ArrayQL Algebra: version 3

July 16, 2012

David Maier
with contributions from
Peter Baumann, Martin Kersten, Kian-Tat Lim, and Mike Stonebraker

This document is a work in progress. Comments should be sent to arraydb-l@slac.stanford.edu.

1. Introduction
This document presents an array algebra that is meant to characterize the expressiveness of the eventual ArrayQL language.

It is not:
• A syntax proposal, either for DDL or DML,
• An implementation proposal.

The feedback should be whether it can express the operations we want to perform in ArrayQL 2012 (or whatever we call the first release).

I’ve tried to be precise about the meanings operations, which might mean I’ve included more parameters than expected. In ArrayQL, some of the parameters might be set implicitly from context or have a default.

This document is currently light on pictures and examples; if there is consensus I am heading in the right direction, I’ll add more where indicated.

2. Array Instance Model
This section gives a model of array instances. How much of the description of an array is part of the type definition, and how much varies from array to array (or across versions of the same array) is TBD. Array instances should be enough to discuss what function each operation performs.

An array instance has three parts:
• A bounding box, which gives the maximum index bounds in each dimension.
• A valid map, which says which array cells are currently allowed to have values.
• A content function that provides the value for each valid cell.

I remind everyone at this point that I’m not proposing an implementation. For example, while I’ve called out a valid map, an implementation need not explicitly store that. It might be that the map is implicit based which index combinations are explicitly stored.

With this model, an array instance A is a 3-tuple: (BoxA, ValidA, ContentA).

The Box is specified as a list of 0 or more named, integer dimensions, with an upper and lower bound for each. Here’s an example:
BoxA = [I:1..3, J:5..8]. (Again, don’t get excited about the syntax.) Unspecified bounds are not allowed, such as I:1..*. While we might allow such things in a type definition, any array instance will have specific bounds. The Box [] is a 0-ary array with one cell, such as an aggregate over a whole array might return.

The Box specifies a Dimension Domain (DD) that Valid and Content operate over. An element of the DD for a Box is a vector of integer indices that lie within the bounds given by the Box. For BoxA above, <I:1, J:6> and <I:2, J:5> are elements of its DD, while <I:4, J:6> and <J:6> are not. If d is an index vector, I’ll write d ∈ BoxA to mean d is in the Dimension Domain of BoxA (and in general overload BoxA as its Dimension Domain). Every DD has an origin, which is the element with the lower bound in all dimensions. For example, <I:1, J:5> is the origin of BoxA above.

The Valid map of an array is a function from the array’s DD to Boolean. For example:
ValidA: BoxA → Boolean.
ValidA(d) = false means no content is allowed in the cell indexed by d.

The Content of an array is a partial function that ranges over the Content Domain (CD) of the array, providing a value for every valid cell of the array. For example
ContentA: BoxA → (iv: Integer, rv: Real).
I’ll assume for the present that the Content Domain is a named-record type over scalar types. For the moment, I’ll allow a record type () with no fields and a single instance called unit.

{Example here, with picture BoxA as before.}

Valid_{A}(<I:1, J:5>) = true
Valid_{A}(<I:1, J:6>) = false

etc.

Content_{A}(<I:1, J:5>) = (iv: 26, rv: 9.95)
Content_{A}(<I:1, J:6>) is undefined etc.

}\ Notes: There is a subtle difference between a cell being “outside the box” versus “inside the box but not valid.” In the former case, the cell can never have a value; in the latter case, it could be assigned a value upon update (though this document takes no position on how arrays are updated). It would also be possible to eliminate Valid and make Content a partial function, but there are use cases where one would want the Valid maps of two arrays to match. Also, as currently formulated, functions used in the array operators do not need to explicitly handle the undefined case.

3. Array-Algebra Operators

This section described the operators in the model. For each operator, I indicate how it affects Box, Valid and Content. The first couple operators below are utility functions for modifying the names and origin of an array.

Rename

Rename(A, OldName, NewName)

Description: Rename changes the name of a dimension or of a cell attribute in array A. The usual strictures apply – OldName must be an existing component name; you can’t end up with two components with the same name. This operation is useful in controlling which dimensions are matched under join.

Definition: The definition here is a bit tedious. I’ll sketch it. Assume Rename(A, OldName, NewName) = R, where
\[
R = (Box_{A}, Valid_{R}, Content_{R})
\]

Case (1) OldName is a dimension name. Then Box_{R} is the same as Box_{A} with dimension NewName instead of dimension OldName. The bounds for NewName are the ones for OldName. For example, if A is as before, then

Rename(A, “I”, “K”)

would give Box_{R} = [K:1..3, J:5..8].

Valid_{R} and Content_{R} are essentially Valid_{A} and Content_{A} but accounting for the renamed dimension. Thus, if d is in Box_{R}, and d’ is the corresponding element of Box_{A} with OldName replaced by NewName, then

Valid_{R}(d) = Valid_{A}(d’)
and
Content_{R}(d) = Content_{A}(d’).

Case (2) OldName is an attribute name. In this case, Box_{R} = Box_{A} and

Valid_{R} = Valid_{A},

If Content_{A}(d) = r, then Content_{R}(d) is r with the OldName attribute renamed to NewName. For example, if A is as before, then

Rename(A, “iv”, “intv”)

would give

Content_{R}(<I:1, J:5>) = (intv: 26, rv: 9.95).

\{Show a picture of shifted A.\}

Shift

Shift(A, NewOrigin)

Description: This operator shifts the indexing of array A to start at a new origin. Upper bounds of dimensions are suitably adjusted, and Valid and Content are shifted onto the new Dimension Domain defined by the new Box. The operator is useful after Rebox (below) if one wants to have array dimensions start at the same index.

Definition: Consider

\[
Shift(A, n) = (Box_{R}, Valid_{R}, Content_{R})
\]

where n is the new origin, which is given as a vector of integers with the same dimension names as Box_{A}. Let m be the origin of Box_{A} and let p = n – m (element-wise subtraction). Then Box_{R} is obtained by adding p to both the lower and upper bounds of Box_{A}. For example, if

n = <I:0, J:0>
then
p = <I:-1, J:-5>
and
Box_{R} = [I:0..2, J:=0..3].

Valid_{R} and Content_{R} are defined from Valid_{A} and Content_{A} in the obvious way. For d \in Box_{R},\n
Valid_{R}(d) = Valid_{A}(d – p)

Content_{R}(d) = Content_{A}(d – p).

For example, if A is as in previous examples, and NewOrigin is n above, then

Valid_{R}(<I:1, J:0>) = Valid_{A}(<I:0, J:0>)
Valid_{R}(<I:1, J:0>) = Valid_{A}(<I:1, J:5>) = true.

\{Show a picture of shifted A.\}

Rebox

Rebox(A, NewBox)

Description: Rebox changes the bounds of array A to be NewBox. NewBox must have the same dimension names as Box_{A}. The position of content doesn’t change, only the bounds on dimensions. Thus Rebox can be used both to clip A to a
smaller region and to expand A to a larger region. If the Dimension Domain of NewBox is larger than the Dimension Domain of BoxA, then any new cell positions are marked invalid.

**Definition:** Consider
Rebox(A, NewBox) = R = (BoxR, ValidR, ContentR).
Then
BoxR = NewBox
For d ∈ BoxR:
ValidR(d) = ValidA(d) if d ∈ BoxA
ValidR(d) = false if d in BoxR − BoxA
ContentR(d) = ContentA(d) if ValidR(d) and d ∈ BoxR.

For example,
Rebox(A, [I=2:3, J=5:9]) removes the 1st row of A and adds a new last column, with all cells set to invalid.

{Show an example that clips I dimension of A and expands the J dimension}

**Filter**
*Filter*(A, pred:CD → Boolean) where CD is the Content Domain of A.

*Description:* Filter operates by changing the Valid map of A to be false anywhere that *pred* is false. Cells that were already invalid stay invalid. The Content map is restricted down to the valid cells.

**Definition:** Consider
Filter(A, pr) = R = (BoxR, ValidR, ContentR).
Then
BoxR = BoxA.
For d ∈ BoxR:
ValidR(d) = ValidA(d) ∧ pr(ContentA(d))
ContentR(d) = ContentA(d) if ValidR(d).

{Example here with picture for pr(c) = c.iv > 10}

**Note:** I currently have *pred* defined only over the Content Domain of A. I’m willing to consider reasons to allow it to be over the combination of the Dimension Domain and the Content Domain.

**Fill**
*Fill*(A, fillfcn:BoxA → CDa) where CDa is the Content Domain of A.

*Description:* Fill applies the *fillfcn* to the index vector of each invalid cell of A, leaving the valid cells unchanged. The result will thus be valid on all cells in BoxA.

**Definition:** Consider fillfcn = f.
Fill(A, f) = R = (BoxR, ValidR, ContentR).
Then
BoxR = BoxA.
For d ∈ BoxR:
ValidR(d) = true.
ContentR(d) = ContentA(d) if ValidA(d) else f(d).

**Apply**
*Apply*(A, fcn:CDa → CDa) where CDa is the Content Domain of A and CDa is the Content Domain of the result.

*Description:* Apply applies *fcn* to all the valid cells of A, generating the cell content of the result array.

**Definition:** Consider
Apply(A, f) = R = (BoxR, ContentR).
Then
BoxR = BoxA.
For d ∈ BoxR:
ValidR(d) = ValidA(d)
ContentR(d) = f(ContentA(d)) if ValidR(d).

{Give an example of Apply}

**Notes:** Apply can perform a Project operation with the appropriate *fcn*. Apply can be used for cell-wise multiplication, addition, etc. on two arrays by first combining the arrays with an InnerDJoin. However, if the arrays match on the Dimension Domain, the Combine operator is a simpler way to express such operations. Using Apply followed by Fill allows one to externalize the Valid map of an array A as a Boolean array:

Fill(Apply(A, true), false)
In this expression, *true* and *false* should be read as constant functions. That is, each returns the same result on every input.

{Example of projection using Apply}

**Combine**
*Combine*(A, B, bfcn, lfcn, rfcn) where
bfcn:CDa × CDa → CDa
lfcn:CDa → CDa
rfcn:CDa → CDa
CDa, CDa, and CDa are the content domains of A, B and the result, respectively.

*Description:* This operator combines two similar arrays, acting somewhat like a Union. (But it’s not commutative.) Arrays A and B must have the same Dimension Domain, but can have different validity maps. The result will be valid where either A or B is valid. The three functions handle the cases of both A and B cells being valid, only the A cell is valid, and only the B cell is valid. (If both are invalid, the result is invalid.) Combine can be used to concatenate arrays along a dimension by padding them with Rebox first.
Definition: Assume
A = (Box_A, Valid_A, Content_A)
B = (Box_B, Valid_B, Content_B)
where Box_A = Box_B. Consider
Combine(A, B, fb, fl, fr) = R = (Box_R, Valid_R, Content_R).
Then
Box_R = Box_A.

For d ∈ Box_R:
Valid_R(d) = Valid_A(d) ∨ Valid_B(d)
Content_R(d) = fb(Content_A(d), Content_B(d))
if Valid_A(d) ∧ Valid_B(d).

\[
\text{Content}_R(d) = fl(Content_A(d)) \text{ if Valid}_A(d) \land \neg \text{Valid}_B(d)
\]
\[
\text{Content}_R(d) = fr(Content_B(d)) \text{ if Valid}_B(d) \land \neg \text{Valid}_A(d).
\]

\text{Notes:} This is a generalized version of the Overlap operator in version 1. \text{Overlap(A, B)} can be expressed as

\[
\text{Combine(A, B, fb, fl, fr)} \text{ where}
fb(v_A, v_B) = v_A
fl(v_A) = v_A
fr(v_B) = v_B.
\]

\text{InnerDJoin}
\text{InnerDJoin(A, B)}

\text{Note:} InnerDJoin is completely generalized by InnerEJoin, which follows. I leave it in as an aid to understanding the more general case.

\text{Description:} InnerDJoin (inner dimension join) joins A and B on their common dimensions. Each common dimension must have the same bounds. (If there are no common dimensions, we get a cross product of the Dimension Domains.) It is an inner join because a result cell is valid only if both contributing cells are valid. (If we need an outer join, we can fill holes first with Apply.) Note that if Box_A = Box_B, then InnerDJoin has a result over the same Box that concatenates corresponding cell values. We also require that the Content Domains of A and B have no attribute names in common.

**Preliminaries:** We need some functions (which we will write in infix notation) that merge various entities. First, the merge of two boxes Box_A and Box_B, with 0 or more dimensions in common, is written Box_A#Box_B. The requirement is that for any common dimension D, D.lower and D.upper are the same. The merge of the two Boxes is essentially the natural join of their Dimension Domains: we get the combination of dimensions, with duplicates removed. For instance, if

Box_A = [I:1..3, J:5..8]
Box_B = [J:5..8, K:10..12]
then
Box_A#Box_B = [I:1..3, J:5..8, K:10..12].

Notice that if Box_A = Box_B, then
Box_A#Box_B = Box_A.

Similarly, we write d_A#d_B for the merge dimension vectors d_A ∈ Box_A and d_B ∈ Box_B when they have the same value on the common dimensions. For instance,


We also use the notation r_A#r_B for merging cell values r_A ∈ CD_A and r_B ∈ CD_B. Here CD_A and CD_B are the Content Domains for A and B, respectively. We require that CD_A and CD_B have no attributes in common (otherwise use Rename to remove the conflict). An example of this function is


Definition: Assume
A = (Box_A, Valid_A, Content_A)
B = (Box_B, Valid_B, Content_B)
where Box_A & Box_B are compatible (same-named dimensions have the same bounds) and Content_A & Content_B have Content Domains with disjoint attribute names. Consider
InnerDJoin(A, B) = R = (Box_R, Valid_R, Content_R).
Then
Box_R = Box_A#Box_B.

For d = d_A#d_B ∈ Box_R:
Valid_R(d) = Valid_A(d_A) ∧ Valid_B(d_B)
Content_R(d) = Content_A(d_A)#Content_B(d_B) if Valid_B(d).

\text{Notes:} InnerDJoin operates only on dimensions. Certain kinds of attribute-based joins can be defined by InnerDJoin followed by Filter.

\text{InnerEJoin}
\text{InnerEJoin(A, B)}

\text{Description:} InnerEJoin (inner extended join) joins A and B on their common components, be they dimensions or attributes. In particular, an attribute in A can join with a dimension with the same name in B (but not a dimension in A with an attribute in B). Matching dimensions must have the same bounds. An attribute that matches a dimension must have integer type. Two matching attributes must have the same type.

In the result of InnerEJoin, any dimension in B that is matched to an attribute in A is “demoted” to an attribute in the result. The Dimension Domain of the result will thus consist of all dimensions in the inputs (duplicates removed) that are not involved in a match to an attribute. We need to extent the merge function ‘#’ to work over the entire schema (Box and Content Domain) of two arrays, to reflect the handling of dimension-to-attribute matches. Examples:

(a) \text{Dimension-only join:} If
Schema_A = (rv: Float) [I:1..3, J:5..8]
Thus we have a specific combination of L and M for indexing that I and J determine a unique value for the L attribute, and D indices determines a unique cell in Note that in this example, a given combination of I, J and M but it is not the case that when they are compatible, written as an entry consisting of cell value E A D A A = (cv: Char) [J:5..8, L:1..256, M:1..8] A D A A = (rv: Float, L: Int) [I:1..3, J:5..8] A D A A = (rv: Float, bv: Bool) [I:1..3, J:5..8] A D A A = (rv: Float, L: Int) [I:1..3, J:5..8] A D A A = (sv: Float) [L:1..256, M:1..8] A D A A = (rv: Float, bv: Bool) [I:1..3, J:5..8] A D A A = (rv: Float, L: Int) [I:1..3, J:5..8] A D A A = (cd: Char) [L:1..256] A D A A = (iv: Bool) [J:5..8, K:10..12] A D A A = (rv: 2.77)<I:2, J:5>. So, for example, an entry from array A in the result array will only be defined in the case that attribute L in A will be the entry in B, meaning a cell index plus the cell value at the corresponding index. We extend ‘#’ to work over pairs of entries, where an entry is a cell index plus the cell value at the corresponding index. We will write an entry consisting of cell value \( r \) and dimension vector \( d \) as \( r,d \). So, for example, an entry from array A in the examples above might be \( r_A,d_A = (rv: 2.77)<I:2, J:5> \).

The merge of two entries \( r_A,d_A \) and \( r_B,d_B \) will only be defined when they are compatible, written \( r_A,d_A \sim r_B,d_B \), meaning same-named components have the same values. So, considering example (c) above, \( (rv: 2.77, L: 118)<I:2, J:5> \sim (sv: 0.2)<L: 118, M:7> \) but it is not the case that \( (rv: 2.77, L: 118)<I:2, J:5> \sim (sv: 0.2)<L: 119, M:7> \). Note that in this example, a given combination of I, J and M indices determines a unique cell in A and at most one cell in D. The reason there can be at most one compatible cell in D is that I and J determine a unique value for the L attribute, and thus we have a specific combination of L and M for indexing in D. Note that if we had an entry \( (rv: 6.28, L: 310)<I:2, J:5> \) in A, there will be no compatible entry in D, because L: 310 isn’t in the range 1..256. I will assume that there is a pre-filter on the left argument to InnerEJoin that sets to invalid any cells with such out-of-range value for an attribute-dimension match.

As another example, consider case (d) above. We have \( (rv: 2.77, bv: T)<I:2, J:5> \sim (bv: T, sv: 0.6)<J:5, M:3> \) but not \( (rv: 2.77, bv: T)<I:2, J:5> \sim (bv: F, sv: 0.8)<J:5, M:4> \). In the latter case, because of the attribute mismatch on bv, the result array will be invalid at \( <I:2, J:5, M:4> \).

Given two entries \( r_A,d_A \) and \( r_B,d_B \) where \( r_A,d_A \sim r_B,d_B \), the result of merging, \( r_A,d_A \# r_B,d_B \), will be the entry \( r_X,d_X \) over the result schema that matches the attributes and dimension values of the two input entries. Taking the two examples above that were compatible, we have \( (rv: 2.77, L: 118)<I:2, J:5> \# (sv: ’d’)<L: 118, M:7> \) \( (rv: 2.77, L: 118, sv: ’d’)<I:2, J:5, M:7> \) and \( (rv: 2.77, bv: T)<I:2, J:5> \# (bv: T, sv: 0.6)<J:5, M:3> \) \( (rv: 2.77, bv: T, sv: 0.6)<I:2, J:5, M:3> \). We extend ‘#’ to work over pairs of entries, where an entry is a cell index plus the cell value at the corresponding index. We will write an entry consisting of cell value \( r \) and dimension vector \( d \) as \( r,d \). So, for example, an entry from array A in the examples above might be \( r_A,d_A = (rv: 2.77)<I:2, J:5> \).

The merge of two entries \( r_A,d_A \) and \( r_B,d_B \) will only be defined when they are compatible, written \( r_A,d_A \sim r_B,d_B \), meaning same-named components have the same values. So, considering example (c) above, \( (rv: 2.77, L: 118)<I:2, J:5> \sim (sv: 0.2)<L: 118, M:7> \) but it is not the case that \( (rv: 2.77, L: 118)<I:2, J:5> \sim (sv: 0.2)<L: 119, M:7> \). Note that in this example, a given combination of I, J and M indices determines a unique cell in A and at most one cell in D. The reason there can be at most one compatible cell in D is that I and J determine a unique value for the L attribute, and thus we have a specific combination of L and M for indexing in D. Note that if we had an entry \( (rv: 6.28, L: 310)<I:2, J:5> \) in A, there will be no compatible entry in D, because L: 310 isn’t in the range 1..256. I will assume that there is a pre-filter on the left argument to InnerEJoin that sets to invalid any cells with such out-of-range value for an attribute-dimension match.

As another example, consider case (d) above. We have \( (rv: 2.77, bv: T)<I:2, J:5> \sim (bv: T, sv: 0.6)<J:5, M:3> \) but not \( (rv: 2.77, bv: T)<I:2, J:5> \sim (bv: F, sv: 0.8)<J:5, M:4> \). In the latter case, because of the attribute mismatch on bv, the result array will be invalid at \( <I:2, J:5, M:4> \).

Given two entries \( r_A,d_A \) and \( r_B,d_B \) where \( r_A,d_A \sim r_B,d_B \), the result of merging, \( r_A,d_A \# r_B,d_B \), will be the entry \( r_X,d_X \) over the result schema that matches the attributes and dimension values of the two input entries. Taking the two examples above that were compatible, we have \( (rv: 2.77, L: 118)<I:2, J:5> \# (sv: ’d’)<L: 118, M:7> \) \( (rv: 2.77, L: 118, sv: ’d’)<I:2, J:5, M:7> \) and \( (rv: 2.77, bv: T)<I:2, J:5> \# (bv: T, sv: 0.6)<J:5, M:3> \) \( (rv: 2.77, bv: T, sv: 0.6)<I:2, J:5, M:3> \).
dimensions, in which case the output is over the Box [] with no dimensions – a single cell value.

**Definition**: Consider

\[
\text{Reduce}(A, DS, a) = R = (\text{Box}_R, \text{Valid}_R, \text{Content}_R).
\]

Then

\[
\text{Box}_R = \text{Box}_A \text{ with dimensions in } DS \text{ removed.}
\]

For example, with \( \text{Box}_A \) defined as before and \( DS = \{J\} \), then \( \text{Box}_R = [I:1..3] \). We also need

\[
\text{Box}_DS = \text{Box}_A \text{ with dimensions in } DS \text{ preserved.}
\]

So in the example, \( \text{Box}_DS = [J:5..8] \).

For \( d \in \text{Box}_R \):

\[
\text{Valid}_R(d) = \text{true}
\]

\[
\text{Content}_R(d) = \text{agg}(A[d])
\]

where \( A[d] \) is the array over \( \text{Box}_DS \) we get by fixing the indexes in \( d \) for the other dimensions. For example,

\[
A[<I:2>] \text{ is the subarray over } [J:5..8] \text{ we get by fixing } I = 2.
\]

{Give an example of reducing one dimension (maybe a matrix multiply case) and reducing all dimensions}

**Notes**: The definition allows aggregation functions that are aware of cell order in the reduced dimensions (as the input is a subarray). However, we may want to restrict ourselves to functions over sets of values (MIN, SUM, etc.) initially. Also, Reduce produces an array that is valid over its whole box, and it’s up to agg to figure out what to do with invalid cells.

**Final Comments**

Not everything is possible with this algebra. There is no way to promote attributes into dimensions, or demote a dimension into an attribute. But I think that’s a reasonable limitation for Version 1.