# LaraDB: A Minimalist Kernel for Linear and Relational Algebra Computation

Dylan Hutchison, Bill Howe, Dan Suciu
{dhutchis,billhowe,suciu}@cs.washington.edu

## ABSTRACT

Analytics tasks manipulate structured data with variants of relational algebra (RA) and quantitative data with variants of linear algebra (LA). The two computational models have overlapping expressiveness, motivating a common programming model that affords unified reasoning and algorithm design. At the logical level we propose LARA, a lean algebra of three operators, that expresses RA and LA as well as relevant optimization rules. We show a series of proofs that position LARA at just the right level of expressiveness for a middleware algebra: more explicit than MapReduce but more general than RA or LA. At the physical level we find that the LARA operators afford efficient implementations using a single primitive that is available in a variety of backend engines: range scans over partitioned sorted maps.

To evaluate these ideas, we implemented the LARA operators as range iterators in Apache Accumulo, a popular implementation of Google's BigTable. First we show how LARA expresses a sensor quality control task, and we measure the performance impact of optimizations LARA admits on this task. Second we show that the LARADB implementation outperforms Accumulo's native MapReduce integration on a core task involving join and aggregation in the form of matrix multiply, especially at smaller scales that are typically a poor fit for scale-out approaches. We find that LARADB offers a conceptually lean framework for optimizing mixed-abstraction analytics tasks, without giving up fast record-level updates and scans.

## CCS Concepts

•**Information systems → Query operators;** *Query languages;* Query optimization; Physical data models;

## 1. INTRODUCTION

Analytics tasks involve preprocessing (ETL, restructuring, cleaning) that are typically expressed using relational algebra-based languages as well as numerical tasks (machine learning, optimization, signal processing) that are typically expressed using linear algebra operations [16]. The distinction between these two programming styles is increasingly blurred: machine learning applications are implemented using RA-oriented interfaces using, for example, Spark, sometimes with extensions for special cases [25, 18, 27, 5].

The relevant tasks involve both programming styles, but systems tend to favor one or the other. An LA-oriented system may emphasize matrix operations in the programming interface but require awkward gymnastics with column and row indices to implement even simple SPJ queries. A relational system, in contrast, obscures matrix properties suitable for optimization and algorithm selection. Some systems allow explicit transformation of relations into matrices and vice versa, exposing a different set of programming idioms for each data type [24]. A common programming environment where both styles can be used interchangeably is desirable, as is being explored by a number of systems and libraries, including Myria [32], Spark [34], and more.

These systems emphasize mixed-programming syntax but assume more conventional internal computational models, many based on RA. The benefit of this approach is that conventional RA properties and rewrites are easy to exploit for optimization. The problem with this approach is that LA properties and rewrites are obscured, or entirely inexpressible. We find it desirable to use theorems from both LA and RA when reasoning about queries. For example, the fact that the inner product $U^\intercal U$ is symmetric suggests an immediate optimization: only produce its upper triangle rather than also computing its lower triangle redundantly. Although this optimization is possible to implement (and prove) using RA, it is far from obvious, and no RDBMS applies this optimization. We seek a new set of abstractions to facilitate the use of similar theorem-oriented optimizations.

We propose a lean algebra of three operators, LARA, that subsumes the operators and rules of both LA and RA. We keep the number of operators to a minimum to simplify an initial implementation on a number of backend systems, and to simplify reasoning during optimization by avoiding large numbers of special cases for relatively simple concepts (pushing selections, etc.). As with other systems for big data processing [11, 14, 2, 8], LARA operators are parameterized by rich user-defined functions, and the properties of these functions are involved in optimization. However, LARA emphasizes a more restricted semiring structure to capture the properties of vector space algebra as opposed to emphasizing the "free-for-all" UDF approach other systems emphasize.

We also propose a physical algebra, PLARA, that allows reasoning about low-level optimizations such as operator fu-

*BeyondMR'17 May 19, 2017, Chicago, IL, USA*
© 2017 ACM. ISBN 978-1-4503-5019-8/17/05. . . $15.00
DOI: http://dx.doi.org/10.1145/3070607.3070608

sion, elimination of unnecessary writes, and shared scans.

Finally we show how the physical algebra can be implemented efficiently in a distributed system using only a single efficient primitive: range scans over partitioned sorted maps. We find this primitive to be nearly universally supported across RDBMS, NoSQL systems, linear algebra libraries, file-based systems, and others.

The object of Lara is the *Associative Table*, a data structure capturing core properties of relations, tensors, and key-values. The operators of Lara are ext (flatmap), join (horizontal concatenation), and union (vertical concatenation).

Our contributions are:

1. A minimal logical algebra, Lara, to unify LA and RA.
2. A physical algebra over sorted partitioned maps that exposes low-level optimization opportunities.
3. A system, LaraDB, implementing the Lara abstractions on the Apache Accumulo database.
4. An evaluation showing that on representative tasks, LaraDB is competitive with a hand-coded MR implementation at scale, and far faster at small scales.
5. An evaluation of LaraDB on a more complex sensor validation task that demonstrates the kinds of optimizations exposed by the Lara abstractions.

## 2. RELATED WORK

MapReduce promoted a minimalist approach to distributed programming, but had no real capabilities for reasoning over and optimizing the resulting programs. As a result, a spate of SQL-on-Hadoop projects emerged in the first few years. Since that early period, a number of projects have proposed more refined approaches that balance flexibility for the programmer and optimizability by the system.

Fegaras et al. superimpose three operations atop MapReduce: cmap (abbreviation of concat-map, which implements flatmap), groupBy, and join [13]. These operators expose optimization opportunities, but the authors do not explore the relationship between these operations and linear algebra.

Elgama et al proposed a variety of sum-product optimizations and operator fusion techniques for SystemML called SPOOF by representing matrix equivalences in RA and applying relational optimizations [12]. This approach is similar to our own but focuses on developing a single, tightly coupled system involving heavy use of code generation, as opposed to our goal of providing a general abstraction that can be naturally implemented in many contexts.

Kunft et al described a vision for optimization involving both linear and relational algebra equivalences, but did not describe an implementation of the ideas [24]. Our paper was inspired in part by this work; we use an adaptation of their running example in our experiments.

Palkar et al proposed Weld, a common runtime that replaces the runtime of libraries like Spark, Pandas, and NumPy in order to optimize within and across them [28]. Weld's algebraic basis is an adaptation of map and reduce termed "loops and builders". We designed Lara with one step more structure by differentiating "horizontal" from "vertical" group-by, in order to obtain greater reasoning capability.

Marker et al built a cost-based optimizer, DxTer, and applied it to the task of tensor contractions in MPI environments [26]. Though DxTer lays a foundation for RA-style reasoning, they focused solely on the domain of dense LA.

| $t$ | $c$ | $v$ [$\perp$] |
|---|---|---|
| 440 | hum | 38.6 |
| 466 | temp | 55.2 |
| 466 | hum | 40.1 |
| 492 | temp | 56.3 |
| 492 | hum | 35.0 |
| 528 | temp | 56.5 |

Figure 1: Table $A$ with (time, class, value) sensor measurements. The default value $\perp$ identifies non-measured values.

Crotty et al developed Tupleware, a cluster programming environment emphasizing code generation and stateful analytics, but the emphasis is on low-level programming idioms rather than marrying logical and physical abstractions [11].

Rheinlander et al proposed a logical optimizer for UDF-centric dataflows called SOFA [29]. SOFA emphasizes properties of UDFs to facilitate optimizations. Our approach is complementary, but we focus on properties that allow reasoning about LA, specifically semiring structures.

Semirings are the main focus of Associative Arrays, a data structure generalizing LA's sparse matrices whose operations were shown to subsume RA [23]. Lara and associative arrays share design choices such as sparsity and pluggable $\oplus$ and $\otimes$. Ultimately, Lara compromises between associative arrays (heavily LA-based) and relations (heavily RA-based).

Lattices were also posed as a basis for RA via two operators: natural join and inner union [31]. The relational lattice is a special case of Lara, especially w.r.t. Section 3.3's distributive laws, when Lara is relaxed to allow tables with infinite support and restricted to only use key attributes.

## 3. LOGICAL ALGEBRA

Our hypothesis underlying Lara is that the objects of RA and LA—relations, scalars, vectors, matrices, and tensors—can be recast into a rectangular representation of multidimensional key-value data. RA and LA operate on "rectangular blocks" of key-value records by either applying a function "record-wise" (think selection, projection, flatmap, function application), by combining blocks "horizontally" (think Cartesian product, tensor product, relational joins), or by combining blocks "vertically" (think relational union, aggregation, tensor contraction). These operations are composable; matrix multiply, for example, is a horizontal operation followed by a vertical one. By expressing these kernels directly, rather than their particular instantiation in RA or LA, we aim to reason about RA and LA tasks uniformly. We call our new representation an *associative table*.

Figure 1 shows an example table consisting of environmental sensor data. It has two keys—time and measurement class—that map to measurement values. The table can be thought of as a lookup function: given a key, return the value it maps to, or the default value ($\perp$, in this table) if the key has no mapping. Default values allow us to model sparse and dense matrices uniformly, for example, in which case we would use numeric 0 as the default value.

### 3.1 Running Example: Sensor Quality

To illustrate the Lara algebra, consider the example task and its translation into Lara in Figure 2. We adapted this example from Kunft et al [24, Listing 1], in which the authors motivated an RA-LA hybrid query language like Lara.

In this example, a manufacturer seeks to calibrate newly produced sensors to a "gold standard" sensor. Each sen-

| | Pseudocode | Lara Logical Plan |
|---|---|---|
| 1 | $A, B = \textsc{Load}$ 's1', 's2' | $A = \textsc{Load}$ 's1' |
| 2 | *// RA-style (SQL) bin, filter* | $A_1 = \textsc{Map}\ A\ \textsc{by}\ [v\text{: if}(460 \le t \le 860)\ v \text{ else } \bot]$ |
| 3 | $A' = \textsc{select}$ bin($t$) $\textsc{as}\ t', c$, avg($v$) $\textsc{from}\ A\ \textsc{where}\ 460 \le t \le 860$ | $A_2 = \textsc{Ext}\ A_1\ \textsc{by}\ \begin{array}{c\|cc} t' & v & cnt \\ \hline \text{bin}(t) & v & v \ne \bot \end{array}$ |
| 4 | $\textsc{group by}\ t', c$ | $A_3 = \textsc{Agg}\ A_2\ \textsc{on}\ t', c\ \textsc{by}\ [v\text{: } +,\ cnt\text{: } +]$ |
| 5 | | $A' = \textsc{Map}\ A_3\ \textsc{by}\ [v\text{: } v/cnt]$ |
| 6 | $B' = \textsc{select}$ bin($t$) $\textsc{as}\ t', c$, avg($v$) $\textsc{from}\ B\ \textsc{where}\ 460 \le t \le 860$ $\textsc{group by}\ t', c$ | $B' = \ldots$ *// repeat above for second sensor* |
| | *// LA-style (MATLAB) mean, covariance of residuals $A' - B'$, viewed as dense matrices:* | |
| 7 | $X = A' - B'$ *// residuals; $\|t'\| \times \|c\|$ matrix* | $X = \textsc{Join}\ A', B'\ \textsc{by}\ [v\text{: } -]$ |
| 8 | $N = \text{size}(X, 1)$ *// # unique t's; scalar* | $X_1 = \textsc{Map}\ X\ \textsc{by}\ [v\text{: } v \ne \bot]$ |
| 9 | | $X_2 = \textsc{Agg}\ X_1\ \textsc{on}\ t'\ \textsc{by}\ [v\text{: any}]$ |
| 10 | | $N = \textsc{Agg}\ X_2\ \textsc{by}\ [v\text{: } +]$ |
| 11 | $M = \text{mean}(X, 1)$ *//c means; $1 \times \|c\|$ vector* | $X_3 = \textsc{Map}\ X\ \textsc{by}\ [v\text{: } v,\ cnt\text{: } v \ne \bot]$ |
| 12 | $\textsc{Store}\ M$ | $X_4 = \textsc{Agg}\ X_3\ \textsc{on}\ c\ \textsc{by}\ [v\text{: } +,\ cnt\text{: } +]$ |
| 13 | | $M = \textsc{Map}\ X_4\ \textsc{by}\ [v\text{: } v/cnt]$ |
| 14 | $U = X - \text{repmat}(M, N, 1)$ *//subtract mean* | $U = \textsc{Join}\ X, M\ \textsc{by}\ [v\text{: } -]$ |
| 15 | $C = U^\intercal U/(N-1)$ *// c covariances;* | $U_1 = \textsc{Rename}\ U\ \textsc{from}\ c\ \textsc{to}\ c'$ |
| 16 | $\textsc{Store}\ C$ *// $\|c\| \times \|c\|$ matrix* | $U_2 = \textsc{Join}\ U_1, U\ \textsc{by}\ [v\text{: } \times]$ |
| 17 | | $U_3 = \textsc{Agg}\ U_2\ \textsc{on}\ c, c'\ \textsc{by}\ [v\text{: } +]$ |
| 18 | | $C = \textsc{Join}\ U_3, N\ \textsc{by}\ [v\text{: } v/(v'-1)]$ |

|  |  |  | [⊥] | [0] |
|---|---|---|---|---|
| $t$ | $c$ | $t'$ | $v$ | $cnt$ |
| 466 | temp | 460 | 55.2 | 1 |
| 466 | hum | 460 | 40.1 | 1 |
| 492 | temp | 520 | 56.3 | 1 |
| 492 | hum | 520 | 35.0 | 1 |
| 528 | temp | 520 | 56.5 | 1 |

(a) Table $A_2$

|  |  | [⊥] |
|---|---|---|
| $t'$ | $c$ | $v$ |
| 460 | temp | 55.2 |
| 460 | hum | 40.1 |
| 520 | temp | 56.4 |
| 520 | hum | 35.0 |

(b) Table $A'$

|  |  | [⊥] |
|---|---|---|
| $t'$ | $c$ | $v$ |
| 460 | temp | -3.1 |
| 460 | hum | 1.6 |
| 520 | temp | -4.0 |
| 520 | hum | -0.8 |

(c) Table $X$

|  | $v$ |
|---|---|
| () | 2 |

(d) $N$

|  | [⊥] |
|---|---|
| $c$ | $v$ |
| temp | 0.4 |
| hum | -3.5 |

(e) Table $M$

|  |  | [⊥] |
|---|---|---|
| $t'$ | $c$ | $v$ |
| 460 | temp | 0.4 |
| 460 | hum | 1.2 |
| 520 | temp | -0.4 |
| 520 | hum | -1.2 |

(f) Table $U$

|  |  | [⊥] |
|---|---|---|
| $c_1$ | $c_2$ | $v$ |
| temp | temp | 0.32 |
| temp | hum | 0.96 |
| hum | temp | 0.96 |
| hum | hum | 2.88 |

(g) Table $C$

Figure 2: Example to compute the mean $M$ and covariance $C$ of sensor residual differences $X$ after filtering and aligning their measurements $A, B$. The example is given in pseudocode in the left panel involving RA and LA operations. The bin function is defined as $\text{bin}(t) = t - \text{mod}(t, 60) + 60\lfloor \frac{\text{mod}(t,60)}{60} + .5 \rfloor$. The right panel shows a translation into the Lara algebra. Figure 1 presents sample data for sensor $A$; sensor $B$ is omitted. The bottom panel displays results calculated from the sample data.

sor records temperature, humidity, and other environmental data, albeit at different rates and times. The goal is to compute means and covariances of the residual difference between new sensors' and the trusted sensor's measurements.

## *Pseudocode.*

In line 1, we load two sensors' measurements into associative tables $A$ and $B$. In lines 3 and 6, the two sensor streams are filtered to a region of interest and binned to minute intervals. This task is naturally expressed as a SQL query over a set of records. In line 7, $A'$ and $B'$ are interpreted as matrices, such that the residuals can be computed directly using element-wise subtraction. This task is possible to express in SQL, but it would involve a multi-attribute join condition and a new column alias.

In line 8, we assume a MATLAB-style function size that computes the number of unique time bins. We use this function to illustrate a different programming style; a simple SQL count distinct query would also suffice. In line 11, the mean of each attribute $c$ across all $t$ is computed. Here, the MATLAB-style syntax is quite useful; it is tedious and error-prone in SQL to aggregate many columns in one query.

Finally, calculating covariance[1] consists of subtracting the mean from each measurement into $U$ (using the MATLAB function repmat, which repeats the row vector $M$ to the same matrix shape as $X$), then computing $C = U^\intercal U$ and dividing by the time count $N - 1$.[2]

## *Lara Logical Plan.*

The right of Figure 2 presents a translation of the pseudocode into a logical plan expressed in the Lara algebra. Each line in the Lara plan consists of a single operator.

Figure 3 lists the three core operators of Lara, defined in Section 3.2. Here we provide intuition using our running example. The operators we use in Figure 2 are Ext (apply a function to each record, possibly adding new keys), Map (apply a function to each record without changing keys), Agg (relational group by), Join (relational join on keys), and Rename (relabel an attribute). Map and Rename are special cases of the core operator Ext. Agg is a special case of the core operator Union. Join is itself a core operator.

Each line generally takes the form <*op*> <*table*> BY <*expressions*>, where <*expressions*> is akin to a SELECT clause in SQL with aggregate and arithmetic expressions. In this example functions handle the default value $\bot$ the same as NULL, but a crucial concept in Lara is that value attributes may have different default values. Table $A_2$ in Figure 2(a) has a 0 default value, for example. Line 3 uses a tableau

[1]Covariance is given by $C = \mathbb{E}\left[(X - \mathbb{E}X)^\intercal (X - \mathbb{E}X)\right]$

[2]Dividing by $N - 1$ forms an unbiased covariance estimator.

| Lara Operator | output $\bar{k}$ | output $\bar{v}$ |
|---|---|---|
| Union $A, B$ by $\oplus$ | $\bar{k}_A \cap \bar{k}_B$ | $\bar{v}_A \cup \bar{v}_B$ |
| Join $A, B$ by $\otimes$ | $\bar{k}_A \cup \bar{k}_B$ | $\bar{v}_A \cap \bar{v}_B$ |
| Ext $A$ by $f$ | $\bar{k}_A$ extended by $f$ | set by $f$ |

Figure 3: Summary of Lara's operators and their effect on table schema, i.e., the names of key and value attributes. For example, union's key names are the intersection of $A$ and $B$'s key names. Union and join are dual in this respect.

notation for the table-valued output of its user-defined function; Equation 1 shows an example of this function in action.

We encourage the reader to trace through the algorithm with the example tables at the bottom of Figure 2, which stem from the first sensor $A$'s data in Figure 1 and a second sensor $B$'s data which is not shown.

## 3.2 Lara Defined

In this section we define associative tables and the three core Lara operators. Figure 3 summarizes the Lara operators and their effect on associative table structure.

### *Tuples and Associative Tables.*

The notation $\bar{a} = [t\colon 135, c\colon \text{temp}, v\colon 55.2]$ defines $\bar{a}$ as a tuple of three elements named $t$, $c$, and $v$. All tuples have names associated with their values. Writing tuples $\bar{a}, \bar{b}$ side-by-side denotes their concatenation. We use $\pi_X$ to denote tuple projection onto names $X$; for example, $\pi_t\, \bar{a} = [t\colon 135]$.

An associative table $A : \bar{k} \to \bar{v} : \bar{0}$ is a total function from key attributes $\bar{k}$ to value attributes $\bar{v}$ with default values $\bar{0}$.

The *support* of $A$ is the set of keys that map to non-default values. Associative tables always have finite support. The expression $A(\bar{k})$ retrieves the $\bar{v}$ associated with $\bar{k}$ if $\bar{k}$ is in $A$'s support, or default values $\bar{0}$ if $k$ is not in $A$'s support.

### *Union.*

As the "vertical concatenation" of tables, the expression

$$\text{Union } A, B \text{ by } \bar{\oplus}$$

creates an associative table over the shared key attributes of $A$ and $B$, aggregating values that map to the same key.

Union takes a tuple of user-defined $\oplus$ functions, one for each value attribute in $A$ and $B$, to sum colliding values. Collisions are keys from $A$ and $B$'s support that match on their common key attributes. The collisions are summed via *structural recursion* [7], a strategy to sum values pair-wise until a single value is obtained. For this paper we assume each $\oplus$ is associative and commutative, which implies that we can sum values in any order. In general we can relax this assumption when $A$ and $B$'s keys have a total order.

We require that each $\oplus$ have $A$ and $B$'s default value as its identity: $0 \oplus v = v \oplus 0 = v$, which forces $A$ and $B$ to have the same 0. This requirement ensures that union has the same result independent of whether default values are stored in $A$ or $B$; extra default values merely add extra 0s.

Often a Lara expression takes the union of a table $A$ with an empty table $E_{\bar{k}}$, i.e., a table with key attributes $\bar{k}$ and empty support. Such a union aggregates $A$ onto the key attributes $\bar{k}$. For this common case we use the shorthand

$$\text{Agg } A \text{ on } \bar{k} \text{ by } \bar{\oplus}$$

### *Join.*

As the "horizontal concatenation" of tables, the expression

$$\text{Join } A, B \text{ by } \bar{\otimes}$$

creates an associative table over the union of $A$ and $B$'s keys. Join forms the Cartesian product of $A$ and $B$'s values that match on their keys in common, and multiplies them.

A tuple of $\otimes$ functions, one for each value attribute present in both $A$ and $B$, multiply their values. The default value of join is the multiplication of $A$ and $B$'s default values.

We require that each $\otimes$ have $A$ and $B$'s default values as its annihilators: $0_A \otimes v = v \otimes 0_B = 0_A \otimes 0_B$. As with union, this requirement ensures that join is independent of whether default values are stored in $A$ or $B$; extra default values merely multiply extra 0s.

### *Ext.*

Also known as "flatmap", ext is the extension[3] of a function $f$ on tuples to a Lara operator on tables written

$$\text{Ext } A \text{ by } f$$

The $f$ in ext is a user-defined function that returns a new associative table for every tuple. The keys in the table returned by $f$ append to $A$'s keys in the result. The values returned by $f$ replace $A$'s values in the result.

We demonstrate this process with an example from our running sensor example. Take the ext from Figure 2 line 3. The action of this ext's $f$ on the second row of Table $A$ is

$$f([t\colon 466, c\colon \text{temp}, v\colon 55.2]) = \begin{array}{c|cc} t' & v & cnt \\ \hline 460 & 55.2 & 1 \end{array} \qquad (1)$$

We require that $f$ satisfy two properties to be used in an ext. The tables produced by $f$ must be valid tables with finite support, and specifically when passed default values, $f$ must produce a table with empty support. These requirements guarantee that the result of ext has finite support, and they offer independence from storing default values in $A$; default values do not produce support in Ext's result.

A common special case of ext produces no additional key attributes. We call out this behavior with the expression

$$\text{Map } A \text{ by } f$$

To illustrate how Map relates to Ext, we show how the Map in Figure 2 line 2 would be written as an Ext:

$$\text{Map } A \text{ by } [v\colon \text{if } (460 \le t \le 860)\ v \text{ else } \bot]$$
$$\equiv \quad \text{Ext } A \text{ by } \begin{array}{c|c} & v \\ \hline () & \text{if } (460 \le t \le 860)\ v \text{ else } \bot \end{array}$$

Another common special case renames keys or values. Renaming is crucial for the correct application of join and union, whose semantics depend on the common and distinct names of their input's keys and values. We write rename as

$$\text{Rename } A \text{ from } x \text{ to } y$$

When $x$ is a value attribute, renaming is straightforward; a map function $f([x\colon v]) = [y\colon v]$ (holding other value attributes constant) performs the renaming. When $x$ is a key, the expression is shorthand for an Ext that adds a new $y$ key and an Agg that removes the old $x$ key. The union does not incur aggregation because collisions cannot occur.

Promoting a value to a key is a simple Ext, and demoting a key to a value is an Agg that may incur aggregation.

[3]Coined by Buneman et al [7], we chose the term "ext" over "flatmap" to emphasize that ext can extend a table's keys. It also indicates monadic bind that is monotonic on key types.

*Formal Definitions.*

We now present a more concise algebraic syntax for the LARA operators that is useful for writing identities and proving theorems. We encourage the reader to use the COBOL-style syntax when writing scripts.

Suppose we have associative tables $A$ and $B$ with types

$$A : a, c \to x, z : 0^x, 0^z \qquad B : c, b \to z, y : 0^z, 0^y$$

i.e., where $c$ and $z$ are keys and values common to $A$ and $B$, and $a, b, x$, and $y$ are keys and values unique to $A$ or $B$. Though we write these as individual attributes, the definitions hold when these are tuples (e.g. $\bar{a}, \bar{c}, \bar{b}, \bar{x}, \bar{z}, \bar{y}, \bar{0}^x, \ldots$).

In the case of $\otimes$ and join, the common value attribute $z$ is allowed to differ in type and default value between $A$ and $B$. We omit this case to maintain clarity.

Suppose we have the user-defined functions

$$\begin{aligned} &\oplus : i \times i \to i \quad \text{for } i \in \{x, y, z\} \\ &\otimes : z \times z \to z' \\ &f : a, c \times x, z \to (k' \to v' : 0') \end{aligned}$$

In the case of $\oplus$, the definition holds when there is a different $\oplus$ for each attribute $x, y, z$. However, we only write the case when all the $\oplus$ are the same for clarity.

We require that $\oplus, \otimes$, and $f$ obey the equations

$$\begin{gathered} \forall i,\, i \oplus 0^i = 0^i \oplus i = 0^i \text{ for } i \in \{x, y, z\} \\ \forall z,\, 0^z \otimes z = z \otimes 0^z = 0^z \otimes 0^z \\ \forall a, c, k',\, f(a, c, 0^x, 0^z)(k') = 0' \\ \forall a, c, x, z,\, f(a, c, x, z) \text{ has finite support} \end{gathered}$$

We also have a technical consistency requirement that $f$ produce tables of the same schema (attribute names) $\forall a, c, x, z$.

Given the above, we define the LARA operators as

$$\begin{aligned} &A \bowtie_\otimes B : a, c, b \to z : 0^z \otimes 0^z \\ &(A \bowtie_\otimes B)(a, c, b) := [z \colon \pi_z A(a, c) \otimes \pi_z B(c, b)] \\ &A ⧖_\oplus B : c \to x, z, y : 0^x, 0^z, 0^y \\ &(A ⧖_\oplus B)(c) := \big[x \colon \bigoplus_a \pi_x A(a, c), \\ &\qquad z \colon \bigoplus_a \pi_z A(a, c) \oplus \bigoplus_b \pi_z B(c, y),\ y \colon \bigoplus_b \pi_x B(c, y)\big] \\ &\mathrm{ext}_f\, A : a, c, k' \to v' : 0' \\ &(\,\mathrm{ext}_f\, A)(a, c, k') := f(a, c, A(a, c))(k') \end{aligned}$$

The "big $\bigoplus_a$" denotes summation over all values of key attribute $a$; the sum is always finite since we sum over associative tables which have finite support. The ext definition can be seen as un-currying the function given by $f(a, c, A(a, c))$.

We use one shorthand notation. When taking a union with an empty table (one with no support), as in the previous AGG syntax, we adopt a unary version of union written $⧖_\oplus^{\mathrm{x}} A$, where $x$ are the key attributes of the empty table. We also use $\mathrm{map}_f$ for cases of $\mathrm{ext}_f$ that add no new keys.

## 3.3 Properties and Rewrites

*Lifted Properties.*

Some properties from the user-defined functions $\oplus$ and $\otimes$ automatically apply to union and join. If $\oplus$ or $\otimes$ are associative, commutative, or idempotent, then so are $⧖_\oplus$ or $\bowtie_\otimes$ respectfully. These follow directly from the definitions.

*Distributive Laws.*

First we examine the conditions for distributing join over union. If $\otimes$ distributes over $\oplus$ such that $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$, then the same law applies to distribute $\bowtie_\otimes$ over $⧖_\oplus$ such that $A \bowtie_\otimes (B ⧖_\oplus C) = (A \bowtie_\otimes B) ⧖_\oplus (A \bowtie_\otimes C)$ under two conditions: $A$ and $B$ must have no keys in common not also present in $C$, and $A$ and $C$ must have no keys in common not also present in $B$. Put more simply, the distributive law requires $(k_B \Delta k_C) \cap k_A = \emptyset$, where $\Delta$ is symmetric difference.

Next we examine how to push union through join. The following result follows from the Generalized Distributive Law [1]. Assuming that $\otimes$ distributes over $\oplus$,

$$\begin{aligned} &(A \bowtie_\otimes B) ⧖_\oplus C = \\ &\qquad ((⧖_\oplus^{\mathrm{k_B \cup k_C}} A) \bowtie_\otimes (⧖_\oplus^{\mathrm{k_A \cup k_C}} B)) ⧖_\oplus (⧖_\oplus^{\mathrm{k_A \cup k_B}} C) \end{aligned}$$

We refer the reader to a previous technical report for proof of the above two laws [19].

*Matrix Equations.*

To illustrate the ability to reason about non-trivial equivalences in LARA, consider the rotation invariance of matrix multiplication inside a matrix trace: $\mathrm{tr}(ABC) = \mathrm{tr}(BCA)$. The trace of a matrix, $\mathrm{tr}(A)$, sums its diagonal entries. We prove the equation's LARA analogue on tables

$$A : i, j \to v : 0 \qquad B : j, k \to v : 0 \qquad C : k, l \to v : 0$$

To reduce notation, we use subscript $A_{ij}$ in place of $A(i, j)$.

$$\begin{aligned} &\ \mathrm{tr}(ABC) && \\ =&\ ⧖_+ \mathrm{ext}_{i=l}(ABC) && \text{tr defn.} \\ =&\ ⧖_+ \mathrm{ext}_{i=l}\, ⧖_+^{\mathrm{i,l}}(⧖_+^{\mathrm{i,k}}(A_{ij} \bowtie_\otimes B_{jk}) \bowtie_\otimes C_{kl}) && ABC \text{ defn.} \\ =&\ ⧖_+ ⧖_+^{\mathrm{i,l}} \mathrm{ext}_{i=l}(⧖_+^{\mathrm{i,k}}(A_{ij} \bowtie_\otimes B_{jk}) \bowtie_\otimes C_{kl}) && \text{push ext into } ⧖ \\ =&\ ⧖_+ \mathrm{ext}_{i=l}(⧖_+^{\mathrm{i,k}}(A_{ij} \bowtie_\otimes B_{jk}) \bowtie_\otimes C_{kl}) && \text{combine } ⧖ \\ =&\ ⧖_+ (⧖_+^{\mathrm{i,k}}(A_{ij} \bowtie_\otimes B_{jk}) \bowtie_\otimes C_{ki}) && \text{apply ext} \\ =&\ ⧖_+ ⧖_+^{\mathrm{i,k}}(A_{ij} \bowtie_\otimes B_{jk} \bowtie_\otimes C_{ki}) && \text{distr. } \bowtie \text{ into } ⧖ \\ =&\ ⧖_+ (A_{ij} \bowtie_\otimes B_{jk} \bowtie_\otimes C_{ki}) && \text{combine } ⧖ \\ =&\ ⧖_+ (B_{jk} \bowtie_\otimes C_{ki} \bowtie_\otimes A_{ij}) && \text{commute } \bowtie_\otimes \\ =&\ \ldots\ \textit{// reversing the above steps} && \\ =&\ \mathrm{tr}(BCA) && \end{aligned}$$

Due to space considerations, we do not include additional proofs of this form, but we have also sketched proofs of all of the simple rules considered in the context of SystemML [12], including $\mathrm{tr}(AB) = \mathrm{sum}(A \otimes B^\intercal)$, $\mathrm{sum}(\lambda \otimes A) = \lambda \otimes \mathrm{sum}(A)$, $\mathrm{sum}(A + B) = \mathrm{sum}(A) + \mathrm{sum}(B)$, and others.

## 3.4 Relationship to RA and LA

Figure 4 summarizes how each RA and LA operator can be written as a LARA expression.

First we examine RA in Figure 4(a). Selection ($\sigma$) by a predicate $p$ is a map that sends tuples failing $p$ to the default value. Projecting away value attributes ($\pi$) is also a map; projecting away keys is treated as an aggregation. Aggregation ($\gamma$) and relational union ($⧖$) are both instances of LARA union. Relational natural join ($\bowtie$) and Cartesian product ($\times$) are LARA joins after ensuring that the join attributes are keys that match in name. General $\theta$-joins can be modeled via $\sigma_\theta(A \times B)$.

| RA | LARA |
|---|---|
| $\sigma_p$ | $\text{map}_p$ |
| $\pi$ | map or $\text{⧖}$ |
| $\times, \bowtie$ | $\bowtie$ |
| $\gamma, \cup$ | $\text{⧖}_{\oplus}$ |

(a) RA to LARA

| LA | LARA |
|---|---|
| $A \oplus.\otimes B$ | $\text{⧖}^{\mathrm{i,k}}_{\oplus}(A \bowtie_{\otimes} B)$ |
| $A \otimes C$ | $A \bowtie_{\otimes} C$ |
| $A \oplus C$ | $A \,\text{⧖}_{\oplus} C$ |
| $\text{reduce}(A, \oplus)$ | $\text{⧖}^{\mathrm{i}}_{\oplus} A$ |
| $A(I, J)$ | $A \bowtie I \bowtie J$ |
| $f(A)$ | $\text{map}_f$ |
| $A^{\intercal}$ | rename |

(b) LA to LARA

Figure 4: Translation of RA and LA expressions to LARA, given tables $A : i,j \to v,\ B : j,k \to v,\ C : i,j \to v$.

Second we examine LA in Figure 4(b). We chose representative operations for LA based on the emerging GraphBLAS standard [22]. Matrix multiply ($\oplus.\otimes$) is a LARA join and union, after ensuring that the correct dimension of the two matrices match in name. Element-wise multiply ($\otimes$) and addition ($\oplus$) are a join and union. Matrix reduction is a union. Matrix sub-referencing by sets of indices ($A(I, J)$) is a join of $A$ with each set $I$ and $J$, treating the sets as indicator vectors with value 1 for each present position and default 0 otherwise. Function application ($f(A)$) is a map. Transpose ($A^{\intercal}$) is a rename.

The matrix sub-reference translation highlights an interesting property: joining a matrix to a vector $A \bowtie_{\otimes} v$ *expands* $v$ to the shape of $A$ and multiplies them together. Dually, the union $A \,\text{⧖}_{\oplus} v$ *reduces* $A$ to the shape of $v$ and sums them together. In LA one must manually adjust shapes before these operations. LARA adjusts them automatically.

RA and LA have more advanced operators we do not cover here, including outer join, difference, division, pivot, masks, and convolution. These too are expressible in LARA [19].

## 4. PHYSICAL ALGEBRA

In this section we extend LARA to a physical algebra atop an abstraction of *partitioned sorted maps.* These maps are a model for many implementations, including matrix systems (e.g., those that support CSR, CSC, and DCSC [6] storage), relational systems (e.g., row and column stores), and NoSQL systems (e.g., BigTable-style [9] key-value databases).

We call the new physical algebra PLARA. We derive it from LARA in three steps. First, we augment the associative table by imposing an order on their key attributes called an *access path.* Second, we extend the three LARA operators with semantics for associative tables with access paths. Third, we add the operators LOAD and **SORT**.

After defining PLara, we show how PLara admits a number of RA and LA-style optimization opportunities in the context of the sensor example, and we describe an implementation of PLara on the Accumulo database.

### 4.1 PLara Defined

A *Sorted Associative Table* is an associative table with an order imposed on its key attributes. We refer to the ordering as an *access path.* For example, the access path of Table $A$ in Figure 1 is $[t, c]$. Its type is written $A : [t, c] \to v : \perp$.

We model the map's backing store as a row-wise layout of $A$'s tuples sorted by access path and partitioned into segments that can be independently processed or stored. "Split points" that delineate partitions are chosen by the implementation, usually in as equal sizes as possible to avoid skew.

The above scheme performs horizontal partitioning. Vertical partitioning can be achieved by separate associative tables; storing $n$ value attributes separately is equivalent to manipulating $A_1 \,\text{⧖}\, A_2 \,\text{⧖} \ldots \text{⧖}\, A_n$, where each $A_i$ is a one-attribute table. More sophisticated 2-D and higher schemes could be designed but fall outside this paper's scope.

#### *Sort-on-Write.*

Writing out an associative table according to an access path sorts and partitions its data as a side effect. This mechanism is natural for many database implementations, where inserts automatically sort and partition on a clustered index (SQL) or by keys (NoSQL). A chief goal of an optimizer is to minimize the number of sort/write operations. We typeset

$$\textbf{SORT}\ A\ \text{TO}\ [\bar{k}]$$

in bold to highlight the performance impact of re-sorting.

#### *Sorted Join, Union, Ext.*

We assume a single primitive for reading data at the physical level: an efficient *range scan* over the keys of a partitioned sorted map. Range scans are often implemented as *range iterators* that execute user-defined code, including filters, transforms, and aggregations, on streams of data.

In previous work we have shown how to re-purpose range iterators, normally designed for single-table parallel scans, to multi-table computation [20, 21]. This approach enables us to implement the LARA operators inside range iterators.

Join and union take the form of *merge-scans*: range scans on one table that themselves scan matching entries from another. Processing tables in this way is efficient when both tables are sorted on the attributes to be merged; if not, one must re-sort the input tables prior to the merge-scan.

Specifically we implement JOIN, UNION, and AGG as

$$\begin{gathered}\text{MERGEJOIN}\ A, B\ \text{BY}\ \bar{\otimes} \\ \text{MERGEUNION}\ A, B\ \text{BY}\ \bar{\oplus} \\ \text{MERGEAGG}\ A\ \text{ON}\ [\bar{k}]\ \text{BY}\ \bar{\oplus}\end{gathered}$$

EXT $A$ BY $f$ maintains the same syntax in PLARA.

The access path of each operation is as follows. Assume $A$ has access path $[c, a]$, and $B$ has access path $[c, b]$. MERGEJOIN has access path $[c, a, b]$. MERGEUNION has access path $[c]$. MERGEAGG has access path $[\bar{k}]$. If $f$ produces tables sorted on $[k']$, then EXT $A$ BY $f$ has access path $[c, a, k']$.

The behavior of the these operators is as follows. MERGEJOIN $A, B$ takes the Cartesian product of tuples that match on their common keys. For each match, it streams through $B$'s matching tuples while holding $A$'s matching tuples in memory, and it applies an $\otimes$ function to each pair. MERGEUNION aggregates tuples by an $\oplus$ for each common key.

The execution of MERGEUNION depends on the properties of the $\oplus$ function. At a minimum, $\oplus$ must have an identity 0 matching the default values of its input tables, or else correctness is not guaranteed. A basic execution strategy folds $\oplus$ across matching tuples in order on a single partition. If $\oplus$ is associative, then $\oplus$ may run across multiple partitions in parallel, computing local sums before combining them into a global sum during the next **SORT** (see optimization (A) in the next section). If $\oplus$ is idempotent, then $\oplus$ can run more than once on the same tuples, which is helpful for guaranteeing correctness when recovering from server failure. If $\oplus$ is commutative, then $\oplus$ can run out of order.

| Lara Physical Plan | Access Path | Optimizations |
|---|---|---|
| 1 $A =$ Load ‘s1’ | $[t, c]$ | (E) Encode numeric attributes in packed byte form |
| 2 $A_1 =$ Map $A$ by $[v\colon \text{if}(460 \le t \le 860)\ v \text{ else } \bot]$ | $[t, c]$ | (F) Push filter into Load ‘s1’ from 460 to 860 |
| 3 $A_2 =$ Ext $A_1$ by $[t'\colon \text{bin}(t) \to v\colon v, cnt\colon v \neq \bot]$ | $[t, c, t']$ | (M) Eliminate **Sort** by $t'$: bin$(t)$ monotone in $t$ |
| 3.5 $A_{20} =$ **Sort** $A_2$ to $[t', c, t]$ | $[t', c, t]$ | |
| 4 $A_3 =$ MergeAgg $A_{20}$ on $t', c$ by $[v\colon +,\ cnt\colon +]$ | $[t', c]$ | |
| 5 $A' =$ Map $A_3$ by $[v\colon v/cnt]$ | $[t', c]$ | |
| 6 $B' = \ldots$ *// repeat above for second sensor* | $[t', c]$ | |
| 7 $X =$ MergeJoin $A', B'$ by $[v\colon -]$ | $[t', c]$ | (P) Propagate $A, B$’s partition splits throughout |
| 8 $X_1 =$ Map $X$ by $[v\colon v \neq \bot]$ | $[t', c]$ | |
| 9 $X_2 =$ MergeAgg $X_1$ on $t'$ by $[v\colon \text{any}]$ | $[t']$ | |
| 10 $N =$ Agg $X_2$ by $[v\colon +]$ | $[]$ | (C) Store scalar $N$ at client instead of a table |
| 10.5 $X_0 =$ **Sort** $X$ to $[c, t']$ | $[c, t']$ | (Z) If $M, C$ relaxed to *sparse* matrix interpretation, |
| 11 $X_3 =$ Map $X_0$ by $[v\colon v, cnt\colon v \neq \bot]$ | $[c, t']$ | identify $\bot$ with 0, discarding 0-valued entries |
| 12 $X_4 =$ MergeAgg $X_3$ on $c$ by $[v\colon +,\ cnt\colon +]$ | $[c]$ | in $X_3$ and all following tables |
| 13 $M =$ Map $X_4$ by $[v\colon v/cnt)]$ | $[c]$ | (D) Defer $X_3, X_4, M$ to future scans on $X_0$, |
| 13.5 Store $M$ | $[c]$ | eliminating write-out of $M$ |
| 14 $U =$ MergeJoin $X_0, M$ by $[v\colon -]$ | $[c, t']$ | (R) Reuse $X_0$ data source (common sub-expression) |
| 14.5 $U_0 =$ **Sort** $U$ to $[t', c]$ | $[t', c]$ | ($U_2$ has a similar sub-expression below) |
| 15 $U_1 =$ Rename $U_0$ from $c$ to $c'$ | $[t', c']$ | (S) $U^\intercal U$ is symmetric; only compute upper triangle |
| 16 $U_2 =$ MergeJoin $U_0, U_1$ by $[v\colon \times]$ | $[t', c, c']$ | via Map filter $c \le c'$ |
| 16.5 $U_{20} =$ **Sort** $U_2$ to $[c, c', t']$ | $[c, c', t']$ | (A) Push sum of partial products into $U_{20}$ compaction and |
| 17 $U_3 =$ MergeAgg $U_{20}$ on $c, c'$ by $[v\colon +]$ | $[c, c']$ | flush; assume no repeated writes due to server failure |
| 18 $C =$ MergeJoin $U_3, N$ by $[v\colon v/(v'-1)]$ | $[c, c']$ | (D) Defer $U_3, C$ to future scans on $U_{20}$, |
| 18.5 Store $C$ | $[c, c']$ | eliminating final pass |

Figure 5: PLara physical plan for Figure 2’s example, with shading and line numbers matching the logical plan. Each line is annotated with its access path. Optimization opportunities are listed at the right; their effect is quantified in Figure 7.

Ext $A$ by $f$ applies $f$ to each tuple, producing a nested table for each tuple which is immediately flattened. Map is similar to Ext, but never needs to flatten.

Load initiates a range scan on an existing table, possibly restricted to a sub-range. Its access path is given by a database catalog. We also include a Store operator, implemented as a **Sort** that does not change the access path.

### *Physical plan for sensor example.*

Figure 5 presents a line-by-line translation of the Lara logical plan from Figure 2 into a PLara physical plan. The bulk of the translation is tracking the access path induced by each Lara operator and inserting a **Sort** where access path requirements are unmet. This occurs for table $A_3$, which aggregates on $t'$ and $c$ but follows an Ext with access path $[t, c, t']$; table $X_4$, which aggregates on $c$ but stems from $X$ with access path $[t', c]$; table $U_2$, which joins on $t'$ but stems from $U$ with access path $[c, t']$; and table $U_3$, which aggregates on $c$ and $c'$ but stems from $U_2$ with path $[t', c, c']$.

## 4.2 Physical Optimizations

Figure 6 illustrates a few optimization rules on the PLara algebra; Figure 5 pinpoints where these and other rules apply to our running sensor example. We evaluate the impact of these optimizations in Section 5.1.

Some of the most important optimizations act on **Sort** operations. Rule (A) pushes aggregations that run after a **Sort** into the **Sort** itself. We call the fused operation **SortAgg**. Speedup from fusing aggregation into sorting can be dramatic, since the implementation can compute partial sums which reduces the burden of sorting and storage.

Rule (M) eliminates **Sort**s after an Ext when they are unnecessary. Normally additional key attributes produced by an Ext append to the end of its input’s access path. Moving new attributes up in the access path past existing key attributes requires re-sorting. If $f$ is *monotone* with respect to existing key attributes, however, the new key attributes may be promoted past those existing ones without sorting. Here, monotone means that $k_1 \le k_2 \Rightarrow f(k_1) \le f(k_2)$, using $f(k)$ to refer to the keys of the tables produced by $f$.

Rule (F) pushes filter operations into the Load statements that start a range scan. These filter operations restrict data to a range of keys on a prefix of the loaded table’s access path. The result restricts scanned data to the desired range, as opposed to naively scanning all data and post-filtering.

Rules (Z-Sort), (Z-Map), (Z-Agg), and (Z-Join) push “discarding zeros” through a Lara expression. These rules generalize to Ext and Union; in fact, they apply at the logical Lara level, but we list them here since they are usually associated with physical storage. The null-to-zero function—ntz$(v) = $ if $(v = \bot)$ 0 else $v$—changes $v$’s default value from $\bot$ to 0. The change allows implementations to discard zeros without fear of incorrectness.

In order to apply the (Z) rewrites, the inference rules check that the function to push ntz through treats $\bot$ and 0 “the same”. For example, read the first as “if we Ext $A$ by $f$ and afterward discard zeros, and it holds that $f(\bot) = \bot$ and $f(0) = 0$, then we can safely discard zeros before the Ext”.

We now discuss a few rules not listed in Figure 6. Rule (S) leverages symmetry of the inner product computed in lines 15–17: $(U^\intercal U)^\intercal = U^\intercal U$. Lara expresses this identity as a Rename. If $C$ is relaxed to restrict its output to its upper triangle, then (S) could push the filter “Map $C$ by $c \le c'$” up through the plan to the point immediately after line 16 by means of rules in the same style as the (Z) rules.

Rule (D) defers the last pass before a Store to scans on the last materialized table, i.e., the last **Sort** result. This rule partitions the plan into parts computed “eagerly” and

$$\text{(A)}\ \frac{\textsc{MergeAgg}\ (\textbf{Sort}\ A\ \textsc{to}\ [k])\ \textsc{on}\ k'\ \textsc{by}\ \oplus}{\textbf{SortAgg}\ A\ \textsc{to}\ [k']\ \textsc{by}\ \oplus} \qquad \text{(F)}\ \frac{\textsc{Load}\ \text{`x'} : [k] \to v : 0 \qquad \textsc{Map}\ (\textsc{Load}\ \text{`x'})\ \textsc{by}\ [v\colon \text{if } (a \le v \le b)\ v \text{ else } 0])}{\textsc{Load}\ \text{`x'}\ \textsc{from}\ a\ \textsc{to}\ b}$$

$$\text{(M)}\ \frac{A : k \to v \qquad f(k,v) : k' \to v' \qquad f \text{ monotone in } k \qquad \textbf{Sort}\ (\textsc{Ext}\ A\ \textsc{by}\ f)\ \textsc{to}\ [k',k]}{\textsc{Ext}\ A\ \textsc{by}\ f\ \textsc{over}\ [k',k]}$$

$$\text{(Z-Sort)}\ \frac{\textsc{Map}\ (\textbf{Sort}\ A\ \textsc{to}\ [k])\ \textsc{by}\ [v\colon \text{ntz}(v)]}{\textbf{Sort}\ (\textsc{Map}\ A\ \textsc{by}\ [v\colon \text{ntz}(v)])\ \textsc{to}\ [k]} \qquad \text{(Z-Map)}\ \frac{\textsc{Map}\ (\textsc{Map}\ A\ \textsc{by}\ [v\colon f(v)])\ \textsc{by}\ [v\colon \text{ntz}(v)] \qquad f(\bot) = \bot,\ f(0) = 0}{\textsc{Map}\ (\textsc{Map}\ A\ \textsc{by}\ [v\colon \text{ntz}(v)])\ \textsc{by}\ [v\colon f(v)]}$$

$$\text{(Z-Agg)}\ \frac{\textsc{Map}\ (\textsc{Agg}\ A\ \textsc{on}\ [k]\ \textsc{by}\ [v\colon \oplus])\ \textsc{by}\ [v\colon \text{ntz}(v)] \qquad \bot \oplus v = v \qquad v \ne \bot \Rightarrow 0 \oplus v = v}{\textsc{Agg}\ (\textsc{Map}\ A\ \textsc{by}\ [v\colon \text{ntz}(v)])\ \textsc{on}\ [k]\ \textsc{by}\ [v\colon \oplus]}$$

$$\text{(Z-Join)}\ \frac{\textsc{Map}\ (\textsc{Join}\ A, B\ \textsc{by}\ [v\colon \otimes])\ \textsc{by}\ [v\colon \text{ntz}(v)] \qquad \bot \otimes v = \bot \qquad v \ne \bot \Rightarrow 0 \otimes v = 0}{\textsc{Join}\ (\textsc{Map}\ A\ \textsc{by}\ [v\colon \text{ntz}(v)]), (\textsc{Map}\ B\ \textsc{by}\ [v\colon \text{ntz}(v)])\ \textsc{by}\ [v\colon \otimes]}$$

Figure 6: A sample of rewrite rules. Rule (A) pushes MergeAggs into **Sort**; (F) pushes filters into Load; (M) eliminates **Sort** after a monotonic Ext; the (Z-) rules push discarding zeros. The "null-to-zero" function is $\text{ntz}(v) = \text{if } (v = \bot)\ 0 \text{ else } v$.

parts computed "lazily". The performance impact to future scans of deferring the last computation is usually minimal, since **Sort**s are never deferred and so the deferred operations can be streamed. In Figure 5, lines 11–13 defer to scans on $X_0$, and lines 17–18 defer to scans on $U_{20}$.

Rule (E) encodes numbers in a packed byte format. Like (Z) and (S), (E) involves a change in the output that, if allowed, can be pushed through the computation, in this case all the way to the original data sources $A$ and $B$.

Rule (R) is a form of common sub-expression elimination. In the Accumulo implementation, (R) entails re-using a single range iterator to serve two separate data streams.

Rule (P) acts on the table splits that partition data. It pre-splits new tables using the splits of existing tables. Presplitting tables improves insert performance by increasing parallelism before the implementation splits data on its own.

Additional optimizations are possible. For instance, we might forgo sorting in favor of hash-shuffling when correct to do so, just as Tenzing employed [10].

### 4.3 Accumulo Implementation of LaraDB

We implemented PLara on the architecture of Google's BigTable [9], a design that closely resembles PLara's sorted partitioned map abstraction. Operations in the BigTable architecture consist of inserts and range scans. During scans, the user can execute arbitrary code in the form of *iterators* that run server-side as data streams from each partition in parallel. Iterator code can even initiate scans on or write entries to additional tables, a fact we previously exploited in the Graphulo matrix math library [20, 21].

BigTable's range iterators suffice to implement Lara. In particular, we implemented PLara on Apache Accumulo, an open-source adaptation of BigTable's design. However, we emphasize that our implementation applies just as much to other BigTable systems, including Hypertable and Apache HBase, and that we see no fundamental barriers to implementing Lara atop other systems with some concept of key and value, including relational and matrix systems. Even nested relational systems for JSON-like data fit into Lara, either by flattening or new indexing techniques [30].

For this prototype implementation, we chose a simple model that stores the first key, subsequent keys, and values in the Accumulo row, column qualifier, and value, respectively. Keys are stored (and sorted) according to the table's access path. We coded Ext, MergeJoin, and MergeUnion as iterator fragments linked by the Graphulo library.

## 5. EXPERIMENTS

### 5.1 Sensor Optimization Experiment

In this section we conduct an experiment with two goals: to assess Lara's ability to express a complex computation with elements of both RA and LA, and to measure the impact of optimizations that Lara affords on this computation. We implemented each optimization manually; building an optimizer that applies them automatically is future work.

The experimental task is the sensor quality control plan detailed in Figure 5. We obtained 1.5 months' data from two "Array of Things" sensors [3] managed by Argonne National Laboratory. The raw data amounts to 1.2 GB; however, this reduces to about 60 MB after parsing, projecting, and storing the data in Accumulo's default compressed format. We partitioned each sensor's data into 3 day segments. The plan's filter step restricts analysis to a 30 day period.

We experimented on an Amazon EC2 `m3.large` cluster of 4 workers, 3 coordinators, and 1 monitor machine. Each has 7.5 GB memory, 2 vCPUs, and a 30 GB SSD drive.

Figure 6 plots sensor task runtime with different optimizations enabled. At the left we plot the baseline, no optimizations, at 1230 seconds. We then plot each optimization individually as well as the combined effect of all optimizations.

Most of the runtime is spent calculating the covariance $C$. This matches our expectations because computing the inner product $U^{\intercal}U$ generates a large number of partial products. For this reason, optimization (A) yields the greatest performance increase, since it drastically increases the efficiency of summing partial products. Without (A), all partial products must be materialized before they can be summed.

Optimizations (D) and (S) both affect the $C$ calculation and deliver the next best performance improvement. (S) eliminates half the computation to compute $C$, and (D) defers finishing the summation to future scans.

Other optimizations proved effective but had less impact since they applied less to the covariance bottleneck. The impact of (Z) depends on the number of zero-valued entries materialized during the $U$ and $C$ computation. (P) increased parallelism in each step, somewhat reducing worker skew.

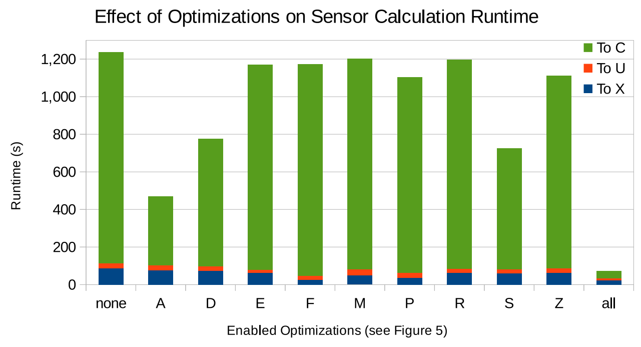

Figure 7: Runtime of Figure 5's PLARA plan with different optimizations enabled on one month's data from two sensors. We decompose runtimes for Figure 5 into the portion scanning $A$ and $B$ to calculating $X$, from $X$ to $U$, and from $U$ to $C$; the $C$ calculation dominates runtime.

(F) sped up the first phase by 4x, decreasing its runtime from 87 to 22 seconds. (E) and (M) had smaller effects.

We conclude that LARA and PLARA are sufficient to express the sensor quality control computation, as well as several optimizations useful for impacting performance.

## 5.2 Competitiveness Experiment

In this section we conduct an experiment to test whether LARADB competes in performance with the analytics engine natively integrated with Accumulo: MapReduce.

The task we run is matrix multiplication (MxM). In terms of RA, MxM consists of a join followed by an aggregation. In terms of LA, many other LA kernels can be simulated by MxM. For example, matrix reduction can be realized as multiplication by a vector of 1s, and matrix subset can be realized as multiplication on the left by a diagonal matrix that selects rows and on the right by a diagonal matrix that selects columns. Composition of these kernels lead to more complex graph algorithms such as triangle enumeration [33], vertex similarity, k-truss, and matrix factorization [15].

Because our goal is to compare the performance of the LARADB and the MapReduce execution engines, rather than the difference between two MxM algorithms, we wrote the LARADB and MapReduce code implementing MxM as similarly as possible. Both read inputs from and write outputs to Accumulo tables. Both implement the the MxM $C = AB$ outer product algorithm [20] on pre-indexed data with $A$ sorted column-major and $B$ sorted row-major. Both have optimizations (A) and (D) from Section 4.2 enabled.

The main operational difference between the LARADB and MapReduce execution is that LARADB executes inside Accumulo's range scan iterators while MapReduce executes as external processes managed by the YARN scheduler. Specifically, MapReduce performs a reduce-side join [13].

We generated test data via the Graph500 unpermuted power law graph generator [4]. We chose the generator because power law distributions well model properties of real world data such as skew [17]. Generated matrices range from $2^{10}$ rows (scale 10) to $2^{19}$ rows (scale 19), each with roughly 16 nonzero entries per row. Multiplying the largest matrices formed close to $2^{33}$ $(= 8 \times 10^9)$ partial products.

We used the same Amazon EC2 experiment environment as Section 5.1, except with 8 workers instead of 4. Each worker allocated 3 GB of memory to YARN and 3 GB to Accumulo. The 8-worker environment is well-suited to gauging inter-node parallelism; intra-node parallelism, however, was limited by the small number of vCPUs (2) per machine.

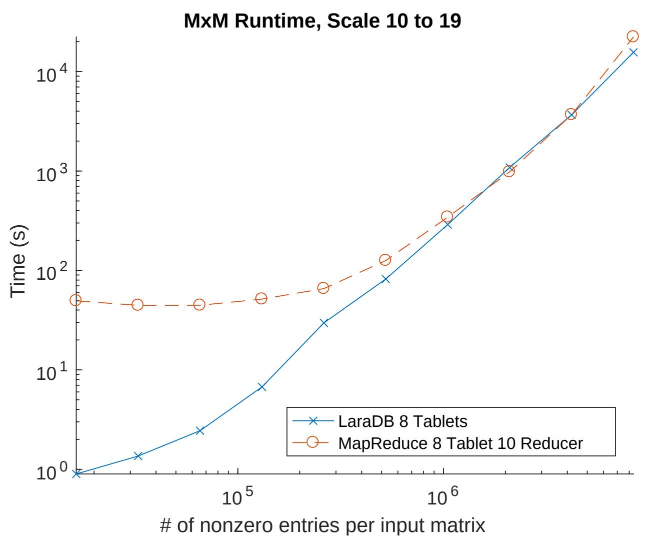

Figure 8: 8-worker $A^{\intercal}B$ experiment runtime as problem size increases. LARADB dominates at smaller sizes, while LARADB and MapReduce converge at larger sizes.

Figure 8 plots MxM runtime as problem size increases. Graphulo dominates MapReduce at smaller problem sizes. This is due to the large startup cost that MapReduce programs are infamous for; the YARN scheduler takes roughly 30s to start any task as a result of job submission, container allocation, jar copying, and other cold start overheads.

LARADB, on the other hand, has a warm start since it runs inside the already-running Accumulo tablet servers. These tablet servers have a standing thread pool ready to service scan requests as soon as they receive a remote procedure call. We conclude that LARADB is much better suited to interactive and small-scale computation, such as analytics on a subset of data extracted from an Accumulo table.

At larger problem sizes, LARADB and MapReduce converge in performance. The convergence meets our expectations because the two libraries run similar code in a similar pattern of parallelism over the same data partitioning. Their execution environment, JVMs over Hadoop, is also similar given sufficient time to amortize YARN's startup cost.

We conclude that our LARADB implementation is competitive with at least one major RA/LA system at scale. We take this as initial evidence that systems built atop the LARA algebra can and do have strong performance.

## 6. CONCLUSION

Linear algebra (LA) and relational algebra (RA) are, in a sense, two sides of the same coin. We offer LARA as that coin, expressive enough to subsume LA and RA yet with more structure than MapReduce that in turn affords greater reasoning. Lowering LARA to a physical algebra brings this reasoning to the domain of partitioned sorted maps, a broad abstraction that encompasses LA, RA, and key-value systems including the LARADB implementation on Accumulo.

Our experiments demonstrate that (1) LARA expresses high and low-level optimizations that make a difference in the execution of real-world tasks, and (2) that the LARADB implementation outperforms an existing data processing system vastly at small scale and competitively at large scale.

In the future, we aim to use LARA as a conduit for studying and computationally exploiting the relationship between LA and RA. A database optimizer is an ideal place to realize the benefits of this study for joint linear-relational analytics.

## Acknowledgments

This material is partially supported by NSF Graduate Research Fellowship DGE-1256082. Thanks to David Maier, Jeremy Kepner, and Tim Mattson for their enthusiasm and comments.

## 7. REFERENCES

[1] S. M. Aji and R. McEliece. The generalized distributive law. *Transactions on Information Theory*, 46(2):325–343, 2000.

[2] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, et al. The stratosphere platform for big data analytics. *The VLDB Journal*, 23(6):939–964, 2014.

[3] Array of things. https://arrayofthings.github.io/.

[4] D. Bader, K. Madduri, J. Gilbert, V. Shah, J. Kepner, T. Meuse, and A. Krishnamurthy. Designing scalable synthetic compact applications for benchmarking high productivity computing systems. *Cyberinfrastructure Technology Watch*, 2:1–10, 2006.

[5] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. The HaLoop approach to large-scale iterative data analysis. *VLDB Journal*, 21(2):169–190, 2012.

[6] A. Buluc and J. R. Gilbert. On the representation and multiplication of hypersparse matrices. In *International Symposium on Parallel and Distributed Processing (IPDPS)*. IEEE, 2008.

[7] P. Buneman, S. Naqvi, V. Tannen, and L. Wong. Principles of programming with complex objects and collection types. *Theoretical Computer Science*, 149(1):3–48, 1995.

[8] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment*, 1(2), 2008.

[9] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[10] B. Chattopadhyay, L. Lin, W. Liu, S. Mittal, P. Aragonda, V. Lychagina, Y. Kwon, and M. Wong. Tenzing a SQL implementation on the mapreduce framework. *PVLDB*, 4:1318–1327, 2011.

[11] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, U. Cetintemel, and S. B. Zdonik. Tupleware: "big" data, big analytics, small clusters. In *Conference on Innovative Data Systems Research (CIDR)*, 2015.

[12] T. Elgamal, S. Luo, M. Boehm, A. V. Evfimievski, S. Tatikonda, B. Reinwald, and P. Sen. Spoof: Sum-product optimization and operator fusion for large-scale machine learning. In *Conference on Innovative Data Systems Research (CIDR)*, Jan. 2017.

[13] L. Fegaras, C. Li, and U. Gupta. An optimization framework for map-reduce queries. In *EDBT*. ACM, 2012.

[14] Apache flume. https://flume.apache.org/.

[15] V. Gadepally, J. Bolewski, D. Hook, D. Hutchison, B. Miller, and J. Kepner. Graphulo: Linear algebra graph kernels for NoSQL databases. In *International Parallel & Distributed Processing Symposium Workshops*. IEEE, 2015.

[16] V. Gadepally and J. Kepner. Big data dimensional analysis. In *High Performance Extreme Computing Conference (HPEC)*. IEEE, 2014.

[17] V. Gadepally and J. Kepner. Using a power law distribution to describe big data. In *High Performance Extreme Computing Conference (HPEC)*. IEEE, 2015.

[18] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, et al. The madlib analytics library: or mad skills, the sql. *Proceedings of the VLDB Endowment*, 5(12):1700–1711, 2012.

[19] D. Hutchison, B. Howe, and D. Suciu. Lara: A key-value algebra underlying arrays and relations. *arXiv preprint arXiv:1604.03607*, 2016.

[20] D. Hutchison, J. Kepner, V. Gadepally, and A. Fuchs. Graphulo implementation of server-side sparse matrix multiply in the Accumulo database. In *High Performance Extreme Computing Conference (HPEC)*. IEEE, 9 2015.

[21] D. Hutchison, J. Kepner, V. Gadepally, and B. Howe. From NoSQL Accumulo to NewSQL Graphulo: Design and utility of graph algorithms inside a BigTable database. In *High Performance Extreme Computing Conference (HPEC)*. IEEE, 9 2016.

[22] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, S. McMillan, J. Moreira, J. D. Owens, C. Yang, M. Zalewski, and T. Mattson. Mathematical foundations of the GraphBLAS. In *HPEC*. IEEE, 9 2016.

[23] J. Kepner, V. Gadepally, D. Hutchison, H. Jananthan, T. Mattson, S. Samsi, and A. Reuther. Associative array model of SQL, NoSQL, and NewSQL databases. In *High Performance Extreme Computing Conference (HPEC)*. IEEE, 9 2016.

[24] A. Kunft, A. Alexandrov, A. Katsifodimos, and V. Markl. Bridging the gap: towards optimization across linear and relational algebra. In *Proceedings of the 3rd SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond*. ACM, 2016.

[25] R. Maas, J. Hyrkas, O. G. Telford, M. Balazinska, A. Connolly, and B. Howe. Gaussian mixture models use-case: in-memory analysis with myria. In *Proceedings of the 3rd VLDB Workshop on In-Memory Data Mangement and Analytics*. ACM, 2015.

[26] B. Marker, M. Schatz, D. Matthews, I. Dillig, R. van de Geijn, and D. Batory. Dxter: An extensible tool for optimal dataflow program generation. Technical report, Technical Report TR-15-03, The University of Texas at Austin, 2015.

[27] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, et al. Mllib: Machine learning in apache spark. *Journal of Machine Learning Research*, 17(34):1–7, 2016.

[28] S. Palkar, J. J. Thomas, A. Shanbhag, D. Narayanan, H. Pirk, M. Schwarzkopf, S. Amarasinghe, M. Zaharia, and S. InfoLab. Weld: A common runtime for high performance data analytics. In *Conference on Innovative Data Systems Research (CIDR)*, Jan. 2017.

[29] A. Rheinländer, A. Heise, F. Hueske, U. Leser, and F. Naumann. SOFA: An extensible logical optimizer for udf-heavy data flows. *Information Systems*, 52, 2015.

[30] D. Shukla, S. Thota, K. Raman, M. Gajendran, A. Shah, S. Ziuzin, K. Sundaram, M. G. Guajardo, A. Wawrzyniak, S. Boshra, R. Ferreira, M. Nassar, M. Koltachev, J. Huang, S. Sengupta, J. J. Levandoski, and D. B. Lomet. Schema-agnostic indexing with azure documentdb. *PVLDB*, 8:1668–1679, 2015.

[31] M. Spight and V. Tropashko. First steps in relational lattice. *arXiv preprint cs/0603044*, 2006.

[32] J. Wang, T. Baker, M. Balazinska, D. Halperin, B. Haynes, B. Howe, D. Hutchison, S. Jain, R. Maas, P. Mehta, et al. The Myria big data management and analytics system and cloud service. Jan. 2017.

[33] M. M. Wolf, J. W. Berry, and D. T. Stark. A task-based linear algebra building blocks approach for scalable graph analytics. In *High Performance Extreme Computing Conference (HPEC)*. IEEE, 2015.

[34] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. *HotCloud*, 10:10–10, 2010.