

Index-Based Multidimensional Array Queries: Safety and Equivalence

[Extended Abstract]

Rona Machlin
New York, New York, USA
rona@panix.com

ABSTRACT

We propose a new multidimensional array query model giving array bounds and other shape-related metadata a central role. Arrays are treated as shaped maps from indices to values. Schemas are augmented by shape constraints. Queries also have shape preconditions. Within this framework, we introduce the *index-based* array queries expressing index reorganizations and value summarizations. We define them via a declarative, rule-based language with shape-membership constraints in its rule bodies and subscripting and aggregation in its rule heads. We explore safety (including bounds analysis) and query equivalence for various subclasses divided according to the aggregator type, whether we allow disjunctions, and whether we allow (limited) Presburger arithmetic in index and shape terms. We show safety is tractable in the nonarithmetic cases, while state safety remains in **P** in the arithmetic ones. We show that, for a class of monoid-based set and bag aggregators, equivalence reduces to equivalence of *index-cores* – core queries collecting array indices rather than values. For set-aggregator queries, we give complete characterizations of equivalence in terms of containment maps and show the equivalence problems are in **P** in the nonarithmetic, conjunctive case and in **coNP** in all others.

Categories and Subject Descriptors: H.2.3 [Languages]:Query languages; H.2.1[Logical Design]:Data models; E.1 [Data Structures]:Arrays; H.2.8 [Database Applications]:Scientific databases.

General Terms: Languages, Theory.

Keywords: array query languages, multidimensional data, scientific data, query equivalence, aggregation, integer linear constraints.

1. INTRODUCTION

Dense, multidimensional arrays form a ubiquitous class of persistent data. In the earth sciences alone, numerous data repositories publish datasets containing data conceptually organized as arrays. The datasets can represent products derived from observed data as well as gridded outputs from model simulations. The volume of data available from such repositories is enormous; for example, the Earth System Grid (ESG) project[1] climate simulation archives are expected to grow to petabytes.

What distinguishes array data from data fitting into multidimensional, functional relations (typical of the models underlying most

OLAP[6] and statistical[7] systems) is that it is *shaped*. By this, we mean that dimensions have associated lower and upper bounds, that the indices in each dimension range over a discrete linear order, and that, for each multidimensional index within the respective bounds, there is always an array value present. In contrast, multidimensional, functional relations can be sparse, and their dimension ranges need not be bounded, nor even linearly ordered.

An array's shape, that is, the tuple of pairs giving lower and upper bounds for each dimension, is a form of metadata. It serves both as typing information and as data. Like types, array bounds describe the data's structure and can be used statically, during bounds analysis, to check for out-of-bounds accesses. On the other hand, since their values vary from instance to instance, array bounds are also a form of data. In fact, a large class of array queries arises from selecting over metadata rather than data.

In this paper we introduce such a class, the *index-based* array queries. These queries combine structural manipulations of array data with summarizations of the data via aggregations. We call them index-based because their selections and joins are over metadata or array indices bounded by metadata. Array values are not used in selection or join conditions, though the values associated with selected indices can be included or aggregated in the result.

Our queries are similar to range queries, but are more general: our ranges can fall along diagonals, include multiple intervals or involve strides or subsampling. In addition to expressing aggregations over these more general ranges, our queries can express restructuring operations – zips, unzips, dices, slices, rotations, shifts, regriddings, merges, to name a few. All this is achieved via a declarative, rule-based language using shape constraints in its rule bodies, subscripting and aggregation in its rule heads, and limited Presburger (integer linear) arithmetic in its index and shape terms.

Our queries are well-behaved in that they have efficient bounds analysis and rule minimization procedures. We discuss these in the text but spend most of the paper exploring our queries' equivalence problem. This is an important problem to study in any model, as it forms the basis for such central issues as optimization and view usability. In the relational setting, equivalence and containment have been investigated for various subclasses. Of relevance to our context are the results for queries with comparisons and constants[14, 26, 20, 9] and for ones involving aggregation[8, 20, 9, 12].

Our query model makes essential use of both comparisons and aggregation. Comparisons are needed because, instead of using relations to limit variable ranges, we use dimension shapes, that is, metadata-bounded intervals. Aggregation comes into play because our query outputs are shaped (dense) arrays. This means every index in the output array shape must have exactly one value associated with it. We use aggregation to coalesce multiple values as well as to produce nulls when no values are generated.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PODS'07, June 11–13, 2007, Beijing, China.

Copyright 2007 ACM 978-1-59593-685-1/07/0006 ...\$5.00.

The problem of characterizing query equivalence in the presence of aggregation is notorious for requiring custom approaches for different aggregators [8]. We limit our focus to *collection aggregators* [15], a monoid-based class for which we obtain a uniform approach. We prove equivalence between collection aggregator queries reduces to equivalence between their index cores, core queries collecting array indices instead of array values. For index cores using set semantics, we give both necessary and sufficient conditions for equivalence via array containment maps.

In the relational model, the equivalence problem for queries with comparisons that use set-based aggregators (such as *Max*) is Π_2^P -complete [20, 9, 14, 26]. This holds for both conjunctive and disjunctive queries. Because we lack relation literals, we can put all our queries into a connected normal form and, thus, achieve much tighter bounds for equivalence. In fact, for conjunctive queries, we obtain polynomial decidability, and with disjunctions decidability only goes up to **coNP**.

The situation with arithmetic is even more dramatic. The decision problem for Presburger arithmetic with as few as two quantifier blocks has an exponential lower bound [11]. For example, this lower bound holds for deciding equivalences of constraint database queries with integer linear constraints [13]. Our complexity stays within **coNP**. Again, this is due to connectedness as well as our ability to eliminate embedded quantifiers using containment maps.

The key idea is we distinguish between *variables*, ranging over shapes, and *parameters*, fixed by the metadata and used to define these shapes. Our terms can contain any number of parameters, but only one variable. This gives our queries enough expressive power. It also leads to polynomial variable elimination, though parameter elimination complexity remains high. We use variable elimination to put our queries into connected normal form. This allows us to define containment maps, despite the arithmetic, thus eliminating the expensive Presburger quantifier alternation.

In addition to array bounds, our schema’s shape parameters can include other scalar (integer) metadata related to array shapes, such as distinguished indices, subranges, offsets or lengths. Their values are left unspecified, allowing for shape information that is dynamic or incomplete or simply varies from instance to instance.

A feature of our framework is that it makes data and query shape semantics fully transparent. Schemas declare constraints that hold of their shape parameters. Each query also declares its intended shape semantics by explicitly defining its output’s shape and listing the shape assumptions it makes of its inputs. Factoring input shape constraints into ones declared in the schema and ones required by the query supports interoperability. It means data can be published without having to tightly couple it with visualization/analysis packages or other domain-specific query software. It also means general query operators (i.e., query templates) can be published without a priori knowledge of the data they will be applied to.

This paper’s main contributions are: (1) an array data model taking into account shape-related metadata, (2) the identification of a broad class of well-behaved queries, (3) an analysis of safety for this new query paradigm, (4) preliminary results on reducing the equivalence problem and (5) a sound and complete characterization of equivalence for those queries that use set aggregators.

Related Work. In [18], Maier and Vance argue relations don’t provide a natural fit for array data because they fail to capture multidimensional index order, contiguity and rectangularity, i.e., shape. Consider applying a local filter involving each cell’s neighbors to every value in an array. Using a relational representation of arrays this can involve an expensive multiway join. However, with shape and subscripting available in our query language, such queries can be expressed more directly, paving the way for more efficiency.

Several other array query models have been proposed. In [16], we added constructs for tabulating arrays, subscripting, and extracting dimension lengths to a nested relational algebra with arithmetic and aggregation. We showed many array optimizations arise automatically from partial function beta-eta normalization when tabulations, subscripting and shape extraction are treated as partial function lambda abstraction, application, and domain extraction.

The most comprehensive effort to date is Baumann et al.’s full commercial DBMS (RasDaMan) [3]. Its high-level array query language (RasQL) is based on a simple array algebra [4] with three operators: a tabulation-style array constructor, a condenser for computing aggregations and an index sorter.

These proposals and others [18, 25] use tabulations as their main constructor. Tabulations define arrays by applying an index-to-value map over the output shape. Another approach is to use comprehensions, which define arrays via collections of (index,value) pairs, where multiple values are coalesced using aggregation. Fe-garas and Maier do this in their object query language based on monoids [10]. While tabulations create their output in bulk and access their inputs in random order, comprehensions do the opposite. Our rule-based approach mixes these two styles, allowing random access order for both input and output.

A third approach is to define a fixed set of higher-level constructs designed to capture specific optimizations. Baumann does this in an earlier version of his array algebra [2]. This is also the approach taken by Marathe and Salem in their array manipulation language, AML [19]. AML’s focus is on exploiting the structural properties of application-specific functions that take array chunks as inputs and produce array chunks as outputs. Its algebra has three operators. The first two, subsample and merge, allow selections and interleavings, respectively, of array hyperslices according to bit patterns. The last one, apply, is a generalized map for applying chunk-to-chunk functions uniformly over an array.

A problem with higher-level algebras is that the shape semantics used is often arbitrary and not transparent. For example, an operator to append two 2-dimensional arrays along their first dimension needs to choose what to do when the arrays’ second dimensions have different bounds. It can be undefined on such arrays or it can pad both out with nulls to make their second dimensions match. For purposes of interoperability it is important that query and data shape semantics be made explicit, as is done in our model.

Several of the models [4, 10, 16] embed their array constructs in a (nested) relational setting allowing selection of array data according to metadata and definition of new arrays over shapes built from selected metadata. However, none models metadata semantics using constraints or gives any account of safety. In fact, for the full generality of index expressions allowed in these approaches, bounds safety is undecidable [16]. Also, though all the proposals define equational transformations for their queries, to our knowledge, we are the first to give a necessary as well as sufficient characterization of array query equivalence.

Since our queries involve integer linear constraints, they share issues of expressivity and complexity with some constraint and temporal database languages [13, 24, 22, 5]. A difference between the constraint model and ours is that our shapes are metadata, not data; a crucial part of array data is the map from index sets to values.

Outline. In Sect.2, we define our new array data model and introduce the class of aggregators studied. In Sect.3, we give the syntax and semantics of our language for index-based querying of arrays. We also discuss safety, including with respect to bounds errors. The next two sections are devoted to the query equivalence problem. In Sect.4, we reduce it to equivalence between index cores and then characterize equivalence for set-aggregator

queries with nonarithmetic terms. In Sect.5, we extend our results to queries with arithmetic. We touch on query minimization and optimization in Sect.6. In Sect.7, we discuss query expressivity.

2. PRELIMINARIES

A k -dimensional *shape* is a k -tuple of ordered pairs of integers $(l_1 : u_1, \dots, l_k : u_k)$ giving lower and upper bounds for each dimension. Its *index set* is the set of k -tuples of integers falling within the bounds: $\{(i_1, \dots, i_k) \mid l_j \leq i_j \leq u_j \text{ for all } 1 \leq j \leq k\}$. A shape is *nonempty* if $l_j \leq u_j$ for all $1 \leq j \leq k$.

A k -dimensional *array* A is a nonempty k -dimensional shape, together with a map from the shape's index set to a domain of values. We use $lb_j(A)$ and $ub_j(A)$ to denote the lower and upper bounds of A 's j th dimension and $Shape(A)$ to denote A 's shape as a whole; we overload $Shape(A)$ to also mean the index set. For any k -ary index d in $Shape(A)$, we use the usual subscript notation $A[d]$ to denote the value A associates with d . Array values can be m -tuples of reals ($m \geq 1$) or collections of m -tuples, where the collections can have either set or bag semantics.

A *flat array declaration* has the form $A : k \rightarrow m$, where A is an array name, k gives the arity of its indices (i.e., its dimensionality), and m gives the arity of its values. A *collection array declaration* has the form $A : k \rightarrow \{m\}$ or $A : k \rightarrow \{\!\{m\}\!\}$, depending on whether the values are sets or bags of m -tuples. An *input schema* $S = (A_1 : k_1 \rightarrow m_1, \dots, A_n : k_n \rightarrow m_n; s_1, \dots, s_p)$ is a list of flat array declarations followed by a list of symbolic constants, which can be used to declare extra (integer) metadata related to shape, such as distinguished indices, offsets, subranges or lengths. An *output schema* is just a flat or collection array declaration.

Every input schema S has an associated set of parameters, $Params(S)$, which includes the symbolic constants of S , and for each array A_i in S and each dimension $1 \leq j \leq k_i$, terms of the form $lb_j(A_i)$ and $ub_j(A_i)$ representing the lower and upper bounds of A_i 's j th dimension. Each schema also has an associated set of parameter constraints, Γ_S , where the constraints are conjunctions of comparisons over terms built up from the parameters and integer constants. We require Γ_S to contain at least the following constraints guaranteeing all arrays in S are non-empty: $NonEmpty(S) = \{lb_j(A_i) \leq ub_j(A_i) \mid A_i \in S, 1 \leq j \leq k_i\}$.

An *array database instance* I for a schema S is a map assigning an array A^I to each array name A and an integer s^I to each symbolic constant s . We extend $(-)^I$ homomorphically to terms (e.g., $(lb_j(A))^I = lb_j(A^I)$), comparison predicates, and conjunctions/sets of comparisons Γ . Then the definitions of satisfaction under I ($I \models \Gamma$) and logical implication ($\Gamma_1 \models \Gamma_2$) are standard. Note that by the way we defined arrays, for any instance I , $I \models NonEmpty(S)$. We only consider instances of S that satisfy all of Γ_S , and to emphasize this we call them S -instances. We write $\Gamma_1 \models_S \Gamma_2$ as shorthand for $\Gamma_1 \cup \Gamma_S \models \Gamma_2$.

Example 1. Consider a two-dimensional array T_{core} giving core temperatures and nitrate concentrations at different depths into the earth each hour throughout a day. For each instance, we wish to flag the water table depth and the hours from sunrise to sunset. This data and metadata is represented by:

$$\begin{aligned} S &= (T_{core} : 2 \rightarrow 2; d_{WT}, h_{SR}, h_{SS}) \\ \Gamma_S &= \{lb_1(T_{core}) \leq d_{WT} \leq ub_1(T_{core}) = 0 \wedge \\ &\quad 0 = lb_2(T_{core}) \leq h_{SR} \leq h_{SS} \leq ub_2(T_{core}) = 23\} \end{aligned}$$

The schema declares T_{core} to have two dimensions, depth and hour, and two value components, temperature and nitrate concentration, and also declares symbolic constants, d_{WT} , h_{SR} , and h_{SS} , for the metadata. The first constraint says the water table depth is a distinguished index of the depth dimension, whose upper bound is 0,

α	$\cup$	Min	Max	ι	$\uplus$	Σ	Π
(a) τ_α	$\{\tau\}$	$\mathbb{R}^+$	$\mathbb{R}^+$	τ^+	$\{\tau\}$	$\mathbb{R}^+$	$\mathbb{R}^+$
op_α	$\cup$	min	max	op_ι	$\uplus$	$+$	$*$
$null_\alpha$	$\{\}$	$+\infty$	$-\infty$	$\perp_\tau$	$\{\!\{\}\!\}$	0	1

where $\mathbb{R}^+ = \mathbb{R} \cup \{-\infty, +\infty\}$ and for any $\tau, \tau^+ = \tau \cup \{\perp_\tau, \top_\tau\}$

(A1) $\pi_l \circ (\alpha_1, \dots, \alpha_m) = \alpha_l \circ map_s(\pi_l)$
(A2) $\alpha \circ \cup = op_\alpha \circ (\alpha \times \alpha)$
(A3) $\alpha(\{a\}) = a$
(b) (A4) $\alpha(\{\!\{\}\!\}) = null_\alpha$
(A5) $(\exists a)\alpha(\underbrace{\{a, \dots, a\}}_{(n \text{ times})}) \neq \alpha(\underbrace{\{a, \dots, a\}}_{(m \text{ times})})$ for all $n \neq m$

where we obtain bag-aggregator versions of (A1)-(A4) by substituting $\{\!\{\}\!\}, \uplus$ and $map_b(\pi_l)$ for $\{\}, \cup$ and $map_s(\pi_l)$ ¹

Figure 1: The collection aggregators and their identities

representing surface-level. The remaining constraints say the daylight period is a distinguished range in the (24-hour) time dimension. Note that $(lb_1(T_{core}))^I$, $(d_{WT})^I$ and $(h_{SR}, h_{SS})^I$ can all vary with I , as might be the case if instances represent cores drilled to different depths at different locations and/or on different days.

Collection Aggregators. An important difference between the relational model and ours is that array queries produce *shaped* arrays as output. This means each query must define an output array shape and also assign exactly one value to each of its indices. Our queries are defined in terms of rules that generate (index,value) pairs. Aggregation is used both to coalesce multiple values generated for the same index into one and also to assign a default value to any index for which nothing is generated. *Collection aggregators*, a class of commutative monoid-based aggregators similar to the one introduced in [15], work well for this purpose.

We divide these into set or bag aggregators according to their equational theory. Let $\{\tau\}$ and $\{\!\{\tau\}\!\}$ denote the types of finite sets and bags with elements of type τ . We use $\|\tau\|$ to refer to these collection types generically. A collection aggregator is a function $\alpha : \|\tau_\alpha\| \rightarrow \tau_\alpha$ defined in terms of an associated monoid $(\tau_\alpha, op_\alpha, null_\alpha)$ as follows: $\alpha(\|a_1, a_2, \dots, a_n\|) = (a_1 op_\alpha (a_2 op_\alpha \dots (a_n op_\alpha null_\alpha) \dots))$. For set aggregators, the associated monoid must be commutative and idempotent, making the aggregation independent of element order or repetitions; for bag aggregators, the monoid need only be commutative.

Fig.1(a) lists some set and bag aggregators and their monoids. (Bag aggregators are on the right.) Note the distinction between the aggregators $\cup / \uplus$, which flatten collections of collections, and their monoid operators $\cup / \uplus$, which are just (binary) set/bag unions. We include a nonstandard set aggregator ι ("the unique"), which we'll use to express nonaggregating, restructuring queries. It has a null, $\perp$, and a zero, $\top$. On a set of values, ι returns the unique non-null element if there is one; otherwise, it returns $\perp$ if all elements are null and $\top$ if there is more than one non-null element.

In addition to these aggregators, for $\alpha_1, \dots, \alpha_m \notin \{\cup, \uplus\}$, we also allow the m -ary *product aggregator* $(\alpha_1, \dots, \alpha_m) : \|\tau_{\alpha_1} \times \dots \times \tau_{\alpha_m}\| \rightarrow \tau_{\alpha_1} \times \dots \times \tau_{\alpha_m}$. Its monoid is defined pointwise: $\tau_{(\alpha_1, \dots, \alpha_m)} = \tau_{\alpha_1} \times \dots \times \tau_{\alpha_m}$, $op_{(\alpha_1, \dots, \alpha_m)}(a, b) = (op_{\alpha_1}(\pi_1 a, \pi_1 b), \dots, op_{\alpha_m}(\pi_m a, \pi_m b))$, and $null_{(\alpha_1, \dots, \alpha_m)} = (null_{\alpha_1}, \dots, null_{\alpha_m})$. If any of the α_j 's are bag aggregators, so is the product aggregator; otherwise, it is a set aggregator. Product aggregators are used in building arrays whose values are m -

¹Here, π_l is tuple projection and map_s/map_b applies it to every element of a set/bag, so $map_s(\pi_l)/map_b(\pi_l)$ is just the familiar relational projection operator for the l th column using set/bag semantics.

tuples. They summarize the value components simultaneously (in lockstep), but independently.

Figure 1(b) lists some identities governing collection aggregators that will be useful to us later (a full axiomatization is given in [15]). (A1) says the components of product aggregators can be computed independently. (A2) shows how aggregators distribute over (binary) unions. Note the difference between $(\alpha, \alpha) : \|\tau \times \tau\| \rightarrow \tau \times \tau$ and $(\alpha \times \alpha) : \|\tau\| \times \|\tau\| \rightarrow \tau \times \tau$. In fact, by (A1), $(\alpha, \alpha) = (\alpha \times \alpha) \circ \langle \text{map}(\pi_1), \text{map}(\pi_2) \rangle$. (A3) says aggregating a singleton returns its value. (A4) says aggregators return their null value on empty collections. Finally, (A5) says bag aggregators have at least one value on which they respect multiplicities. These identities are only guaranteed to hold for collection aggregators. For example, (A2) fails for Avg and (A3) fails for Cnt.

3. INDEX-BASED ARRAY QUERIES

The syntax for our new class of queries is similar to relational rule-based syntax [20] but takes into account our query inputs and outputs are *shaped* arrays. On the output side, this means our queries must each define an output array shape and also assign a (possibly aggregated or null) value to each index. On the input side, it means instead of using relations to establish variable ranges, our queries must somehow use arrays. We choose to let variables range over meta-data-bounded intervals, that is, shapes, and to use subscripting to access array values. The fact that our inputs and outputs are shaped also introduces safety issues unique to our model. After presenting our query syntax, we detour to discuss safety before going on to present our query semantics.

Query Terms and Comparison Atoms. Because of the different roles they play, we separate our query terms into three sorts: *Bounds terms* are used to define output array shapes and to give ranges for input and output array indices. *Index terms* are used to give these indices. *Value terms* are used to give the output array values associated with output array indices.

In building up these terms, we make use of the integer functions in Fig.2(a). The bounds functions, $\mathcal{F}_b$, include addition, multiplication by an integer constant, two forms of division by an integer constant – rounding down and rounding up, and (binary) minimum and maximum. The index functions, $\mathcal{F}_i$, include only addition and multiplication by an integer constant.

Fix a schema $S = (A_1 : k_1 \rightarrow m_1, \dots, A_n : k_n \rightarrow m_n; s_1, \dots, s_p)$ and a set of variables V . Figure 2(b) gives the formation rules for query terms according to their sort. A *bounds term* is either a parameter in $\text{Params}(S)$ (i.e., a symbolic constant or an array lower or upper bound) or an integer constant,² or it is built up from these using functions $f_b \in \mathcal{F}_b$. An *index term* is built up from variables and bounds terms using $f_i \in \mathcal{F}_i$. A *value term* is a component of an array value obtained by subscripting index terms, that is, a term of the form $\pi_l(A_i[\mathbf{i}, \dots, \mathbf{i}])$, for some $1 \leq l \leq m_i$. When $m_i = 1$, we simply write $A_i[\mathbf{i}, \dots, \mathbf{i}]$.

In building index terms, we constrain the functions $f_i \in \mathcal{F}_i$ to take bounds terms in all arguments except one. This causes index terms to contain at most one variable. The number of parameters in index or bounds terms is unrestricted. We call index terms containing a variable *open* and ones that don't *closed* and write $t(x)$ to indicate t is open with variable x . Any index term $t(x)$ can be put into the canonical form $n * x + b$, where n is an integer constant and b is a bounds term. When a bounds term contains no min, max,

²Note how symbolic constants $s \in S$ differ from integer constants $(1, 2, \dots, n, m, \dots)$. The former, along with array bounds, are schema parameters and are uninterpreted. (Their values vary from instance to instance.) The latter, along with function symbols, are interpreted. (They always take on the same values.)

$$\begin{array}{ll}
 \text{(a)} & \mathcal{F}_b = \{+, -, *, n, \text{div}^\downarrow n, \text{div}^\uparrow n, \min, \max\} \\
 & \mathcal{F}_i = \{+, -, * n\} \\
 & \text{where } n \in \mathbb{Z} \text{ and } n \neq 0 \\
 \text{(b)} & \mathbf{b} ::= s_j \mid lb_j(A_i) \mid ub_j(A_i) \mid n \mid f_b(\mathbf{b}, \dots, \mathbf{b}) \\
 & \mathbf{i} ::= x \mid \mathbf{b} \mid f_i(\mathbf{i}, \mathbf{b}, \dots, \mathbf{b}) \\
 & \mathbf{v} ::= \pi_l(A_i[\mathbf{i}, \dots, \mathbf{i}]) \\
 & \text{where } s_j, A_i \in S, n \in \mathbb{Z}, x \in V, f_b \in \mathcal{F}_b, f_i \in \mathcal{F}_i \\
 \text{(c)} & \mathbf{C} ::= [\mathbf{b} \leq \mathbf{i} \leq \mathbf{b}] \mid \mathbf{i} = \mathbf{i} \mid \mathbf{b} \mathbf{R} \mathbf{b} \\
 & \text{where } R \in \{\leq, <, =\}
 \end{array}$$

Figure 2: The array query (a) term constructors (b) terms and (c) comparison atoms

$\text{div}^\downarrow$ or $\text{div}^\uparrow$, it too can be put into a canonical form, $\sum_i n_i p_i + n$, where the p_i are parameters and the n, n_i are integers.

Index and bounds terms containing no functions must be either variables, parameters or constants. We refer to such terms (and the query fragment built up from them) as *nonarithmetic*.

A *query comparison atom* (see Fig.2(c)) has the form $[l \leq t \leq u]$, $t_1 = t_2$, $b_1 \leq b_2$ or $b_1 < b_2$, where the t 's are index terms and l , u , and the b 's are bounds terms. The first form is called an *index bounds atom* and is just shorthand for the conjunction $l \leq t \wedge t \leq u$. We use the specialized syntax to limit how variables can occur. In fact, below, we define a notion of range-restriction in which index bounds atoms play a syntactic role (similar to relation literals in the relational paradigm) in limiting variables' ranges.

A *conjunctive variable constraint* (CVC) is a conjunction of query comparison atoms. We will use CVCs as our query rule bodies. A *conjunctive parameter constraint* (CPC) is a CVC containing no variables. The constraints in $\text{NonEmpty}(S)$ are of this form. We require all parameter constraints in Γ_S to be CPCs. We also will use CPCs to express query shape preconditions.

Rule and Query Syntax. An array *index:value rule* with output schema $Q : k \rightarrow m$ or $Q : k \rightarrow \|m\|$ has the form:

$$ti_1, \dots, ti_k : \alpha(tv_1, \dots, tv_m) \leftarrow C_1 \wedge \dots \wedge C_p$$

Everything to the left of the arrow is the rule *head* and everything to the right is the *body*. The head consists of a k -tuple of index terms and an m -tuple of value terms giving index:value pairs for the output array, together with a collection aggregator α consistent with the output schema Q (i.e., whose result arity is m and/or collection type is $\| \cdot \|$.) The rule body is a (possibly trivial) CVC.

The aggregator α serves two purposes. First, whenever more than one index:value pair is generated for the same index, the aggregator combines the multiple values. Second, if for any output shape index no index:value pair is generated, the aggregator produces a default null value for that index. When α is a product aggregator $(\alpha_1, \dots, \alpha_m)$, we will sometimes write the rule as $ti_1, \dots, ti_k : \alpha_1(tv_1), \dots, \alpha_m(tv_m) \leftarrow C_1 \wedge \dots \wedge C_p$.

An *index-based array query* Q over the output schema $Q : k \rightarrow m$ or $Q : k \rightarrow \|m\|$ is an expression of the form:

$$\begin{array}{l}
 D_1 \wedge \dots \wedge D_q !! \\
 l_1 : u_1, \dots, l_k : u_k :: \\
 ti_1^1, \dots, ti_k^1 : \alpha(tv_1^1, \dots, tv_m^1) \leftarrow C_1^1 \wedge \dots \wedge C_{p_1}^1; \\
 \dots \\
 ti_1^n, \dots, ti_k^n : \alpha(tv_1^n, \dots, tv_m^n) \leftarrow C_1^n \wedge \dots \wedge C_{p_n}^n.
 \end{array}$$

It has three parts: a (possibly trivial) CPC $D_1 \wedge \dots \wedge D_q$ (referred to as *Guard*(Q)) giving the conditions under which the query is defined, a k -tuple of pairs of bounds terms $l_1 : u_1, \dots, l_k : u_k$ (referred to as *Shape*(Q)) giving the shape of the output array, and a (possibly empty) list of index:value rules (referred to as *Rules*(Q)) defining the output array contents. If there is at most one rule, the query is *conjunctive*; otherwise, it is *disjunctive*. In disjunc-

tive queries, all the rules must use the same aggregator α . We call queries that use α as their aggregator α -queries. We call Q m -ary if its output schema is $Q : k \rightarrow m$ or $Q : k \rightarrow \|m\|$.

The query guard is used to give shape preconditions that must hold of the inputs in order for the query to apply. These are in addition to the schema's parameter constraints, Γ_S , which are declared by the data publisher and can be assumed to hold.

Example 2. Consider the schema $S = (T : 4 \rightarrow 1, P : 3 \rightarrow 1; l_{\text{surf}}, h_{\text{sr}}, h_{\text{ss}})$ where T is a 4-dimensional array giving a day's hourly temperatures by longitude, latitude, atmospheric level and time, P is a 3-dimensional array giving precipitation totals by longitude, latitude and time and l_{surf} , h_{sr} and h_{ss} are symbolic constants representing the surface level index and sunrise and sunset hours. Let Γ_S contain the constraints $lb_3(T) \leq l_{\text{surf}} \leq ub_3(T)$, $\bigwedge_{i=1}^2 lb_i(T) = lb_i(P) \wedge ub_i(T) = ub_i(P)$, $lb_4(T) = lb_3(P) = 0 \leq h_{\text{sr}} \leq h_{\text{ss}} \leq 23 = ub_4(T) = ub_3(P)$, saying l_{surf} is an atmospheric level index, the arrays share bounds in their other dimensions, and the daylight period falls within the (24-hour) time dimensions. Then the following query produces a 2-dimensional, 2-ary array pairing each position's total daytime precipitation with its maximum daytime surface level temperature:

$$\begin{aligned} &!! lb_1(T) : ub_1(T), lb_2(T) : ub_2(T) :: \\ &x, y : \Sigma(P[x, y, t]), \text{Max}(T[x, y, l_{\text{surf}}, t]) \leftarrow [lb_1(T) \leq x \leq ub_1(T)] \\ &\quad \wedge [lb_2(T) \leq y \leq ub_2(T)] \wedge [h_{\text{sr}} \leq t \leq h_{\text{ss}}]. \end{aligned}$$

The query's first line gives the output array shape. It has two dimensions – longitude and latitude – whose lower and upper bounds are copied from T 's corresponding dimension bounds. No guard conditions are included, since we intend the query to be defined on all S -instances. The query has a single rule. The conditions in its body constrain x and y to range over all indices in T 's longitude and latitude dimensions and t to range over daylight hours. The terms in its head associate every longitude, latitude pair (x, y) with the sum of P 's values and the maximum of T 's surface-level values at that position over all hours between sunrise and sunset.

Since the output array's longitude and latitude bounds are equal to T 's, the (x, y) indices generated by the rule cover the entire output shape; had they not, the aggregators' nulls would have been assigned to any output indices missed by the rule. In general, the groupby implicitly computed by such a rule differs from a standard relational one in that it is shaped: every tuple in the output shape gets a (possibly empty) group and thus a (possibly null) value.

The queries in Fig.3 illustrate the use of guards, arithmetic and disjunctions. Suppose $A, B : 1 \rightarrow 1$. Then (Q1) and (Q2) give two typical ways to implement their zips (i.e., joins along their indices). The first version is only defined when A and B have the same bounds; the second is defined over their shapes' intersection, if it is nonempty. Notice how the use of both guards and output shapes makes the intended shape semantics of our queries explicit. (Q3) merges two zero-based arrays, putting A in even positions and B in odd ones; it is defined only if they have the same length. (Q4) reverses this, picking out the even positions of a zero-based array.

Let $M, N : 2 \rightarrow 1$ be two matrices. (Q5) appends them along their first dimension and is defined only if their bounds match in the second dimension. (Q6) computes $M[l_1 : n_1 : u_1, l_2 : n_2 : u_2]$, a variation on dicing that takes into account strides, where we assume l_1, l_2, u_1, u_2 are bounds terms and the strides, n_1, n_2 , are integer constants. It is only defined if $(l_1 : u_1, l_2 : u_2)$ is a non-empty shape falling within M 's bounds. If so, it picks out every n_j th index between l_j and u_j starting with l_j , along each dimension j . The resulting elements are returned in a zero-based matrix. This operation, referred to as “subsetting”, is typically included as part of the server-side functionality for OpenDap-compliant[1] datasets.

$$\begin{aligned} (Q1) \quad &lb(A) = lb(B) \wedge ub(A) = ub(B) !! lb(A) : ub(A) :: \\ &i : \iota(A[i]), \iota(B[i]) \leftarrow [lb(A) \leq i \leq ub(A)] \\ (Q2) \quad &\max(lb(A), lb(B)) \leq \min(ub(A), ub(B)) !! \\ &\max(lb(A), lb(B)) : \min(ub(A), ub(B)) :: \\ &i : \iota(A[i]), \iota(B[i]) \leftarrow [\max(lb(A), lb(B)) \leq i \leq \min(ub(A), ub(B))] \\ (Q3) \quad &lb(A) = lb(B) = 0 \wedge ub(A) = ub(B) !! 0 : ub(A) + ub(B) + 1 :: \\ &2 * i : \iota(A[i]) \leftarrow [0 \leq i \leq ub(A)] \\ &2 * i + 1 : \iota(B[i]) \leftarrow [0 \leq i \leq ub(B)] \\ (Q4) \quad &lb(A) = 0 !! 0 : ub(A) \text{div}^\downarrow 2 :: \\ &i : \iota(A[2 * i]) \leftarrow [0 \leq 2 * i \leq ub(A)] \\ (Q5) \quad &lb_2(M) = lb_2(N) \wedge ub_2(M) = ub_2(N) !! \\ &lb_1(M) : ub_1(M) + ub_1(N) - lb_1(N) + 1, lb_2(M) : ub_2(M) :: \\ &i_1, i_2 : \iota(M[i_1, i_2]) \\ &\quad \leftarrow [lb_1(M) \leq i_1 \leq ub_1(M)] \wedge [lb_2(M) \leq i_2 \leq ub_2(M)] \\ &\quad ub_1(M) + 1 + (i_1 - lb_1(N)), i_2 : \iota(N[i_1, i_2]) \\ &\quad \leftarrow [lb_1(N) \leq i_1 \leq ub_1(N)] \wedge [lb_2(N) \leq i_2 \leq ub_2(N)] \\ (Q6) \quad &lb_1(M) \leq l_1 \leq u_1 \leq ub_1(M) \wedge lb_2(M) \leq l_2 \leq u_2 \leq ub_2(M) !! \\ &0 : (u_1 - l_1) \text{div}^\downarrow n_1, 0 : (u_2 - l_2) \text{div}^\downarrow n_2 :: \\ &i_1, i_2 : \iota(M[l_1 + n_1 * i_1, l_2 + n_2 * i_2]) \\ &\quad \leftarrow [0 \leq i_1 \leq (u_1 - l_1) \text{div}^\downarrow n_1] \wedge [0 \leq i_2 \leq (u_2 - l_2) \text{div}^\downarrow n_2] \\ (Q7) \quad &lb(A) = 0 \wedge ub(A) \geq 1 !! \\ &0 : (ub(A) - 1) \text{div}^\downarrow 6 + (ub(A) - 3) \text{div}^\downarrow 6 + 1 :: \\ &i * 2 : \iota(A[j]) \leftarrow [0 \leq j \leq ub(A)] \wedge j = i * 6 + 1 \\ &i * 2 + 1 : \iota(A[j]) \leftarrow [0 \leq j \leq ub(A)] \wedge j = i * 6 + 3 \end{aligned}$$

Figure 3: Some sample restructuring queries.

The last query in Fig.3 shows we can express the AML[19] subsample operator. In particular, (Q7) gives subsample(010100, A), which picks out all positions i such that $i \bmod 6 = 1$ or $i \bmod 6 = 3$ from the zero-based array A . In general, we need one rule (and one output upper bound $\text{div}^\downarrow$ term) for each “1” in the bit pattern.

Restructuring queries, such as those in Fig.3, can be combined with each other and with aggregations. We already saw this in our first query, which takes the surface-level slice of T , zips it with P , dices the result to focus on daylight values and, finally, aggregates along the time dimension using (Σ, Max) . Using disjunctions, different arrays or different parts of an input or output array can be processed independently. In the full version of this paper[17], we give an extended example that illustrates how this is done and also highlights the use of our framework to facilitate interoperability.

Safety. All index-based array queries must meet three safety requirements: *finiteness*, *full-guardedness*, and *nonemptiness*.

Finiteness is the requirement that our queries' variables range over only finite sets of values. As in the relational model, we can enforce it syntactically; however, instead of using relation literals to limit our variables' ranges, we use index bounds atoms. We say a variable x is *range-restricted* in a set of query comparisons C if it occurs in some atom $[l \leq t(x) \leq u]$ in C or some equality $t_1(x) = t_2$ in C , with t_2 closed, or some equality $t_1(x) = t_2(y)$, with y already range-restricted in C . Observe that, (1) the values of bounds terms (closed terms) are fixed by each instance and (2) if $t(x) = nx + b$ can only take on a finite number of values, the same is true of x . Thus, to guarantee finiteness, we simply require every variable occurring in a rule to be range-restricted in its body.

The second safety requirement is specific to our model and arises because our inputs and outputs are shaped. In our rules, accesses to both input arrays (via subscripting) and the output array (via output index terms) can cause bounds errors. Full-guardedness is the requirement that the query's guard conditions be strong enough to guarantee no bounds errors occur on any S -instances satisfying them. Formally, Q is *fully-guarded* if:

- **(FG-I)** All input indices fall within bounds: for each $R \in \text{Rules}(Q)$ and subscript term $A_i[t_1, \dots, t_{k_i}]$ in $\text{Head}(R)$, $\text{Guard}(Q) \cup \text{Body}(R) \models_S \bigwedge_{j=1}^{k_i} [lb_j(A_i) \leq t_j \leq ub_j(A_i)]$.
- **(FG-O)** All output indices fall within bounds: for each $R \in \text{Rules}(Q)$, $\text{Guard}(Q) \cup \text{Body}(R) \models_S \bigwedge_{j=1}^k [l_j \leq ti_j \leq u_j]$, where $\text{Shape}(Q) = (l_1 : u_1, \dots, l_k : u_k)$ and $ti_1, \dots, ti_k$ are the output index terms in $\text{Head}(R)$.

In other words, a query is fully guarded if no bounds errors can ever occur in a rule head on any S -instance for which the query is defined and that rule fires. (Here and in the rest of our discussion on safety, we assume the definition of CPC inference given earlier has been extended to account for variables. We formalize this below.)

A fully guarded query Q is defined on an S -instance I iff $I \models \text{Guard}(Q)$. When $I \not\models \text{Guard}(Q)$, the query is considered undefined on I even if the bare query without the guard would have been error-free on that instance. A fully guarded query Q is defined on all S -instances iff $\models_S \text{Guard}(Q)$; however, for safety we only require that it be fully guarded. Our use of guards to give query shape preconditions thus decouples definedness from errors, giving control over a query's domain of definition to the query writer. By making full-guardedness a safety issue, our approach also conceptually separates error detection (bounds analysis) from data access.

The third safety requirement is a form of closure. We defined arrays to always have nonempty shapes and used the constraint $\text{NonEmpty}(S)$ to make sure input arrays met this requirement. We also must guarantee query output arrays have nonempty shapes:

- **(NE)** $\text{Guard}(Q) \models_S \bigwedge_{j=1}^k l_j \leq u_j$, where $\text{Shape}(Q) = (l_1 : u_1, \dots, l_k : u_k)$.

While finiteness is guaranteed syntactically via range restriction, the definitions of full-guardedness and nonemptiness have been formulated in semantic terms. The question arises whether these safety requirements can also be easily checked.

PROPOSITION 3.1. *Let C, C' be CVCs over only non-arithmetic terms such that all the variables in C' already occur in C , we can determine if $C \models C'$ in time polynomial in $|C| + |C'|$. In particular, non-arithmetic CPC inference is polynomial.*

Since rule bodies are CVCs and guards and parameter constraints are CPC's, and since by range-restriction all variables occurring in a rule head must occur in its body, safety can be verified in polynomial time for non-arithmetic queries.

State safety. When we allow arithmetic, we lose polynomial inference. However, since inference remains decidable, we have two choices. For queries used repeatedly (e.g., queries embedded in systems), we can invest the time to verify safety ahead of time. But for queries written on the fly, we can verify safety on an instance-by-instance basis. In particular, we can (1) access an instance's metadata, (2) replace the safety conditions with versions of these conditions in which all closed terms have been computed down to integers, and (3) verify these new parameter-less safety conditions. This is related to what is called state safety in [5], but in our case only metadata (i.e., parameters) need to be accessed. This has two consequences. First, as with safety, state safety lets us avoid late bounds errors (once data access has begun). Second, its complexity depends on the sizes of the query and metadata but not on the size of the data. As we will show in Sect.5, our version of state safety remains tractable in the presence of arithmetic.

Non-CPC Inference. Before giving our query semantics, we extend the inference relation ($\models$) beyond CPC's to take into account variables and value terms. A valuation v for a set of comparisons C

is an assignment of the variables in C to integers. We write $\text{Vals}(C)$ for the set of all valuations for C . For any term t , $v(t^I)$ denotes the value given t by instance I using v to interpret any variables, where both $(_)^I$ and v extend outward homomorphically. For example, $v((A[x, lb_2(A)], B[3])^I) = (A^I[v(x), lb_2(A^I)], B^I[3])$.

Satisfaction of a set (or conjunction) of comparisons by an instance with respect to a valuation, denoted $[I, v] \models C$ and inference between such sets, denoted $C_1 \models C_2$, are defined as expected. (Though we mostly deal with the conjunctive, universal case, in which all variables of C_2 already appear in C_1 , for generality, we also extend $\models$ to FO formulas in the usual way.) As before, $C_1 \models_S C_2$ stands for $C_1 \cup \Gamma_S \models C_2$. Also, for rule R of query Q with schema S , we abbreviate $\text{Body}(R) \cup \text{Guard}(Q) \cup \Gamma_S$ as $\text{Body}(R)^+$. We say R fires under instance I if $I \models \text{Body}(R)^+$.

Query Semantics. Our semantics differs from standard relational semantics in two ways. First, because of guards, array queries are only *partial* functions from input to output schema instances. Second, because our query outputs are shaped arrays, every index in the output shape must be associated with exactly one value. The collection aggregators are used both to coalesce multiple values for an index into a single value and also to guarantee each index within the output shape is assigned *some* (possibly null) value.

The semantics of α -queries is defined in terms of the semantics of $\bigcup$ - or $\biguplus$ -queries, so we begin with the latter. Let Q be a *safe* $\bigcup$ - or $\biguplus$ -query over input schema S with $\text{Shape}(Q) = (l_1 : u_1, \dots, l_k : u_k)$ and $\text{Rules}(Q) = \{R_1, \dots, R_n\}$ where each rule $R_j = ti_1^j, \dots, ti_k^j : \alpha(tv_1^j, \dots, tv_m^j) \leftarrow \text{Body}(R_j)$. The query Q is defined on an instance I iff I is an S -instance satisfying the guard conditions. When Q is defined on I , the *result of Q on I* is an array Q^I defined as follows. Its shape is given by setting $\text{Shape}(Q^I) = (\text{Shape}(Q))^I = (l_1^I : u_1^I, \dots, l_k^I : u_k^I)$. (Because Q is safe, this a valid shape for a nonempty array.) Then if Q is a $\bigcup$ -query, its value at any index $d = (d_1, \dots, d_k)$ in $\text{Shape}(Q^I)$ is given by setting

$$\begin{aligned} Q^I[d] &= \{v_1((tv_1^1, \dots, tv_m^1)^I) \mid v_1 \in \text{Vals}(\text{Body}(R_1)), \\ &\quad [I, v_1] \models \text{Body}(R_1) \text{ and } v_1((ti_1^1, \dots, ti_k^1)^I) = d\} \\ &\cup \dots \cup \{v_n((tv_1^n, \dots, tv_m^n)^I) \mid v_n \in \text{Vals}(\text{Body}(R_n)), \\ &\quad [I, v_n] \models \text{Body}(R_n) \text{ and } v_n((ti_1^n, \dots, ti_k^n)^I) = d\}. \end{aligned}$$

When Q is a $\biguplus$ -query, its values are defined similarly using bags and bag unions. Note that for bags, multiple copies of the same value can arise from different rules or from the same rule if either the body contains variables not in the head or else some tv_i (with a noninjective array) contains variables not occurring among the ti_j .

The semantics of α -queries for α other than $\bigcup$ or $\biguplus$ is defined in terms of the semantics of the $\bigcup$ - or $\biguplus$ -queries giving their intermediary collection array results. For any α -query Q , let $\text{core}(Q)$ be the query obtained from Q by replacing α with $\bigcup$, if α is a set aggregator, or $\biguplus$, if it is a bag aggregator. (For $\bigcup$ - or $\biguplus$ -queries, $\text{core}(Q) = Q$.) Then for any α -query Q with $\alpha \neq \bigcup, \biguplus$ and any S -instance I satisfying $\text{Guard}(Q)$, the result of Q on I is defined by setting $\text{Shape}(Q^I) = (\text{Shape}(Q))^I$, as before, and setting $Q^I[d] = \alpha(\text{core}(Q)^I[d])$ for every d in $\text{Shape}(Q^I)$. By identity (A4), we get $Q^I[d] = \text{null}_\alpha$ when no values are generated for d .

4. ARRAY QUERY EQUIVALENCE

Two arrays A_1 and A_2 are equal iff $\text{Shape}(A_1) = \text{Shape}(A_2)$ and for every index d in this shape $A_1[d] = A_2[d]$. In defining equivalence of array queries, we must take into account their partiality. Let Q_1 and Q_2 be safe queries over input schema S and the same output schema. Then Q_1 is equivalent to Q_2 ($Q_1 \equiv Q_2$) iff for all S -instances I , Q_1 is defined on I iff Q_2 is, and when-

ever either is defined, $(Q_1)^I = (Q_2)^I$, that is, $\text{Shape}((Q_1)^I) = \text{Shape}((Q_2)^I)$ and $(Q_1)^I[d] = (Q_2)^I[d]$, for all $d \in \text{Shape}((Q_1)^I)$.

For $\bigcup$ -queries, we can completely characterize equivalence in terms of the inference relation:

PROPOSITION 4.1. *Let Q_1, Q_2 be two $\bigcup$ -queries over schema S with $\text{Rules}(Q_i) = \{R_1^i, \dots, R_{n_i}^i\}$. Wlog, let their variables be disjoint, with $\text{Vars}(R_p^1) = \{x_1^p, \dots, x_{m_p}^p\}$ and $\text{Vars}(R_q^2) = \{y_1^q, \dots, y_{n_q}^q\}$. Then $Q_1 \equiv Q_2$ iff (1) $\text{Guard}(Q_1) \models_S \text{Guard}(Q_2)$ and vice versa, (2) $\text{Guard}(Q_1) \models_S \text{Shape}(Q_1) = \text{Shape}(Q_2)$, (3a) for each p , $\text{Body}(R_p^1)^+ \models \bigvee_{q=1}^{n_2} \exists y_1^q \dots \exists y_{n_q}^q \text{Body}(R_q^2) \wedge \text{Head}(R_p^1) = \text{Head}(R_q^2)$, and (3b) for each q , $\text{Body}(R_q^2)^+ \models \bigvee_{p=1}^{n_1} \exists x_1^p \dots \exists x_{m_p}^p \text{Body}(R_p^1) \wedge \text{Head}(R_q^2) = \text{Head}(R_p^1)$.*

Since (1) and (2) involve only CPC inference, they can be verified in polynomial time for nonarithmetic queries. Prop.3.1 doesn't apply to (3a) and (3b) because of the disjunctions and existentials, as well as the presence of value terms involving array subscripting. We can reformulate these conditions in terms of universal inferences by using variations on containment maps appropriate to our model. We do this for nonarithmetic queries at the end of this section and arithmetic ones in Sect.5. First, however, we show how to reduce equivalence of arbitrary α -queries to equivalence of $\bigcup$ - and $\biguplus$ -queries whose value terms are actually index terms.

4.1 Reducing Queries to their Index Cores

We proceed in three steps. First, equivalence of queries with product aggregators reduces to equivalence of unary queries. So we need only consider queries with α unary or $\alpha \in \{\bigcup, \biguplus\}$. Second, for any such queries, equivalence reduces to equivalence of queries whose rule heads all reference the same components of the same arrays. We call such queries *monolithic*. Third, for monolithic unary, $\bigcup$ -, or $\biguplus$ -queries, we can ignore both the array subscripting/projection structure and the particular aggregator used. This allows us to characterize equivalence for all queries in terms of *index cores*, $\bigcup$ - and $\biguplus$ -queries that use index terms in place of value terms. We expand on each of these results next.

Reducing Product Aggregators. Let $Q : k \rightarrow m$ be an m -ary α -query, with $\alpha = (\alpha_1, \dots, \alpha_m)$. For each $1 \leq j \leq m$, let $\pi_j(Q) : k \rightarrow 1$ be the unary α_j -query obtained from Q by keeping Q 's guard and output shape, but replacing each rule $R := t_{i_1}, \dots, t_{i_k} : \alpha_1(tv_1), \dots, \alpha_m(tv_m) \leftarrow \text{Body}(R)$ in $\text{Rules}(Q)$ with the rule $\pi_j(R) := t_{i_1}, \dots, t_{i_k} : \alpha_j(tv_j) \leftarrow \text{Body}(R)$. (Only the rule heads change; the bodies are kept as before.) The following is a consequence of identity (A1) given in Sect.2.

PROPOSITION 4.2. *For α -queries $Q_1, Q_2 : k \rightarrow m$, with $\alpha = (\alpha_1, \dots, \alpha_m)$, $Q_1 \equiv Q_2$ iff $\bigwedge_{1 \leq j \leq m} \pi_j(Q_1) \equiv \pi_j(Q_2)$.*

Monolithic Queries. We say two rules R_1, R_2 over the same schema S , with $\text{Head}(R_i) = (t_{i_1}^i, \dots, t_{i_k}^i) : \alpha(tv_1^i, \dots, tv_m^i)$ are *compatible* if for each j , tv_j^1 and tv_j^2 involve the same projection π_l of the same array A_i . More formally, we define the compatibility signature $\sigma(R)$ for a rule R with $\text{Head}(R) = t_{i_1}, \dots, t_{i_k} : \alpha(tv_1, \dots, tv_m)$ to be the m -tuple of pairs whose j th component is (i, l) if $tv_j = \pi_l(A_i[t_1, \dots, t_{k_i}])$ for some index terms $t_1, \dots, t_{k_i}$. Then two rules R_1 and R_2 are compatible if $\sigma(R_1) = \sigma(R_2)$. If the rules of a query Q are all compatible, we say the query is *monolithic* and define $\sigma(Q)$ to be the common compatibility signature of its rules. We say two α -queries Q_1 and Q_2 are compatible if they are both monolithic and $\sigma(Q_1) = \sigma(Q_2)$.

Index Cores. Let Q be a monolithic α -query Q . We define *indexcore*(Q) to be the query obtained from Q by replacing α with $\bigcup$ or $\biguplus$ (depending on whether α is a set or bag aggregator) and replacing each subscript term with its list of index terms;

that is, we replace each value term of the form $\pi_l(A_i[t_1, \dots, t_{k_i}])$ with the k_i value terms $t_1, \dots, t_{k_i}$. If Q is an m -ary query with $\sigma(Q) = ((i_1, l_1), \dots, (i_m, l_m))$, then *indexcore*(Q) is an m' -ary query with $m' = \sum_{j=1}^m k_{i_j}$ (where $A_{i_j} : k_{i_j} \rightarrow m_{i_j}$). For example, if Q has the rules $3x : \Sigma(A[4x + 2, 5y]), \text{Max}(B[x]) \leftarrow B_1$ and $7y : \Sigma(A[8, y]), \text{Max}(B[4y]) \leftarrow B_2$, then *indexcore*(Q) has the rules $3x : \biguplus(4x + 2, 5y, x) \leftarrow B_1$ and $7y : \biguplus(8, y, 4y) \leftarrow B_2$.

As index terms can now occur in both index and value positions, to avoid confusion we henceforth refer to index and bounds terms, wherever they occur, as *simple* and to value terms involving arrays as *nonsimple*. Index cores can be used to reduce monolithic queries with nonsimple terms to ones with only simple terms.

LEMMA 4.3. *Let Q_1, Q_2 be monolithic α -queries with compatible heads, where $\alpha \in \{\bigcup, \biguplus\}$ or α is unary. Then $Q_1 \equiv Q_2$ iff *indexcore*(Q_1) $\equiv$ *indexcore*(Q_2).*

PROOF SKETCH. From any instance I distinguishing the index cores we construct an instance distinguishing the full queries by leaving I 's action on parameters as is and changing its action on array values. For $\bigcup/\biguplus$ -queries, differences are maintained by using array values that are 1-1 functions of their indices. For other α -queries, differences are maintained by using a non-null value at some index for which the index cores differ and using null_α at all others. Three facts make this work. (1) Because array subscript terms don't appear in rule bodies or indices, we can change the array values in an instance without affecting which rules will fire or the valuations of indices. (2) By (A3), the aggregations depend on the values assigned to the arrays. (3) In the case of bag-aggregators, (A5) propagates differences in multiplicities. $\square$

The restriction to unary queries is essential, as this lemma fails for m -ary product queries with $m \geq 2$. What remains to be shown is that equivalence between arbitrary disjunctive queries reduces to equivalence between monolithic ones.

Query Restrictions. For m -ary query Q and compatibility signature s , the *restriction of Q to s* is a query $Q|_s$ defined by keeping $\text{Guard}(Q|_s) = \text{Guard}(Q)$ and $\text{Shape}(Q|_s) = \text{Shape}(Q)$, but adding to $\text{Rules}(Q|_s)$ only those $R \in \text{Rules}(Q)$ with $\sigma(R) = s$. While Prop.4.2 says we can slice product aggregator queries vertically into their unary projections, the following says we can slice disjunctive queries horizontally into their monolithic restrictions.

LEMMA 4.4. *Let Q_1, Q_2 be two α -queries with $\alpha \in \{\bigcup, \biguplus\}$ or α unary and let $\Sigma = \{\sigma(R) \mid R \in \text{Rules}(Q_1) \cup \text{Rules}(Q_2)\}$. Then $Q_1 \equiv Q_2$ iff for all $s \in \Sigma$, $Q_1|_s \equiv Q_2|_s$.*

PROOF SKETCH. The if direction uses (A2). The other direction follows the previous lemma, but we need to make sure differences due to (index cores of) rules in one restriction aren't made up for by rules in other restrictions. In the $\bigcup/\biguplus$ case, we use array values that are 1-1 functions of indices and compatibility signatures. $\square$

COROLLARY 4.5. *Let Q_1, Q_2 be α -queries, where α is any set or bag-aggregator. Then for some n (polynomial in the queries' size), there are $\bigcup$ and/or $\biguplus$ -queries $Q_1^1, \dots, Q_1^n$ and $Q_2^1, \dots, Q_2^n$ containing only simple terms s.t. $Q_1 \equiv Q_2$ iff $\bigwedge_{1 \leq i \leq n} Q_1^i \equiv Q_2^i$.*

This reduction holds for conjunctive or disjunctive queries with or without arithmetic and using set or bag aggregators.

4.2 Nonarithmetic Set-Aggregator Queries

We now limit our focus to nonarithmetic set-aggregator queries. By the last result, equivalence for these reduces to equivalence of $\bigcup$ -queries containing only simple terms. Prop.4.1 then applies to give a complete characterization of equivalence via inferences. We

can reformulate these inferences without the embedded existentials by using containment maps. For conjunctive queries they become CVC inferences, which we know to be tractable.

We consider the conjunctive case first. As defined, conjunctive queries can have no rules. But any query with no rules is equivalent to one with an unsatisfiable rule. So we henceforth assume all queries have nonempty rule sets. For a conjunctive query, Q , $Head(Q)$ and $Body(Q)$ denote its single rule's head and body.

Containment Maps. Let Q_1, Q_2 be two conjunctive, nonarithmetic queries over the same I/O schemas, with $Head(Q_i) = ti^i : \alpha(tv^i)$, for some $ti^i = (ti_1^i, \dots, ti_k^i)$, $tv^i = (tv_1^i, \dots, tv_m^i)$, and set-aggregator α . An array query containment map $h: Q_2 \rightarrow Q_1$ is a substitution of the variables of Q_2 with simple terms from $Vars(Q_1) \cup Consts(Q_1, Q_2) \cup Params(S)$ such that:

$$(C) \quad Body(Q_1)^+ \models h(ti^2) = ti^1 \wedge h(tv^2) = tv^1 \wedge h(Body(Q_2))$$

Containment maps provide sufficient conditions for equivalence of conjunctive set-aggregator queries (with simple or nonsimple output value terms).

THEOREM 4.6. *For Q_1, Q_2 as above, if conditions (1) and (2) of Prop.4.1 hold (i.e., their guards and shapes are equivalent) and (3) there are containment maps $h_{2,1} : Q_2 \rightarrow Q_1$ and $h_{1,2} : Q_1 \rightarrow Q_2$, then $Q_1 \equiv Q_2$.*

PROOF. Fix an $I \models \Gamma_S \cup Guard(Q_1)$ and a $d \in Shape(Q_1)^I$. By (1), (2), and the fact that $(Q_i)^I[d] = \alpha(core(Q_i)^I[d])$ we need only show that $core(Q_1)^I[d] \subseteq core(Q_2)^I[d]$ and vice versa. Suppose $t \in core(Q_1)^I[d]$. Then there is a valuation v on $Vars(Q_1)$ with $d = v((ti^1)^I)$, $t = v((tv^1)^I)$ and $[I, v] \models Body(Q_1)$. Let $h_{2,1}$ be the map specified in the hypothesis. Since $[I, v] \models Body(Q_1)^+$, using the definition of containment maps (with $h_{2,1}$ for h) we can conclude $v \circ h_{2,1}((ti^2)^I) = v((ti^1)^I) = d$, $v \circ h_{2,1}(tv^2)^I = v((tv^1)^I) = t$, and $[I, v \circ h_{2,1}] \models Body(Q_2)$. So $v \circ h_{2,1}$ is a valuation getting t into $core(Q_2)^I[d]$. $\square$

The converse of this theorem, i.e., a characterization of containment maps as *necessary* conditions for equivalence, holds for $\bigcup$ -queries in a special connected normal form.

Reduced, Connected Normal Forms. We say a rule is *reduced* if its body contains no equations involving variables. A reduced rule is *connected* if all variables appearing in its body also appear in its head. A query is reduced (connected) if all its rules are.

PROPOSITION 4.7. *Every set-aggregator array query with nonarithmetic terms can be converted to an equivalent reduced, connected query in polynomial time.*

This result holds for disjunctive as well as conjunctive set-aggregator queries with or without arithmetic. We prove it separately for arithmetic queries in Sect.5. Connectedness is a useful property to establish when studying equivalence, since for connected queries, the candidate containment maps are fixed by the structural correspondence between rule heads. We use this fact in the proof of the next lemma giving the converse of Thm.4.6.

LEMMA 4.8. *Let Q_1, Q_2 be connected, conjunctive $\bigcup$ -queries with simple, nonarithmetic terms over the same input and output schemas. Then whenever $Q_1 \equiv Q_2$ there are containment maps from Q_2 to Q_1 and Q_1 to Q_2 .*

PROOF SKETCH. We define the containment map $h : Q_2 \rightarrow Q_1$ by setting $h(x)$ to the term in $Head(Q_1)$ corresponding to the leftmost occurrence of x in $Head(Q_2)$. The key idea in proving h is a containment map is that since $Q_1 \equiv Q_2$, for any instance I and valuation v_1 satisfying $Body(Q_1)^+$ there is a valuation v_2 agreeing

with v_1 on the head terms (1) and for which $[I, v_2] \models Body(Q_2)$ (2). From the agreement on head terms, it follows that for any simple term t_2 occurring in Q_2 , $v_2(t_2^I) = v_1(h(t_2)^I)$ (3). Using facts (1)-(3), it is then easy to establish (C). $\square$

This gives us our main result – a characterization of equivalence for connected, conjunctive, nonarithmetic $\bigcup$ -queries in terms of easily verified necessary and sufficient conditions.

THEOREM 4.9. *Let Q_1, Q_2 be connected, conjunctive, nonarithmetic $\bigcup$ -queries over the same I/O schemas. Then $Q_1 \equiv Q_2$ iff conditions (1) and (2) of Prop.4.1 hold, and (3) there are containment maps from Q_2 to Q_1 and Q_1 to Q_2 .*

COROLLARY 4.10. *The equivalence problem for conjunctive set-aggregator queries with nonarithmetic terms is in $\mathbf{P}$.*

Compare this result to the complexity of containment for relational queries with comparisons, shown to be Π_2^P -complete by van der Meyden[26]. Why do we get a low complexity? First, our queries contain no comparisons involving array values. We took advantage of this array value genericity to prove lemmas 4.3-4.4 on reducing equivalence to index cores of compatible (monolithic) queries. Second, the absence of relation/array literals from our query bodies means we can convert our queries into connected ones. We can use compatibility and connectedness to pick out the required containment maps. So our complexity depends only on the time needed to verify these are indeed containment maps and not on the time needed to search through a large space of candidate maps, as must be done for relational queries [14, 20].

For array queries, we get a jump in complexity when we allow either disjunctions or arithmetic terms.

THEOREM 4.11. *The equivalence problem for disjunctive set-aggregator queries with nonarithmetic terms is $\mathbf{coNP}$ -complete.*

Hardness follows by an easy reduction from 3DNF-TAUT. Membership in $\mathbf{coNP}$ is just a special case of the same result for arithmetic queries given in the following section.

5. ADDING ARITHMETIC

We now explore safety and set-aggregator query equivalence for queries with arithmetic terms. To get a handle on complexity in the arithmetic case, we relate the inference problem for the kinds of inferences needed to establish safety and equivalence to the decision problem for the relevant syntactic fragments of Presburger arithmetic (PA). To this end, we first prove we can eliminate all occurrences of $\text{div}^\perp$, $\text{div}^\top$, min , and max from our inferences without adding any quantifier alternations or blowing up the problem size.

We focus on inferences of the form $\phi_1 \models (\exists \bar{y})\phi_2$, where ϕ_1 and ϕ_2 are quantifier-free formulas and $\bar{y}$ contains all variables in ϕ_2 not occurring in ϕ_1 . We say an inference is universal or has one quantifier alternation (more accurately, one quantifier block) if $\bar{y}$ is empty; we say it has two alternations otherwise. For example, all inferences needed to verify safety have one quantifier alternation as do all CPC inferences. The inferences used to establish equivalence have one if we can use containment maps and two otherwise.

In the following lemma, we assume ϕ_1 and ϕ_2 are boolean combinations of comparisons whose terms are built up from variables, parameters and constants using $\mathcal{F}_b$ functions. that is, we treat variables and parameters uniformly. This means the equations and inequations can have any number of variables.

LEMMA 5.1. *ϕ_1, ϕ_2 as above can be converted to quantifier-free formulas ϕ'_1 and ϕ'_2 containing no $\text{div}^\perp$, $\text{div}^\top$, min , or max*

s.t. $\phi_1 \models (\exists \bar{y})\phi_2$ iff $\phi'_1 \models (\exists \bar{z})\phi'_2$, where $\bar{z}$ contains all variables in ϕ'_2 not already in ϕ'_1 . Moreover, this can be done with at most polynomial increases in the sizes of the formulas and without affecting quantifier alternations (i.e., s.t. $\bar{y}$ is nonempty iff $\bar{z}$ is).

Remark. Though we can eliminate the extra functions from inferences, we cannot eliminate them from queries. In fact, including them in $\mathcal{F}_b$ increases the expressive power of our query language because they can't be translated out of output shape terms.

Inference Complexity. PA is traditionally formulated using order, addition and the naturals, but it can also be given using order, addition, multiplication by constants and the integers. By Lem.5.1, our inferences correspond to sentences in this latter formulation. In particular, $\phi'_1 \models (\exists \bar{z})\phi'_2$ iff the PA sentence $(\forall \bar{p})(\forall \bar{w})(\exists \bar{z})(\phi'_1 \Rightarrow \phi'_2)$ is valid, where $\bar{w} = \text{Vars}(\phi'_1)$ and $\bar{p}$ contains all the schema parameters (viewed as extra variables). While the full theory has a doubly exponential lower bound complexity, the decision problem for PA sentences with two quantifier blocks has “only” an exponential lower bound [11].

We can do better. Using containment maps, we can eliminate all embedded existentials from the inferences characterizing equivalence. We detail how this is done in the presence of arithmetic below. For now, we simply state that, though the maps introduce new occurrences of $\text{div}^\perp$ into our inferences, by Lem.5.1, these can be eliminated without reintroducing the embedded existentials. Thus, all the inferences we need correspond to universal PA sentences, whose complexity is **coNP**-complete [23, 27]. The lower bound (hardness) already holds for conjunctive PA sentences, so, unlike the nonarithmetic case, we can't attain lower complexity results for conjunctive arithmetic queries.

Polynomial Variable Elimination. A crucial step towards efficient equivalence in the nonarithmetic case was our polynomial algorithm for converting queries to equivalent reduced, connected ones. This enabled us to pick out the specific containment map to test, rather than have to search a large space of candidate maps.

LEMMA 5.2. *All arithmetic set-aggregator queries can be converted to equivalent reduced, connected ones in polynomial time.*

PROOF SKETCH. Our algorithm for eliminating equalities involving variables is based on the one given in [21], modified to take into account parameters and the fact that our index terms contain at most one variable. We give the algorithm and prove its correctness and polynomial complexity in the full version of this paper [17].

Once we have a way to convert rules to equivalent reduced ones, it is easy to get connectedness. After eliminating all equations involving variables, we can eliminate any remaining unconnected variable x as follows: First, we solve for x in all inequations involving x ; for example, $l \leq n * x + b \leq u$ becomes $(l - b) \text{div}^\perp n \leq x \leq (u - b) \text{div}^\perp n$. Next, we gather all the solved inequations into one inequation using max and min: $l_1 \leq x \leq u_1 \wedge \dots \wedge l_n \leq x \leq u_n$ becomes $\max(l_1, \dots, l_n) \leq x \leq \min(u_1, \dots, u_n)$. Finally, we eliminate x using $(\exists x)l \leq x \leq u$ iff $l \leq u$. $\square$

We don't get tractable inference from efficient variable elimination because we don't have efficient parameter elimination. But when there are no parameters, as in the inferences needed for state safety, inference is tractable.

PROPOSITION 5.3. *For arithmetic queries, though full safety is in **coNP**, the state safety conditions for any particular instance can be verified in time polynomial in the metadata and query sizes.*

Arithmetic Query Equivalence. We now put everything together to characterize set-aggregator query equivalence in the arithmetic case. By Cor.4.5 and Lem.5.2, we need only consider (possibly disjunctive) connected $\bigcup$ -queries with simple terms. Once

again, Prop.4.1 applies to give us a characterization in terms of the inference relation. To avoid the exponential jump in complexity, we use containment maps to eliminate the embedded existentials. Since universal PA inference is already **coNP**-complete in the conjunctive case and remains so when we add disjunctions, we treat the conjunctive and disjunctive cases uniformly.

Containment Maps (Arithmetic Case). Let Q_1 and Q_2 be two connected $\bigcup$ -queries with simple arithmetic terms (i.e., connected index-cores) over the same input and output schemas. For $i = 1, 2$, let $\text{Rules}(Q_i) = \{R_{i,1}, \dots, R_{i,n_i}\}$. An *arithmetic containment map* from Q_2 to Q_1 is a family of maps $\{h_{q,p}^{2,1}\}_{q \in [1..n_2], p \in [1..n_1]}$, where each $h_{q,p}^{2,1}$ is a substitution of the variables of rule $R_{2,q}$ with terms built up from $\text{Vars}(R_{1,p}) \cup \text{Consts}(R_{2,q}, R_{1,p}) \cup \text{Params}(S)$ using functions in $\mathcal{F}_b^3$ such that for each $p \in [1..n_1]$, $\text{Body}(R_{1,p})^+ \models \bigvee_{q=1}^{n_2} \text{Head}(R_{1,p}) = h_{q,p}^{2,1}(\text{Head}(R_{2,q})) \wedge h_{q,p}^{2,1}(\text{Body}(R_{2,q}))$.

THEOREM 5.4. *Let Q_1, Q_2 be two connected $\bigcup$ -queries with simple arithmetic terms. Then $Q_1 \equiv Q_2$ iff conditions (1) and (2) of Prop.4.1 hold and (3) there are arithmetic containment maps from Q_2 to Q_1 and from Q_1 to Q_2 .*

PROOF SKETCH. We assume all open terms in both queries are in the form $n * x + b$ for some integer $n \neq 0$ and bounds term b . We use the structural correspondence between rule heads to define candidate containment maps. For each pair of rules $R_{1,p} \in \text{Rules}(Q_1)$ and $R_{2,q} \in \text{Rules}(Q_2)$, we define $h_{p,q}^{1,2}$ as follows: for each $x \in \text{Vars}(R_{1,p})$, if $t_1(x) = n * x + b$ is the leftmost term containing x in $\text{Head}(R_{1,p})$ and t_2 is the term in the corresponding position of $\text{Head}(R_{2,q})$, we set $h_{p,q}^{1,2}(x) = (t_2 - b) \text{div}^\perp n$. We define similar maps $h_{q,p}^{2,1}$ in the other direction. Verifying the queries are equivalent iff these are indeed containment maps is not hard. $\square$

COROLLARY 5.5. *The equivalence problem for set-aggregator queries with arithmetic is **coNP**-complete.*

6. RULE MINIMIZATION

Though, semantically, an index bounds atom $[l \leq t(x) \leq u]$ is just a conjunction of inequations, operationally, it can be thought of as a for-loop. For set-aggregator queries, we can easily minimize the number and sizes of “loops” in any rule. First, we convert the rule to a reduced, connected one. Next, we solve for loop variables and gather the loops as described in Lem.5.2. The resulting rule has one loop of the form $[\max(l_1, \dots, l_p) \leq x \leq \min(u_1, \dots, u_q)]$ per variable and no further selections on the variables.

Any satisfiable rule of this form is minimal in the following sense. If we treat it as a conjunctive query by ignoring all other rules, then, in any instance, the product of its loop sizes is minimal among all equivalent conjunctive queries. (Operationally, unsatisfiable rules are also minimal, since their loops are never executed.)

In minimizing rules, we leave metadata expressions as is. Simplifying these would raise minimization complexity from **P** to **coNP**. This doesn't pay, since the time it takes to access the metadata and compute (even overly complex) metadata expressions is usually negligible compared to the time it takes to access the data.

The fact that rule minimization is trivial in our context doesn't mean query optimization is trivial. The jump in complexity from the conjunctive to disjunctive nonarithmetic case suggests there are non-trivial choices between equivalent disjunctive queries. Rule minimization says nothing about how to optimize a disjunction of rules. Is it best to minimize the number of rules? Or is it best

³Applying $f_b \in \mathcal{F}_b$ to variables is ok because we use these terms only in inferences, not queries. By lemma 5.1, we can remove integer divisions, etc., from the inferences without affecting their complexity.

to minimize the average loop sizes, even if this means adding more rules? The answer depends on cost-based modeling of the available access paths and storage layouts. For sequential access to tertiary storage, the former might be better. For parallel access, the latter might be, with different rules executed in parallel. For striped layouts, we'd want separate rules for the separate strides.

7. EXPRESSIVITY

We believe the way we defined index and bounds terms strikes the right balance between complexity and expressivity. By limiting index terms to one variable, we got polynomial variable elimination. This led to tractable rule minimization and state safety. It also kept equivalence within **coNP**. Because our variable elimination procedure didn't require efficient parameter elimination, we could include more general functions in our bounds terms without sacrificing tractability. We now show this is all we need to capture a large class of interesting queries, assuming we add value functions.

We omitted functions from value terms in order to get a complete characterization of equivalence. However, including them doesn't interfere with our characterization's soundness. We can extend compatibility to include function application as well as array subscripting and define index cores to ignore functions as well as arrays. Then the "if" direction of Cor.4.5 still holds, giving us sufficient conditions for equivalence. So, for studying expressivity, we assume our queries can use arbitrary functions in their value terms.

Without arithmetic or value functions, conjunctive index-based queries can express the following kinds of restructuring operators: zips (index joins), unzips (value component projections), cross-products, slices/diagonals (selections of lower-dimensional arrays by fixing/equating indices), dices (shape restrictions), dimension permutations (transpositions, rotations), stacks (embeddings into higher dimensions via repetition), paddings (embeddings into larger shapes). Furthermore, they can combine these restructuring operations with aggregations. Using disjunctions different arrays or parts of an array can be processed independently.

When we add arithmetic, we can express shifts, appends, rectangular regriddings, subsamplings, merges, and many more. This means we can express most kinds of data "subsetting" done by dataset interfaces and many queries implicit in array visualization tools. With output value functions, we can express scalings, interpolations, many filtering operations (as in image processing), as well as aggregations over a masked area. Using boolean valued functions, we can even express selections involving array values.

What we can't express are histogram-style queries or index permutations given by an array P , as these would require nonsimple input or output *index* terms or equalities between index and value terms. We also can't express queries involving multivariable or non-linear index terms, such as ones used to flatten an array into row-major order or reshape it back.

Assuming we include value functions for all their base type operators, nonarithmetic index-based queries can express all queries in Baumann's original algebra[2]. With arithmetic, we can express all AML queries, though our encoding of *apply* fails to capture their chunk reordering optimizations.

8. CONCLUSION

We introduced a new query paradigm for array data giving shape-related metadata a prominent role in both the data model and the query language. We defined the class of index-based array queries and showed that, while expressive enough to subsume previously proposed array algebras, this class also admits efficient bounds analysis and rule minimization.

We studied the query equivalence problem for index-based array queries with collection aggregators. For these, equivalence reduces to equivalence between index cores. For queries using set aggregators, equivalence can be characterized completely in terms of containment maps. We studied both conjunctive and disjunctive queries, both with and without limited Presburger arithmetic, and showed the equivalence problem is tractable in the nonarithmetic, conjunctive case and **coNP**-complete in all others.

One obvious future research direction is to characterize equivalence for more aggregators. Which methods developed in the relational context carry over to our model? Another topic is the addition of value functions. What classes of value functions admit complete characterizations of equivalence? How do value functions interact with aggregations and their underlying operators?

Acknowledgements. The author wishes to thank the anonymous reviewers for helpful comments and suggestions.

9. REFERENCES

- [1] B. Allcock, I. Foster, et al. High-performance remote access to climate simulation data: A challenge problem for data grid technologies. In *Parallel Computing*, 29(10):1335–1356, 2003.
- [2] P. Baumann. On the management of multidimensional discrete data. *Vldb Journal*, 4(3):401–444, 1994.
- [3] P. Baumann, A. Dehmelt et al. The multidimensional database system RasDaMan. In *SIGMOD'98*, pp.575–577.
- [4] P. Baumann. A database array algebra for spatio-temporal data and beyond. In *NGITS'99*, pp.76–93.
- [5] M. Benedikt and L. Libkin. Safe constraint queries. In *PODS'98*, pp.99–108.
- [6] L. Cabibbo and R. Torlone. Querying multidimensional databases. In *DBPL'97*.
- [7] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Record*, 26(1):65–74, Mar. 1997.
- [8] S. Cohen. Containment of aggregate queries. *SIGMOD Record*, 34(1):77–85, Mar. 2005.
- [9] S. Cohen, W. Nutt, and A. Serebrenik. Rewriting aggregate queries using views. In *PODS'99*, pp.155–166.
- [10] L. Fegaras and D. Maier. Optimizing object queries using an effective calculus. *TODS*, 25(4):457–516, 2000.
- [11] E. Grödel. Subclasses of Presburger arithmetic and the polynomial-time hierarchy. *TCS*, 56:289–301, 1988.
- [12] S. Grumbach, M. Rafanelli and L. Tininini. Querying aggregate data. In *PODS'99*, pp.174–184.
- [13] O. H. Ibarra and J. Su. On the containment and equivalence of database queries with linear constraints. In *PODS'97*, pp.32–43.
- [14] A. Klug. On conjunctive queries containing inequalities. *JACM*, 35(1):146–160, 1988.
- [15] K. Lellahi and V. Tannen. A calculus for collections and aggregates. In *LNCIS 1290: CTCS'97*, pp.261–280.
- [16] L. Libkin, R. Machlin, and L. Wong. A query language for multidimensional arrays: design, implementation, and optimization techniques. In *SIGMOD'96*, pp.228–239.
- [17] R. Machlin. Index-based multidimensional queries: safety and equivalence. (Available online at www.panix.com/~tona.)
- [18] D. Maier and B. Vance. A call to order. In *PODS'93*, pp.1–16.
- [19] A. Marathe and K. Salem. Query processing techniques for arrays. In *SIGMOD'99*, pp.323–334.
- [20] W. Nutt, Y. Sagiv, and S. Shurin. Deciding equivalences among aggregate queries. In *PODS'98*, pp.214–223.
- [21] W. Pugh. A practical algorithm for exact array dependence analysis. *CACM*, 35(8):102–114, 1992.
- [22] P. Revesz. Safe datalog queries with linear constraints. In *CP'98*, pp.355–369.
- [23] B. Scarpellini. Complexity of subcases of Presburger arithmetic. *Trans.AMS*, 284(1):203–218, 1984.
- [24] D. Toman and J. Chomicki. Datalog with integer periodicity constraints. *J.Log.Prog.* 35(3):263–290, 1998.
- [25] A. van Ballegooij, A. P. de Vries and M. Kersten. RAM: Array processing over a relational DBMS. Technical Report INS-R0301, CWI, March 2003.
- [26] R. van der Meyden. The complexity of querying indefinite data about linearly ordered domains. In *PODS'92*, pp.331–345.
- [27] J. von zur Gathen and M. Sieveking. A bound on solutions of linear integer equalities and inequalities. *Proc. AMS*, 72(1):155–158, 1978.