if necessary
15: end while

**FUNCTION** $\text{LoadNextChunk}(cid, A \rightarrow gs, A \rightarrow cs)$

1: Obtain the top-left corner coordinate $c_{tl}$ and the below-right corner coordinate $c_{br}$ from $cid$ \hspace{1cm} //See Section 4.2.2.1
2: $c_{tl} = c_{tl} - A \rightarrow gs$; $c_{br} = c_{br} + A \rightarrow gs$ \hspace{1cm} //expand chunk with ghost cells
3: if checking array boundary fails then
4: return NULL
5: end if
6: Read all cells from $c_{tl}$ to $c_{br}$ within $A$ into buffer $cbuf$
7: return $cbuf$

Figure 4.6: Apply algorithm in ArrayUDF. We use “$\rightarrow$” symbol to denote the components of $A$. For example, $A \rightarrow \text{Apply}$ is the Apply method of $A$ and $A \rightarrow cs$ is the chunk size metadata of $A$. 

74
when the UDF source code is not accessible to the user. In such cases, ArrayUDF provides a mechanism based on a “trial run” to determine the ghost zone size. During the trial run, ArrayUDF gathers all the relative coordinates used in the UDF. We assume the relative coordinates do not depend on the values of the input array, and, therefore, only perform the trial run on a small number of cells. At this time, we only perform this trial run on the first data point on each PE. After the trial run, ArrayUDF chooses the maximum absolute value among the gathered relative coordinates for a dimension as the number of ghost cells of this dimension. Therefore, different dimensions can have a different number of ghost cells. For a certain dimension, the current ArrayUDF implementation requires that both directions have the same number of ghost cells. This reduces the number of input parameters needed by ArrayUDF, but it also causes reading unnecessary ghost cells. However, as observed in most example applications, they require the same number of cells in both directions of a given dimension. We can easily extend ArrayUDF to set the sizes of a ghost zone for different directions of a dimension. Sometimes, it is impossible to determine the ghost zone size using the trial run. In this situation, ArrayUDF uses the default ghost zone size. Moreover, users can specify a default ghost zone size when initializing Array object, as shown in Fig. 4.1.

4.2.2.3 Putting It All Together: the Apply Algorithm

The Apply function of Array is the entry point to execute a UDF and it also contains a skeleton of our runtime system. The pseudocode of Apply is outlined in Fig. 4.6. Overall, the pseudocode shows that the UDF \( P_{udf} \) processes array \( A \) and stores the results in array \( B \). As shown in Fig. 4.1, users can provide their own chunk size \( (cs) \) and ghost zone size \( (gs) \), both of which are therefore considered as the input. To support parallel processing, each process (PE) initializes an Array instance by itself and follows the same algorithm without any communication. During the initialization, each process obtains the total number of processes \( P \) and the rank \( R \) of itself among all processes using certain library calls such as MPI. In the following, we describe how Apply works on a single process.

From line 1 to line 3, the Apply algorithm determines the chunk size and ghost zone size, if a user does not provide them. The methods for determining these two parameters are discussed in Section 4.2.2.1 and Section 4.2.2.2. Then, the algorithm starts to read a chunk into memory for processing (lines 4 and 6). The ID of the first chunk to be read by a process is equal to the rank of the process, which permits each process to have different chunks to process concurrently. Reading the chunk is performed by the LoadNextChunk function, which first converts the chunk id to the coordinates of the top-left corner and bottom-right corner of a chunk. Then, it extends the coordinates of the chunk to include the ghost zone. Finally, the LoadNextChunk function reads the chunk into memory if the coordinates are valid.

Once the Apply algorithm receives the chunk data from LoadNextChunk, it starts to apply the UDF (i.e., \( P_{udf} \)) on each cell of the chunk (lines 9 – 12). Specifically, a new instance of Stencil is initialized. Then, \( P_{udf} \) is called with the new Stencil instance. The returned result of the UDF is stored in \( B \). After a PE processes a chunk, it reads the next chunk to process. The ID of the next chunk is the current id plus the total number of processes (line 13). The data stored in \( B \) can be flushed to persistent storage, depending on the available memory space.
4.2.3 Analytical Model for ArrayUDF

In this section, we present an analytical model to characterize the performance of ArrayUDF, with the primary goal of deriving a chunking strategy. We first build a generic model without considering array layout and then adapt the model for specific array layouts. The notations used in the model are shown in Table 4.1.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_i$</td>
<td>Chunk size in the $i$th dimension, $i \in (1, \ldots, d)$</td>
</tr>
<tr>
<td>$c$</td>
<td>Number of cells in a chunk, $c \equiv \prod_{i=1}^{d} c_i$</td>
</tr>
<tr>
<td>$C$</td>
<td>Number of chunks $C \equiv \prod_{i=1}^{d} \lceil N_i/c_i \rceil$</td>
</tr>
<tr>
<td>$\delta$</td>
<td>Size of a ghost zone</td>
</tr>
<tr>
<td>$e$</td>
<td>Number of elements in a chunk plus its ghost zones</td>
</tr>
<tr>
<td>$G$</td>
<td>Number of elements in all ghost zones</td>
</tr>
<tr>
<td>$d$</td>
<td>Rank of an array (number of dimensions)</td>
</tr>
<tr>
<td>$M$</td>
<td>Memory size of a single process</td>
</tr>
<tr>
<td>$N_i$</td>
<td>Size of the $i$th dimension, $i \in (1, \ldots, d)$</td>
</tr>
<tr>
<td>$N$</td>
<td>Number of array elements, $N \equiv \prod_{i=1}^{d} N_i$</td>
</tr>
<tr>
<td>$P$</td>
<td>Number of parallel processes</td>
</tr>
</tbody>
</table>

Table 4.1: Notations used in the chunking strategy model

Execution of a computation function typically contain three overheads: computation time, communication time, and I/O time. In the following analysis, we will ignore the computation and communication costs. By design, each process in ArrayUDF can evaluate the UDF in parallel without inter-process communication. We will, therefore, assume that there is no communication overhead. The computation time for each element of the output array can be reasonably represented as a constant, independent of the logical array layout and the physical chunk layout; therefore, as long as there are enough array cells to be divided evenly among processes, the computation cost can be divided evenly and will not depend on how we partition the chunks. Similarly, selection of the layout of the chunks does not affect the computation time. To further simplify the analysis, we also assume the ghost zones are of the same width in each direction and every dimension.

Overall, ArrayUDF divides the evaluation into chunks whose shape and size is determined analytically. On each process, the evaluation proceeds independently. From Fig. 4.6, we see that the key I/O cost is to read a chunk of the input array and the ghost zones. Let $c$ and $e$ denote the number of elements in a chunk and the extended region including the chunk and its surrounding ghost zones, as shown in Table 4.1. Given the size of a chunk $(c_1 \times c_2 \times \ldots \times c_d)$, the total volume of data in the chunks is $c = \prod_{i=1}^{d} c_i$. To evaluate the size of $e$, we need to know how many ghost zones are present. For simplicity, if a part of the ghost zone is available, we will count it as present. As shown in Figure 4.4, it is possible for a chunk to have two ghost zones along each dimension $i$. However, those ghost zones might not be present if the expected ghost zones are outside of the extent of the array\(^2\), as shown in Fig. 4.2. For dimension $i$ with chunk size $c_i$, the $N_i$ points are divided into $\lceil N_i/c_i \rceil$ chunks, which creates need for $2(\lceil N_i/c_i \rceil − 1)$ ghost zones along this dimension. Given the surface area perpendicular to this dimension to be $\prod_{j \neq i} N_j$, the total volume of ghost zones along dimension $i$ is $2\delta(\lceil N_i/c_i \rceil − 1) \prod_{j \neq i} N_j$, and the total volume of data in ghost zones over

\(^2\)It is possible to have a periodic boundary condition, which has a different requirement on ghost zones.
all dimensions is:
\[
G = 2\delta \sum_{i=1}^{d} \left( \left\lfloor \frac{N_i}{c_i} \right\rfloor - 1 \right) \prod_{j \neq i} N_j
\]  
(4.8)

The average number of elements for processing a chunk is:
\[
e = c + G/C
\]  
(4.9)

### 4.2.3.1 Generic Performance Model

In this performance model, we assume the layout of the array dimensions is unknown or the time to read an arbitrarily shaped subarray is strictly a linear function of the number of elements in the subarray. Thus, the time to read a chunk plus its surrounding ghost zones is given by:
\[
t_{io} = \alpha_0 e + \beta_0
\]  
(4.10)

For \(C\) chunks and \(P\) processes, each process has at most \(\left\lceil \frac{C}{P} \right\rceil\) chunks and the maximum read time is:
\[
T_{io}^{ge} = (\alpha_0 e + \beta_0) \left\lceil \frac{C}{P} \right\rceil
\]  
(4.11)

The selection of the optimal chunk size can be formulated as an optimization problem:
\[
\begin{align*}
\min_{c_1, c_2, \ldots, c_d} & \quad T_{io}^{ge}(c_1, c_2, \ldots, c_d) \\
\text{s.t.} & \quad (1) \ c_i \leq N_i, 1 \leq i \leq d; \quad (2) \ \prod_{i=1}^{d} c_i \leq M; \quad (3) \ C \geq P
\end{align*}
\]  
(4.12)

Constraint (2) guarantees that each chunk fits within the available memory while constraint (3) enforces that each process has at least one chunk to work on. Given a large array – the case we are interested in – constraint (3) is easily satisfied and constraint (2) can be turned into:
\[
\prod_{i=1}^{d} c_i = M
\]  
(4.13)

To generate an analytical solution for the above optimization problem, we further assume the ceiling operator can be removed from all the above expressions, which leads to:
\[
T_{io}^{ge} \sim \left( \alpha_0 \sum_{i=1}^{d} c_i + \frac{2\delta \sum_{i=1}^{d} \left( \left\lfloor \frac{N_i}{c_i} \right\rfloor - 1 \right) \prod_{j \neq i} N_j}{\prod_{i=1}^{d} N_i / c_i} \right) + \beta_0 \frac{\prod_{i=1}^{d} N_i / c_i}{P}
\]
\[
= \frac{\alpha_0 N}{P} + \frac{2\alpha_0 \delta N}{P} \sum_{i=1}^{d} \left( \frac{1}{c_i} - \frac{1}{N_i} \right) + \frac{\beta_0 N}{PM}
\]  
(4.14)

In the above expression for \(T_{io}^{ge}\), the only term that is affected by the choices of \(c_i\) is the expression \(\sum 1/c_i\) (multiplied by a positive constant). Given the constraint in Eq. 4.13, the minimal value of \(T_{io}^{ge}\) is obtained with \(c_1 = c_2 = \ldots = \sqrt[d]{M}\), which minimizes the total number of ghost cells \(G\). In
short, when there are no preferred dimensions, the chunks should be as large as they can fit in the available memory and have each side of the same size.

4.2.3.2 Layout-Aware Performance Model

When the layout of an array in a file is known, typically there are preferred dimensions for partitioning the array into chunks. The key reason for this is that the sequential read operations are significantly more efficient than random reads. In many cases, a multi-dimensional array is organized in row-major ordering. Hence, there are significant advantages to partition along the slowest varying dimension:

- Under the row-major ordering, when partitions are based on the slowest varying dimension, each partition can be read with a single sequential scan operation.
- When a dimension is fully contained in a chunk \((c_i = N_i)\), there is no ghost zone for the dimension, which reduces the number of read operations needed for ghost cells.
- When the partition is only along the slowest varying dimension, the ghost cells follow the chunk in the file, which allows the ghost cells to be read with the chunk in a single sequential scan.

Therefore, partitioning along the first dimension (i.e., the slowest varying dimension) is highly desirable as long as the array can be evenly divided onto the \(P\) processes. In this case, Eq. 4.8 turns into:

\[
G = 2\delta([N_1/c_1] - 1) \prod_{j=2}^{d} N_j \sim 2\delta N(1/c_1 - 1/N_1)
\]

This leads to the following expression for the read time:

\[
T_{\text{io}}^1 \sim \frac{\alpha_1 N}{P} + \frac{2\alpha_1 \delta N}{P} \left( \frac{1}{c_1} - \frac{1}{N_1} \right) + \frac{\beta_1 N}{PM}
\]

(4.15)

As \(c_1\) approaches 1, the value of \(G\) approaches \(2N\), which means each process reads more ghost cells than in the generic case considered above. However, since the read operations in this case are large sequential scans, the value of \(\alpha_1\) is considerably smaller than \(\alpha_0\) in the generic case. Thus, dividing chunks along the slowest varying dimension still reduces the overall execution time.

In practice, we find that dividing the array according to the linearization order gives the same advantage as sequential scans, while maintaining better load balance among the processes. This chunking approach may produce ghost zones of irregular shape as illustrated in Fig. 4.5, however, the ghost zones can still be read with the array elements in the chunk in a single sequential scan operation.

4.3 Performance Evaluation

We have evaluated ArrayUDF extensively to demonstrate its effectiveness. We explored the design considerations of ArrayUDF and the assumptions used by its performance model with syn-
thetic datasets. We have also performed several tests to compare ArrayUDF with RasDaMan [16], SciDB [23], and EXTASCID [32, 34], where we used the latest versions of these systems available online. Finally, we compared ArrayUDF with Spark – the system with the state-of-the-art generic UDFs – using four real-world scientific datasets and analysis operations.

**Experimental setup.** We ran our tests on Edison, a Cray XC30 supercomputer at the National Energy Research Scientific Computing Center (NERSC). Edison is equipped with 5576 computing nodes. Each computing node has two 12-core 2.4 GHz Intel “Ivy Bridge” processors and 64 GB DDR3 memory. All tested datasets are stored as HDF5 files in a Lustre parallel file system. The ArrayUDF implementation is compiled with the Intel C/C++ Compiler version 16.0. The Spark installation used as the main comparison is state of the art on HPC [27]. Since Spark does not have native support for HDF5 files, we have use H5Spark [98] to read HDF5 data directly into its RDDs, and therefore reduce the potential impact of the file system and have a fair comparison between ArrayUDF and Spark on the same storage model.

4.3.1 Synthetic Data and Poisson Equation Solver

To explore the impact of chunking parameters, to verify the analytical model, and to compare with the performance of SciDB, RasDaMan, EXTACID, and Spark, we have used synthetic two datasets. We have created these datasets that contain two multi-dimensional arrays, $S_1$ and $S_2$, containing floating point data. $S_1$ is a 2D array with ranges (100000, 100000) for $(x, y)$ dimensions, giving 38 GB in file size. $S_2$ is a 3D array with ranges (10000, 1000, 1000) for $x$, $y$, and $z$, respectively, resulting in 38 GB in file size. We have used the Poisson equation solver, which is widely used in financial mathematics, Riemannian geometry, and, thus, topology. Using the stencil operator of ArrayUDF, we can express the 2D Poisson equation solver as $4S(0, 0) - S(-1, 0) - S(0, 1) - S(1, 0) - S(-1, 0)$. Similarly, the 3D Poisson equation solver can be expressed as $6S(0, 0, 0) - S(-1, 0, 0) - S(0, 1, 0) - S(1, 0, 0) - S(-1, 0, 0) - S(0, 0, -1) - S(0, 0, 1)$. For the tests describing SciDB in this subsection, we used at most 11 compute nodes. These 11-node tests were dedicated to match the SciDB installation at NERSC supercomputing center, where 10 nodes are used for the data instances of SciDB and 1 node for the metadata instance of SciDB.

![Figure 4.7: Linear relationship between the chunk size and the cost of reading a chunk.](image-url)
**Linear relationship between the chunk size and the time of reading it.** In our performance model, we assume that the cost of reading a chunk is proportional to its size. To evaluate this assumption, we ran tests on $S_1$ and $S_2$ multiple times for a certain configuration and then to build a theoretical model. We show the I/O cost with different chunk sizes for $S_1$ and $S_2$ in Fig. 4.7. It is obvious that as the chunk size increases, the time to read the chunk increases linearly as well. The residual standard deviations of this fitting are 3.53 and 1.30 for $S_1$ and $S_2$, respectively. Thus, we can conclude that the linear relationship exists between the size of a chunk and its read time.

![Graph](image)

Figure 4.8: Cost of reading ghost cells for different chunk shapes and for different layouts.

**Impact of chunk shape on the cost of reading ghost cells.** Using the general performance model (i.e., Eq. 4.12), we predict that the chunk shape has a significant impact on data read performance. Specifically, for a fixed size chunk, the square shape guarantees that the number of ghost cells is minimum and, therefore, it has minimum I/O cost. Since we develop the general performance model without relying on any specific data organization, it can characterize the average performance of different organizations. To justify this result, we consider two data organizations, including row-major and column-major, in this test. We compare the performance of reading a chunk with the same shape from these two organizations. As the HDF5 format uses row-major organization to store data, we use the transpose-based data reorganization service of SDS framework [51, 149] to turn the row-major organization into the column-major one. We report the results in Fig. 4.8. As the chunk shape changes from the left (row-major) to the right (column-major), the time for reading the chunk from column-major organization decreases but the time for row-major organization increases. The square shaped chunk in the middle has the smallest overhead when we consider both organizations together. In other worlds, without considering the organization, the square-shaped chunk has minimum overhead. Taking only the row-major data organization as an example, we can observe that the squared chunk is not always the optimal shape, although it needs minimum amount of ghost cells. Actually, at this time, the chunk layout on disk is the dominant factor for the I/O performance. Our analysis in Section 4.2.3.2 takes this into account. In summary, these test results confirm our model formed based on a theoretical analysis.

**Overhead of reading ghost cells with layout-aware chunking.** In the layout-aware performance model, we assume that the overhead of reading ghost zone can be ignored. To justify this assumption, we have designed a test to measure the time for reading ghost zone sizes, from 1 to
64. In Fig. 4.9, we show the performance of reading ghost cells from $S_1$ and $S_2$ datasets. From the figure, we conclude that by increasing the ghost zone from 1 to 64, the time spent to read a single chunk remains flat. Linear regression of multiple measured times show a flat line to represent constant time. With the layout-aware chunking, the ghost cells tend to be contiguously organized with other cells on disk, resulting in the same read performance. Therefore, reading a small amount of ghost zone cells has negligible impact on the performance of reading a chunk. We conclude that during the layout-aware chunking, the assumption we used in the model is reasonable.

Table 4.2: Overhead of the trial run (microsecond).

<table>
<thead>
<tr>
<th>Data sets</th>
<th>The number of cells used by UDF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td>S1</td>
<td>0.37</td>
</tr>
<tr>
<td>S2</td>
<td>0.48</td>
</tr>
</tbody>
</table>

**Overhead of a “trial run” to detect the size of the ghost zone.** ArrayUDF uses a “trial-run” approach to decide the size of the ghost zone for a given UDF. In this test, we have measured the overhead of a “trial run” with respective to different numbers of array cells used by UDF. These numbers range from 4 to 256. When the number of the tested cells is larger than the number of cells required by the Poisson equation solver, we append random array cells at the end. In Table 4.2, we show the overhead of a “trial run”. Overall, we observe that the overhead of trial run is less than one microsecond. Compared with the other components of ArrayUDF presented above, this overhead is negligible.

**Comparing different methods to handle ghost zone.** Array data management systems, such as ArrayStore [145], used a dynamic loading method to access ghost cells. In this method, when a chunk needs to access ghost cells from its neighborhood chunk, this method reads the neighborhood chunk from disk into memory. Based on the fact that a UDF has a well-determined access pattern for ghost cells, ArrayUDF optimizes the accessing of ghost cells via statically extending each chunk by a ghost zone when the chunk is first read from disk. As a result, separate disk accesses are not needed for ghost cells in ArrayUDF. We use the low-overhead “trial run” approach to obtain accurate size estimates for a ghost zone. In this test, we compare these two methods by implementing the
dynamic chunk loading in ArrayUDF. As shown in Fig. 4.10, in reading ghost zones from S1 and S2 datasets, the optimized method in ArrayUDF is on average four times faster than the dynamic chunk loading method. Thus, ArrayUDF has an efficient way in handling access to ghost zones.

Comparing ArrayUDF with SciDB, RasDaMan, EXTASCID, and Spark in executing the standard “window” operator. As we discussed in the previous sections, the Poisson equation solver cannot be expressed directly using the “window” operators of SciDB and RasDaMan. To compare the performance and versatility of ArrayUDF, we compare the performance of the “window” operator these systems provide, where the operation is to compute the average for a $2 \times 2$ window on 2D data and a $2 \times 2 \times 2$ window on 3D data. We also include EXTASCID and Spark for a complete comparison. For fairness of the evaluation platform for all these systems, we have installed them on a single Linux desktop. The desktop has two CPU cores (Intel i7-5557U CPU with 3.10 GHz), two threads per core, and a local disk (Seagate ST1000LM014-1EJ1) with the EXT4 file system. We show these test results in Fig. 4.11(a). The original datasets S1 and S2 are too large for the processing on a single node. Thus, we used smaller size datasets for tests. The tested 2D dataset has an array with a size of (10000, 30000) and the 3D dataset has a size of (1000, 1000, 400).
chunk size for all the systems is set to be (5000, 15000) for 2D and (1000, 1000, 100) for 3D which give 4 chunks per array—equivalent the number of threads in the system.

RasDaMan has the highest execution time because the version that is publicly available only supports “inter-query parallelization” and, thus, it can only use one core to perform the calculation. Spark can use its Map and Reduce interface to implement the window-based average, however, it needs to duplicate the data for different windows. As a result, it has the second highest execution time. In terms of SciDB, it handles each window independently and inefficiently [79], thus SciDB is also much slower than ArrayUDF. We also scale the SciDB tests to 11 nodes on the Edison system for $S_1$ and $S_2$ datasets. In these tests, we observe similar results. EXTASCID provides a robust array data storage model, but it has no portable window-like operator yet. For comparison, we have implemented the window function manually as a generalized linear aggregate (GLA) function, which is the abstract interface in EXTASCID. Basically, the window functions are written in C++ and they use the EXTASCID I/O driver to access the arrays. Even with this hand-optimized code, the execution times of EXTASCID are similar to those of ArrayUDF. In other words, the performance of ArrayUDF is very close to the C++ hand-optimized code. Thus, we conclude that ArrayUDF provides more flexibility to define operations, and it is as efficient as highly-optimized code in performing “window” based analysis tasks.

Comparing ArrayUDF with Spark on solving the Poisson equation on datasets $S_1$ and $S_2$. In this test, we compare the performance of using ArrayUDF and Spark to solve the Poisson equation with 64 computing nodes. By using ArrayUDF, we can directly define and execute the Poisson equation solver on arrays stored in datasets $S_1$ and $S_2$, as shown at the beginning of this section. To use Spark for expressing these operations, we apply the “flatMap” and “reduceByKey” functions. The general idea is that all the adjacent stencil cells required by a cell are viewed as a group and all cells within the group are aggregated onto a single reducer to perform the computation. Since each cell belongs to multiple groups (of its neighbors), we use “flatMap” to transform each cell into multiple (key, value) pairs, where the key is the group ID and the value is the actual data. After “flatMap” finishes, we use “reduceByKey” to consolidate all the cells with the same group ID together. Basically, each reducer is responsible for computing a cell in the result array. One may argue that users can write a specific Map function without a Reduce funtion to solve the Poisson equation. However, as discussed in previous sections, both ArrayUDF and Spark aim at providing a generic UDF mechanism for users to express high-level operations without burdening users to write custom programs for performing different operations.

We compare the performance of ArrayUDF and Spark in executing the Poisson equation solver in Fig. 4.11(b). For datasets $S_1$ and $S_2$, ArrayUDF is 11 and 118 times faster than Spark, respectively. Since the Poisson equation solver on 3D data needs to access more adjacent cells than on 2D data, ArrayUDF achieves a higher performance improvement for processing $S_2$. Basically, the more neighbor cells are required, the more times the entire array is replicated in Spark. As the size of the data to be replicated increases, Spark spends more time on communication and shuffling for “flatMap” and “reduceByKey” functions. Compared with Spark, ArrayUDF allows users to define and execute the Poisson equation solver directly on an array. Thus, there is no data replication during the runtime. Moreover, as ArrayUDF can automatically build the ghost zone using a trial-run, the expensive communication is avoided.
4.3.2 ArrayUDF for Real Scientific Data Analysis

We have evaluated ArrayUDF to perform several real-world analysis tasks on four scientific data sets: S3D, VPIC, MSI, and CoRTAD. We summarize the properties of these datasets and the analysis operations performed in this study in Table 4.3. A brief background is presented in Section 4.1. To compare performance, we use Spark to implement the same analysis tasks on these datasets. The method to implement these analysis operations is the same as the one we use for the Poisson solver on the S1 and S2 datasets. We test each task with different numbers of CPU cores, scaling from 384 (i.e., 16 compute nodes) to 1536 (i.e., 64 nodes). As shown in Fig. 4.12, for S3D, VPIC, and MSI analysis, ArrayUDF outperforms Spark by up to 26×, 220×, and 2070×, respectively. For CoRTAD, Spark crashes due to out-of-memory (OOM) errors. We discuss the performance of each of these analysis tasks in the following.

Table 4.3: Real-world scientific datasets and operations

<table>
<thead>
<tr>
<th>dataset</th>
<th>Rank</th>
<th>Size (GB)</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>S3D</td>
<td>3D</td>
<td>301</td>
<td>Vorticity computation</td>
</tr>
<tr>
<td>MSI</td>
<td>3D</td>
<td>21</td>
<td>Laplacian calculator</td>
</tr>
<tr>
<td>VPIC</td>
<td>3D</td>
<td>36</td>
<td>Trilinear interpolation</td>
</tr>
<tr>
<td>CoRTAD</td>
<td>3D</td>
<td>225</td>
<td>Simple moving average</td>
</tr>
</tbody>
</table>

S3D. For the S3D dataset, we compute the vorticity, which is defined in Eq. 4.2. For a single point in S3D, the vorticity computation needs the values of four neighbors in total and two neighbors per direction. In this test, we use the dataset in the x direction. This data has $1100 \times 1080 \times 1408$ dimensions, resulting in a file size of 22 GB. We show the execution time for vorticity computation using ArrayUDF and Spark in Fig. 4.12(a). On average, ArrayUDF is 18 times faster than Spark. We observe similar performance speedup in the y and z dimensions. In summary, we observe that ArrayUDF is more efficient than Spark in computing vorticity over the S3D dataset.

MSI. The MSI data [132] used in our test contains a $123 \times 463 \times 188960$ 3D array which has 21 GB in file size. This array contains images of a potato eye. The operation of interest on this dataset is the Laplacian calculator, as presented in Eq. 4.3. In our tests, we observe that ArrayUDF is 196 times faster than Spark. Since the Laplacian calculator needs five neighbor cells, but vorticity...
on S3D only needs two, ArrayUDF achieves a higher speedup in this test. As discussed before, the more adjacent cells are needed in UDF, the higher speedup ArrayUDF can achieve.

**VPIC.** In the space weather simulation using VPIC, the total size for the magnetic field data is 36 GB and the data has x, y and z dimensions [25]. The dimensions of the dataset are 2000 × 2000 × 800. We use both ArrayUDF and Spark to implement the trilinear interpolation shown in Eq. 4.4. For simplicity, we assume that there is a particle in a cell. But, our ArrayUDF can be extended to support more flexible interpolation. We report the results for the x dimension in Fig.4.12(c)—we obtain similar results for dimensions. On average, ArrayUDF is 1607X faster than Spark in this test.

**CoRTAD.** CoRTAD is a collection of sea surface temperatures (SST) [1]. It contains the weekly temperature for 1617 × 4320 sites from 1981 to 2010. Thus, it is a 3D array with size 1617 × 4320 × 8640, where the third dimension is the number of weeks (i.e., 8640 weeks). We compute a moving average based smoothing on this data. Basically, for each site, we smoothen its temperature based on the month, i.e., compute the average of four neighbor array cells. In Fig. 4.12(d), we depict only the performance of ArrayUDF. For the tests with the same number of CPU cores, Spark crashes with out-of-memory errors. The reason for the OOM errors is the need for Spark to replicate the entire dataset 4 times in order to compute the moving average. Moreover, since we also need to use the (key, value) pair structure and the Scala object type to store the data, the total data size increases by more than 8 times, which contributes to the OOM errors. Meanwhile, ArrayUDF successfully completes the moving average computation for this large dataset without any memory footprint pressure.

### 4.4 Related Work

UDFs are widely supported by relational database management systems, such as SQL Server [29], IBM DB2 [177], MySQL [162], and PostgreSQL [112]. MonetDB [129] has an extension to support vector-based UDFs that take advantage of the columnar data representation. The UDFs within these systems are based on the relational set semantics and permit users to define operations at tuple or table level—known as User-Defined Aggregates (UDA) [31]. The key difference between ArrayUDF and these types of UDFs is that ArrayUDF is developed for multi-dimensional arrays and allows more general structural locality based operations.

In MapReduce [47], UDFs consist of two steps—Map and Reduce. Map applies the UDF on a single key-value pair and produces one or more key-value pairs as output. Reduce consolidates key-value pairs having the same key and then applies another UDF. Reduce requires expensive data shuffling to repartition data across nodes. Among all extensions to MapReduce [11, 24, 59, 70, 158], Spark provides a much richer set of UDFs for iterative in-memory analysis. Compared to MapReduce and its extensions, ArrayUDF requires a single step to express UDFs on a set of related array cells, thus avoiding the expensive shuffle stage.

Array database systems such as RasDaMan [16], AML [106], SciDB [23], SciQL [88] and EXTASCID [34], have UDF support. Rusu et al. [133] provides a complete survey on this topic. Typically, these array UDFs follow a similar idea to relational database systems, where users define an operation on a single element, i.e., array cell. If the UDF requires multiple adjacent cells, these have to be mapped into tuples and then apply the UDF. ArrayUDF is a novel UDF type for arrays that
allows users to define operations directly on adjacent array cells, without any mapping. Moreover, to support efficient data access in a shared-disk system, ArrayUDF performs dynamic chunking and ghost zone building. In contrast, array database management systems have a shared-nothing architecture and rely on data ingestion to handle chunking and ghost zones. A specific type of array UDF is the window-based Apply operator [106]. SciDB provides this function via a highly optimized window operator. RasDaMan has a similar operator named condense. While similar to ArrayUDF, these operators support only fixed-size windows and the operations on a window are limited. ArrayUDF generalizes the window shape and the operations for the cells within a window. Join operators are found to be expensive to support these operations because of data replication [52, 172]. SAGA [160] explores aggregate operations on scientific arrays stored in native data formats.

The domain-specific languages (DSL) [18, 109] share similarity with our ArrayUDF in leveraging stencil behaviors to improve performance of array based data analytics. These DSLs are mostly developed as programming language and compiler extensions with the goal of increasing the efficiency of calculation and memory data access. But, in ArrayUDF, we developed a flexible computing model towards large-scale data analytics, i.e., to derive knowledge from the multidimensional arrays in data files directly. The ArrayUDF generalizes the MapReduce -- like systems to realize a wider range of operations on arrays, including the stencil operator supported by these DSLs. Beyond that, ArrayUDF has the capability to express other types of operations, such as aggregation, filtering, etc. Moreover, ArrayUDF provides optimizations (e.g., trial run based ghost zone determination, disk layout aware/unaware partitions for multidimensional array) for efficiently processing out-of-core data sets, which are too large to be quickly loaded into memory, even with multiple nodes. Our analytical performance model for ArrayUDF can also be used as the foundation to optimize these existing stencil DSL systems in handling large-scale scientific datasets on disk.

4.5 Conclusions and Future Work

Customized data analysis, especially for data stored as multidimensional arrays in large files, is a common method to extract insights from data and the UDF mechanism is a general strategy to support this analysis. However, the current UDF implementations are not able to effectively express the structural locality present in most array based data analysis operations, such as computing the vorticity for a 3D flow field, the moving average for a time series, etc. These operations have to be expressed as expensive join or reduction operations. We design ArrayUDF to easily capture the generalized structural locality and implement an in situ processing system optimized for multidimensional arrays. The generalized structural locality mechanism of ArrayUDF allows users to define a different operation on each neighbor separately and, therefore, express more complex customized data analysis. The in situ processing system automatically partitions data stored in raw data files and creates ghost zones for the arrays stored in files without an expensive data ingestion phase. Our automatic data partitioning minimizes the execution time based on a an analytical model of the expected performance. It is also able to take advantage of the layout of arrays in input data files.

Our evaluation using a number of different scientific datasets show that ArrayUDF is up to three orders of magnitude faster than Apache Spark. In future, we plan to enhance ArrayUDF with a filter feature to reduce loading of unnecessary array cells in the result arrays. We will also explore the use of ArrayUDF to process stream and real-time data.
Chapter 5

Distributed Caching for Processing Raw Arrays

As applications continue to generate multi-dimensional data at exponentially increasing rates, fast analytics to extract meaningful results is becoming extremely important. The database community has developed array databases that alleviate this problem through a series of techniques. In-situ mechanisms provide direct access to raw data in the original format—without loading and partitioning. Parallel processing scales to the largest datasets. In-memory caching reduces latency when the same data are accessed across a workload of queries. However, we are not aware of any work on distributed caching of multi-dimensional raw arrays. In this section, we introduce a distributed framework for cost-based caching of multi-dimensional arrays in native format. Given a set of files that contain portions of an array and an online query workload, the framework computes an effective caching plan in two stages. First, the plan identifies the cells to be cached locally from each of the input files by continuously refining an evolving R-tree index. In the second stage, an optimal assignment of cells to nodes that collocates dependent cells in order to minimize the overall data transfer is determined. We design cache eviction and placement heuristic algorithms that consider the historical query workload. A thorough experimental evaluation over two real datasets in three file formats confirms the superiority – by as much as two orders of magnitude – of the proposed framework over existing techniques in terms of cache overhead and workload execution time.

5.1 Background

In the era of big data, many scientific applications – from high-energy physics experiments to cosmology telescope observations – collect and analyze immense amounts of data at an unprecedented scale. For example, projects in astronomy such as Sloan Digital Sky Survey\(^1\) (SDSS) and Palomar Transient Factory\(^2\) (PTF) collect the observations of stars and galaxies at a nightly rate of hundreds of gigabytes. The newly-built Zwicky Transient Facility\(^3\) (ZTF) records the sky at a 15X

\(^1\)http://www.sdss.org/dr13/
\(^2\)http://www.ptf.caltech.edu/iptf/
\(^3\)https://www.ptf.caltech.edu/ztf
higher rate than PTF—up to 7 TB per night. A common characteristic of the datasets produced by astronomy projects is that the data are natively organized in a multi-dimensional array rather than an unordered set. Due to the inefficacy of traditional relational databases to handle ordered array data [43, 122], a series of specialized array processing systems [16, 24, 34, 122, 154, 170] have emerged. These systems implement natively a distributed multi-dimensional array data model in which arrays are chunked across a distributed shared-nothing cluster and processed concurrently.

Automatic transient classification. The PTF project aims to identify and automatically classify transient astrophysical objects such as variable stars and supernovae in real-time. A list of potential transients—or candidates—is extracted from the images taken by the telescope during each night. They are stored as a sparse array:

\[
\text{candidates} \langle \text{bright}, \text{mag}, \ldots \rangle \langle \text{ra}, \text{dec}, \text{time} \rangle
\]

with three dimensions—the equatorial coordinates \( \text{ra} \) and \( \text{dec} \), and \( \text{time} \)—and tens to a few hundred attributes such as brightness and magnitude. The range of each dimension is set according to the accuracy of the telescope which increases with every generation of lenses. In PTF [114], an array cell corresponds to 1 arcsecond on \( \text{ra} \) and \( \text{dec} \), and 1 minute on \( \text{time} \), which generate more than \( 10^{15} \) cells. The candidates array is stored in several FITS\(^4\) files—one for every night. Since the telescope is pointed to distinct parts of the sky each night, the files cover different ranges of the array space. However, the ranges are large enough to overlap. The number of candidates in a file has high variance—there are sparse files with only tens of candidates and skewed files with millions of candidates.

Transient classification consists of a series of array queries that compute the similarity join [172] between candidates, i.e., find all the candidates identified at similar coordinates, possibly in a time window. In order to execute these queries inside an array database, the candidates array has to be loaded and partitioned. This process is time-consuming—it takes hours in SciDB according to [34, 166]—and duplicates the data. As a result, it is infeasible for the real-time processing required in PTF. In-situ processing over raw files [9, 21, 33] is an alternative that eliminates loading altogether and provides instant access to data. While there is a large body of work on in-situ processing for CSV and semi-structured files such as JSON and Parquet [9, 33, 118, 148], there are only two extensions to multi-dimensional arrays [66, 166]. Both of them act as optimized HDF5\(^5\) connectors for SciDB. They allow SciDB to execute declarative array queries over dense arrays stored in HDF5 format by pushing down the subarray operator into the HDF5 read function. None of the connectors support the sparse and skewed files corresponding to PTF candidates because these files are not organized and partitioned along the dimensions of the array. Even for dense arrays, the initial partitioning specified at array definition may be suboptimal with respect to the query workload. For example, a tight band range on one dimension triggers a high number of partition accesses even though the number of queried cells is small. Moreover, the connectors do not provide any caching mechanism. This is pushed entirely to the SciDB buffer manager which operates at instance-level, i.e., it caches only local arrays. This precludes any optimization in accessing the raw data because the granularity and the content to cache are query-dependent rather than partition-dependent—partition or chunk caching is suboptimal. As pointed out in [9, 82, 118], caching is extremely important for in-situ processing because repeated access to raw data is expensive—the

\[^4\]https://fits.gsfc.nasa.gov/fits_documentation.html
\[^5\]https://support.hdfgroup.org/HDF5/
case for the set of queries in PTF transient classification.

**Problem statement.** We tackle the problem of distributed caching for raw arrays with the goal to accelerate queries over frequently accessed ranges. Given a set of raw files that contain portions of an array and an online dynamic query workload, we have to determine which cells to cache in the distributed memory of an array database system. Rather than having each node manage its memory buffer to cache local cells, we aim for a global caching infrastructure that automatically identifies both the cells to cache and the instance where to cache them. Our ultimate goal is to provide both instant access to distributed multi-dimensional raw arrays and optimal query performance through caching.

**Challenges.** Distributed caching for arrays poses two main challenges. The first challenge is to determine the cells kept in the cache at query time. This is a problem because cache misses over raw arrays are very expensive. Even if we have only one cache miss for a cell, it requires us to scan all the raw files whose bounding box contains the cell—for dense arrays, although direct access eliminates scanning, a complete disk read is still necessary. However, at most one file contains the required cell. The second challenge comes from the distributed nature of array database. The conventional approach caches the requested data at the origin instance—where it is stored. This approach works well in content delivery networks (CDN) [121, 146] where data co-locality is not required for query execution. However, most of the array queries specify shape-based relationships between cells in the form of stencil operators and their generalization to similarity joins [172]. Given the current placement of raw and cached data, the challenge is to coordinate and organize the caches across nodes—decide what cached data are placed on which nodes—in order to preserve data co-locality and provide efficient data access. Direct application to arrays of generic distributed memory caching [10, 58, 95, 115, 157] implemented in Hadoop and Spark suffers from excessive communication and load imbalance due to the skewed distribution of cells across files.

**Approach.** We design a distributed caching framework that computes an effective caching plan in two stages. First, the plan identifies the cells to be cached locally from each of the input files by continuously refining an evolving R-tree index. Each query range generates a finer granularity bounding box that allows advanced pruning of the raw files that require inspection. This guarantees that—after a sufficiently large number of queries—only relevant files are scanned. In the second stage, an optimal assignment of cells to nodes that collocates dependent cells in order to minimize the overall data transfer is determined. We model cache eviction and placement as cost-based heuristics that generate an effective cache eviction plan and reorganize the cached data based on a window of historical queries. We design efficient algorithms for each of these stages. In the long run, the reorganization improves cache data co-locality by grouping relevant portions of the array and by balancing the computation across nodes.

**Contributions.** The specific contributions of this work can be summarized as follows:

- We introduce distributed caching for raw arrays (Section 5.3). While caching over raw data has been explored in a centralized setting, this is the first work that investigates in-situ processing with distributed caching.

- We design an evolving R-tree index that refines the chunking of a sparse array to efficiently find the cells contained in a given subarray query (Section 5.3.1). The index is used to eliminate unnecessary raw files from processing.
• We propose efficient cost-based algorithms for distributed cache eviction and placement that consider a historical query workload (Section 5.3.3 and 5.3.4). The goal is to collocate dependent cells in order to minimize data transfer and balance computation.

• We evaluate experimentally our distributed caching framework over two real sparse arrays with more than 1 billion cells stored in three file formats—CSV, FITS, and HDF5 (Section 5.4). The results prove the effectiveness of the caching plan in reducing access to the raw files and the benefits of the eviction algorithm.

While our solution is presented for sparse arrays, the proposed framework is also applicable to dense data. While access to a specific cell is much faster in a dense array, caching entire chunks incurs unnecessary memory usage. Instead of caching the complete chunks as specified in the array definition, our solution infers a workload-derived caching scheme that may have smaller granularity. This allows for more relevant data to be cached. Another important application of this work is to distributed linear algebra which is the fundamental building block in large-scale machine learning (ML). Stochastic gradient descent (SGD) [104] – the primary algorithm to train ML models – requires access to rows in the example matrix that have non-zero entries for certain columns—or features. Existing solutions in Spark MLlib [55] and TensorFlow [101] access all the blocks containing the columns of the required example—which is unnecessary and expensive. Our framework builds an evolving R-tree index that allows for non-zero columns to be easily found without accessing the block.

5.2 Preliminaries

In this section, we introduce multi-dimensional arrays, in-situ data processing, and distributed caching—the foundation for distributed caching of raw arrays.

![Array A](image)

Figure 5.1: Array A\texttt{:char,f:int}[i=1,6; j=1,8]. This is a raw array consisting of 7 files distributed over 3 nodes. Attributes \texttt{n} and \texttt{f} correspond to the node and the file id on the node, e.g., \texttt{<X,2>} is a cell in the second file on node X. The cell color also shows the server on which the file is stored. The dashed rectangle specifies a query.

5.2.1 In-Situ Processing over Raw Data

Multiple systems that perform queries over raw data that do not reside in databases have been proposed [5, 9, 33, 71, 73, 82, 147] recently. They are extensions of the external table mechanism
supported by standard database servers. These systems execute SQL queries directly over raw data while optimizing the conversion process into the format required by the query engine. This eliminates loading and provides instant access to data. Several systems [5, 9, 33] provide a dynamic tradeoff between the time to access data and the query execution time by adaptively loading a portion of the data during processing. This allows for gradually improved query execution times while reducing the amount of storage for replication. In-situ processing over scientific data – specifically, multi-dimensional arrays – resumes to connectors from a query processing engine, e.g., SciDB, to an I/O-library for a specific data format, e.g., HDF5 [21, 66, 166]. These systems allow SciDB to execute declarative array queries exclusively over dense HDF5 arrays by pushing-down the subarray operator. They do not provide integrated optimizations across layers. The focus of this work is on the more general case of unorganized (sparse) arrays—not only HDF5.

5.2.2 Caching

All database systems cache disk pages in memory buffers after they are read from disk. The fundamental problem in caching is to decide which pages to keep in memory when the available budget is exhausted—or, equivalently, which pages to evict. The provably optimal eviction algorithm is to remove the page that will be accessed farthest in the future. Since the future is generally unknown, existing eviction algorithms are based on the past query workload. The least-recently-used (LRU) and least-frequently-used (LFU) are the most common eviction algorithms widely implemented in computer systems. They can be easily extended to array databases by replacing the page with the chunk [122]. Online cost-based caching algorithms [26, 72] prioritize the eviction of pages/chunks with lower cost in order to keep the expensive items in cache, where the cost is application-dependent. Since there are no provable polynomial time cost-based cache eviction algorithms [26], Greedy heuristics are the only solution. Nonetheless, these algorithms are preferred in a distributed setting where the cost of accessing data is variable. The standard approach in distributed caching is to define a global shared memory that is mapped to the local memory of physical nodes [95, 115]. This allows for an immediate extension of the centralized algorithms. Parallel databases use a simpler approach in which each node is managing its local cache—there is no global cache manager. The reason is transactional cache consistency. However, in read-intensive scientific applications, transactions are not an issue. Caching for raw data processing is necessary because of the high cost to extract data. The NoDB [9] and Proteus [82] systems cache everything in memory and use LRU for eviction, while SCANRAW [171] materializes the evicted pages adaptively in the database storage. However, they do not target raw sparse arrays and are centralized solutions.

5.3 Raw Array Distributed Caching

In this section, we present the first distributed caching mechanism over raw array data proposed in the literature. We begin with a high-level description of the distributed caching inner-workings that identifies the main processing stages. Then, we delve into the details of each stage and introduce our technical contributions.

Problem setting. Given a collection of raw files distributed across the nodes of the array database and a query workload, the goal is to process the queries optimally with limited memory
budget for caching. We assume the entire workload is unknown before processing—queries are processed one after another in online fashion. Figure 5.2 illustrates the proposed distributed caching architecture which consists of a cache coordinator and a cache module for every node of the array database. These can run either as independent processes or as threads that are part of the array database coordinator and nodes, respectively. Each file \( f_{i,j} \) – the \( j \)th file stored in its entirety on node \( i \) – has a bounding box \( B(f_{i,j}) \) – stored in the catalog at the cache coordinator – that contains the range of each dimension for the cells in the file. The cache coordinator has a complete view of the distributed cache memory across nodes both in terms of content and size. This allows for optimizing in-memory chunk location, thus, query execution, and is a major departure from distributed buffer managers such as Spark’s. We emphasize that the cache coordinator does not receive any file data—exclusively managed by the cache nodes.

**Approach.** The array distributed caching framework works as follows (Figure 5.2). First, the execution engine sends the subarray and the shape in the array similarity join query to the cache coordinator which has to determine the chunk pairs that have to be joined. While these can be immediately inferred from the catalog for loaded data, there is no chunking—or the chunking is not known—for raw data. The naive solution to handle this problem is to treat each file as a chunk and perform the similarity join between pairs of files. Since a file covers a much larger range and is not organized based on cell locality, this solution accesses many unnecessary cells and uses the cache budget inefficiently. Our approach is to build in-memory chunks incrementally based on the query—workload-driven chunking [133]. These chunks are built by the cache nodes by accessing the files that overlap with the query—the only data the cache coordinator forwards to the nodes. Instead of extracting chunks only for the queried data, the cache nodes create higher granularity chunks for the entire file. Since chunk building requires raw file access—which is expensive—caching is very important for efficient processing. In the best case, the entire file can be cached. Otherwise, a cache replacement policy is required to determine the chunks to keep in memory. Once evicted from memory, the chunks are lost—they have to be recreated from the raw data. While it is possible
to materialize the evicted chunks in some representation, e.g., R-tree, this replicates data and adds another dimension—this representation to read data from—to the problem.

Instead of allowing each node to execute a local cache replacement algorithm, e.g., LRU, we develop a global cost-based caching mechanism [12] executed at the coordinator. This algorithm takes the metadata of all the created chunks across nodes and determines the chunks to be evicted from the cache in a unified way—this may incur chunk transfer across nodes. Since query evaluation also requires chunk transfer and replication, we go one step further and combine query processing with cache replacement. Specifically, cache replacement piggybacks on the query execution plan and eliminates extra chunk transfer. Moreover, the location of a cached chunk is computed by considering the relationship with other chunks in the historical query workload. This is implemented as follows (Figure 5.2). Upon receiving the chunk metadata from the nodes, the coordinator—query optimizer—computes the optimal query execution plan which specifies what node joins each of the chunk pairs. This is passed to the cache coordinator which computes the cache replacement plan by combining the execution plan with chunk usage statistics and chunk co-locality information. The cache replacement plan informs every node on what chunk replica to cache and which chunks to evict. Finally, these two plans are forwarded to the nodes for execution—the engine executes the query and the cache executes the replacement plan, in this sequence.

![Figure 5.3: Raw array query-driven chunking. (a) depicts a raw array and its bounding box. (b), (c), and (d) show how the chunking evolves based on a workload of three queries. (e) depicts the resulting four chunks into which the initial array is partitioned.](image)

**5.3.1 Raw Array Chunking**

The goal of raw array chunking is to infer the chunks of the array file dynamically at runtime from the query workload. Instead of creating arbitrary chunks during loading—which is time-consuming, delays the time-to-query, and may not be optimal for the actual workload—we build chunks incrementally one query-at-a-time. Given a subarray in the domain of the raw arrays over which to evaluate a similarity join query, we have to identify the cells that are relevant for the query while minimizing the total number of inspected cells. We have to avoid access to the raw files because they are unorganized and require full scan. This cannot be realized initially unless there is no overlap between a file and the query. A straightforward solution is to load the coordinates of all the cells after the first access. However, this is impractical because it requires complete data replication. Instead, we build chunks that group close cells and maintain only the much smaller bounding box. The chunks are cached in the memory of the nodes, while the bounding boxes are managed by the cache coordinator in memory. Moreover, not all the chunks can be cached.
We design a novel incremental chunking algorithm that builds an evolving R-tree [64] based on the queries executed by the system. The invariant of this algorithm is that the set of chunks cover all the cells of the array at any time instant. Moreover, the chunks are non-overlapping. The central point of this algorithm is splitting a chunk that overlaps with the query subarray (Algorithm 5). Two questions require answer.

**When to split?** A chunk is split in two cases. First, if there are a sufficiently large number of cells in the chunk. This threshold is a configurable parameter that can be set by the user. In the second case, even when the number of cells is below the threshold, if the query subarray does not contain any cell, the chunk is split further. Creating a large number of (small) chunks has both positive and negative impact. On the positive side, the likelihood that a chunk that overlaps with the query contains relevant cells is higher. This avoids inspecting unnecessary chunks. On the negative side, the number of bounding boxes that have to be managed by the cache coordinator increases and this makes query and cache optimization more time-consuming.

**How to split?** The number of additional chunks generated by a split varies between 1 and $3^d$, where $d$ is the dimensionality of the array. We opt for always splitting a chunk into two chunks. This is done by selecting a single splitting dimension. The algorithm enumerates over the queried subarray boundaries that intersect with the chunk bounding box and chooses to split into those two chunks that have the minimum combined hyper-volume. Rather than computing the hyper-volume from the query-generated chunks, we derive the bounding box of a chunk only from the cells assigned to it. Typically, this results in smaller and more condensed chunks. We increase the number of chunks conservatively because we do not want the number of bounding boxes managed by the cache coordinator to explode with the number of queries. Chunks that cover a smaller hyper-volume are more compact, thus the probability to contain relevant cells is higher.

**Algorithm 5 Chunk Split**

FUNCTION Chunk $\alpha$ with bounding box $BB_\alpha$ that intersects query subarray $Q$; Minimum number of cells threshold $MinC$

Output: Chunks $\beta$ and $\gamma$ after splitting $\alpha$

1. if (cells in $\alpha \cap MinC$) and ($\exists$ cell in $\alpha \in Q$) then return
2. $min\_vol = +\infty$
3. for each boundary $b \in Q$ that intersects with $BB_\alpha$ do
4. $(\beta_b, \gamma_b) \leftarrow$ split cells in $\alpha$ into two sets by boundary $b$
5. if $\text{vol}(\beta_b) + \text{vol}(\gamma_b) \leq min\_vol$ then
6. $min\_vol \leftarrow \text{vol}(\beta_b) + \text{vol}(\gamma_b)$
7. $\beta \leftarrow \text{bounding}_\text{box}(\beta_b), \gamma \leftarrow \text{bounding}_\text{box}(\gamma_b)$
8. end if
9. end for

We illustrate how the algorithm works for a 2-D array in Figure 5.3. The dashed box is the query subarray $Q$, while the set of chunks at a given instant is represented by solid bounding boxes. Initially, there is a single chunk with a large bounding box. This corresponds to the root of the R-tree. Since only two boundaries of query $Q_1$ overlap with the original chunk, only two potential splits are considered. The one that is chosen corresponds to the horizontal axis. Two smaller chunks
are generated. Query \( Q_2 \) overlaps with both of these two chunks. Since the number of cells in the upper chunk is 4 – smaller than the splitting threshold of 5 – and there is a cell relevant to the query, no split is triggered. The lower chunk is split based on the single query boundary that intersects with it. Query \( Q_3 \) overlaps only with one of the input 3 chunks. Although the number of cells in the chunk is below the splitting threshold, there is no cell in the query subarray. This triggers a split into two condensed chunks, for a total of four chunks overall.

Figure 5.4: High-level view of raw array distributed caching. The chunks in Figure 5.1 are shown together with the corresponding bounding box separately for each server. The queried subarray is depicted as a dashed rectangle.

5.3.2 Cost-Based Chunk Caching

Given the chunks extracted from the raw array and a memory caching budget at each server, the goal is to find an optimal caching plan to execute the query workload. In a centralized setting, cache optimality is defined as minimizing the number of cache misses, i.e., disk misses. In a distributed setting, another type of cache misses are present—chunks that have to be transferred over the network, i.e., network misses. The impact of these two measures depends on the relative throughput of the disk and the network. In the case of raw arrays, however, disk access always requires a complete file scan because chunks are not organized. Intuitively, this suggests that cache optimality should be defined exclusively in terms of disk misses. While this is true for queries that consider the chunks independently, in the case of array join queries, network misses are important because of chunk co-locality—chunk pairs that have to be processed together.

We illustrate the issues in array chunk caching on the example in Figure 5.4. We set the memory budget on each node to at most 3 cells. A self-similarity join query with a cross shape, i.e., \( L^1(1) \), is executed over the dashed rectangle subarray. The example starts with a cold cache—the cache is empty in the beginning. The cache coordinator determines that chunks \( f_{1,1}, f_{2,1}, f_{3,1} \), and \( f_{3,2} \) overlap with the queried subarray. Since chunk \( f_{3,2} \) does not have any cells that are contained by the query subarray, it is not considered for caching—it is split into two smaller chunks based on Algorithm 5. The other 3 chunks overlap with the query, thus they are cached. In local caching, a chunk is cached on the node where it is stored. This is suboptimal for chunk \( f_{3,1} \) which has 4 cells—it cannot be cached entirely. In distributed caching, the memory budget is aggregated. As a result, chunk \( f_{3,1} \) is cached completely by using the available entry at node \( Y \). This eliminates the
expensive disk misses incurred by partially cached chunks. In order to retrieve the uncached cell [1,4], a full scan over the file containing chunk $f_{3,1}$ is required. In order to execute the self-similarity join query, cell pairs ([1,3],[1,4]), ([1,3],[2,3]), ([2,2],[2,3]), ([2,3],[3,3]), and ([4,2],[5,2]) have to be resident on the same node. Since distributed caching does not consider network misses, only cell pairs ([1,3],[1,4]) and ([4,2],[5,2]) are collocated. A better caching configuration that collocates 3 cell pairs is shown in Figure 5.4.

We design a cost-based caching algorithm that weighs the chunks based on how they are accessed in the query workload. There are two components to the cost—access frequency and co-locality. Access frequency prioritizes chunks that are accessed in the near past. While this is similar to standard LRU, access is biased towards chunks belonging to the same raw file—a cache miss inside a file requires a complete scan of the file. The access frequency cost is included in the cache eviction section of the algorithm. Co-locality assigns chunk pairs that are joined together to the same node in the cluster. This cost is based on the query workload and is also biased towards more recent queries. The co-locality cost is included in the chunk placement section of the algorithm. The separation of cost-based caching into two sections is necessary because of the interaction between query processing and caching. If we consider them together, it is not clear how to define—not to mention solve—a cost formula that combines access and frequency.

5.3.3 Cache Eviction

After we generate the currently queried chunks, we have to identify chunks to evict from the cache—when the cache is full. Notice that only the chunks that overlap with the current query are considered for caching—chunks created because of non-overlapping splits are discarded. The important observation in caching raw array chunks is that we have to scan a file entirely even if only one accessed chunk is not cached. Therefore, we aim to cache all the queried chunks in a file. Conversely, if we evict a chunk from a file, we should evict all the other chunks from that file—indeed of when they have been accessed. Since chunk-level LRU does not consider this correlation between chunks, it is likely suboptimal. File-level LRU—on the other hand—uses ineffectively the cache budget by caching non-accessed chunks. We include experiments for both of these two alternatives in Section 5.4.

We propose a novel cost-based cache eviction algorithm that integrates access with the raw file read savings for each unit price we pay, i.e., the additional cache budget we use. Specifically, the cost of a cached chunk is defined as:

$$\text{cost}_{\text{evict}}(Q_l, f_i, \{C_j\}) = w_{Q_l} \cdot \frac{\text{size}(f_i)}{\sum \text{size(uncached } C_j)}$$

where $\{C_j\}$ is the set of chunks from file $f_i$ accessed by query $Q_l$. The weight of a query $w_Q$ is defined as an exponential function of the index $l$ of the query in the workload, e.g., $w(1) = 2^1, \ldots, w(l) = 2^l$. Essentially, the weight generates an exponential decay for the importance of the queries in the workload. The ratio between the file size and the size of the uncached chunks is larger when the number of chunks in the file required by a given query is smaller. If all the queried chunks are cached—the case for the current query—the ratio is infinity, thus the file is not evicted.

Based on this eviction cost, we design an efficient greedy heuristic (Algorithm 6) that evicts
the chunks with the lowest cost. The algorithm takes as input the cache state, the chunks accessed by the current query, and the cache budget. It outputs the updated cache state that includes the newly accessed chunks. Instead of incrementally evicting chunks with the lowest cost, the heuristic selects the chunks with the highest cost to keep from the original cache state. This is necessary because of the correlation between chunks across queries. For example, a chunk with a low cost in a query may be evicted even though it is also accessed by an important query. If we decide to include the chunk, the decision is final. Moreover, this decision has an impact on other chunks. Thus, we increase the cost of all the correlated chunks to boost their chance of being cached (line 6). This is realized by evaluating the eviction cost dynamically without materialization.

**Algorithm 6 Cache Eviction**

**FUNCTION**

Cache state as set of triples $S = \{(Q_l, f_i, \{C_j\})\}$ consisting of the chunks $j$ accessed from file $i$ at query $l$; Set of pairs $(f_i, \{C_j\})$ consisting of the chunks $j$ accessed from file $i$ at the current query $Q_{l+1}$; Cumulated cache budget $B = \sum_k B_k$ consisting of local budgets $B_k$ at each node

**Output:** Updated cache state $S' = \{(Q_{l+1}, f_i, \{C_j\})\}$

1. $S' \leftarrow \{(Q_{l+1}, f_i, \{C_j\})\}$
2. while $\text{budget}(S') < B$ do
3. Extract triple $t = (Q_l, f_i, \{C_j\})$ from $S$ such that $\text{budget}(t \cup S') \leq B$ and $\text{cost}(t)$ is maximum
4. $S' \leftarrow S' \cup t$
5. $S \leftarrow S \setminus t$
6. Increase cost of triples $t' \in S$ that contain chunks in $t$
7. end while

The time complexity of Algorithm 6 is proportional to the number of chunks in the cache state summed-up with the number of chunks accessed by the current query. If the total number of chunks – cached and accessed – is $N$, the complexity is $O(N \log N)$. The most time consuming operation is finding the best chunk to cache. This can be done by using a max binary heap having the eviction cost as key. Binary heap supports an efficient key increase operation required in line 6 of the algorithm. Compared to LRU, the proposed eviction algorithm also considers all the cached chunks. Since the memory budget is bounded, the number of cached chunks cannot grow dramatically. Moreover, we can control the number of generated chunks by the threshold on the minimum number of cells inside a chunk (Algorithm 5).

### 5.3.4 Cache Placement

Given the chunks selected to be cached, the next decision to make is how to allocate them over the distributed cache budget at the nodes. The main idea is to piggyback on the replication induced by query execution. Specifically, chunk-based join processing requires the chunk pairs to join to be collocated on the same node. Since one chunk joins with several other chunks, this induces replication. Notice that complete reshuffling is not desirable for sort-based partitioned arrays. The cache placement algorithm is performed after query execution and takes as input the location of all the currently cached chunks—including their replicas. The goal is to preserve a single copy of every
chunk to better utilize the cache. Thus, we have to decide the node where to cache every chunk. We propose a solution that maximizes the co-locality of correlated chunks based on the entire query workload in order to reduce network traffic during query processing. Moreover, the communication cost of moving cached chunks across nodes is also considered.

To this end, we design a local search greedy algorithm for cache placement (Algorithm 7) derived from incremental array view maintenance [174]. The algorithm takes as input the correlated chunk pairs across the performed queries, the location of the cached chunks, and the cache budget at every node. It outputs the updated cached chunk locations determined based on a cost function that measures the number of correlated collocated chunk join pairs:

\[
\text{cost}_{\text{placement}}(C_i, n, P', W) = \sum_{Q \in W} w_Q \cdot |C_j \in P'_n \land (C_i, C_j) \in Q|
\]

In this equation, \(C_i\) is the current chunk to place, \(n\) is a candidate node – typically chosen from where a replica of \(C_i\) exists – \(P'\) is the location of already placed chunks, and \(W\) is the query workload. In general, the larger the number of collocated chunks, the larger the cost is. However, the contribution of a chunk pair is scaled by an exponentially decayed query weight that gives priority to recent queries. The order in which chunks are considered has an important impact on the algorithm. Our strategy is to take the chunks in increasing order of their number of replicas. Chunks with more replicas have more choices without network transfer when the local cache budget decreases.

### Algorithm 7 Cache Placement

**FUNCTION** Set \(W = \{(Q_l, [(C_i, C_j)])\}\) consisting of chunk pairs \((i, j)\) that join at query \(l\); Set of locations \(P = \{(C_i, \{N_k\})\}\) specifying all the nodes \(k\) that have a copy of cached chunk \(i\) at current query \(Q_{l+1}\); Cache budget \(B_k\) at node \(k\)

**Output:** Updated locations \(P' = \{(C_i, N_k)\}\)

1. \(P' \leftarrow \{p \in P\}, \) where \(p\) has no replicas
2. for each \(p = (C_i, \{N_k\}) \in P\) with multiple replicas do
3.    Select node \(n \in \{N_k\}\) such that \(\text{budget}_n(C_i) \leq B_n\) and \(\text{cost}(C_i, n, P', W)\) is maximum
4.    \(P' \leftarrow P' \cup (C_i, n)\)
5. end for

The complexity of Algorithm 7 is \(O(|P| \cdot N + |W|)\), where \(|P|\) is the number of cached chunks, \(N\) is the number of nodes, and \(|W|\) is the size of the query workload. Since \(N\) is a small constant and \(|P|\) can be bounded during the chunking process, the first term cannot become too large. \(|W|\) can also be bounded by considering a limited number of previous queries. Moreover, the weight \(w\) of “old” queries is small enough to be ignored due to exponential decay.

### 5.4 Experimental Evaluation

The objective of the experimental evaluation is to investigate the overall query processing performance and the overhead of cost-based caching for two types of array similarity join query workloads over the PTF catalog stored in two different file formats. These are real queries executed
Figure 5.5: Query execution time as a function of the cache memory budget. Three caching strategies – file\_LRU, chunk\_LRU, and cost-based caching (evict) – are compared across three workloads and three file formats. The time for cost-based caching includes raw file chunking, cache eviction, and cache placement. Chunk\_LRU includes only raw file chunking, while file\_LRU is the time for caching complete raw files using LRU.

in the PTF pipeline for transient detection. We use the CSV LinkedGeoData dataset\(^6\) in order to confirm the behavior of array caching on another file format. Specifically, the experiments are targeted to answer the following questions:

- Does the proposed cost-based distributed caching improve upon file- and chunk-level LRU for a query workload?

- How sensitive is cost-based caching to query patterns, cache budgets, and file formats?

- How effective is query-driven array chunking?

- Does the cache assignment improve the query communication time across a series of queries?

- What is the execution time of the cost-based caching heuristics?

\(^6\)http://linkedgeodata.org
5.4.1 Setup

**Implementation.** We implement caching for raw arrays as a C++11 distributed multi-thread prototype that supports executing the array similarity join operator proposed in [172]. The catalog is stored at the coordinator. The distributed caching heuristic is also executed at the coordinator. It takes the input query from the query engine and generates the data communication plan containing information on chunk transfer and cache placement as output to the query engine. The query engine transfers chunks among nodes according to the data communication plan. The similarity join operator runs as a server on each node in the cluster. It manages a pool of worker threads equal to the number of CPU cores on each node. A worker thread is invoked with a pair of chunks that have to be joined. Chunks are retrieved directly from memory since the cache coordinator has already instructed each node to load the relevant chunks. This happens concurrently across all the workers. We use the CFITSIO\(^7\) and HDF5 libraries to read FITS and HDF5 files, respectively. Data serialization for network communication is optimized with the ProtoBuffers\(^8\) library.

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

**Data.** We use the same two real datasets as in [172] for experiments. The PTF catalog consists of 1 billion time-stamped objects represented in the equatorial coordinate system (ra, dec). The range of the time coordinate spans over 153,064 distinct values, while for ra and dec we use ranges of 100,000 and 50,000, respectively. In array format, this corresponds to:

\[
\text{PTF}[\text{time}=1,153064;\text{ra}=1,100000;\text{dec}=1,50000]
\]

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. Since HDF5 does not support sparse arrays natively, we store the PTF objects as an HDF5 table, i.e., relation. In FITS, data are stored as a binary table. Each tuple in HDF5 and FITS contains the dimensions and attributes for each cells in the original sparse array. The size of the PTF catalog is 343 GB in CSV, 262 GB in HDF5, and 221 GB in FITS.

**LinkedGeoData** stores geo-spatial data used in OpenStreetMap. We use the “Place” dataset which contains location information on roughly 3 million 2-D (long, lat) points-of-interest (POI). Since this is a too small dataset, we synthetically generate a larger dataset by adding 9,999 synthetic points with coordinates derived from each original point using a Gaussian distribution with \(\mu = 0\) and \(\sigma = 10\) miles \([137]\). In array format, this corresponds to:

\[
\text{GEO}[\text{long}=1,100000;\text{lat}=1,50000]
\]

With this replication, the size of GEO in CSV format is 1.6 TB. We split the entire dataset into 8,000 files with equal sizes.

\(^7\)https://heasarc.gsfc.nasa.gov/fitsio/
\(^8\)https://developers.google.com/protocol-buffers/
Query workloads. We extract 3 query workloads from the original datasets using four methods—PTF-1, PTF-2, and GEO. Each workload contains 10 queries. PTF-1 takes data exploration queries from the real workload that performs array similarity joins through all the detections on the time dimension. For PTF-2, we simulate a typical data exploration pattern—shifting ranges and alternating queries. We extract 4 range shifting queries from real workloads, enlarge the query ranges by a factor of $4 - 2X$ on $ra \times 2X$ on $dec$ — and let them appear in the workload alternatively, e.g., 1, 2, 3, 4, 1, 2, 3, 4, 1, 2. GEO is a workload that contains queries with shifting ranges. The size of the query range is fixed. During query 1 through 5, the queried range shifts in the same direction—we increase latitude with a constant, e.g., 3, 000. Then we shift them back to the original position, e.g., 1, 2, 3, 4, 5, 4, 3, 2, 1. Our small-size workloads mimic a data exploration behavior over raw arrays. In addition, the caching mechanism plays an even more important role when dealing with small workloads. Nonetheless, we also extract 100 queries from the real PTF workload and perform a stress test of our caching algorithms.

5.4.2 Results

The code contains special functions to harness detailed profiling data. 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 in the first query, i.e., cold caches.

5.4.2.1 Execution Time

The execution time for caching is the time to read the raw files, extract queried data, and execute the query. The execution time is depicted in Figure 5.5 for each individual query in the workload. We use file-level LRU – denoted $file\_lr$ – as the baseline. It utilizes all the memory across the cluster as a unified distributed memory and caches the least recent used files that contain queried data. $file\_lr$ works at file granularity because caching a subset of a file without the uncached bounding box still requires a full scan of the entire file—we have insufficient information to determine whether all the required data in the file are cached. $chunk\_lr$ computes bounding boxes during the query-driven chunking (Algorithm 5). It works at chunk level—caching the most recent queried chunks. $evict$ is the proposed cost-based caching that includes both cache eviction (Algorithm 6) and cache placement (Algorithm 7) on top of $chunk\_lr$. We use a cache budget of 40 GB, 80 GB, and 160 GB, respectively, to evaluate the scalability and behavior of the proposed algorithms. 40 GB is for the entire 8-node cluster, i.e., 5 GB for each node.

PTF-1. The workload execution time for PTF-1 on HDF5 format exhibits large variations among queries. This is mostly due to the skewed nature of the dataset—some regions of the sky contain more detections than others. Some of the query execution times are very close to 0 because raw file reading is not performed and the cache layout is not changed when all the queried data hit the cache. $evict$ always outperforms both LRU solutions. When we have 40 GB as cache budget, the difference varies across queries and is larger for the queries that share ranges—by as much as a factor of $20X$ for query 2, 6, 7, and 10. $chunk\_lr$ and $evict$ improve upon $file\_lr$ for all the queries, except the first—when they cannot even be applied. In the case of a larger cache budget, the difference between the LRU solutions and our cost-based caching is reduced because more available
caching space makes the cache selection problem easier. In this case, evict outperforms chunk_lru by 10-40%.

PTF-2. The execution time for PTF-2 on FITS format exhibits higher variance across the caching algorithms—especially when we have a small cache budget. The difference between file_lru and chunk_lru is smaller than for PTF-1 because the query ranges are enlarged and we only have a small budget. Since LRU always caches the most recent queried file, chunk_lru degenerates into file_lru. evict outperforms chunk_lru starting at query 3 and becomes 5-10X faster after query 4. This is because evict eliminates the case when large files are fully read only to extract a small amount of data. After increasing the cache budget to 160 GB, chunk_lru gets closer, however, evict is still more than 3X faster on query 5, 6, and 10.

GEO. Unlike the chunks in the real PTF dataset, chunking GEO files generates larger and sparser bounding boxes. file_lru cannot take advantage of the small cache budget. It is more than 10X slower than evict which manages to decrease the chunk size whenever no overlaps are detected. Moreover, evict outperforms chunk_lru by considering a weighted workload history to cover and read less files. This is exemplified by queries 6-10 which are the repetition of queries 1-5 in reverse order, where evict is 10X faster than chunk_lru.

5.4.2.2 Scalability with the Number of Queries

Figure 5.6 depicts the improvement in execution time generated by chunk_lru and cost-based caching over file_lru for every query in a workload of 100 PTF real queries. The cache budget is 160 GB, thus it favors the LRU algorithms. We observe that cost-based caching always outperforms the LRU algorithms by at most 20%. This is a significant value for such a large cache budget and such a large number of queries since LRU has sufficient time to optimize its execution. Moreover, the chunks are dense enough to generate overlap with the queries.

![Figure 5.6: Execution time improvement over file_lru on PTF in HDF5 with 160 GB cache budget.](image)

5.4.2.3 Optimization Time

The time to compute raw file chunking, eviction plan, and placement plan on LinkedGeoData with 40 GB cache budget is depicted in Figure 5.7. chunking is the time used to generate and look-up query-driven chunking for raw files. evict+place corresponds to the cost-based eviction plan.
and placement algorithms. Since the chunking is updated while we extract data from raw files, the updating time for splitting chunks is included in chunking. For the first query, it takes 8 seconds to compute the original chunking and fill the cache budget. After that, all the optimizations finish within 1.5 seconds. The optimization time is acceptable considering the significant reduction in execution time it brings—as much as 800 seconds or more.

Figure 5.7: Optimization time on LinkedGeoData.

### 5.4.2.4 Impact of Chunk Placement on Query Execution

Figure 5.8 shows the reduction in similarity join execution time due to cost-based cache placement. static is the time without the cost-based cache placement, while dynamic employs Algorithm 7 to cache chunks considering query co-locality. Since we process queries in a multi-thread pipeline that overlaps CPU computation and network communication, the execution is I/O-bound with 16 threads enabled on each node—the network communication time is the bottleneck. Cache placement piggybacks on the join data transfer and reassigns cached chunks to other nodes. Thus, the placement algorithm improves the cache organization and alleviates the network I/O by assigning the data required to be joined on the same node. As expected, there is no difference for the first query because cache placement is not invoked in the beginning. For PTF-1 the effect of placement is not significant since there is no pattern in the workload. In contrast, for PTF-2 and GEO that have a shifting range query workload, dynamic is 2-10X faster than the distributed LRU cache placement static. This gap clearly proves the benefit of the placement algorithm.

### 5.4.3 Discussion

The experimental results show that the proposed cost-based caching provides considerable improvement over distributed LRU for a query workload. Query-driven chunking improves over file-level LRU by 10-100X. On top of this improvement, the cost-based eviction algorithm accelerates caching by up to 100X. The file format has only a constant factor impact on data chunking—caused by the I/O format API libraries. When the cache budget is increased to 160 GB, the proposed algorithm is still more than 2X faster than chunk-level LRU. The cost-based cache placement algorithm improves the query communication time across a series of queries by a factor of 2-10X.
Figure 5.8: Reduction in similarity join execution time with cache placement (40 GB cache): (a) PTF-1, (b) PTF-2, and (c) GEO.

The time taken by the optimizations is a small fraction of the data chunking time—more so when compared to the reduction it generates.

5.5 Related Work

Array databases. While many array databases have been proposed over the years, they implement caching following the standard parallel database approach. That is, LRU is executed locally at each node and only for the chunks assigned to the node. In the following, we focus only on how these systems handle caching and point the interested reader to [133] for a comprehensive discussion on array database systems in general. RasDaMan [16] is a general middleware for array processing with chunks stored as BLOBs in a back-end relational database. RAM [154] and SRAM [42] provide support for array processing on top of the MonetDB [135] columnar database. They do not provide native support for arrays since arrays are represented as relations and array operations are mapped over relational algebra operators. While caching can be implemented in the middleware, this would replicate the functionality of the buffer manager in the back-end database system used to store the chunks. With integrated memory management and careful mapping from array chunks to disk pages, it is conceivable that an optimized array cache manager can be designed. RIOT [169] is a prototype system for linear algebra operations over large vectors and matrices mapped into a standard relational database representation. Linear algebra operations are rewritten into SQL views and evaluated lazily. Caching is offloaded entirely to the underlying database. SciDB [122] is the most advanced shared-nothing parallel database system designed specifically for dense array processing. It supports multi-dimensional nested arrays with cells containing records, which in turn can contain multi-dimensional arrays. The buffer manager in SciDB is independent for each node and caches only local chunks. In order to support non-chunked sparse arrays, these have to be projected to a single dimension. During the loading process, SciDB performs repartitioning to create compact chunks. This process is extremely inefficient [34, 166], though. SciHadoop [24] implements array processing on top of the popular Hadoop Map-Reduce framework which has only primitive caching support—the cache is replicated across all the nodes. ArrayStore [145], TrajStore [44], and TileDB [124] are storage managers optimized for multi-dimensional arrays and trajectories, respectively. They rely on the file system buffer manager for caching.
**Raw data processing.** At a high level, we can group raw data processing into two categories. In the first category, we have extensions to traditional database systems that allow raw file processing inside the execution engine. Examples include external tables [102, 113, 163] and various optimizations that eliminate the requirement for scanning the entire file to answer the query [9, 71, 73]. Modern database engines – e.g., Oracle, MySQL, Impala – provide external tables as a feature to directly query flat files using SQL without paying the upfront cost of loading the data into the system. NoDB [9] and Proteus [82] enhance external tables by extracting only the attributes required in the query and caching them in memory for use in subsequent queries. Data vaults [73] and SDS/Q [21] apply the same idea of query-driven just-in-time caching to scientific repositories. OLA-RAW [36] caches samples rather than full columns. Adaptive partial loading [71] materializes the cached data in NoDB to secondary storage before query execution starts—loading is query-driven. ReCache [12] chooses the format in which to cache nested in-memory data adaptively. SCANRAW [33] is a super-scalar adaptive external tables implementation that materializes data only when I/O resources are available. Instant loading [147] introduces vectorized SIMD implementations for tokenizing. RAW [84] and its extensions VIDa [82, 83] generate scan operators just-in-time for the underlying data file and the incoming query. The second category is organized around Hadoop MapReduce which processes natively raw data by including connector code in the Map and Reduce functions. Invisible loading [5] focuses on eliminating the connector code by loading the already converted data inside a database. While similar to adaptive partial loading, instead of saving all the tuples into the database, only a fraction of tuples are stored for every query. None of these solutions supports in-situ processing over sparse arrays—the central contribution of this work. SciDB connectors for multi-dimensional arrays stored in HDF5 are introduced in [66, 166]. They allow SciDB to execute declarative array queries over dense arrays stored in HDF5 format by pushing down the subarray operator into the HDF5 read function. However, they do not support sparse arrays and caching—pushed to SciDB.

**Distributed caching.** In the context of relational databases and storage systems, there is extensive work on cache eviction algorithms such as the LRU-K [120], DBMIN [38], and LRFU [94]. Unlike our framework, these algorithms operate on fixed size pages – not raw files of sparse arrays – or for semantic caching [46]. In the web context, many caching policies have been developed for variable-size objects. Some of the most well-known algorithms in this space are LRU-Threshold [6], Lowest-Latency-First [164], and Greedy-Dual-Size [26]. In the context of Hadoop systems, Impala⁹ and Hortonworks¹⁰ allow users to manually pin HDFS files or partitions in the HDFS cache. This can be done automatically with adaptive algorithms in [58]. In Spark¹¹, RDDs can be cached manually in Tachyon [95], a distributed in-memory file system. While these solutions work for raw files, they do not consider the multi-dimensional structure of arrays—they simply cache non-structured chunks. Distributed caching for spatial mobile data is considered in [99] and [69]. They cache the chunks based on the spatial distance between chunks over the non-overlapping partition of the data—that is not applicable to the unorganized raw arrays.

---

⁹https://impala.apache.org/
¹⁰https://hortonworks.com/
¹¹https://spark.apache.org/
5.6 Conclusions and Future Work

In this chapter, we introduce a distributed framework for cost-based caching of multi-dimensional arrays in native format. The framework computes an effective caching plan in two stages. First, the plan identifies the cells to be cached locally from each of the input files by continuously refining an evolving R-tree index. In the second stage, a cost-based assignment of cells to nodes that collocates dependent cells in order to minimize the overall data transfer is determined. We provide a cost-based formulation for cache eviction that considers the historical query workload. A thorough experimental evaluation over two real datasets in three file formats confirms the superiority of the proposed framework over existing techniques in terms of caching overhead and workload execution time by as much as two orders of magnitude. In future work, we plan to explore how the proposed framework can be integrated with high-performance computing caching architectures such as the Burst Buffer\textsuperscript{12} and with the SciDB buffer manager.

\textsuperscript{12}www.nersc.gov/users/computational-systems/cori/burst-buffer
Chapter 6

Conclusions

In this chapter, we conclude this dissertation by reflecting on array data chunking strategies, communication and computation balances for distributed array databases, limitations of our approach, and opportunities for future work in array data processing. We conclude with a final discussion on the results contained herein.

6.1 Array Data Organization

We encountered a set of recurring patterns that assisted in our design process in querying and processing array data during the development of the algorithms and systems described in this dissertation. We outline the three in the hope that they act as helpful rules of thumb and guidelines for future systems.

On-the-fly reorganization. In the online settings, queries are given at runtime. In distributed settings, reorganization plays a pivotal role in communication and computation balances. A static chunk-to-node assignment is suboptimal for online data exploration queries or data updates in the view maintenance scenario. A workload-based reassignment can be performed without any additional cost by piggybacking the necessary communications in the query execution [174].

I/O and computation balancing. Modern computing clusters have tens of cores on each node. When the computation workload is balanced, the bottleneck of scientific data processing becomes I/O—network communications and disk accesses. Scientific data analysis issues complex queries—convolution, stencil. Those theta-join queries can be generalized in array similarity join [172]. The query execution requires co-locating multiple chunks into one node to perform the join computation. A pre-computed optimized communication plan that contains data transfer plan, node communication schedule and disk access plan significantly outperforms MapReduce or structural join algorithms [172].

Caching/partial loading. Unlike the optimized database binary representation, the raw array data formats are not efficient and natively designed to be used in the high-performance and large-scale analysis. One noticeable characteristic of in-situ array data processing is that only a small portion
of data is heavily used in data explorations [171, 173]. Reorganizing the raw data and caching them in the memory is a generalization of the partial loading techniques. Unlike the conventional cache eviction strategies, caching on raw data focuses on minimizing the raw file reading costs instead of cache misses [173]—because reading any portion of a file requires a complete file scan in many raw file formats, e.g., CSV. A distributed caching over raw array files improves the data exploration workload executions by as much as two orders of magnitude.

6.2 Limitations and Future Work

While we believe that the frameworks and techniques in this dissertation are beneficial and useful, they have several limitations. In this section, we outline four of them, address the possible solutions, and discuss some promising research directions for future work.

Online workload prediction. In the current online solutions, we use a window of historical workload in a LRU fashion to optimize the execution. In real data analyses, data scientists issue queries in a specific pattern following their research logic. We plan to find out and understand the pattern, and predict the future workload based on the historical queries. After a query pattern is recognized, we can use it to hint the optimization procedure to produce better execution plan.

Coordinator. Most of the techniques in this dissertation requires a lightweight coordinator that is able to harness tens of computing nodes. The coordinator will eventually be overwhelmed after adding more and more nodes into the cluster. In order to scale up further, we target a coordination-free execution [13] or a self-election decentralized system.

Replication. The system we designed follows the shared-nothing architecture. Since scientific data are mostly immutable—there is no data consistency problem when editing data, replications can be considered as a trade-off between storage and communication cost. In the future work, we plan to build a storage engine that dynamically figures out the replication level for each chunk. Previously, the chunk replicated in the execution time is dropped after query is finished. The dynamical replication can also piggyback on the runtime chunk communications.

Multi-query processing in array database. Our system processes query batches sequentially, and treats those queries independently. However, the queries across the batch may share some execution components. We plan to share the query processing and execute multiple queries together to reduce communication and computation costs. Meanwhile, processing as many queries as possible in one batch is not guaranteed to be optimal because threads and working memory budgets are limited. Finding the optimal method to partition query workloads into batches is also a challenging problem.

6.3 Closing Thoughts

Scientific data processing is different from the traditional relational model—scientific data contain hundreds of attributes in one single table and scientific analysis applies complex queries
over the big volume and variety of the multidimensional data. It creates an opportunity to build specialized scientific databases to tackle these challenges.

In this dissertation, we develop an end-to-end solution to process scientific array data and queries. First, we propose and optimize array similarity join operator that is a generalization of convolution, stencil and other distance-based operators. Second, we define view for array data and investigate incremental view maintenance for arrays. After that, we propose an effective user-defined function execution framework on dense arrays. Finally, we propose a distributed caching over raw arrays that enables scientists to perform complex array queries on raw files. Our system is integrated in PTF pipeline, and it plays a pivotal role in the first observation of a neutron star merger [85].
Bibliography


Y. Cheng and F. Rusu. Parallel In-Situ Data Processing with Speculative Loading. In *SIGMOD 2014*.


Y. Cheng, W. Zhao, and F. Rusu. Bi-Level Online Aggregation on Raw Data. In *SSDBM 2017*.


115


118


