UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

ON THE EXPRESIVENESS OF LARA: A UNIFIED LANGUAGE FOR LINEAR AND RELATIONAL ALGEBRA

TESIS PARA OPTAR AL GRADO DE MAGÍSTER EN CIENCIAS,
MENCIÓN COMPUTACIÓN

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERIO CIVIL EN COMPUTACIÓN

NELSON NICOLAS HIGUERA RUIZ

PROFESORES GUÍAS:
PABLO BARCELÓ BAEZA
JORGE PÉREZ ROJAS

MIEMBROS DE LA COMISIÓN:
CLAUDIO GUTIERREZ GALLARDO
FEDERICO OLMEDO BERON
CRISTIAN RIVEROS JAEGER

Este trabajo ha sido completamente financiado por el
Instituto Milenio Fundamentos de los Datos (IMFD).

SANTIAGO DE CHILE
2019

Resumen

Estudiamos el poder de expresividad del lenguaje Lara – un álgebra unificador recientemente propuesto para expresar operaciones del álgebra relacional y lineal – ambos en términos de lenguajes de consultas de bases de datos tradicionales y algunas tareas analíticas frecuentemente realizadas en tareas de machine learning.

El modelo de datos de Lara es la tabla asociativa, una estructura similar a una tabla que tiene una cabecera y contiene datos en filas. Está dividida en dos: las llaves y los valores. La tabla puede verse como una función, donde toda llave está asociada a sus valores respectivos. El lenguaje tiene tres operaciones: join, union, and extension, las cuales están parametrizadas por funciones. Las primeras dos son parametrizadas por funciones de agregación (suma, promedio, desviación estándar, etc.) mientras que la última está parametrizada por una función definida por el usuario, la cual puede permitir transformar filas en tablas.

Comenzamos mostrando como Lara es completamente expresivo con respecto a la lógica de primer orden con agregación. Dado que Lara está parametrizado por un conjunto de funciones definidas por el usuario, el poder expresivo exacto del lenguaje depende en cómo estas funciones están definidas. Distinguimos dos casos principales dependiendo del nivel de genericidad al que las consultas están impuestas a satisfacer.

Bajo suposiciones de genericidad fuerte el lenguaje no puede expresar la convolución de matrices, una operación muy importante en las operaciones actuales de machine learning. El lenguaje es también local, y por lo tanto no puede expresar operaciones que exhiben un comportamiento recursivo, como la inversión de matrices.

Para expresar la convolución, podemos relajar el requerimiento de genericidad añadiendo un orden lineal subyacente al dominio. Esto, sin embargo, destruye la localidad y convierte el poder expresivo del lenguaje en algo mucho más difícil de entender. En particular, aunque bajo supuestos de complejidad el lenguaje resultante aún no puede expresar la inversión de matrices, una prueba de este hecho sin estos supuestos parece ser difícil de obtener.

Abstract

We study the expressive power of the LARA language – a recently proposed unifying algebra for expressing relational and linear algebra operations – both in terms of traditional database query languages and some analytic tasks often performed in machine learning pipelines.

The data model of LARA is the associative table, a table-like structure which has a header and contains rows that follows the header. It is divided in two: the keys and the values. The table can be seen as a function, where each key is associated with their respective values. The lenguage has three operations: join, union and extension, where each is parameterized by a function. The first two are parameterized by aggregate functions (sum, average, standard deviation, etc.) while the latter is parameterized by an user-defined function, which can allow to transform rows into tables.

We start by showing LARA to be expressive complete with respect to first-order logic with aggregation. Since LARA is parameterized by a set of user-defined functions, the exact expressive power of the language depends on how these functions are defined. We distinguish two main cases depending on the level of genericity queries are enforced to satisfy.

Under strong genericity assumptions the language cannot express matrix convolution, a very important operation in current machine learning operations. This language is also local, and thus cannot express operations such as matrix inverse that exhibit a recursive behavior.

For expressing convolution, one can relax the genericity requirement by adding an underlying linear order on the domain. This, however, destroys locality and turns the expressive power of the language much more difficult to understand. In particular, although under complexity assumptions the resulting language can still not express matrix inverse, a proof of this fact without such assumptions seems challenging to obtain.

A mi familia, por su perpetuo apoyo.
A mis amigos, por todos los buenos momentos.
Y a la matemática, por iluninar mi camino.

Agradecimientos

El viaje que me ha traído hasta este momento está lleno de altos y bajos pero ya aquí, agradezco las experiencias que me han formado. Ciertamente no podría haber terminado esto de no ser por todas las personas que me han apoyado, enseñado y acompañado en el camino.

Primeramente debo agradecer a mi familia, ya que ellos han sido mi mayor fuente de motivación. A mi madre, que cuida de mi, a mi hermana y a mi abuelo, y que gracias a su fortaleza y ternura me ha hecho ser duro sin perder la compasión. A mi padre, que gracias a su sacrificio y constancia me ha demostrado que siempre es posible salir adelante, por mal que estén las cosas. A mi hermana, que me complementa y hace todo lo que yo hago mal, bien. Esto también aplica al revés. A mi tía y primos, que me han recibido en su hogar por ya siete años nunca exigiendo nada a cambio. Y a mis dos gatos, que me acompañan en la soledad de mi pieza mientras escribo esto.

Luego, debo agradecer a mis profesores de tesis, ambos personas excelentes. No espero retribuir todo lo que me han enseñado en estas pocas líneas, pero al menos espero sientan mi afecto. A Pablo, por su estilo retórico de enseñanza, que me ha hecho volver a tener confianza en lo que escribo y digo. A Jorge, por los múltiples consejos acertados que en su momento no comprendí. A retos y palos me quitaron el miedo y trajeron a mi una nueva oportunidad de seguir haciendo lo que siempre me ha gustado, que es estudiar y aprender. Espero nunca dejen de enseñar, sea donde se lleven a si mismos.

Finalmente, agradezco a mis amigos, quienes han alegrado mi vida incontables veces. Debo partir por mis mś viejos amigos, aquellos que hice en Villarrica. A pesar de la distancia, siempre se mantienen en mi recuerdo: Rayen, Fabian, Zepu, Robi, Coni, Felipe, Raul, Nunivek, Hector, Moises, Renzo, Bastian y Daniel, gracias. Dos de ellos me han motivado en particular a crecer como persona y admiro por su inteligencia, Joaquin y Cristian. Desgraciadamente nuestros caminos han divergido pero espero poder volver a encontrarlos en cualquier momento. Siempre me han tratado bien y recibido con mucho cariño en sus hogares, a pesar de mi lejania. También agradezco a mis amigos de Santiago, quienes han formado parte de mi vida desde que llegue. A mis amigos de plan común, por las intensas sesiones de estudio y posteriores salidas, que hicieron mucho más llevaderos esos años. A aquellos que conocí en la zona D del CEC, con los que jugamos mas de lo que deberíamos. A los que conocí por mis tropiezos vocacionales, por quienes ahora se que uno puede encontrar amigos en lugares insospechados. Y a los que conocí en la carrera de computación, con quienes comparto a diario y me hacen sonreir. En particular destaco a Pablo, Tomas y Benjamin,

con los que he compartido muchas experiencias y lo sigo haciendo. Tambien agradezco a los que compartieron su tiempo conmigo mientras estaba en el magister: Valentina, por su timidez y alegría, y Bernardo, por el trabajo que hicimos y las interesantes conversaciones que tuvimos. Finalmente, agradezco a Angélica, por brindarme su amor en tiempos difíciles.

A todas estas personas y mas, doy mi más sincero agradecimiento y les deseo lo mejor en sus vidas.

Contents

Introduction

Most of the actual analytic systems require relational algebra and statistical operations to handle the data. In particular, relational algebra is used for pre-processing the data (cleaning, structuring, etc.), to then be used as input of statistical operations that allows us to reason quantitatively about the data. A setting of much importance these days, and the one we will be focusing, is the one of machine learning (ML). ML tasks also can be fitted in the schema described above, as pre-processing of the data is required in many applications to then apply the corresponding operations to train ML models. Since the data models of relational algebra and ML tend to be different (relations and tensors), this creates an impedance mismatch [22]. It is worth to say that several of the needed functionalities for ML applications, and in particular, for tensor manipulation can be performed in languages such as SQL. But it is not reasonable to force ML practitioners to adapt to already high-level formalisms python, Matlab or R and migrate all their pipeline to a relational database. Because of this, the database theory community is looking for some way to integrate these two [12]. To do this, what we need is a common data model for relations and tensors and also a language capable of expressing relational algebra and ML operations. Although the latter has received some proposals and wide acceptance, there is an interest in redesigning the way in which tensors are used in deep learning applications, due to problems [18, 32, 33] with the current abstractions.

Hutchinson et al. [20, 19] recently proposed a data model and a language that aims to become the "universal connector" for solving this impedance mismatch. First, the data model proposed corresponds to the *associative tables*, which generalize relational tables, tensors, graphs, and others. Associative tables are two-sorted, consisting of *keys* and *values*, such that a key maps to their corresponding values. Second, the language is called LARA, and subsumes several known languages for the data models mentioned. LARA is algebra designed in a minimalistic way by only including three operators; namely, *join*, *union*, and *extend*, and all of them are parameterized by functions. Roughly, the first one corresponds to some variation of the join from relational algebra, the second one to the operation of aggregation or reduction, and the third one to the extension defined by a function as in a *flatmap* operation. It has been shown that LARA subsumes all relational algebra operations and is capable of expressing several interesting linear algebra operations used in graph algorithms [19].

Given that LARA is a unified language for relational and linear algebra, it is relevant to study its expressive power, both in terms of the query languages traditionally studied in database theory and some important and common ML operations. We start with the former and show that LARA is expressive complete with respect to *first-order logic with aggregation* ($\mathrm{FO}_{\mathsf{Agg}}$), a language that has been deeply studied. This expressive completeness can be seen

as a sanity check for LARA. In fact, this language is specifically tailored to handle aggregation in conjunction with relational algebra operations, and a classical result in database theory establishes that the latter is expressive complete with respect to first-order logic (FO). Our result is parameterized by the set of functions allowed to parameterize the extend operator. For each such a set Ω we allow FO_{Agg} to contain all built-in predicates that encode the functions in Ω. To understand which ML operators LARA can express, one then needs to bound the class Ω of extension functions allowed in the language, as these could be very powerful. We start with a set that can still express several relevant functions. These are the FO-definable functions that allow to compute arbitrary numerical predicates on values, but can only compare keys with respect to equality or inequality. This restriction makes the algebra quite amenable for theoretical exploration. In fact, it is easy to show that this version of LARA which we call "tame" LARA satisfies a strong *genericity* property, which means that the result of every query written in LARA does not depend on the database, and is also *local*, in the sense that queries in the language can only see up to a fixed-radius neighborhood from its free variables; cf., [25]. The first property implies that this tame version of LARA cannot express operations as matrix *convolution*, and the second one that it cannot express inherently recursive queries, such as matrix *inverse*. Both operations are very relevant for ML applications; e.g., matrix convolution is routinely applied in dimension-reduction tasks, while matrix inverse is used for learning the matrix of coefficient values in linear regression. We also study the expressiveness of a sort of "mini language", which is the Einstein summation notation. This notation has multiple applications in ML, as it generalizes operations such as the inner product of vectors, matrix trace, matrix multiplication, tensor product and tensor contraction.

We then look more deeply at the case of matrix convolution, and show that this query can be expressed if we relax the genericity properties of the language by assuming the presence of a linear order on the domain of keys, which means that we now can compare the domain *keys* and *values* by all sorts of inequalities. This relaxation implies that queries expressible in the resulting version of LARA are no longer invariant with respect to key-permutations. This language, however, is much harder to understand in terms of its expressive power. In particular, it can express non-local queries, and hence we cannot apply locality techniques to show that the matrix inversion query is not expressible in it. To prove this result, then, one would have to apply techniques based on the *Ehrenfeucht-Fraïssé* games that characterize the expressive power of the logic. Showing results based on such games in the presence of a linear order, however, is often combinatorially difficult, and currently we do not know whether this is possible. In turn, it is possible to obtain that matrix inversion is not expressible in a natural restriction of our language under complexity-theoretic assumptions. This is because the data complexity of queries expressible in such a restricted language is LOGSPACE, while matrix inversion is complete for a class that is believed to be a proper extension of the latter.

The main objective of our paper is connecting the study of the expressive power of tensor-based query languages, in general, and of LARA, in particular, with traditional database theory concepts and the arsenal of techniques that have been developed in this area to study the expressiveness of query languages. We also aim at identifying potential lines for future research that appear in connection with this problem. Our work is close in spirit to the recent study of MATLANG [6, 15], a matrix-manipulation language based on elementary linear algebra operations. It is shown that this language is contained in the three-variable

fragment of relational algebra with summation and, thus, it is local. This implies that the core of MATLANG cannot check for the presence of a four-clique in a graph (represented as a Boolean matrix), as this query requires at least four variables to be expressed, and neither it can express the non-local matrix inversion query. We show that MATLANG is strictly contained in the tame version of LARA mentioned above, and thus some of our results can be seen as generalizations of the ones for MATLANG.

Outline and Main Contributions

Chapter 1: We introduce the context of ML and databases and the study of expressiveness. For that we present the standard operations in ML and databases, numerous systems arranged in families, relational and array languages, our study object, LARA, and a brief introduction to the study of expressiveness presenting related work.

Chapter 2: We formally introduce *associative tables* and the language LARA by presenting its syntax and semantics. We then present the base version of FO, to then expand it to be two-sorted and be able to deal with aggregation, which we call $\mathrm{FO}_{\mathsf{Agg}}$.

Chapter 3: We show that "tame" LARA is equally expressive to $\mathrm{FO}_{\mathsf{Agg}}$ when both languages are restricted to the same functions that parameterize the *extend* operator. As a corollary, when these functions are those $\mathrm{FO}_{\mathsf{Agg}}$-definable, the version of LARA becomes strongly generic and local, as $\mathrm{FO}_{\mathsf{Agg}}$ is.

Chapter 4: We study the expressiveness of common ML operations in "tame" LARA. These are the matrix convolution, the matrix inverse, and the Einstein summation notation.

Chapter 5: We allow *keys* and *values* by inequalities in LARA and study the expressiveness of this language, which is more powerful than the "tame" version as it can express non-local queries as the transitive closure.

Conclusion and Future Work: We draw our conclusions and further lines of research that could be explored based on the results obtained in this work.

Publications

Preliminary parts of this work were accepted in the 3rd Workshop on Data Management for End-to-End Machine Learning (DEEM 2019), the 13th Alberto Mendelzon International Workshop on Foundations of Data Management (AMW 2019) and the 23rd International Conference on Database Theory (ICDT 2020).

Chapter 1

Context

The goal of this chapter is to provide the necessary background to understand why studying and comprehending the expressiveness of algebras is useful for databases and in our particular case, machine learning. We start by presenting the current relation of machine learning and databases, and by that showing the impedance mismatch described in the previous chapter. We then go deeper into the database families to reach a new kind, which is the polystore system. After seeing all the practical aspects, we move on to our main interest, which are the languages that allow us to manage data. Concluding this section we briefly present the Lara language we are studying. Finally, we introduce the reader to the study of expressiveness and reference related work that is in the same vein as our own.

1.1 Machine Learning and Databases

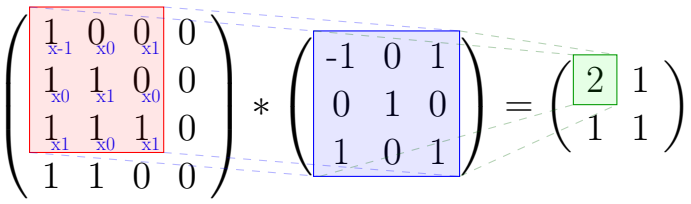
Machine Learning (ML) struck the world with new kinds of intense data use applications in image analysis [23], advertising [35], voice recognition [30, 7], natural language processing [41, 40] and much more [3, 26, 42, 43]. There are many ML models, but they all share one truth: you need data to make them work. For industrial applications, the volume of data is so big that it is natural for databases to appear in the picture. The relationship between databases and ML is not something new, as data mining tools like clustering and support vector machines have been part of the database stack for a decade or more. As things escalate, the industry is pushing to marry statistical analytic frameworks like R and Python with almost every data processing engines like Cloudera's Impala or SparkSQL [31]. One approach is to look at the operations done by machine learning tasks and the operations available in database systems, and think about how we can satisfy both through some unifying mechanism.

1.1.1 Operations in Machine Learning and Relational Algebra

Suppose we have unstructured data that we want to store. The first step is to choose a structure to save our data in the computer, which is the *data model*. There are many data models, which include relations, arrays, graphs, etc. We also need *operations* to do something with the stored data. Some common ones could be the multiplication of matrices or the join of two tables. When a data model has operations associated to them, we say that they constitute a *language*.

For the data handling of ML applications, a good choice is to use tensors as the data model [11]. Let us think of two simple tensors: one-dimensional and two-dimensional ones, more commonly called, arrays and matrices. Common neural network architectures use arrays and matrices; the former for the biases of neurons and the latter for the weights. These structures come with their own operations, which are grouped under the name of matrix operations and linear algebra operations. The distinction between matrix operations and linear algebra is blurry, and because of that, we refer to both as the latter, which is more general. Here we present two of the most commonly used linear algebra operations in ML applications:

- In Deep Learning, a convolution between a matrix A and a *kernel* K is a sort of 2-dimensional dot product. It works by adding each element of A to its local neighbors, weighted by K, the *kernel*.



- In mathematics, especially in applications of linear algebra to physics and recently in ML, the Einstein summation is a notational convention that implies summation over a set of indexed terms in a formula, thus achieving brevity. Next is the general formula for the Einstein Summation:

$$\text{einsum}((\alpha, \beta \to \gamma), \mathsf{A}, \mathsf{B})$$

Here we show two simple applications of this notation. The first serves to calculate the dot product of vectors and the second to calculate the matrix multiplication:

$$\text{einsum}((\mathrm{i}, \mathrm{i} \to \emptyset), \mathsf{a}, \mathsf{b}) = \mathsf{a} \cdot \mathsf{b}$$

$$\text{einsum}((\mathrm{i}j, jk \to \mathrm{i}k), \mathsf{A}, \mathsf{B}) = \mathsf{A} * \mathsf{B}$$

For multidimensional arrays, the Einstein summation has many more applications. In ML tasks, we are doing operations that are related to the indices of our array. Using the Einstein summation eases the writing of algorithms and operations.

Now lets look at the other side of things. Databases have a long history in computer science and with further research, old models get obsolte as new models appear all the time. Regarding inflection in database history, Codd proposed the relational model as the candidate for "large shared data banks" [8]. Although alternatives to the relational models have been proposed under the names of NoSQL and NewSQL, the relational model has dominated the database since 1980. Considering this, we concentrate on relational algebra as it constitutes the basis of the relational model [38].

1.1.2 A Practical Example

Let us think about a common situation in an industrial ML application pipeline: from various "factories", you gather and clean data to then train a ML model and finalize by saving the results.

If we analyze this task, we can see that the operations needed to complete it can be classified in the two algebras previously seen: relational and linear. The white boxes correspond to relational algebra operations as $\rho, \sigma, \Pi, \bowtie, \cup, \times$, etc., while the black box contains ML operations as matrix transpose, matrix multiplication, convolution, tensor contraction, etc. This tells us something: if we could unify linear and relational algebra models, we could use this to have a "mono-color" pipeline. This would solve the aforementioned impedance mismatch, as changing the data model would no longer be necessary. The benefits from this are immediate, as changing from one language to another is a cost not only for the computer, but also for the user. Discovering ways to execute computation among this segmented pipeline and also finding an optimal execution plan is challenging given semantic differences between disparate languages, so another benefit from unifying this pipeline would be to discover optimizations for the whole process.

1.2 Database Families

A *database* is a collection of data, generally stored and accessed from a computer. The *database management system* (DBMS) is the software that interacts with end users, applications, and the database itself to do all operations on the data. The functionality provided by a DBMS can vary enormously. Codd proposed the following functions and services a fully-fledged general purpose DBMS should provide [8]: data storage, retrieval and update, user accessible catalog or data dictionary describing the metadata, support for transactions and concurrency, ease of recovering the database should it become damaged, support for authorization of access and update of data, access support from remote locations and enforcing

constraints to ensure data in the database abides by certain rules.

Most database management systems include their own *application programming interface* (API) that allows the user to interact with their engine without going through the user interface of the DBMS. Interact how? Writing *queries*. A query is a request for data or information from a database. *Query languages* are computer languages used to make queries in databases. The query languages implement what we referred as the API. API's as being only an interface, can be implemented in various ways, which yields us various query languages. The core part of the DBMS interacting between the database and the application interface is sometimes referred to as the *database engine*, which is the underlying software component that a DBMS uses to create, read, update and delete (CRUD) data from a database. The total sum of the database, the DBMS and the query languages or APIs can be referred to as the *database system*. A *database model* determines the logical structure of a database and fundamentally determines in which manner data can be stored, organized and manipulated. This is very similar to what we referred as the data model, but in this case we refer to databases.

Computer scientists may classify database systems according to the database models that they support. Database systems span several families, as the relational, array, graph and file systems [20]. Before continuing, we must make a distinction between a logical and physical representation. Users relate to data logically by name, however, the actual data is physically located in sectors on a disk. Each family has a unifying logical representation, but particular systems within a family may implement a distinct physical representation of its family's logical representation. Our focus will be the former. Now we give a short review on some of the database families.

1.2.1 Relational Systems

The logical data representation of relational systems are relations. Computation takes the form of relational algebra, invoking methods such as selection, projection, Cartesian product, union, etc. PostgreSQL, Myria, and many other systems are members of the relational family, each having their own physical data representation following the relation pattern, and each having their own API supporting relational algebra calls.

1.2.2 Array Systems

The logical data representation of array systems is, as the name suggests, multidimensional arrays. Computation takes the form of linear algebra operations, invoking methods such as matrix multiply, reduce, element-wise function applications, and index reference. ScaLAPACK is an array system with a physical data representation following the matrix pattern. After placing data into the physical format required by ScaLAPACK, one can make BLAS API calls that perform linear algebra optimizations. SciDB is another member of the array systems family: a column-oriented DBMS designed for multidimensional data management and analytics common to scientific, geospatial, financial, and industrial applications.

1.2.3 Graph Systems

The logical data representation of graph systems can be relations or arrays. Many graph systems like AllegroGraph or Neo4J have physical data formats following the graph pattern, some in the case of adjacency lists, others as incidence matrices, etc. Graph databases are part of the NoSQL databases created to address the limitations of the existing relational databases. While the graph model explicitly lays out the dependencies between nodes of data, the relational model and other NoSQL database models link the data by implicit connections. The graph systems support APIs of vertex and edge processing with their APIs. They also commonly support sending messages through the nodes of a graph. For a survey on graph database systems see [2].

1.2.4 Polystore Systems

Many of the natural algorithms a family has can also be implemented in the other families. Of course, computations appearing concise and natural in one family may require gymnastics in another. Said another way, each family is similar in *expressiveness*. Computation efficiency, on the other hand, depends heavily on choice of family and even particular systems. MapReduce systems execute best for computations that truly fit the pattern of a map and reduce, such as counting words. Other systems tend to execute problems that don not fit the map-reduce pattern faster, such as array systems for matrix inversion and convolution. All-to-all shortest paths may execute best on graph systems, though many relational and array systems are strong contenders.

Therefore, there is motivation to use a variety of system families for execution, not because any one family can express computations that the others cannot, but because we gain performance by leveraging systems from different families for computations they perform best. We call such a hybrid, federated system a polystore system. A polystore system is any DBMS that is built on top of multiple, heterogeneous, integrated engines. The motto of polystore systems is simple: "Use the right system for the right job." Many tasks can be decomposed into a set of jobs that execute most efficiently when run on different systems.

Here we give a brief polystore terminology review:

- Island: An abstraction of database engines having a similar data model and query language.
- Shim: A query transformation operation written in one language, but intended for a separate Island. Handles the intended operation on the target system in terms of another language.
- Cast: A data transformation operation from one data model to another.

BigDAWG is an open source project from researchers within the Intel Science and Technology Center for Big Data and is a reference implementation of a polystore database. The current release includes support for 3 database engines: PostgreSQL, SciDB and Accumulo. For more information about federated systems and polystores, look at this short article by

Stonebreaker [39].

1.3 Languages of Relations and Arrays

To operate the DBMS, we have the languages. When we said that the expressiveness of the families described was similar, we were actually saying that the expressiveness of the query languages that implements the API of a database family are similar. As mentioned in the first section, data scientists mostly choose between using arrays to represent their data, or using the relational data model and use relations. For that the scope of this section considers only languages from the relational or array family. Here we give a brief overview of the representing languages of each family.

1.3.1 Relational Languages

Structured Query Language (SQL) is a programming language used to communicate with data stored in a relational database management system or RDBMS. SQL syntax is similar to the English language, which makes it relatively easy to write, read, and interpret. SQL offers two main advantages: first, it introduces the concept of accessing many records with one single command, and second, it eliminates the need to specify how to reach a record. SQL was one of the first commercial languages for Codd's relational model. Despite not entirely adhering to the relational model as described by Codd, it became the most widely used database language. There are many classes of SQL. For example, SQLite is a relational database management system that contains a minimal set of SQL commands. A distinction should be made between alternatives to SQL as a language, and alternatives to the relational model itself.

Next are proposed relational alternatives to the SQL language:

- Datalog: this language has two advantages over SQL: it has cleaner semantics, which facilitates program understanding and maintenance, and it is more expressive, in particular for recursive queries.
- IBM Business System 12: one of the first fully relational database management systems, introduced in 1982.

There are more, but SQL remains being the dominant till this day.

1.3.2 Array Languages

In computer science, array programming refers to solutions which allow the application of operations to an entire set of values at once. Modern programming languages that support array programming (also known as vector or multidimensional languages) have been engineered specifically to generalize operations on scalars to apply transparently to vectors, matrices,

and higher-dimensional arrays. These include Fortran 90, MATLAB, Analytica, Octave, R, Julia, Wolfram Language, and the NumPy extension to Python. Array programming primitives concisely express broad ideas about data manipulation. The level of concision can be dramatic in certain cases: it is not uncommon to find array programming language one-liners that require more than a couple of pages of SQL code. Providing database support for matrices and multidimensional arrays has been a long-standing research topic [34], originally geared towards applications in scientific data management, and more recently motivated by machine learning over big data [4, 36, 27].

1.4 LARA

Polystores connect back-end systems with front-end languages through a unifying narrow API, using each system where it performs best. With all that has been said, we want a query language that can implement all the algorithms in all the families API's. Discovering ways to execute computation among multiple available systems, let alone discovering an optimal execution plan, is challenging given semantic differences between disparate families of systems. A key step to building a polystore is to devise a common data representation which facilitates translation between the languages and systems composing it: revealing joint optimization opportunities, promoting more efficient data transfer, and reducing the semantic gap that programmers face when writing code across families. So, which language should we choose? The answer: No choice necessary: Use LARA. In short, LARA proposes the following scheme:

- Write algorithms in any/all algebras.
- Translate to/from LARA common algebra.
- Use any/all execution engines.

It is conjectured that translating computation through LARA leads to better performance, gained from running parts of algorithms on different systems. The common abstraction of Lara enables multi-system optimizations difficult to capture otherwise. LARA underlies and unifies algebras representing the families above in order to facilitate translation between systems, for which is a candidate for the proposed BigDAWG polystore system. In particular, LARA unifies relational and array algebras.

1.5 The Study of Expressiveness

In logic and computer science, the *expressive power* (also called expressiveness or expressivity) of a logic, language or algebra is the breadth of ideas that can be represented in such abstraction. It is proposed that the expressiveness has two common meanings [14]: The theoretical expressivity, which is the measure of what ideas can be expressed without regard to how the ideas are expressed, and the practical expressivity, which is the measure of how readily ideas can be expressed. The former sense dominates in areas of mathematics and

logic that deal with the formal description of languages and their meaning, such as formal language theory, mathematical logic and process algebra, so it is the one that attains to us. The more expressive a language is, the greater the variety and quantity of ideas it can be used to represent. For example, take a language that has natural numbers and only the sum. Is there an expression for the square of a number x? The answer is negative, as expressing a square with only sum give us an expression that depends on the number x, or said in the other way, there is no fixed formula for calculating the square of x. Instead, if we include the multiplication operator, the square of a number x is easily expressible as $x \times x$. If we label the names of these languages as A and B respectively, we say that the expressive power of B is greater than the one of A. More over, as we can express everything of A in B, we say that B captures the expressive power of A.

1.5.1 Descriptive Complexity

Descriptive complexity is a branch of computational complexity theory and of finite model theory that characterizes complexity classes by the type of logic needed to express the languages in them. In the beginning, there were two measures of computational complexity: time and space. From an engineering standpoint, these were very natural measures, quantifying the amount of physical resources needed to perform a computation. From a mathematical viewpoint, time and space were somewhat less satisfying, since neither appeared to be tied to the inherent mathematical complexity of the computational problem.

In 1974, Ronald Fagin changed this. Fagin's theorem is the oldest result of descriptive complexity theory. It states that the set of all expressible properties in existential second-order logic is exactly the complexity class NP. The connection between complexity and the logic of finite structures allows results to be transferred from one area to the other, enabling new proof methods [21]. The queries, when restricted to finite structures, correspond to the computational problems of finite model theory [24]. Using first-order languages, this approach demonstrated that almost all measures of complexity can be explained in logic. Descriptive complexity has a myriad of applications to database theory. Many questions about the expressibility of query languages and the efficiency of their evaluation have been resolved using it.

1.5.2 First-order Logic

Mathematical logic seeks to formalize the process of mathematical reasoning and turn this process itself into a subject of mathematical inquiry. First-order logic is very successful at this purpose. Many natural mathematical theories can be expressed as first-order theories. These include set theory, fundamental to the foundations of mathematics. Godel's completeness theorem guarantees that the consequences of these theories can be effectively obtained.

In descriptive complexity, FO is a complexity class of structures that can be recognized by formulas of first-order logic, and also equals the complexity class AC0. Restricting predicates to be from a set X yields a smaller class FO[X]. For instance, FO[$<$] is the set of star-free

languages. It turns out that first-order logic is lacking in expressive power: it cannot express certain queries, as queries involving transitive closure. However, adding expressive power must be done with care: it must still remain possible to evaluate queries with reasonable efficiency, which is not the case, for second-order logic.

1.5.3 Relational Algebra

The relational algebra used by the relational model directly inherits from first-order logic as Edgar Codd used the latter to formulate operations in the former. For instance, one of the primitive relational operators, the Cartesian Product can be written in logic like this:

$$X \times Y = \{(x, y) \parallel x \in X \wedge y \in Y\}$$

The other four operators were also formulated using FO. There are other more complex operators based on the five primitive operators, they also use FO.

In a relational database, the information is described by a number of relations. An answer to a question is represented by a new relation, that can be calculated from the relations of the data base, using the relational algebra. Codd's theorem states that relational algebra and the domain-independent relational calculus queries, two well-known foundational query languages for the relational model, are precisely equivalent in expressive power. That is, a database query can be formulated in one language if and only if it can be expressed in the other. Codd's theorem is notable since it establishes the equivalence of two syntactically quite dissimilar languages: relational algebra is a variable-free language, while relational calculus is a logical language with variables and quantification. Query languages that are equivalent in expressive power to relational algebra were called relationally complete by Codd. Relational completeness clearly does not imply that any interesting database query can be expressed in relationally complete languages. Well-known examples of inexpressible queries include simple aggregations (counting tuples, or summing up values occurring in tuples, which are operations expressible in SQL but not in relational algebra, and computing the transitive closure of a graph given its adjacency matrix relation. Nevertheless, relational completeness constitutes an important yardstick by which the expressive power of query languages can be compared.

1.5.4 SQL

Libkin studies the expressive power of SQL [24]. We note that SQL is based on relational algebra, but they are not the same. Libkin states that the SQL langugage can not express reachability queries. An example of this is computing the transitive closure of a graph. Before formal proofs, it was a folk result that first-order logic, relational algebra and calculus, and SQL could not express recursive queries such as reachability, and for this SQL3 introduced a recursion operator. But Libkin points out that SQL has more expresiveness that relational algebra and calculus, as this language can express queries than compare the cardinalities of

relations. He says that relational algebra is distinct from SQL because the latter possess aggregate functions, grouping and arithmetic. In the first section of his paper he does a survey of work related to the expressive power of SQL, which ends in a proof given here [17]. In the second section, he gives a new proof of why SQL cannot express the transitive closure. It follows roughly like this:

- He presents relational algebra with aggregates. This consists of relational algebra operations: permutation, boolean operations, cartesian product, projection and selection, arithmetic operations: numerical selection and function application, and aggregation and grouping operations.
- He then introduces the concept of *locality* of queries. He says that the general idea is that a query can only look at a small portion of its input, and by small we mean a neighborhood of a fixed radius.
- He introduces a logic $\mathcal{L}_{\mathsf{agg}}$ into which he translates $\text{ALG}_{\mathsf{aggr}}$.
- Then he translates $\mathcal{L}_{\mathsf{agg}}$ into $\mathcal{L}_{\mathbf{C}}$, which is a logic that contains infinite conjunction and disjunction, but does not have aggregation.
- Finally he shows that $\mathcal{L}_{\mathbf{C}}$ is not local, finishing the proof.

For the complete and original proof, refer to the paper [24].

1.5.5 MATLANG

MATLANG [5] is a language for matrices based in common matrix operations and linear algebra that serves as "an analog for matrices of the relational algebra for relations". Quoting the paper where it first appeared: "Indeed, given their popularity, we believe the expressive power of matrix sublanguages also deserves to be understood in its own right".

The five operations at disposal are the one-vector; turning a vector in a diagonal matrix; matrix multiplication; matrix transposition; and pointwise function application. This seemingly simple language is capable of expressing the minimum element of some vector v and the Google matrix of any directed graph G, having its adjacency matrix. It is also proven that the expressive power of MATLANG is bounded by the one of relational algebra with aggregations (only sumation) and also is subsumed by aggregate logic with only three nonnumerical variables and thus, MATLANG can express all queries from graph databases to binary relations expressible in first-order logic with three variables. This also means that asking if a graph contains a four-clique is not expressible, as this demands the use of at least four nonnumerical variables.

MATLANG is then extended with and operation for inverting matrices and named MATLANG + inv, which increases its expressive power. A prove of this is that the transitive closure of binary relations can be expressed. That this can not be expressed without inversion, follows from the locality of relational algebra with aggregation [24]. One final extension is made adding an operation that can return the eigenvectors and eigenvalues of a matrix, which is called MATLANG + eigen. This version of MATLANG can express the inversion. An open question is posed: are there boolean queries about matrices, or generic queries about

graphs, expressible in MATLANG + eigen but not in MATLANG + inv?

Chapter 2

LARA and $\mathrm{FO}_{\mathsf{Agg}}$

Here we give a formal introduction to the study object, LARA, and the logic we use as a yardstick, first order logic with aggregation ($\mathrm{FO}_{\mathsf{Agg}}$).
For this we express the basic notions of the relational model such as schema, sort and relation. Then, we present the data model of LARA, the associative table, and the operations in the language, namely, *join*, *union*, and *extend*. Finally we present the syntax and semantics of first order logic and from there we expand it to deal with aggregation.
For integers $m \leq n$, we write $[m, n]$ for $\{m, \ldots, n\}$ and $[n]$ for $\{1, \ldots, n\}$. If $\bar{v} = (v_1, \ldots, v_n)$ is a tuple of elements, we write $\bar{v}[\mathrm{i}]$ for v_i. We denote multisets as $\{\!\{a, b, \ldots\}\!\}$.

2.1 Basic Notions

First, we need a universe from where the elements that will populate our database come. Let that universe be $\mathcal{U}$. Next, we will need two things to compose a relation, which are a *relation name* $(A, B, C, \ldots)$ and a set of *attributes names* of that relation, i.e., the name of each column of the relation (ID, employee, salary, etc.). Formally speaking, we assume two sets: a set of names, Names, and a set of attributes, $\mathsf{Attributes}$. These sets are infinite, and although in finite model theory we deal with finite sets, most database systems do not have problems with the number of names available, so it is safe to assume this. Each relation name is associated with a finite set of attributes. We model this by means of a function **sort** from Names to $\mathsf{Attributes}$, that assigns such a set of attributes to a relation name. Let $\mathcal{P}_{\mathrm{fin}}(X)$ be the finite power set of a set X. Then $\mathbf{sort} : \mathsf{Names} \to \mathcal{P}_{\mathrm{fin}}(\mathsf{Attributes})$. If R is a relation name and A a finite set of attribute names, we write $R[A]$ when $\mathbf{sort}(R) = A$. For example, we write $R[city, name, country]$ when the attributes associated with relation name R are *city*, *name*, and *country*. A *relational schema* is a finite set $\sigma = \{R_1, \ldots, R_n\}$ of relation names. A *tuple of sort* A is a function $t : A \to \mathcal{U}^{|A|}$. Finally, a *database* D over schema σ is a mapping that assigns a finite set R^D of tuples of sort A to each relation symbol $R[A] \in \sigma$.

2.2 LARA: Linear Algebra and Relational Algebra

We expand upon the definition of relations to make them *two-sorted*. This means that now a relation name is associated with two different sets of attributes names. We assume two set of attribute names: $\mathcal{K}$ and $\mathcal{V}$. A *two-sorted relational schema* is a finite collection σ of two-sorted relation symbols. We say that the first sort consists of *key-attributes* and the second one of *value-attributes*. So in this context, $\mathbf{sort} : \mathsf{Names} \to \mathcal{P}_{\text{fin}}(\mathcal{K}) \times \mathcal{P}_{\text{fin}}(\mathcal{V})$. Each relation symbol $R \in \sigma$ is then associated with a pair $(\bar{K}, \bar{V})$, where $\bar{K}$ and $\bar{V}$ are (possibly empty) tuples of different key- and value- attributes, respectively. We write $R[\bar{K}, \bar{V}]$ to denote that $(\bar{K}, \bar{V})$ is the *sort* of R. We do not distinguish between $\bar{K}$, resp., $\bar{V}$, and the set of attributes mentioned in it. There are two countably infinite sets of objects over which databases are populated: A domain of *keys*, which interpret key-attributes and is denoted Keys, and a domain of *values*, which interpret value-attributes and is denoted Values. A *tuple of sort* $(\bar{K}, \bar{V})$ is a function $t : \bar{K} \cup \bar{V} \to \mathsf{Keys} \cup \mathsf{Values}$ such that $t(A) \in \mathsf{Keys}$ if $A \in \bar{K}$ and $t(A) \in \mathsf{Values}$ if $A \in \bar{V}$. A *database* D over schema σ is a mapping that assigns with each relation symbol $R[\bar{K}, \bar{V}] \in \sigma$ a finite set R^D of tuples of sort $(\bar{K}, \bar{V})$. We often see D as a set of *facts*, i.e., as the set of expressions $R(t)$ such that $t \in R^D$. For ease of presentation, we write $R(\bar{k}, \bar{v}) \in D$ if $R(t) \in D$ for some tuple t with $t(\bar{K}) = \bar{k}$ and $t(\bar{V}) = \bar{v}$ (where $\bar{k} \in \mathsf{Keys}^{|\bar{K}|}$ and $\bar{v} \in \mathsf{Values}^{|\bar{V}|}$).

2.2.1 Data model

For a database D to be a LARA *database* we need D to satisfy an extra restriction: Key attributes define a key constraint over the corresponding relation symbols. That is,

$$R(\bar{k}, \bar{v}), R(\bar{k}, \bar{v}') \in D \quad \Longrightarrow \quad \bar{v} = \bar{v}',$$

for each $R[\bar{K}, \bar{V}] \in \sigma$, $\bar{k} \in \mathsf{Keys}^{|\bar{K}|}$, and $\bar{v}, \bar{v}' \in \mathsf{Values}^{|\bar{V}|}$. Relations of the form R^D are called *associative tables* [20].

Employee	**PhD**	**Salary**	**Hobby**
Ringo	Philosophy	2000	Tennis
Jhon	Linguistics	1500	Guitar
Paul	Biology	2350	Bonsai

An Associative Table.

Yet, we call associative table to any set of tuples A of the same sort $(\bar{K}, \bar{V})$ such that $\bar{v} = \bar{v}'$ for each $(\bar{k}, \bar{v}), (\bar{k}, \bar{v}') \in A$. In such a case, A is of sort $(\bar{K}, \bar{V})$. Notice that for a tuple $(\bar{k}, \bar{v})$ in A, we can safely denote $\bar{v} = A(\bar{k})$. From this perspective, we can see an associative table as a function.

2.2.2 Syntax

First we present the grammar of the language.

$$\begin{aligned}\mathbf{e} := \ & \mathsf{T} \\ & \mid \mathsf{T}_1 \bowtie_{\oplus} \mathsf{T}_2 \\ & \mid \mathsf{T}_1 \between_{\otimes} \mathsf{T}_2 \\ & \mid \mathrm{ext}_f \mathsf{T}\end{aligned}$$

Only three expressions are available in the LARA language, which are the *join*, *union* and *ext*. The first is parameterized by a binary function, as can be the sum of two numbers or the division of them. The second is parameterized by an aggregate operator which we define next. The final one is parameterized by an user-defined function which we will see, can be very powerful.

An *aggregate operator* over domain U is a family $\oplus = \{\oplus_0, \oplus_1, \ldots, \oplus_\omega\}$ of functions, where each $\oplus_k$ takes a multiset of k elements from U and returns a single element in U. If u is a collection of k elements in U, we write $\oplus(u)$ for $\oplus_k(u)$. We only deal with aggregate operators that in their binary form, i.e. when there is an aggregation between two elements, the aggregate operator is associative, commutative and has an identity element. This is a standard restriction which encompasses most aggregate operators used in practical query languages; e.g., SUM, AVG, MIN, MAX, and COUNT.

The syntax of LARA is parameterized by a set of *extension functions*. This is a collection Ω of user-defined functions f that map each tuple t of sort $(\bar{K}, \bar{V})$ to a finite associative table of sort $(\bar{K}', \bar{V}')$, for $\bar{K} \cap \bar{K}' = \emptyset$ and $\bar{V} \cap \bar{V}' = \emptyset$. We say that f is of sort $(\bar{K}, \bar{V}) \mapsto (\bar{K}', \bar{V}')$. From the definition of the extension functions we see that they can be used to add new keys and values to a table, and also can convert rows of a table to a new table. This is the reason we had to parameterize them.

Now, we inductively define the set of expressions in LARA(Ω) over schema σ as follows.

- *Empty associative table.* $\emptyset$ is an expression of sort $(\emptyset, \emptyset)$.
- *Atomic expressions.* If $R[\bar{K}, \bar{V}]$ is in σ, then R is an expression of sort $(\bar{K}, \bar{V})$.
- *Join.* If e_1 and e_2 are expressions of sort $(\bar{K}_1, \bar{V}_1)$ and $(\bar{K}_2, \bar{V}_2)$, respectively, and $\oplus$ is an binary function over Values, then $\mathrm{e}_1 \bowtie_{\oplus} \mathrm{e}_2$ is an expression of sort $(\bar{K}_1 \cup \bar{K}_2, \bar{V}_1 \cup \bar{V}_2)$.
- *Union.* For e_1, e_2, and $\otimes$ is an aggregate operator, it is the case that $\mathrm{e}_1 \between_{\otimes} \mathrm{e}_2$ is an expression of sort $(\bar{K}_1 \cap \bar{K}_2, \bar{V}_1 \cup \bar{V}_2)$.
- *Extend.* For e an expression of sort $(\bar{K}, \bar{V})$ and f a function in Ω of sort $(\bar{K}, \bar{V}) \mapsto (\bar{K}', \bar{V}')$, it is the case that Ext_f e is an expression of sort $(\bar{K} \cup \bar{K}', \bar{V}')$.

We write $\mathrm{e}[\bar{K}, \bar{V}]$ to denote that expression e is of sort $(\bar{K}, \bar{V})$.

Before going through the semantics of LARA, we shall give examples of what these operations do to ease their understanding. We will assume the existence of three associative

tables: A, B and T.

Example 2.1 *We begin by presenting the* join *operator. We assume we have multiplication between two numbers,* $\times$.

i	j	v
0	0	1
0	1	2

A

$\bowtie_{\times}$

j	k	v
0	0	1
0	1	1

B

$=$

i	j	k	v
0	0	0	1
0	0	1	1

A $\bowtie_{\times}$ B

This is very similar to the join of relational algebra. The resulting table contains the key and value attributes of both original tables.

Example 2.2 *We now present an example of the* union *operator. We will assume an aggregate operator* $+$*, which returns the sum of its arguments.*

i	j	v
0	0	1
0	1	2

A

$\barwedge_{+}$

j	k	v
0	0	1
0	1	1

B

$=$

j	v
0	3
1	2

A $\barwedge_{+}$ B

This operation can be seen as a vertical concatenation of tables. It maintains the common key attributes and has all the value attributes of the original tables.

Example 2.3 *The most interesting example is for the* extend *operator. We will assume the existence of a function* wrdcnt, which counts the words of a given text.

id	text
1	"La Lo La Lo"
2	"Some random phrase"

T

$\text{ext}_{\text{wrdcnt}}$ T

id	wrd	count
1	"La"	2
1	"Lo"	2
2	"Some"	1
2	"random"	1
2	"phrase"	1

We see that the operator added a new key value to the original table and changed the value **text** to a new one **count**.

2.2.3 Semantics

We assume that for every aggregate operator $\oplus$ over domain U there is a *neutral value* $0_{\oplus}$ such that $u \oplus 0_{\oplus} = u$, for every $u \in U$. Given an aggregate operator $\oplus$ and tuples $\bar{u} = (u_1, \ldots, u_n)$

and $\bar{v} = (v_1, \ldots, v_n)$ in Values^n, we define $\bar{u} \oplus \bar{v} := (u_1 \oplus v_1, \ldots, u_n \oplus v_n)$. An important notion is *padding*. Let $\bar{V}_1$ and $\bar{V}_2$ be tuples of value-attributes, and $\bar{v}$ a tuple over Values of sort $\bar{V}_1$. Then $\mathrm{pad}_{\oplus}^{\bar{V}_2}(\bar{v})$ is a new tuple $\bar{v}'$ over Values of sort $\bar{V}_1 \cup \bar{V}_2$ such that for each $V \in \bar{V}_1 \cup \bar{V}_2$ we have that $v'(V) = v(V)$, if $V \in \bar{V}_1$, and $v'(V) = 0_{\oplus}$, otherwise.

Consider tuples $\bar{k}_1$ and $\bar{k}_2$ over key-attributes $\bar{K}_1$ and $\bar{K}_2$, respectively. We say that $\bar{k}_1$ and $\bar{k}_2$ are *compatible*, if $\bar{k}_1(K) = \bar{k}_2(K)$ for every $K \in \bar{K}_1 \cap \bar{K}_2$. If $\bar{k}_1$ and $\bar{k}_2$ are compatible, one can define the extended tuple $\bar{k}_1 \cup \bar{k}_2$ over key-attributes $\bar{K}_1 \cup \bar{K}_2$. Also, given a tuple $\bar{k}$ of sort $\bar{K}$, and a set $\bar{K}' \subseteq \bar{K}$, the restriction $\bar{k}{\downarrow_{\bar{K}'}}$ of $\bar{k}$ to attributes $\bar{K}'$ is the only tuple of sort $\bar{K}'$ that is compatible with $\bar{k}$. Another way to see the restriction of a tuple is as a shorter version of that tuple restricted to the attributes in $\bar{K}'$. Finally, given an aggregate operator $\oplus$ (which we also allow to be a binary operator) and a multiset T of elements of the same type that $\oplus$ takes, we define $\mathsf{Solve}_{\oplus}(T)$ as

$$\mathsf{Solve}_{\oplus}(T) := \{(\bar{k}, \bar{v}) \mid \text{there exists } \bar{u} \text{ such that } (\bar{k}, \bar{u}) \in T \text{ and } \bar{v} = \bigoplus\{\{\bar{u} \mid (\bar{k}, \bar{u}) \in T\}\}\}.$$

The intuition behind the name is that it allows us to "solve" conflicts between tuples. These conflicts arise because key values can only appear once per table.

The evaluation of a LARA(Ω) expression e over schema σ on a LARA database D, denoted e^D, is inductively defined as follows.

- *Empty associative table.* if $\mathrm{e} = \emptyset$ then $\mathrm{e}^D := \emptyset$.
- *Atomic expressions.* If $\mathrm{e} = R[\bar{K}, \bar{V}]$, for $R \in \sigma$, then $\mathrm{e}^D := R^D$.
- *Join.* If $\mathrm{e}[\bar{K}_1 \cup \bar{K}_2, \bar{V}_1 \cup \bar{V}_2] = \mathrm{e}_1[\bar{K}_1, \bar{V}_1] \bowtie_{\oplus} \mathrm{e}_2[\bar{K}_2, \bar{V}_2]$, then

$$\mathrm{e}^D := \{(\bar{k}_1 \cup \bar{k}_2, \bar{v}_1 \oplus \bar{v}_2) \mid \bar{k}_1 \text{ and } \bar{k}_2 \text{ are compatible tuples such that } \bar{v}_1 = \mathrm{pad}_{\oplus}^{\bar{V}_2}(\mathrm{e}_1^D(\bar{k}_1)) \text{ and } \bar{v}_2 = \mathrm{pad}_{\oplus}^{\bar{V}_1}(\mathrm{e}_2^D(\bar{k}_2))\}.$$

- *Union.* If $\mathrm{e}[\bar{K}_1 \cap \bar{K}_2, \bar{V}_1 \cup \bar{V}_2] = \mathrm{e}_1[\bar{K}_1, \bar{V}_1] \between_{\oplus} \mathrm{e}_2[\bar{K}_2, \bar{V}_2]$, then

$$\mathrm{e}^D := \mathsf{Solve}_{\oplus}\{\{(\bar{k}, \bar{v}) \mid \bar{k} = \bar{k}_1{\downarrow_{\bar{K}_1 \cap \bar{K}_2}} \text{ and } \bar{v} = \mathrm{pad}_{\oplus}^{\bar{V}_2}(\mathrm{e}_1^D(\bar{k}_1)) \text{ for some } \bar{k}_1 \in \mathrm{e}_1^D, \text{ or } \bar{k} = \bar{k}_2{\downarrow_{\bar{K}_1 \cap \bar{K}_2}} \text{ and } \bar{v} = \mathrm{pad}_{\oplus}^{\bar{V}_1}(\mathrm{e}_2^D(\bar{k}_2)) \text{ for some } \bar{k}_2 \in \mathrm{e}_2^D\}\}.$$

- *Extend.* If $\mathrm{e}[\bar{K} \cup \bar{K}', \bar{V}'] = \mathsf{Ext}_f\, \mathrm{e}_1[\bar{K}, \bar{V}]$ and f is of sort $(\bar{K}, \bar{V}) \mapsto (\bar{K}', \bar{V}')$, then

$$\mathrm{e}^D := \{(\bar{k} \cup \bar{k}', \bar{v}') \mid (\bar{k}, \bar{v}) \in \mathrm{e}_1^D, \text{ and } (\bar{k}', \bar{v}') \in f(\bar{k}, \bar{v})\}.$$

  Notice that in this case $\bar{k} \cup \bar{k}'$ always exists as $\bar{K} \cap \bar{K}' = \emptyset$.

Several useful operators, as described below, can be derived from the previous ones.

- *Map operation.* An important particular case of Ext_f occurs when f is of sort $(\bar{K}, \bar{V}) \mapsto (\emptyset, \bar{V}')$, i.e., when f does not extend the keys in the associative table but only modifies the values. Following [20], we write this operation as Map_f.

- *Aggregation.* This corresponds to an aggregation over some of the key-attributes of an associative table. Consider a LARA expression $\mathrm{e}_1[\bar{K}_1, \bar{V}_1]$, an aggregate operator $\oplus$ over Values, and a $\bar{K} \subseteq \bar{K}_1$, then $\mathrm{e} = \bigotimes_{\oplus}^{\bar{K}} \mathrm{e}_1$ is an expression of sort $(\bar{K}, \bar{V}_1)$ such that $\mathrm{e}^D := \mathsf{Solve}_{\oplus}\{\{(\bar{k}, \bar{v}) \mid \bar{k} = \bar{k}_1{\downarrow_{\bar{K}}} \text{ and } \bar{v} = \mathrm{e}_1^D(\bar{k}_1)\}\}$. We note that $\bigotimes_{\oplus}^{\bar{K}} \mathrm{e}_1$ is equivalent to the expression $\mathrm{e}_1 \bigotimes_{\oplus} \mathsf{Ext}_f(\emptyset)$, where f is the function that associates an empty table of sort $(\bar{K}, \emptyset)$ with every possible tuple.
- *Reduction.* The reduction operator, denoted by $\bar{\bigotimes}$, is just a syntactic variation of aggregation defined as $\bar{\bigotimes}_{\oplus}^{\bar{L}} \mathrm{e}_1 \equiv \bigotimes_{\oplus}^{\bar{K}\setminus\bar{L}} \mathrm{e}_1$.

Next we provide an example that applies several of these operators.

Example 2.4 *Consider the schema* $\mathrm{Seqs}[(time, batch, features), (val)]$*, which represents a typical tensor obtained as the output of a recurrent neural network that processes input sequences. The structure stores a set of features obtained when processing input symbols from a sequence, one symbol at a time. For efficiency the network can simultaneously process a batch of examples and provide a single tensor as output.*

Assume that, in order to make a prediction one wants to first obtain, for every example, the maximum value of every feature over the time steps, and then apply a softmax *function. One can specify all this process in* LARA *as follows.*

$$\begin{aligned}
\mathrm{Max} &= \bar{\bigotimes}_{\max(\cdot)}^{(time)} \mathrm{Seqs} && (2.1)\\
\mathrm{Exp} &= \mathsf{Map}_{\exp(\cdot)}\mathrm{Max} && (2.2)\\
\mathrm{SumExp} &= \bar{\bigotimes}_{\mathrm{sum}(\cdot)}^{(features)} \mathrm{Exp} && (2.3)\\
\mathrm{Softmax} &= \mathrm{Exp} \bowtie_{\div} \mathrm{SumExp} && (2.4)
\end{aligned}$$

Expression (2.1) *performs an aggregation over the time attribute to obtain the new tensor* $\mathsf{Max}[(batch, features), (val)]$ *such that* $\mathsf{Max}(b, f) = \max_{u=\mathsf{Seqs}(t,b,f)} u$*. That is,* Max *stores the maximum value over all time steps (for every feature of every example). Expression* (2.2) *applies a point-wise exponential function to obtain the tensor* $\mathsf{Exp}[(batch, features), (val)]$ *such that* $\mathsf{Exp}(b, f) = \exp(\mathsf{Max}(b, f))$*. In expression* (2.3) *we apply another aggregation to compute the sum of the exponentials of all the (maximum) features. Thus we obtain the tensor* $\mathsf{SumExp}[(batch), (val)]$ *such that*

$$\mathsf{SumExp}(b) \;=\; \sum_f \mathsf{Exp}(b, f) \;=\; \sum_f \exp(\mathsf{Max}(b, f)).$$

Finally, expression (2.4) *applies point-wise division over the tensors* $\mathsf{Exp}[(batch, features), (val)]$ *and* $\mathsf{SumExp}[(batch), (val)]$*. This defines a tensor* $\mathsf{Softmax}[(batch, features), (val)]$ *such that*

$$\mathsf{Softmax}(b, f) \;=\; \frac{\mathsf{Exp}(b, f)}{\mathsf{SumExp}(b)} \;=\; \frac{\exp(\mathsf{Max}(b, f))}{\sum_{f'} \exp(\mathsf{Max}(b, f'))}.$$

Thus, we effectively compute the softmax *of the vector of maximum features over time for every example in the batch.* □

It is easy to see that for each LARA expression e and LARA database D, the result $\mathrm{e}(D)$ is always an associative table. Moreover, although the elements in the evaluation $\mathrm{e}(D)$ of an

expression e over D are not necessarily in D (due to the applications of the operator $\mathsf{Solve}_{\oplus}$ and the extension functions in Ω), all LARA expressions are *safe*, i.e., $|\mathrm{e}^D|$ is finite.

Proposition 2.1 *Let* e *be a* LARA(Ω) *expression. Then* e^D *is a finite associative table, for every* LARA *database* D*.*

2.3 First Order Logic

A vocabulary σ is a collection of constant symbols $(c_0, \ldots)$, relation symbols $(R_0, \ldots)$ and function symbols $(f_0, \ldots)$. Each relation and function symbol has an associated arity. A *structure* or *model*

$$\mathcal{A} = \{A, \{c_{\mathrm{i}}^{\mathcal{A}}\}, \{R_{\mathrm{i}}^{\mathcal{A}}\}, \{f_{\mathrm{i}}^{\mathcal{A}}\}\}$$

consists of an universe A along with an interpretation of every symbol of the vocabulary σ.

- each constant symbol $c_{\mathrm{i}} \in \sigma$ is interpreted as an element of A, i.e., $c_{\mathrm{i}}^{\mathcal{A}} \in A$
- each relation symbol $R_{\mathrm{i}} \in \sigma$ of arity k is interpreted as a relation $R_{\mathrm{i}}^{\mathcal{A}} \in A^k$.
- each constant symbol $f_{\mathrm{i}} \in \sigma$ of arity k is interpreted as a function $f_{\mathrm{i}}^{\mathcal{A}}$ such that $f_{\mathrm{i}}^{\mathcal{A}} : A^k \to A$.

For example, if σ has constant symbols $0, 1$, a binary relation symbol $<$, and two binary function symbols $\times$ and $+$, then one possible structure for σ is the natural field $\mathbf{N} = \{\mathbb{N}, 0^{\mathbf{N}}, 1^{\mathbf{N}}, <^{\mathbf{N}}, +^{\mathbf{N}}, \times^{\mathbf{N}}\}$. Naturally, $0^{\mathbf{N}}, 1^{\mathbf{N}}, <^{\mathbf{N}}, +^{\mathbf{N}}, \times^{\mathbf{N}}$, have their expected interpretations, so when this is clear we will omit the superscript leaving our structure simply as $\mathbf{N} = \{\mathbb{N}, 0, 1, <, +, \times\}$.

2.3.1 Syntax

We assume a countable infinite set of variables $(x, y, z, \ldots)$. We inductively define the terms and formulae of FO as follows:

- Each variable x is a term.
- Each constant symbol c is a term.
- If $t_1, \ldots, t_k$ are terms and f is a k-ary function symbol, then $f(t_1, \ldots, t_k)$ is a term.
- If t_1 and t_2 are terms, then $t_1 = t_2$ is a formula.
- If $t_1 \ldots, t_k$ are terms and R is a k-ary relation symbol, then $R(t_1, \ldots, t_k)$ is a formula.
- If φ_1, φ_2 are formulae, then $\varphi_1 \vee \varphi_2$, $\varphi_1 \vee \varphi_2$ and $\neg\varphi_1$ are formulae.
- If φ is a formula and x a variable, then $\exists x\varphi$ and $\forall x\varphi$ are formulae.

We also introduce the concept of free variables of a term or formula as follows:

- The only free variable of term x is x.
- Constant terms c do not have free variables.

- The free variables of the formula $t_1 = t_2$ are the free variables of t_1 and t_2.
- The free variables of $R(t_1, \ldots, t_k)$ or $f(t_1, \ldots, t_k)$ are the free variables of $t_1, \ldots, t_k$.
- The free variables of a negated formula are the same of the non-negated one.
- The free variables of $\varphi_1 \vee \varphi_2$ and $\varphi_1 \vee \varphi_2$ are the free variables of φ_1 and φ_2.
- The free variables of $\exists x\varphi$ and $\forall x\varphi$ are the free variables of φ except x.

Variables that are not free are called bound. If $\bar{x}$ is the tuple of all the free variables of φ, we write $\varphi(\bar{x})$.

2.3.2 Semantics

Given a model $\mathcal{A}$, we inductively define for each term t with free variables $(x_1, \ldots, x_n)$ the value $t^{\mathcal{A}}(\bar{a})$, where $\bar{a} \in \mathcal{A}^n$, and for each formula $\varphi(x_1, \ldots, x_n)$, the notion of $\mathcal{A} \models \varphi(\bar{a})$, which means that the formula $\varphi(\bar{a})$ is true in $\mathcal{A}$.

- If t is a constant symbol c, then the value of t in $\mathcal{A}$ is $t^{\mathcal{A}}$.
- If t is a variable x_{i}, then the value of $t^{\mathcal{A}}(\bar{a})$ is a_{i}.
- If t is $f(x_1 \ldots, x_k)$, then the value of $t^{\mathcal{A}}(\bar{a})$ is $f^{\mathcal{A}}(t_1^{\mathcal{A}}(\bar{a}), \ldots, t_k^{\mathcal{A}}(\bar{a}))$.
- If φ is $t_1 = t_2$, then $\mathcal{A} \models \varphi(\bar{a})$ if and only if $t_1^{\mathcal{A}}(\bar{a})$ equals $t_2^{\mathcal{A}}(\bar{a})$.
- If φ is $R(x_1, \ldots, x_k)$, then $\mathcal{A} \models \varphi(\bar{a})$ if and only if $(t_1^{\mathcal{A}}, \ldots, t_k^{\mathcal{A}}) \in R^{\mathcal{A}}$.
- $\mathcal{A} \models \neg\varphi(\bar{a})$ if and only if $\varphi(\bar{a})$ is false in $\mathcal{A}$.
- $\mathcal{A} \models \varphi_1 \vee \varphi_2$ if and only if $\mathcal{A} \models \varphi_1(\bar{a})$ or $\mathcal{A} \models \varphi_2(\bar{a})$.
- $\mathcal{A} \models \varphi_1 \wedge \varphi_2$ if and only if $\mathcal{A} \models \varphi_1(\bar{a})$ and $\mathcal{A} \models \varphi_2(\bar{a})$.
- If $\psi(\bar{x})$ is $\exists y\varphi(y, \bar{x})$, $\mathcal{A} \models \psi(\bar{a})$ if and only if $\mathcal{A} \models \varphi(a', \bar{a})$ for some $a' \in A$.
- If $\psi(\bar{x})$ is $\forall y\varphi(y, \bar{x})$, $\mathcal{A} \models \psi(\bar{a})$ if and only if $\mathcal{A} \models \varphi(a', \bar{a})$ for all $a' \in A$.

2.4 First Order Logic with Aggregation

We build upon the first order logic definition a two-sorted version with aggregation. We thus assume the existence of two disjoint and countably infinite sets of *key-variables* and *value-variables*. The former are denoted $x, y, z, \ldots$ and the latter $\mathrm{i}, j, k, \ldots$. In order to cope with the demands of the extension functions used by LARA, we allow the language to be parameterized by a collection Ψ of user-defined relations R of some sort $(\bar{K}, \bar{V})$. For each $R \in \Psi$ we blur the distinction between the symbol R and its interpretation over $\mathsf{Keys}^{|\bar{K}|} \times \mathsf{Values}^{|\bar{V}|}$.

2.4.1 Syntax

The language contains terms of two sorts.

- *Key-terms*: Composed exclusively by the key-variables $x, y, z \ldots$.

- *Value-terms*: Composed by the constants of the form $0_\oplus$, for each aggregate operator $\oplus$, the value-variables $\mathrm{i}, j, \ldots$, and the *aggregation terms* defined next. Let $\tau(\bar{x}, \bar{y}, \bar{\mathrm{i}}, \bar{j})$ be a value-term mentioning only key-variables among those in $(\bar{x}, \bar{y})$ and value-variables among those in $(\bar{\mathrm{i}}, \bar{j})$, and $\phi(\bar{x}, \bar{y}, \bar{\mathrm{i}}, \bar{j})$ a formula whose *free* key- and value-variables are those in $(\bar{x}, \bar{y})$ and $(\bar{\mathrm{i}}, \bar{j})$, respectively (i.e., the variables that do not appear under the scope of a quantifier). Then for each aggregate operator $\oplus$ we have that

$$\tau'(\bar{x}, \bar{\mathrm{i}}) \;:=\; \mathsf{Agg}_\oplus \bar{y}, \bar{j}\, \big(\tau(\bar{x}, \bar{y}, \bar{\mathrm{i}}, \bar{j}), \phi(\bar{x}, \bar{y}, \bar{\mathrm{i}}, \bar{j})\big) \tag{2.5}$$

  is a value-term whose free variables are those in $\bar{x}$ and $\bar{\mathrm{i}}$.

Let Ψ be a set of relations R as defined above. The set of formulas in the language $\mathrm{FO}_{\mathsf{Agg}}(\Psi)$ over schema σ is inductively defined as follows:

- Atoms $\bot$, $x = y$, and $\iota = \kappa$ are formulas, for x, y key-variables and ι, κ value-terms.
- If $R[\bar{K}, \bar{V}] \in \sigma \cup \Psi$, then $R(\bar{x}, \bar{\iota})$ is a formula, where $\bar{x}$ is a tuple of key-variables of the same arity as $\bar{K}$ and $\bar{\iota}$ is a tuple of value-terms of the same arity as $\bar{V}$.
- If ϕ, ψ are formulas, then $(\neg\phi)$, $(\phi \vee \psi)$, $(\phi \wedge \psi)$, $\exists x \phi$, and $\exists \mathrm{i} \phi$ are formulas, where x and i are key- and value-variables, respectively.

Before getting into the semantics, we present a example which illustrates what this extension can achieve.

Example 2.5 *Lets say we have a ternary* FO *relation U whose tuples are (n, s, g), where n stands for the name of a student, s for a subject, and g for their grade in that subject. Suppose we want to have the average grade of each student. This is given by the following* $\mathrm{FO}_{\mathsf{Agg}}$ *formula $\varphi(n, a)$:*

$$\big(\exists s \exists g\; U(n, s, g)\big) \wedge \big(a = \mathsf{Agg}_{\mathsf{AVG}}(s, g)\, \big(g, U(n, s, g)\big)\big)$$

2.4.2 Semantics

We now define the semantics of $\mathrm{FO}_{\mathsf{Agg}}(\Psi)$. Let D be a LARA database and η an *assignment* that interprets each key-variable x as an element $\eta(x) \in \mathsf{Keys}$ and value-variable i as an element $\eta(\mathrm{i}) \in \mathsf{Values}$. If $\tau(\bar{x}, \bar{\mathrm{i}})$ is a value-term only mentioning variables in $(\bar{x}, \bar{\mathrm{i}})$, we write $\tau^D(\eta(\bar{x}, \bar{\mathrm{i}}))$ for the *interpretation* of τ over D when variables are interpreted according to η. Also, if $\phi(\bar{x}, \bar{\mathrm{i}})$ is a formula of the logic whose free key- and value-variables are those in $(\bar{x}, \bar{\mathrm{i}})$, we write $D \models \phi(\eta(\bar{x}, \bar{\mathrm{i}}))$ if D *satisfies* ϕ when $\bar{x}, \bar{\mathrm{i}}$ is interpreted according to η, and ϕ^D for the set of tuples $\eta(\bar{x}, \bar{\mathrm{i}})$ such that $D \models \phi(\eta(\bar{x}, \bar{\mathrm{i}}))$ for some assignment η..

The notion of satisfaction is inherited from the semantics of two-sorted FO. The notion of interpretation, on the other hand, requires explanation for the case of value-terms. Let η be an assignment as defined above. Constants $0_\oplus$ are interpreted as themselves and value-variables are interpreted over Values according to η. Consider now an aggregate term of the form (2.5). Let D be a LARA database and assume that $\eta(\bar{x}) = \bar{k}$, for $\bar{k} \in \mathsf{Keys}^{|\bar{x}|}$, and $\eta(\bar{\mathrm{i}}) = \bar{v}$, for $\bar{v} \in \mathsf{Values}^{|\bar{\mathrm{i}}|}$. Let $(\bar{k}'_1, \bar{v}'_1), (\bar{k}'_2, \bar{v}'_2), \ldots$, be an enumeration of all tuples

$(\bar{k}', \bar{v}') \in \mathsf{Keys}^{|\bar{y}|} \times \mathsf{Values}^{|\bar{j}|}$ such that $D \models \phi(\bar{k}, \bar{k}', \bar{v}, \bar{v}')$, i.e. there is an assignment η' that coincides with η over all variables in $(\bar{x}, \bar{\imath})$ and satisfies $\eta'(\bar{y}, \bar{j}) = (\bar{k}', \bar{v}')$. Then

$$\tau'(\eta(\bar{x}, \bar{\imath})) \;=\; \tau'(\bar{k}, \bar{v}) \;:=\; \bigoplus \{\!\{\tau(\bar{k}, \bar{k}'_1, \bar{v}, \bar{v}'_1), \tau(\bar{k}, \bar{k}'_2, \bar{v}, \bar{v}'_2), \dots\}\!\} \in \mathsf{Values}.$$

Notice that the aggregation of $\mathrm{FO}_{\mathsf{Agg}}$ in essence does the same that the Solve operator.

Chapter 3

Expressive Completeness of LARA with respect to $\mathrm{FO}_{\mathsf{Agg}}$

Having our language and tool defined, we proceed to measure the expressiveness of LARA. In order to do this we first have to notice that the user-defined functions can be very powerful, as the counting words example showed. For this, we parameterize LARA by a set of user-defined functions available to parameterize the *extend* operator. After this, we proceed to capture the expressiveness of LARA by the one of $\mathrm{FO}_{\mathsf{Agg}}$, which means that every expression of LARAcan be translated to a $\mathrm{FO}_{\mathsf{Agg}}$ formula and vice versa.

3.1 From LARA to $\mathrm{FO}_{\mathsf{Agg}}$

We prove that LARA(Ω) has the same expressive power as a suitable restriction of $\mathrm{FO}_{\mathsf{Agg}}(\Psi_\Omega)$, where Ψ_Ω is a set that contains relations that represent the extension functions in Ω. In order to achieve this, we have that for every extension function $f \in \Omega$ of sort $(\bar{K}, \bar{V}) \mapsto (\bar{K}', \bar{V}')$, there is a relation $R_f \subseteq \mathsf{Keys}^{|\bar{K}|+|\bar{K}'|} \times \mathsf{Values}^{|\bar{V}|+|\bar{V}'|}$ in Ψ_Ω such that for every $(\bar{k}, \bar{v}) \in \mathsf{Keys}^{|\bar{K}|} \times \mathsf{Values}^{|\bar{V}|}$:

$$f(\bar{k}, \bar{v}) \;=\; \{(\bar{k}', \bar{v}') \mid (\bar{k}, \bar{k}', \bar{v}, \bar{v}') \in R_f\}.$$

We show first that the expressive power of LARA(Ω) is bounded by that of $\mathrm{FO}_{\mathsf{Agg}}(\Psi_\Omega)$.

Theorem 3.1 *For every expression* $\mathrm{e}[\bar{K}, \bar{V}]$ *of* LARA(Ω)*, there is a formula* $\phi_\mathrm{e}(\bar{x}, \bar{\imath})$ *of* $\mathrm{FO}_{\mathsf{Agg}}(\Psi_\Omega)$ *such that* $\mathrm{e}^D = \phi_\mathrm{e}^D$*, for every* LARA *database* D*.*

PROOF. By induction on e.

- If $\mathrm{e} = \emptyset$, then $\phi_\mathrm{e} = \bot$.
- If $\mathrm{e} = R[\bar{K}, \bar{V}]$, for $R \in \sigma$, then $\phi_\mathrm{e}(\bar{x}, \bar{\imath}) = R(\bar{x}, \bar{\imath})$, where $\bar{x}$ and $\bar{\imath}$ are tuples of distinct key- and value-variables of the same arity as $\bar{K}$ and $\bar{V}$, respectively.

- Consider the expression $\mathrm{e}[\bar{K}_1 \cup \bar{K}_2, \bar{V}_1 \cup \bar{V}_2] = \mathrm{e}_1[\bar{K}_1, \bar{V}_1] \bowtie_{\oplus} \mathrm{e}_2[\bar{K}_2, \bar{V}_2]$, and assume that $\phi_{\mathrm{e}_1}(\bar{x}_1, \bar{\mathrm{i}}_1)$ and $\phi_{\mathrm{e}_2}(\bar{x}_2, \bar{\mathrm{i}}_2)$ are the formulas obtained for $\mathrm{e}_1[\bar{K}_1, \bar{V}_1]$ and $\mathrm{e}_2[\bar{K}_2, \bar{V}_2]$, respectively, by induction hypothesis. Let us first define

$$\alpha_{\mathrm{e}}(\bar{x}, \bar{\mathrm{i}}) := \exists \bar{x}_1, \bar{x}_2 \exists \bar{\mathrm{i}}_1, \bar{\mathrm{i}}_2 \big(\phi_{\mathrm{e}_1}(\bar{x}_1, \bar{\mathrm{i}}_1) \wedge \phi_{\mathrm{e}_2}(\bar{x}_2, \bar{\mathrm{i}}_2) \wedge \chi_{\bar{K}_1 \cap \bar{K}_2}(\bar{x}, \bar{x}_1, \bar{x}_2) \wedge \mu_{\bar{V}_1 \cup \bar{V}_2}(\bar{\mathrm{i}}, \bar{\mathrm{i}}_1, \bar{\mathrm{i}}_2) \big),$$

assuming that $\bar{x}_1$ and $\bar{x}_2$ share no variables, the same holds for $\bar{\mathrm{i}}_1$ and $\bar{\mathrm{i}}_2$, the formula $\chi_{\bar{K}_1 \cap \bar{K}_2}$ expresses that $\bar{x}$ is the result of "joining" $\bar{x}_1$ and $\bar{x}_2$ with respect to positions that represent same key-attributes $K \in \bar{K}_1 \cap \bar{K}_2$, and $\mu_{\bar{V}_1 \cup \bar{V}_2}(\bar{\mathrm{i}}, \bar{\mathrm{i}}_1, \bar{\mathrm{i}}_2)$ expresses that $\bar{\mathrm{i}}$ is any tuple that satisfies one of two following statements:

  - $\bar{\mathrm{i}} = (\mathrm{pad}_{\oplus}^{\bar{V}_2}(\bar{\mathrm{i}}_1), 0)$, i.e., $\bar{\mathrm{i}}[|\bar{\mathrm{i}}|] = 0$, and for every $j \in [|\bar{\mathrm{i}}| - 1]$ it is the case that $\bar{\mathrm{i}}[j] = \bar{\mathrm{i}}_1[j]$, if the jth position of $\bar{\mathrm{i}}$ represents a value-attribute V in $\bar{V}_1$, and $\bar{\mathrm{i}}[j] = 0_{\oplus}$ otherwise; or
  - $\bar{\mathrm{i}} = (\mathrm{pad}_{\oplus}^{\bar{V}_1}(\bar{\mathrm{i}}_2), 1)$, i.e., $\bar{\mathrm{i}}[|\bar{\mathrm{i}}|] = 1$, and for every $j \in [|\bar{\mathrm{i}}| - 1]$ it is the case that $\bar{\mathrm{i}}[j] = \bar{\mathrm{i}}_2[j]$, if the jth position of $\bar{\mathrm{i}}$ represents a value-attribute V in $\bar{V}_2$, and $\bar{\mathrm{i}}[j] = 0_{\oplus}$ otherwise.

  The tuple $\bar{\mathrm{i}}$ contains a 0 or a 1 at the end to simply distinguish whether we are padding $\bar{\mathrm{i}}_1$ or $\bar{\mathrm{i}}_2$, respectively. Notice that both 0 and 1 are constants in the language of $\mathrm{FO}_{\mathsf{Agg}}$, as they correspond to the neutral values for the aggregate operations of addition and multiplication, respectively.

  The evaluation of α_{e} on e_1^D and e_2^D, for D a Lara database, thus contains all tuples $(\bar{k}_1 \cup \bar{k}_2, \bar{v}_1, 0)$ and $(\bar{k}_1 \cup \bar{k}_2, \bar{v}_2, 1)$ such that $\bar{k}_1$ and $\bar{k}_2$ are compatible tuples in e_1^D and e_2^D, respectively, and it is the case that $\bar{v}_1 = \mathrm{pad}_{\oplus}^{\bar{V}_2}(\mathrm{e}_1^D(\bar{k}_1))$ and $\bar{v}_2 = \mathrm{pad}_{\oplus}^{\bar{V}_1}(\mathrm{e}_2^D(\bar{k}_2))$. We then define the formula

$$\phi_{\mathrm{e}}(\bar{x}, \bar{\mathrm{i}}) := \exists \bar{j} \alpha_{\mathrm{e}}(\bar{x}, \bar{j}) \wedge \bigwedge_{\ell \in [|\bar{j}| - 1]} \bar{\mathrm{i}}[\ell] = \mathsf{Agg}_{\oplus} \bar{j} \, (\bar{j}[\ell], \alpha_{\mathrm{e}}(\bar{x}, \bar{j})).$$

  Clearly, the evaluation of ϕ_{e} on e_1^D and e_2^D, for D a Lara database, contains all tuples $(\bar{k}_1 \cup \bar{k}_2, \bar{v}_1 \oplus \bar{v}_2)$ such that $\bar{k}_1$ and $\bar{k}_2$ are compatible tuples in e_1^D and e_2^D, respectively, and it is the case that $\bar{v}_1 = \mathrm{pad}_{\oplus}^{\bar{V}_2}(\mathrm{e}_1^D(\bar{k}_1))$ and $\bar{v}_2 = \mathrm{pad}_{\oplus}^{\bar{V}_1}(\mathrm{e}_2^D(\bar{k}_2))$.
- Consider the expression $\mathrm{e}[\bar{K}_1 \cap \bar{K}_2, \bar{V}_1 \cup \bar{V}_2] = \mathrm{e}_1[\bar{K}_1, \bar{V}_1] \between_{\oplus} \mathrm{e}_2[\bar{K}_2, \bar{V}_2]$, and assume that $\phi_{\mathrm{e}_1}(\bar{x}_1, \bar{\mathrm{i}}_1)$ and $\phi_{\mathrm{e}_2}(\bar{x}_2, \bar{\mathrm{i}}_2)$ are the formulas obtained for $\mathrm{e}_1[\bar{K}_1, \bar{V}_1]$ and $\mathrm{e}_2[\bar{K}_2, \bar{V}_2]$, respectively, by induction hypothesis. Let us define the formula

$$\alpha_1(\bar{x}, \bar{\mathrm{i}}) := \exists \bar{y} \exists \bar{j} \, \phi_{\mathrm{e}_1}(\bar{x}, \bar{y}, \bar{j}) \wedge \mathrm{i}[j' + 1] = 0 \bigwedge_{\ell \in [|\bar{j}'|]} \mathrm{i}[\ell] = \mathsf{Agg}_{\oplus} \bar{y}, \bar{j}' \, \big(\bar{j}[\ell], \exists \bar{j} (\phi_{\mathrm{e}_1}(\bar{x}, \bar{y}, \bar{j}) \wedge j' = \mathrm{pad}_{\oplus}^{V_2}(\bar{j})) \big),$$

  where we assume that $\bar{x}$ is positioned over the attributes representing $\bar{K}_1 \cap \bar{K}_2$ in ϕ_{e_1}. It is easy to see that when evaluated over e_1^D, for D a Lara database, the result of α_1 corresponds to the set of tuples $(\bar{k}, \bar{v}, 0)$ such that $(\bar{k}, \bar{v})$ belongs to the multiset

$$\mathsf{Solve}_{\oplus} \{\{ (\bar{k}', \bar{v}') \mid \bar{k}' = \bar{k}_1 {\downarrow}_{\bar{K}_1 \cap \bar{K}_2} \text{ and } \bar{v}' = \mathrm{pad}_{\oplus}^{V_2}(\mathrm{e}_1^D(\bar{k}_1)) \text{ for some } \bar{k}_1 \in \mathrm{e}_1^D, \text{ or } \bar{k}' = \bar{k}_2 {\downarrow}_{\bar{K}_1 \cap \bar{K}_2} \text{ and } \bar{v}' = \mathrm{pad}_{\oplus}^{V_1}(\mathrm{e}_2^D(\bar{k}_2)) \text{ for some } \bar{k}_2 \in \mathrm{e}_2^D \}\}.$$

Analogously, we can then define a formula

$$\alpha_2(\bar{x},\bar{\mathrm{i}}) \;:=\; \exists\bar{y}\exists\bar{j}\,\phi_{\mathrm{e}_2}(\bar{x},\bar{y},\bar{j}) \,\wedge\, \mathrm{i}[j'+1]=1 \\ \bigwedge_{\ell\in[|\bar{j}'|]} \mathrm{i}[\ell] = \mathsf{Agg}_{\oplus}\bar{y},\bar{j}'\,\big(\,\bar{j}[\ell],\,\exists\bar{j}(\phi_{\mathrm{e}_2}(\bar{x},\bar{y},\bar{j})\wedge j'=\mathrm{pad}^{V_1}_{\oplus}(\bar{j}))\,\big),$$

with a similar interpretation over e_2^D (notice, however, that tuples in the interpretation of α_2 are of the form $(\bar{k},\bar{v},1)$).

Finally, we define

$$\phi_{\mathrm{e}}(\bar{x},\bar{\mathrm{i}}) \;:=\; \exists\bar{j}\,\big(\,\alpha_1(\bar{x},\bar{j})\,\vee\,\alpha_2(\bar{x},\bar{j})\,\big)\,\wedge\,\bigwedge_{\ell\in[|\bar{j}|-1]}\bar{\mathrm{i}}[\ell]=\mathsf{Agg}_{\oplus}\bar{j}\,\big(\,\bar{j}[\ell],\,\alpha_1(\bar{x},\bar{j})\vee\alpha_2(\bar{x},\bar{j})\,\big).$$

- Consider the expression $\mathrm{e}[\bar{K}\cup\bar{K}',\bar{V}'] = \mathsf{Ext}_f\,\mathrm{e}_1[\bar{K},\bar{V}]$, where f is of sort $(\bar{K},\bar{V})\mapsto(\bar{K}',\bar{V}')$, and assume that $\phi_{\mathrm{e}_1}(\bar{x}_1,\bar{\mathrm{i}}_1)$ is the formula obtained for $\mathrm{e}_1[\bar{K}_1,\bar{V}_1]$ by induction hypothesis. Then

$$\phi_{\mathrm{e}}(\bar{x}_1,\bar{x},\bar{\mathrm{i}}) \;:=\; \exists\bar{\mathrm{i}}_1\,\big(\,\phi_{\mathrm{e}_1}(\bar{x}_1,\bar{\mathrm{i}}_1)\,\wedge\,R_f(\bar{x}_1,\bar{x},\bar{\mathrm{i}}_1,\bar{\mathrm{i}})\,\big).$$

This finishes the proof of the theorem.

□

3.2 From $\mathrm{FO}_{\mathsf{Agg}}$ to LARA

We now prove the converse, which is a that all $\mathrm{FO}_{\mathsf{Agg}}$ formulas have a translation to a LARA expression. To do this, however, we need the following two restrictions on $\mathrm{FO}_{\mathsf{Agg}}$ formulas, which ensure that the semantics of the formulas considered matches that of LARA. In particular, we need to ensure that the evaluation of $\mathrm{FO}_{\mathsf{Agg}}$ formulas is safe and that they only output associative tables.

- *Safety.* Formulas of $\mathrm{FO}_{\mathsf{Agg}}(\Psi_\Omega)$ are not necessarily safe, i.e., their evaluation can have infinitely many tuples (think, e.g., of the formula $\mathrm{i}=j$, for i,j value-variables, or $R_f(\bar{x},\bar{x}',\bar{\mathrm{i}},\bar{\mathrm{i}}')$, for $R_f\in\Psi_\Omega$). While safety issues relating to the expressive completeness of relational algebra with respect to first order logic are often resolved by relativizing all operations to the *active* domain of databases (i.e., the set of elements mentioned in relations in databases), such a restriction only makes sense for keys in our context, but not for values. In fact, several useful formulas compute a new value for a variable based on some aggregation terms over precomputed data (see, e.g., the translations of the join and union operator of LARA into $\mathrm{FO}_{\mathsf{Agg}}$ provided in the proof of Theorem 3.1).

  To overcome this issue we develop a suitable syntactic restriction of the logic that can only express safe queries. This is achieved by "guarding" the application of value-term equalities, relations encoding extension functions, and Boolean connectives as follows.
  - We only allow equality of value-terms to appear in formulas of the form

$$\phi(\bar{x},\bar{x}',\bar{\mathrm{i}},\bar{\mathrm{i}}')\,\wedge\,j=\tau(\bar{x},\bar{\mathrm{i}}),$$

where j is a value-variable that does not necessarily appear in $\bar{\imath}$ and τ is an arbitrary value-term whose value only depends on $(\bar{x}, \bar{\imath})$. This formula computes the value of the aggregated term τ over the precomputed evaluation of ϕ, and then output it as the value of j. In the same vein, atomic formulas of the form $R(\bar{x}, \bar{\iota})$ must satisfy that every element in $\bar{\iota}$ is a value-variable.

- Relations $R_f \in \Psi_\Omega$ can only appear in formulas of the form

$$\phi(\bar{x}, \bar{y}, \bar{\imath}, \bar{j}) \wedge R_f(\bar{x}, \bar{x}', \bar{\imath}, \bar{\imath}'),$$

  i.e., we only allow to compute the set $f(\bar{x}, \bar{\imath})$ for specific precomputed values of $(\bar{x}, \bar{\imath})$.
- Also, negation is only allowed in the restricted form $\phi(\bar{x}, \bar{\imath}) \wedge \neg\psi(\bar{x}, \bar{\imath})$ and disjunction in the form $\phi(\bar{x}, \bar{\imath}) \vee \psi(\bar{x}, \bar{\imath})$, i.e., when formulas have exactly the same free variables. Although we did not use the negation in the translation from LARA to FO_{Agg}, we have to take this restriction in order to translate all formulas from this fragment of FO_{Agg} to LARA. The reason we did not use negation, was because this is implicitly encoded in the extension functions.

We denote the resulting language as $\text{FO}^{\text{safe}}_{\mathsf{Agg}}(\Psi_\Omega)$. These restrictions are meaningful, as the translation from $\text{LARA}(\Omega)$ to $\text{FO}_{\mathsf{Agg}}(\Omega_\Psi)$ given in the proof of Theorem 3.1 always builds a formula in $\text{FO}^{\text{safe}}_{\mathsf{Agg}}(\Psi_\Omega)$.

- *Key constraints.* We also need a restriction on the semantics of $\text{FO}^{\text{safe}}_{\mathsf{Agg}}(\Psi_\Omega)$ formulas that ensures that the evaluation of any such formula on a LARA database is an associative table. For doing this, we assume that every formula ϕ of $\text{FO}^{\text{safe}}_{\mathsf{Agg}}(\Psi_\Omega)$ is given in conjunction with an aggregate operator $\oplus$ over Values. We denote this as $op(\phi) = \oplus$. The operator $\oplus$ is used to "solve" the key violations introduced by the evaluation of ϕ. In particular, then, we are interested in capturing the semantics of $\mathsf{Solve}_\oplus(\phi^D)$ over every LARA database D. This is the typical key restriction of databases, where we allow a particular set of keys to only appear once per relation. What we are doing here is recovering this restriction using $\oplus$ as an aggregate operator when some keys appear more than once in a relation. For a formula ϕ of FO_{Agg} and its operator $op(\phi)$, we say that $\mathsf{Solve}_\oplus(\phi)$ is the formula with the key restriction applied.

In addition, we need one natural assumption about the extension functions that LARA can use. In particular, we need these functions to be able to express traditional relational algebra operations that are not included in the core of LARA; namely, copying attributes, selecting rows based on (in)equality, and projecting over value-attributes (the projection over key-attributes, in turn, can be expressed with the union operator). Formally, we assume that Ω contains the following families of extension functions.

- $\mathsf{copy}_{\bar{K},\bar{K}'}$ and $\mathsf{copy}_{\bar{V},\bar{V}'}$, for $\bar{K}, \bar{K}'$ tuples of key-attributes of the same arity and $\bar{V}, \bar{V}'$ tuples of value-attributes of the same arity. Function $\mathsf{copy}_{\bar{K},\bar{K}'}$ takes as input a tuple $t = (\bar{k}, \bar{v})$ of sort $(\bar{K}_1, \bar{V})$, where $\bar{K} \subseteq \bar{K}_1$ and $\bar{K}' \cap \bar{K}_1 = \emptyset$, and produces a tuple $t' = (\bar{k}, \bar{k}', \bar{v})$ of sort $(\bar{K}_1, \bar{K}', \bar{V})$ such that $t'(\bar{K}') = t(\bar{K})$, i.e., $\mathsf{copy}_{\bar{K},\bar{K}'}$ copies the value of attributes $\bar{K}$ in the new attributes $\bar{K}'$. Analogously, we define the function $\mathsf{copy}_{\bar{V},\bar{V}'}$.
- $\mathsf{add}_{V,0_\oplus}$, for V an attribute-value and $\oplus$ an aggregate operator. Function $\mathsf{add}_{V,0_\oplus}$ takes as input a tuple $t = (\bar{k}, \bar{v}')$ of sort $(\bar{K}, \bar{V}')$, where $V \notin \bar{V}'$, and produces a tuple $t' = (\bar{k}, \bar{v}', 0_\oplus)$ of sort $(\bar{K}, \bar{V}', V)$, i.e., $\mathsf{add}_{V,0_\oplus}$ adds a new value-attribute V that always takes value $0_\oplus$.

- $\mathsf{eq}_{\bar{K},\bar{K}'}$ and $\mathsf{eq}_{\bar{V},\bar{V}'}$, for $\bar{K},\bar{K}'$ tuples of key-attributes of the same arity and $\bar{V},\bar{V}'$ tuples of value-attributes of the same arity. The function $\mathsf{eq}_{\bar{K},\bar{K}'}$ takes as input a tuple $t = (\bar{k},\bar{v})$ of sort $(\bar{K}_1,\bar{V})$, where $\bar{K},\bar{K}' \subseteq \bar{K}_1$, and produces as output the tuple $t' = (\bar{k},\bar{v})$ of sort $(\bar{K}_1,\bar{V})$, if $t(\bar{K}) = t(\bar{K}')$, and the empty associative table otherwise. Hence, this function acts as a filter over an associative table of sort $(\bar{K}_1,\bar{V})$, extending only those tuples t such that $t(\bar{K}) = t(\bar{K}')$. Analogously, we define the function $\mathsf{eq}_{\bar{V},\bar{V}'}$.
- In the same vein, extension functions $\mathsf{neq}_{\bar{K},\bar{K}'}$ and $\mathsf{neq}_{\bar{V},\bar{V}'}$, for $\bar{K},\bar{K}'$ tuples of key-attributes of the same arity and $\bar{V},\bar{V}'$ tuples of value-attributes of the same arity. These are defined exactly as $\mathsf{eq}_{\bar{K},\bar{K}'}$ and $\mathsf{eq}_{\bar{V},\bar{V}'}$, only that we now extend only those tuples t such that $t(\bar{K}) \neq t(\bar{K}')$ and $t(\bar{V}) \neq t(\bar{V}')$, respectively.
- The projection $\pi_{\bar{V}}$, for $\bar{V}$ a tuple of value-attributes, takes as input a tuple $(\bar{k},\bar{v}')$ of sort $(\bar{K},\bar{V}')$, where $\bar{V} \subseteq \bar{V}'$, and outputs the tuple $(\bar{k},\bar{v})$ of sort $(\bar{K},\bar{V})$ such that $\bar{v} = \bar{v}'{\downarrow_{\bar{V}}}$.

Finally we need a lemma to aid our results.

Lemma 3.2 *For every formula $\alpha(\bar{x},\bar{\imath})$ with $op(\alpha) = \oplus_\alpha$ and value-term $\lambda(\bar{x},\bar{\imath})$ of $\mathrm{FO}^{\mathrm{safe}}_{\mathsf{Agg}}(\Psi_\Omega)$, there is an expression $\mathrm{e}_{\alpha,\lambda}[\bar{K},\bar{V},V_1]$ of $\textsc{Lara}(\Omega)$ such that for every $\textsc{Lara}$ database D it holds that $(\bar{k},\bar{v},v_1) \in \mathrm{e}^D_{\alpha,\lambda}$ iff $(\bar{k},\bar{v}) \in \mathsf{Solve}_{\oplus_\alpha}(\alpha(D))$ and $\lambda(\bar{k},\bar{v}) = v_1$.*

Proof. We prove this by induction on λ.

- Consider the base case when $\lambda = \ell$, for some variable $\ell \in \bar{\imath}$. Then $\mathrm{e}_{\alpha,\lambda}[\bar{K},\bar{V},V_1] := \mathsf{copy}_{V,V_1}\mathrm{e}_{\alpha,\oplus_\alpha}[\bar{K},\bar{V}]$, where V is the value-attribute corresponding to variable ℓ in $\bar{V}$ and V_1 is a fresh value-attribute.
- Consider now the base case when $\lambda = 0_\oplus$. Then $\mathrm{e}_{\alpha,\lambda}[\bar{K},\bar{V},V_1] := \mathsf{add}_{V_1,0_{\oplus_\alpha}}\mathrm{e}_{\alpha,\oplus_\alpha}[\bar{K},\bar{V}]$, where V_1 is a value-attribute not in $\bar{V}$.
- For the induction hypothesis, assume that

$$\lambda(\bar{x},\bar{\imath}) \;=\; \mathsf{Agg}_\oplus \bar{x}',\bar{\imath}' \left(\lambda'(\bar{x},\bar{x}',\bar{\imath},\bar{\imath}'),\alpha'(\bar{x},\bar{x}',\bar{\imath},\bar{\imath}')\right),$$

  for a formula α' and a value-term λ' of $\mathrm{FO}^{\mathrm{safe}}_{\mathsf{Agg}}(\Psi_\Omega)$. In addition, assume that

$$\mathrm{e}_{\alpha',\lambda'}[\bar{K},\bar{K}',\bar{V},\bar{V}',V_1]$$

  is the formula that is obtained for α' and λ' by induction hypothesis. Then

$$\mathrm{e}_{\alpha,\lambda}[\bar{K},\bar{V},V_1] \;:=\; \pi^{\oplus}_{\bar{K}}\,\pi_{\bar{V}_1}\big(\,\mathrm{e}_{\alpha',\lambda'}[\bar{K},\bar{K}',\bar{V},\bar{V}',V_1]\,\big) \bowtie \mathrm{e}_{\alpha}[\bar{K},\bar{V}],$$

  where $\bowtie$ does not need to specify which aggregate operator we are using.

□

We now establish our result.

Theorem 3.3 *Let us assume that Ω contains all extension functions specified above. For every formula $\phi(\bar{x},\bar{\imath})$ of $\mathrm{FO}^{\mathrm{safe}}_{\mathsf{Agg}}(\Psi_\Omega)$ with $op(\phi) = \oplus$ there is a $\textsc{Lara}(\Omega)$ expression $\mathrm{e}_{\phi,\oplus}[\bar{K},\bar{V}]$ such that $\mathrm{e}^D_{\phi,\oplus} = \phi^D$, for each $\textsc{Lara}$ database D.*

PROOF. When f is one of the distinguished extension functions f defined above, we abuse notation and write simply f instead of Ext_f. We first define several useful operations and expressions.

- The projection $\pi^{\oplus}_{\bar{K}}\,\mathrm{e}$ over keys with respect to aggregate operator $\oplus$, defined as $\bar{\sum}^{\bar{K}}_{\oplus}\,\mathrm{e}$. Notice that this removes key-, but not value-attributes from e, i.e., if e is of sort $[\bar{K}', \bar{V}]$ then $\pi^{\oplus}_{\bar{K}}\,\mathrm{e}$ is of sort $[\bar{K}, \bar{V}]$.
- The *rename* operator $\rho_{\bar{K}\to\bar{K}'}\,\mathrm{e}$ as $\pi_{\bar{K}}\,(\mathsf{copy}_{\bar{K},\bar{K}'}\,\mathrm{e})$, where π has no superscript $\oplus$ as no aggregation is necessary in this case. This operation simply renames the key-attributes $\bar{K}$ to a fresh set of key-attributes $\bar{K}'$. Analogously, we define $\rho_{\bar{V}\to\bar{V}'}\,\mathrm{e}$.
- The *active domain* expression $\mathrm{e}_{\mathsf{ActDom}}$, which takes as input a LARA database D and returns all elements $k \in \mathsf{Keys}$ that appear in some fact of D. It is defined as follows. First choose a key attribute not present in any table of D; say it is Z. For each $R[\bar{K}, \bar{V}] \in \sigma$ we define an expression $R^{\mathsf{Keys}} := \pi_{\emptyset}\,R$, which removes all attribute-values in $\bar{V}$ from R. For each $K \in \bar{K}$ we then define $R^{\mathsf{Keys}}_K := \pi_K\,R^{\mathsf{Keys}}$ as the set of keys that appear in the position of attribute K in $R[\bar{K}, \bar{V}]$ (no need to specify superscript $\oplus$ on π in this case). Finally, we define $\mathrm{e}^R_{\mathsf{ActDom}} := \bar{\sum}_{K\in\bar{K}}\,\rho_{K\to Z}R^{\mathsf{Keys}}_K$ and $\mathrm{e}_{\mathsf{ActDom}} := \bar{\sum}_{R\in\sigma}\,\mathrm{e}^R_{\mathsf{ActDom}}$.

We now prove the theorem by induction on a formula ϕ of $\mathrm{FO}_{\mathsf{Agg}}$.

- If $\phi = \bot$ then $\mathrm{e}_{\phi,\oplus} = \emptyset$ for every aggregate operator $\oplus$.
- If $\phi = (x = y)$, for x, y key-variables, then $\mathrm{e}_{\phi,\oplus}[K, K'] := \mathsf{eq}_{K,K'}\big(\mathrm{e}_{\mathsf{ActDom}}[K] \bowtie \rho_{K\to K'}\mathrm{e}_{\mathsf{ActDom}}[K]\big)$ for every aggregate operator $\oplus$.
- Consider now $\phi = R(\bar{x}, \bar{\mathrm{i}})$, for $R \in \sigma$. We assume all variables in $\bar{x}$ and $\bar{\mathrm{i}}$, respectively, to be pairwise distinct, as repetition of variables can always be simulated with equalities. Then $\mathrm{e}_{\phi,\oplus}[\bar{K}, \bar{V}] := R[\bar{K}, \bar{V}]$.
- Assume that $op(\phi) = \oplus$ and that $\phi = \phi'(\bar{x}', \bar{\mathrm{i}}') \wedge \neg\phi''(\bar{x}', \bar{\mathrm{i}}')$, for formulas ϕ', ϕ'' of $\mathrm{FO}^{\mathrm{safe}}_{\mathsf{Agg}}(\Psi_\Omega)$ such that $op(\phi') = \oplus'$ and $op(\phi'') = \oplus''$, respectively. Let $\mathrm{e}_{\phi',\oplus'}[\bar{K}', \bar{V}']$ and $\mathrm{e}_{\phi'',\oplus''}[\bar{K}', \bar{V}']$ be the expressions obtained for $(\phi', \oplus')$ and $(\phi'', \oplus'')$, respectively, by induction hypothesis. Then

  $$\mathrm{e}_{\phi,\oplus}[\bar{K}, \bar{V}] \;:=\; \mathsf{Solve}_{\oplus}(\rho_{\bar{K}'\to\bar{K}}\,\rho_{\bar{V}'\to\bar{V}}\,\pi_{\bar{V}'}\,\mathsf{neq}_{\bar{V}',\bar{V}''}\,\big(\mathrm{e}_{\phi',\oplus'}[\bar{K}', \bar{V}'] \;\bowtie\; \rho_{\bar{V}'\to\bar{V}''}(\mathrm{e}_{\phi'',\oplus''}[\bar{K}', \bar{V}'])\big)).$$

  We leave the join $\bowtie$ without subscript $\oplus$, as no aggregation is needed.
- Assume now that $op(\phi) = \oplus$ and that $\phi = \phi'(\bar{x}', \bar{\mathrm{i}}') \wedge \phi''(\bar{x}'', \bar{\mathrm{i}}'')$, for formulas ϕ', ϕ'' of $\mathrm{FO}^{\mathrm{safe}}_{\mathsf{Agg}}(\Psi_\Omega)$ such that $op(\phi') = \oplus'$ and $op(\phi'') = \oplus''$, respectively. Let $\mathrm{e}_{\phi',\oplus'}[\bar{K}', \bar{V}']$ and $\mathrm{e}_{\phi'',\oplus''}[\bar{K}'', \bar{V}'']$ be the expressions obtained for $(\phi', \oplus')$ and $(\phi'', \oplus'')$, respectively, by induction hypothesis. Then

  $$\mathrm{e}_{\phi,\oplus}[\bar{K}, \bar{V}] \;:=\; \mathsf{Solve}_{\oplus}(\rho_{\bar{K}'\cup\bar{K}''\to\bar{K}}\,\rho_{\bar{V}'\cup\bar{V}''\to\bar{V}}\,\pi_{\bar{V}^*}\,\mathsf{eq}_{\bar{V}'\cap V'',\bar{V}^*}\,\alpha),$$

  where α is defined as

  $$\mathrm{e}_{\phi',\oplus'}[\bar{K}', \bar{V}'] \;\bowtie\; \rho_{\bar{V}'\cap\bar{V}''\to\bar{V}^*}(\mathrm{e}_{\phi'',\oplus''}[\bar{K}'', \bar{V}'']).$$

  As before we leave the join $\bowtie$ without subscript $\oplus$, as no aggregation is needed.

- Assume that $op(\phi) = \oplus$ and that $\phi = \phi'(\bar{x}, \bar{\mathrm{i}}) \wedge k = \tau(\bar{x}, \bar{\mathrm{i}})$, for a formula ϕ' and a value-term τ of $\mathrm{FO}^{\mathrm{safe}}_{\mathsf{Agg}}(\Psi_\Omega)$ such that $op(\phi') = \oplus'$ and k a value-variable not necessarily present in $\bar{\mathrm{i}}$. We only consider the case when k is not in $\bar{\mathrm{i}}$. The other case is similar.

  We now show how to construct $\mathrm{e}_{\phi,\oplus}$ for $\phi = \phi'(\bar{x}, \bar{\mathrm{i}}) \wedge k = \tau(\bar{x}, \bar{\mathrm{i}})$. Assume $\mathrm{e}_{\phi',\oplus'}[\bar{K}, \bar{V}]$ is the expression obtained for $(\phi', \oplus')$ by induction hypothesis, and $\mathrm{e}_{\phi',\tau}[\bar{K}, \bar{V}, V_1]$ is the expression constructed for ϕ' and τ, as described above. Then

$$\mathrm{e}_{\phi,\oplus}[\bar{K}, \bar{V}, V_1] \;:=\; \mathsf{Solve}_{\oplus}(\pi_{\bar{V}}\, \mathsf{eq}_{\bar{V},\bar{V}'}\left(\mathrm{e}_{\phi',\oplus'}[\bar{K}, \bar{V}] \bowtie \rho_{\bar{V}\to\bar{V}'}(\mathrm{e}_{\phi',\tau})[\bar{K}, \bar{V}, V_1]\right)),$$

  where there is no need to specify the aggregate operator on $\bowtie$.
- The cases $\phi = \exists x \phi'(x, \bar{x}, \bar{\mathrm{i}})$ and $\phi = \exists \mathrm{i} \phi'(\bar{x}, \bar{\mathrm{i}}, \mathrm{i})$, when $op(\phi) = \oplus$, can be translated as $\pi^{\oplus}_{K}\, \mathrm{e}_{\phi',\oplus'}[K, \bar{K}, \bar{V}]$ and $\pi_{V}\, \mathrm{e}_{\phi',\oplus'}[\bar{K}, \bar{V}, V]$, respectively, assuming that $op(\phi') = \oplus'$ and $\mathrm{e}_{\phi',\oplus'}$ is the expression obtained for $(\phi', \oplus')$ by induction hypothesis.

This finishes the proof of the theorem. □

Although this result characterizes the expressive power of LARA as the one of $\mathrm{FO}_{\mathsf{Agg}}$, the proof is heavily dependant on the extension functions allowed. As these are user-defined, the proof does not shed light on the real expressive power of LARA, which is still dependant on the election of these functions.

Chapter 4

Expressiveness of LARA in terms of ML Operators

In the previous chapter we showed that LARA and FO_{Agg} have the same expressive power when the user-defined functions allowed to parameterize the *extend* operator are encoded via built-ins in the logic. Taking restrictions on the set Ω of functions allows us to completely capture the expressivity of LARA in terms of FO_{Agg}. FO_{Agg} has very useful properties that we use to prove results about the expressiveness of ML operators in LARA.

4.1 Tame LARA

We assume from now on that $\mathsf{Values} = \mathbb{Q}$. Since extension functions in Ω can a priori be arbitrary, to understand what LARA can express we first need to specify which classes of functions are allowed in Ω. In rough terms, this is determined by the operations that one can perform when comparing keys and values, respectively. We explain this below.

- Extensions of two-sorted logics with aggregate operators over a *numerical* sort $\mathcal{N}$ often permit to perform arbitrary numerical comparisons over $\mathcal{N}$ (in our case $\mathcal{N} = \mathsf{Values} = \mathbb{Q}$). It has been noted that this extends the expressive power of the language, while at the same time preserving some properties of the logic that allow to carry out an analysis of its expressiveness based on well-established techniques (see, e.g., [25]).
- In some cases in which the expressive power of the language needs to be further extended, one can also define a linear order on the non-numerical sort (which in our case is the set Keys) and then perform suitable arithmetic comparisons in terms of such a linear order. A well-known application of this technique is in the area of descriptive complexity [21].

In this section we consider only the first possibility only. That is, we allow comparing elements of $\mathsf{Values} = \mathbb{Q}$ in terms of arbitrary numerical relations. Elements of Keys, in turn, can only be compared with respect to equality. This yields a logic that is amenable for theoretical exploration – in particular, in terms of its expressive power – and that at the

same time is able to express many extension functions of practical interest (e.g., several of the functions used in examples in [19, 20]).

We design a simple logic FO(=, All) for expressing extension functions. Intuitively, the name of this logic states that it can only compare keys with respect to equality but it can compare values in terms of arbitrary numerical predicates. The formulas in the logic are standard FO formulas where the only atomic expressions allowed are of the following form:

- $x = y$, for x, y key-variables;
- $P(\mathrm{i}_1, \ldots, \mathrm{i}_k)$, for $P \subseteq \mathbb{Q}^k$ a numerical relation of arity k and $\mathrm{i}_1, \ldots, \mathrm{i}_k$ value-variables or constants of the form $0_\oplus$.

The semantics of this logic is standard. In particular, an assignment η from value-variables to $\mathbb{Q}$ *satisfies* a formula of the form $P(\mathrm{i}_1, \ldots, \mathrm{i}_k)$, for $P \subseteq \mathbb{Q}^k$, whenever $\eta(\mathrm{i}_1, \ldots, \mathrm{i}_k) \in P$.

Let $\phi(\bar{x}, \bar{y}, \bar{\mathrm{i}}, \bar{j})$ be a formula of FO(=, All). For a tuple $t = (\bar{k}, \bar{k}', \bar{v}, \bar{v}') \in \mathsf{Keys}^{|\bar{k}|+|\bar{k}'|} \times \mathsf{Values}^{|\bar{v}|+|\bar{v}'|}$ we abuse terminology and say that $\phi(\bar{k}, \bar{k}', \bar{v}, \bar{v}')$ *holds* if $D_t \models \phi(\bar{k}, \bar{k}', \bar{v}, \bar{v}')$, where D_t is the database composed exclusively by tuple t. In addition, an extension function f of sort $(\bar{K}, \bar{V}) \mapsto (\bar{K}', \bar{V}')$ is *definable* in FO(=, All), if there is a formula $\phi_f(\bar{x}, \bar{y}, \bar{\mathrm{i}}, \bar{j})$ of FO(=, All), for $|\bar{x}| = |\bar{K}|$, $|\bar{y}| = |\bar{K}'|$, $|\bar{\mathrm{i}}| = |\bar{V}|$, and $|\bar{j}| = |\bar{V}'|$, such that for every tuple $(\bar{k}, \bar{v})$ of sort $(\bar{K}, \bar{V})$ it is the case

$$f(\bar{k}, \bar{v}) \;=\; \{(\bar{k}', \bar{v}') \mid \phi(\bar{k}, \bar{k}', \bar{v}, \bar{v}') \text{ holds}\}.$$

This gives rise to the definition of the following class of extension functions:

$$\Omega_{(=,\mathsf{All})} = \{f \mid f \text{ is an extension function that is definable in FO}(=, \mathsf{All})\}.$$

Recall that extension functions only produce finite associative tables by definition, and hence only some formulas in FO(=, All) define extension functions.

The extension functions $\mathsf{copy}_{\bar{K},\bar{K}'}$, $\mathsf{copy}_{\bar{V},\bar{V}'}$, $\mathsf{add}_{V,0_\oplus}$, $\mathsf{eq}_{\bar{K},\bar{K}'}$, $\mathsf{eq}_{\bar{V},\bar{V}'}$, $\mathsf{neq}_{\bar{K},\bar{K}'}$, $\mathsf{neq}_{\bar{V},\bar{V}'}$, and $\pi_{\bar{V}}$, as defined in the previous section, are in $\Omega_{(=,\mathsf{All})}$. Next we provide more examples.

Example 4.1 *We use* $\mathrm{i}+j = k$ *and* $\mathrm{i}j = k$ *as a shorthand notation for the ternary numerical predicates of addition and multiplication, respectively. Consider first a function f that takes a tuple t of sort (K_1, K_2, V) and computes a tuple t' of sort (K_1', K_2', V') such that $t(K_1, K_2) = t'(K_1', K_2')$ and $t'(V') = 1 - t(V)$. Then f is definable in* FO(=, All) *as $\phi_f(x, y, x', y', \mathrm{i}, j) := \big(x = x' \wedge y = y' \wedge \mathrm{i} + j = 1\big)$. This function can be used, e.g., to interchange 0s and 1s in a Boolean matrix.*

Consider now a function g that takes a tuple t of sort (K, V_1, V_2) and computes a tuple t' of sort (K', V') such that $t(K) = t'(K')$ and $t'(V)$ is the average between $t(V_1)$ and $t(V_2)$. Then g is definable in FO(=, All) *as $\phi_g(x, y, \mathrm{i}_1, \mathrm{i}_2, j) := \big(x = y \wedge \exists \mathrm{i}\,(\mathrm{i}_1 + \mathrm{i}_2 = \mathrm{i} \wedge 2j = \mathrm{i})\big)$.* □

As an immediate corollary to Theorem 3.1 we obtain the following result, which formalizes the fact that – in the case when $\mathsf{Values} = \mathbb{Q}$ – for translating $\textsc{Lara}(\Omega_{(=,\mathsf{All})})$ expressions it is not necessary to extend the expressive power of $\mathrm{FO}_{\mathsf{Agg}}$ with the relations in $\Psi_{\Omega_{(=,\mathsf{All})}}$ as long as one has access to all numerical predicates over $\mathbb{Q}$. Formally, let us denote by $\mathrm{FO}_{\mathsf{Agg}}(\mathsf{All})$

the extension of $\mathrm{FO}_{\mathsf{Agg}}$ with all formulas of the form $P(\iota_1, \ldots, \iota_k)$, for $P \subseteq \mathbb{Q}^k$ and $\iota_1, \ldots, \iota_k$ value-terms, with the expected semantics. Then one can prove the following result.

Corollary 4.1 *For every expression* $\mathrm{e}[\bar{K}, \bar{V}]$ *of* $\textsc{Lara}(\Omega_{(=,\mathsf{All})})$ *there is a formula* $\phi_{\mathrm{e}}(\bar{x}, \bar{\mathrm{i}})$ *of* $\mathrm{FO}_{\mathsf{Agg}}(\mathsf{All})$ *such that* $\mathrm{e}^D = \phi_{\mathrm{e}}^D$*, for every* $\textsc{Lara}$ *database* D*.*

Formulas definable in $\mathrm{FO}_{\mathsf{Agg}}(\mathsf{All})$ satisfy two important properties, namely, *genericity* and *locality*, which allow us to prove that neither convolution of matrices nor matrix inversion can be defined in the language. From Corollary 4.1 we obtain then that none of these queries is expressible in $\textsc{Lara}(\Omega_{(=,\mathsf{All})})$. We explain this next.

4.2 Matrix Convolution

Let A be an arbitrary matrix and K a square matrix. For simplicity we assume that K is of odd size $(2n+1) \times (2n+1)$. The convolution of A and K, denoted by $A * K$, is a matrix of the same size as A whose entries are defined as

$$(A * K)_{k\ell} = \sum_{s=1}^{2n+1} \sum_{t=1}^{2n+1} A_{k-n+s,\ell-n+t} \cdot K_{st}. \tag{4.1}$$

Notice that $k-n+s$ and $\ell-n+t$ could be invalid indices for matrix A. The standard way of dealing with this issue, and the one that we use here, is *zero padding*. This simply assumes those entries outside A to be 0. In the context of the convolution operator, one usually calls K a *kernel*.

We represent A and K over the schema $\sigma = \{\mathsf{Entry}_A[K_1, K_2, V], \mathsf{Entry}_K[K_1, K_2, V]\}$. Assume that $\mathsf{Keys} = \{\mathsf{k}_1, \mathsf{k}_2, \mathsf{k}_3, \ldots\}$ and $\mathsf{Values} = \mathbb{Q}$. If A is a matrix of values in $\mathbb{Q}$ of dimension $m \times p$, and K is a matrix of values in $\mathbb{Q}$ of dimensions $(2n+1) \times (2n+1)$ with $m, p, n \geq 1$, we represent the pair (A, K) as the $\textsc{Lara}$ database $D_{A,K}$ over σ that contains all facts $\mathsf{Entry}_A(\mathsf{k}_\mathrm{i}, \mathsf{k}_j, A_{\mathrm{i}j})$, for $\mathrm{i} \in [m]$, $j \in [p]$, and all facts $\mathsf{Entry}_K(\mathsf{k}_\mathrm{i}, \mathsf{k}_j, K_{\mathrm{i}j})$, for $\mathrm{i} \in [2n+1]$, $j \in [2n+1]$. The query $\mathsf{Convolution}$ over schema σ takes as input a $\textsc{Lara}$ database of the form $D_{A,K}$ and returns as output an associative table of sort $[K_1, K_2, V]$ that contains exactly the tuples $(\mathsf{k}_\mathrm{i}, \mathsf{k}_j, (A * K)_{\mathrm{i}j})$. We can then prove the following result:

Proposition 4.2 *Convolution is not expressible in* $\textsc{Lara}(\Omega_{(=,\mathsf{All})})$*.*

The proof is based on a simple *genericity* property for the language that is not preserved by convolution. This property intuitively expresses that, when $\textsc{Lara}(\Omega_{(=,\mathsf{All})})$ expressions are interpreted as expressions over matrices, they are invariant under reordering of rows and columns of those matrices. More formally, we make use of *key-permutations* and *key-generic* queries. A *key-permutation* is an injective function $\pi : \mathsf{Keys} \to \mathsf{Keys}$. We extend a key-permutation π to be a function over $\mathsf{Keys} \cup \mathsf{Values}$ by letting π be the identity over Values. A formula $\phi(\bar{x}, \bar{\mathrm{i}})$ is *key-generic* if for every $\textsc{Lara}$ database D, key-permutation π, and assignment ν, we have that $D \models \phi(\nu(\bar{x}, \bar{\mathrm{i}}))$ iff $\pi(D) \models \phi(\pi(\nu(\bar{x}, \bar{\mathrm{i}}))$. The following lemma expresses the self-evident property that formulas in $\mathrm{FO}_{\mathsf{Agg}}(\mathsf{All})$ are key-generic.

$\mathsf{Entry}_A =$

K_1	K_2	V
k_1	k_1	1
k_1	k_2	0
$\vdots$	$\vdots$	$\vdots$
k_2	k_1	0
k_2	k_2	1
k_2	k_3	0
$\vdots$	$\vdots$	$\vdots$
k_4	k_3	0
k_4	k_4	1

$\mathsf{Entry}_K =$

K_1	K_2	V
k_1	k_1	1
k_1	k_2	1
$\vdots$	$\vdots$	$\vdots$
k_3	k_2	1
k_3	k_3	1

$\mathsf{Entry}_{A'} =$

K_1	K_2	V
k_1	k_1	1
k_1	k_2	0
$\vdots$	$\vdots$	$\vdots$
k_3	k_2	0
k_3	k_3	1
k_3	k_4	0
$\vdots$	$\vdots$	$\vdots$
k_4	k_3	0
k_4	k_4	1

Figure 4.1: LARA representations for matrices A and K in the proof of Proposition 4.2

Lemma 4.3 *Every formula $\phi(\bar{x}, \bar{\mathrm{i}})$ of $\mathrm{FO}_{\mathsf{Agg}}(\mathsf{All})$ is key-generic.*

With the aid of Lemma 4.3 we can now prove Proposition 4.2, as it is easy to show that Convolution is not key-generic (even when the kernel K is fixed).

PROOF OF PROPOSITION 4.2. We show next that Convolution is not expressible in $\mathrm{FO}_{\mathsf{Agg}}(\mathsf{All})$. To obtain a contradiction assume that there exists a formula $\varphi(x, y, \mathrm{i})$ in $\mathrm{FO}_{\mathsf{Agg}}(\mathsf{All})$ such that for every LARA database $D_{A,K}$ we have that $D_{A,K} \models \varphi(\mathsf{k}_\mathrm{i}, \mathsf{k}_j, v)$ iff $(A * K)_{\mathrm{i}j} = v$. Let A, K, and A' be the following matrices

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad K = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \qquad A' = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The LARA representations for these matrices are depicted in Figure 4.1. Consider now the key-permutation π such that $\pi(\mathsf{k}_2) = \mathsf{k}_3$, $\pi(\mathsf{k}_3) = \mathsf{k}_2$, and π is the identity for every other value in Keys. It is not difficult to see that $\pi(D_{A,K}) = D_{A',K}$. Now, the convolutions $(A * K)$ and $(A' * K)$ are given by the matrices

$$(A * K) = \begin{bmatrix} 2 & 2 & 1 & 0 \\ 2 & 2 & 1 & 0 \\ 1 & 1 & 2 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \qquad (A' * K) = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 2 & 1 & 1 \\ 0 & 1 & 2 & 2 \\ 0 & 1 & 2 & 2 \end{bmatrix}$$

We know that $D_{A,K} \models \varphi(\mathsf{k}_1, \mathsf{k}_1, 2)$ (since $(A*K)_{11} = 2$), then, since φ is generic, we have that $\pi(D_{A,K}) \models \varphi(\pi(\mathsf{k}_1, \mathsf{k}_1, 2))$. Thus, since $\pi(D_{A,K}) = D_{A',K}$, $\pi(\mathsf{k}_1) = \mathsf{k}_1$, and π is the identity over Values, we obtain that $D_{A',K} \models \varphi(\mathsf{k}_1, \mathsf{k}_1, 2)$ which is a contradiction since $(A' * K)_{11} = 1 \neq 2$. This proves that Convolution is not expressible in $\mathrm{FO}_{\mathsf{Agg}}(\mathsf{All})$. Hence from Corollary 4.1 we obtain that $\mathrm{LARA}(\Omega_{\mathsf{Agg}})$ cannot express Convolution. □

4.3 Matrix inverse

It has been shown by Brijder et al. [6] that matrix inversion is not expressible in MATLANG by applying techniques based on locality. The basic idea is that MATLANG is subsumed by $\mathrm{FO}_{\mathsf{Agg}}(\emptyset) = \mathrm{FO}_{\mathsf{Agg}}$, and the latter logic can only define *local* properties. Intuitively, this means that formulas in $\mathrm{FO}_{\mathsf{Agg}}$ can only distinguish up to a *fixed-radius* neighborhood from its free variables (see, e.g., [25] for a formal definition). On the other hand, as shown in [6], if matrix inversion were expressible in MATLANG there would also be a $\mathrm{FO}_{\mathsf{Agg}}$ formula that defines the transitive closure of a binary relation (represented by its adjacency Boolean matrix). This is a contradiction as transitive closure is the prime example of a non-local property. We use the same kind of techniques to show that matrix inversion is not expressible in $\textsc{Lara}(\Omega_{(=,\mathsf{All})})$. For this, we use the fact that $\mathrm{FO}_{\mathsf{Agg}}(\mathsf{All})$ is also local.

We represent Boolean matrices as databases over the schema $\sigma = \{\mathsf{Entry}[K_1, K_2, V]\}$. Assume that $\mathsf{Keys} = \mathbb{N}$ and $\mathsf{Values} = \mathbb{Q}$. The Boolean matrix M of dimension $n \times m$, for $n, m \geq 1$, is represented as the LARA database D_M over σ that contains all facts $\mathsf{Entry}(\mathsf{k}_\mathrm{i}, \mathsf{k}_j, b_{\mathrm{i}j})$, for $\mathrm{i} \in [n]$, $j \in [m]$, and $b_{\mathrm{i}j} \in \{0, 1\}$, such that $M_{\mathrm{i}j} = b_{\mathrm{i}j}$. Consider the query Inv over schema σ that takes as input a LARA database of the form D_M and returns as output the LARA database $D_{M^{-1}}$, for M^{-1} the inverse of M. Then:

Proposition 4.4 $\textsc{Lara}(\Omega_{(=,\mathsf{All})})$ *cannot express* Inv *over Boolean matrices. That is, there is no* $\textsc{Lara}(\Omega_{(=,\mathsf{All})})$ *expression* $\mathrm{e}_{\mathsf{Inv}}[K_1, K_2, V]$ *over* σ *such that* $\mathrm{e}_{\mathsf{Inv}}(D_M) = \mathsf{Inv}(D_M)$*, for every* LARA *database of the form* D_M *that represents a Boolean matrix* M*.*

PROOF OF PROPOSITION 4.4. Assume for the sake of contradiction that $\mathrm{e}_{\mathsf{Inv}}$ exists. From Theorem 3.1 there is a $\mathrm{FO}^{+,\times}_{\mathsf{Agg}}$ formula $\phi(x_1, x_2, \mathrm{i})$ that expresses Inv over Boolean matrices, i.e., $\mathrm{e}_{\mathsf{Inv}}(D_M) = \phi(D_M) = D_{M^{-1}}$, for every LARA database of the form D_M that represents a Boolean matrix M. For reasons similar to those observed in [6, Example 12], this implies that there is an $\mathrm{FO}^{+,\times}_{\mathsf{Agg}}$ formula $\alpha(x_1, x_2, \mathrm{i})$ over σ that expresses the *transitive closure* query over the class of binary relations represented as Boolean matrices.[1] That is, for every LARA database of the form D_M that represents a Boolean matrix M of $m \times n$, it is the case $\phi(D_M)$ is the set of tuples (u, v, b_{uv}) such that $u \in [m]$, $v \in [n]$, and $b_{uv} \in \{0, 1\}$ satisfies that $b_{uv} = 1$ iff (u, v) belongs to the transitive closure of the binary relation represented by M.

It is well known, on the other hand, that $\mathrm{FO}^{+,\times}_{\mathsf{Agg}}$ can only express local queries (cf., [25]). In particular, this implies that there is no $\mathrm{FO}^{+,\times}_{\mathsf{Agg}}$ formula $\beta(x, y)$ such that for each finite binary relation R over $\mathbb{N}$ represented as a database $D_R = \{\mathsf{Rel}(u, v) \mid (u, v) \in R\}$, and each pair $(u, v) \in \mathbb{N} \times \mathbb{N}$, it is the case that $\beta(D_R)$ is the set of pairs (u, v) in the transitive closure of R. But this is a contradiction, as from $\alpha(x, y, \mathrm{i})$ we can construct a formula $\beta(x, y)$ of $\mathrm{FO}^{+,\times}_{\mathsf{Agg}}$ that satisfies this condition. In fact, we can define $\beta(x, y)$ as $\alpha'(x, y, 1)$, where α' is obtained by

- first replacing each subformula of the form $\exists \mathrm{i}' \psi(\bar{x}, \mathrm{i}', \bar{\mathrm{i}})$ in α with $\psi(\bar{x}, 0, \bar{\mathrm{i}}) \vee \psi(\bar{x}, 1, \bar{\mathrm{i}})$, where 0 is a shorthand for $\mathsf{Zero}(\mathrm{i}') := \neg \exists j (j + \mathrm{i}' \neq j)$ and 1 for $\mathsf{One}(\mathrm{i}') := \neg \exists j (j \cdot \mathrm{i}' \neq j)$; and then

[1] A Boolean matrix M represents binary relation R iff $R = \{(u, v) \mid M(u, v) = 1\}$.

- replacing each atomic formula of the form $\mathsf{Entry}(x', y', 1)$ with $\mathsf{Rel}(x', y')$, and each of the form $\mathsf{Entry}(x', y', 0)$ with $\neg\mathsf{Rel}(x', y')$.

This finishes the proof of the proposition. □

4.4 Einstein Sum

Einstein summation notation is another popular operator on tensors that generalizes operations such as the inner product of vectors, matrix trace, and tensor product and contraction. It is based on implicitly specifying summation over indexes in a formula. For instance, let A and B be two matrices and consider the expression $A_{\mathrm{i}j}B_{jk}$. When viewed as an Einstein summation, it is interpreted as a tensor T with two indexes (that is, a matrix), such that $\mathsf{T}_{\mathrm{i}k} = \sum_j A_{\mathrm{i}j} \cdot B_{jk}$, and thus, defines matrix product. In general, every repeated index in the expression implies a summation over such an index, while indexes that are not repeated are consider *free* and thus part of the output.

In modern tensor libraries [28, 29], Einstein summation is implemented as an even more general function `einsum` that gives a specific indexing to every input tensor, and specifies also the indexes expected in the output tensor. For simplicity, we focus on the case when `einsum` receives only two input tensors. It also receives a specification of the form $(\alpha, \beta \to \gamma)$ such that α and β are the indexing of the input tensors A and B, respectively, and $\gamma \subseteq \alpha \cup \beta$ is the indexing of the output. Every index in $\alpha \cup \beta$ that is not in γ is summed in the output. For instance, $\texttt{einsum}((\mathrm{i}j, jk \to \mathrm{i}k), A, B)$ is the multiplication of matrices A and B. As a more general example, consider tensors A and B of rank 3 and 4 respectively. Then the expression $\texttt{einsum}((\mathrm{i}jk, k\ell js \to \ell\mathrm{i}), A, B)$ produces a tensor T of rank 2 such that $\mathsf{T}_{\ell\mathrm{i}} = \sum_j \sum_k \sum_s A_{\mathrm{i}jk} \cdot B_{k\ell js}$. Notice that γ can be the empty set, in which case the output is a scalar. For example, $\texttt{einsum}((\mathrm{i}, \mathrm{i} \to \emptyset), \mathsf{a}, \mathsf{b})$ is the inner product of a and b.

LARA can easily express a named version of `einsum` in which α and β are renamings of the key attributes of the input tensors. In fact, it can be shown that LARA can express a *renaming* operator ρ such that $\rho_\alpha A$ is a tensor with exactly the same data as A but with its key attributes renamed according to α. This is done via the *extend* operator and the *union* operator. Thus $\texttt{einsum}((\alpha, \beta \to \gamma), A, B)$ can be expressed simply as

$$\textstyle\bigboxplus^{\gamma}_{+} (\rho_\alpha A \bowtie_{\cdot} \rho_\beta B)$$

.

Proposition 4.5 $\textsc{Lara}(\Omega_{(=,\mathsf{All})})$ *can express the general Einstein summation.*

Given that we are dealing with named tensors, a special (and more natural) case is when we use the original attribute names without renaming and, instead of specifying the key attributes in the output, we specify the attributes δ over which we want to sum. In such a case we obtain the expression $\bar{\textstyle\sum}^{\delta}_{+} (A \bowtie_{\cdot} B)$, which resembles the *named tensor contraction* [32, 33].

Chapter 5

Adding Built-in Predicates over Keys

We proceed to relax the restriction of only comparing *keys* and *values* by equality to arbitrary comparisons between them. We will see that this increases the expressiveness of LARA to the point that we can now express recursive queries. This comes at the cost of losing the nice properties we had, which are strong genericity and locality.

5.1 Convolution in LARA with comparisons

Previously we have seen that there are important linear algebra operations, such as matrix inverse and convolution, that $\text{LARA}(\Omega_{(=,\mathsf{All})})$ cannot express. The following result, on the other hand, shows that a clean extension of $\text{LARA}(\Omega_{(=,\mathsf{All})})$ can express matrix convolution. This extension corresponds to the language $\text{LARA}(\Omega_{(<,\mathsf{All})})$, i.e., the extension of $\text{LARA}(\Omega_{(=,\mathsf{All})})$ in which we assume the existence of a asymmetric linear-order $<$ on Keys and extension functions are definable in the logic $\text{FO}(<,\mathsf{All})$ that extends $\text{FO}(=,\mathsf{All})$ by allowing atomic formulas of the form $x < y$, for x, y key-variables. Even more, the only numerical predicates from All we need are $+$ and $\times$. We denote the resulting logic as $\text{LARA}(\Omega_{(<,\{+,\times\})})$.

Proposition 5.1 CONVOLUTION *is expressible in* $\text{LARA}(\Omega_{(<,\{+,\times\})})$.

PROOF OF PROPOSITION 5.1. The important observation is that once we have a linear order over Keys, we can define a bijection among elements of the two sorts, which allows us to compare keys with the numerical predicates $+$ and $\times$. Assume for simplicity that $\mathsf{Keys} = \mathbb{N}$ and $\mathsf{Values} = \mathbb{Q}$. Consider first the extension function $f : (K, K', \emptyset) \mapsto (\emptyset, V)$ defined as the following $\text{FO}(<, \{+, \times\})$ formula:

$$\phi_f(x, y, \mathrm{i}) \;:=\; (x < y) \rightarrow \mathrm{i} = 0 \;\wedge\; \neg(x < y) \rightarrow \mathrm{i} = 1.$$

Now consider a relation Entry_A of sort $[K, K', V]$ that represents a square matrix of dimension $n \times n$. By our definition of f we have that $\mathsf{Map}_f\, \mathsf{Entry}_A$ has sort $[K, K', V]$ and its evaluation consists of all triples $(x, y, \mathrm{i}) \in [n]^3$ such that $\mathrm{i} = 1$ if $x \geq y$ and $\mathrm{i} = 0$ otherwise. Consider

now the expression

$$\mathsf{Ind}_{K,V} \;:=\; \textstyle\sum_{+}^{K} \mathsf{Map}_f\, \mathsf{Entry}_A$$

that aggregates $\mathsf{Map}_f\, \mathsf{Entry}_A$ by summing over V by grouping over K. The evaluation of $\mathsf{Ind}_{K,V}$ contains all pairs $(x, \mathrm{i}) \in [n] \times [n]$ such that i is the natural number that represents the position of x in the linear order over Keys. Hence, we have that $\mathsf{Ind}_{K,V}$ contains all pairs $(x, \mathrm{i}) \in [n] \times [n]$ such that x is a key, i is a value, and $x = \mathrm{i}$. This simple fact allows us to express extension functions using arbitrary properties in $\mathrm{FO}(<, \{+, \times\})$ over key- and value-attributes together, without actually mixing sorts. For example, consider an associative table R of sort $[K, V_1]$ with $[n]$ as set of keys, and assume that we want to construct a new table R' of sort $[K, V_2]$ such that, for every tuple $(x, v) \in R$, relation R' contains the tuple (x, j) with $j = 2x + v$. We make use of the extension function $g : (K, V, V_1) \mapsto (\emptyset, V_2)$ defined by the formula $\phi_g(x, \mathrm{i}, v, j) := (j = 2\mathrm{i} + v)$. Notice that ϕ_g only mentions variables of the second sort. We can construct R' as

$$R' := \mathsf{Map}_g(\mathsf{Ind}_{K,V} \bowtie_+ R)$$

To see that this works, notice first that $\mathsf{Ind}_{K,V}$ and R has the same set of keys (the set $[n]$). Moreover, given that $\mathsf{Ind}_{K,V}$ and R has no value attribute in common, $\mathsf{Ind}_{K,V} \bowtie_+ R$ is just performing a natural join. Thus the result of $\mathsf{Ind}_{K,V} \bowtie_+ R$ is a table of sort $[K, V, V_1]$ that contains all tuples (x, i, v) such that $(x, \mathrm{i}) \in \mathsf{Ind}_{K,V}$ and $(x, v) \in R$, or equivalently, all tuples (x, i, v) such that $(x, v) \in R$ and $x = \mathrm{i}$. Then with $\mathsf{Map}_g(\mathsf{Ind}_{K,V} \bowtie_+ R)$ we generate all tuples (x, j) such that $(x, v) \in R$, $x = \mathrm{i}$, and $(j = 2\mathrm{i} + v)$, or equivalently, all tuples (x, j) such that $(x, v) \in R$, and $(j = 2x + v)$, which is what we wanted to obtain. Given the above discussion, we assume in the following that extension functions are defined by expressions over Keys and Values, and then we can write the above expression simply as

$$R' := \mathsf{Map}_{(j=2x+v)} R.$$

To complete the example we need to introduce some additional notions. Let $\mathrm{e}_1[\bar{K}_1, \bar{V}_1]$ and $\mathrm{e}_2[\bar{K}_2, \bar{V}_2]$ be expressions such that $\bar{K}_1 \cap \bar{K}_2 = \bar{V}_1 \cap \bar{V}_2 = \emptyset$. The cartesian product is a new expression $\mathrm{e}[\bar{K}_1, \bar{K}_2, \bar{V}_1, \bar{V}_2]$ such that for every tuple $(\bar{k}_1, \bar{v}_1) \in \mathrm{e}_1^D$ and $(\bar{k}_2, \bar{v}_2) \in \mathrm{e}_2^D$ we have that $(\bar{k}_1, \bar{k}_2, \bar{v}_1, \bar{v}_2) \in \mathrm{e}^D$ for every database D. It is not difficult to prove that the cartesian product, denoted by $\times$ is expressible in Lara. Another operator that we need is the Filter operator. Given an expression $\mathrm{e}_1[K, V]$ and a logical expression $\varphi(\bar{x}, \bar{y})$, filtering e_1 with φ, is a new expression $\mathrm{e}_2 = \mathsf{Filter}_\varphi(\mathrm{e}_1)$ that has sort $[K, V]$ and such that for every database D it holds that $(\bar{k}, \bar{v})$ is in e_2^D if and only if $(\bar{k}, \bar{v}) \in \mathrm{e}_1^D$ and $\varphi(\bar{k}, \bar{v})$ holds. It is not difficult to prove that the filter operator is also expressible in Lara. Moreover, by the discussion above, for expressions in $\textsc{Lara}(\Omega_{(<,\{+,\times\})})$ we can use filter expressions as arbitrary $\mathrm{FO}(<, \{+, \times\})$ formulas over keys and values. Finally, the renaming operator $\rho_\alpha(A)$ is a simple operator that just changes the name of attributes of A according to the assignment α. Both, filtering and renaming can be defined as a special case of Ext.

We now have all the ingredients to express the convolution. We first write the convolution definition in a more suitable way so we can easily express the sums required with $\mathrm{FO}(<, \{+, \times\})$ formulas. Let K be a kernel of dimensions $\mathsf{m} \times \mathsf{m}$ with m an odd number. First define mid as $\frac{\mathsf{m}-1}{2}$. Consider now a matrix A of dimension $\mathsf{n}_1 \times \mathsf{n}_2$. Now for every $(\mathsf{i}, \mathsf{j}) \in [\mathsf{n}_1] \times [\mathsf{n}_2]$

one can write the following expression for $(A * K)_{\mathsf{ij}}$

$$(A*K)_{\mathsf{ij}} = \text{sum}\ \{\{\ A_{\mathsf{st}} \cdot K_{\mathsf{kl}}\ \mid\ \mathsf{s} \in [\mathsf{n}_1], \mathsf{t} \in [\mathsf{n}_2], \mathsf{k}, \mathsf{l} \in [\mathsf{m}]\quad \text{and}\quad \mathsf{i} - \mathsf{mid} \leq \mathsf{s} \leq \mathsf{i} + \mathsf{mid}\quad \text{and}\quad \mathsf{j} - \mathsf{mid} \leq \mathsf{t} \leq \mathsf{j} + \mathsf{mid}\quad \text{and}\quad \mathsf{k} = \mathsf{s} - \mathsf{i} + \mathsf{mid} + 1\quad \text{and}\quad \mathsf{l} = \mathsf{t} - \mathsf{j} + \mathsf{mid} + 1\ \}\} \quad (5.1)$$

Now, in order to implement the above definition we use the following extension functions:

$$\text{diag}(k, \ell, m) := \quad (k = \ell) \to m = 1 \quad \wedge \quad \neg(k = \ell) \to m = 0$$

$$\text{neighbors}(\mathsf{i}, j, s, t, m) := \quad \exists \text{mid}\big(2 \times \text{mid} = m - 1 \quad \wedge \quad \mathsf{i} - \text{mid} \leq s \quad \wedge \quad s \leq \mathsf{i} + \text{mid} \quad \wedge \quad j - \text{mid} \leq t \quad \wedge \quad t \leq j + \text{mid}\big)$$

$$\text{kernel}(\mathsf{i}, j, k, \ell, s, t, m) := \quad \exists \text{mid}\big(2 \times \text{mid} = m - 1 \ \wedge \quad k = s - \mathsf{i} + \text{mid} + 1 \quad \wedge \quad \ell = t - j + \text{mid} + 1\big)$$

In what follows and for simplicity we will use lowercase letters for key and value attributes in associative tables, to facilitate correspondence between attributes and variables in the extension functions. Let $\mathsf{Entry}_A[(\mathsf{i}, j), (v)]$ and $\mathsf{Entry}_K[(k, \ell), (u)]$ be two associative tables that represents a matrix A and the convolution kernel K, respectively. We first construct an expression that computes the dimension of the kernel:

$$M = \textstyle\bigtriangledown_{+}^{\emptyset}\ \mathsf{Map}_{\text{diag}}\mathsf{Entry}_K.$$

By the definition of diag, we have that $M[\emptyset, (m)]$ has no key attributes and a has single value attribute m that contains one tuple storing the dimension of K. Now we proceed to make the cartesian product of Entry_A with itself, Entry_K and M. For that we need to make a copy of Entry_A with renamed attributes. The cartesian product is

$$C = \mathsf{Entry}_A \times \mathsf{Entry}_K \times (\rho_{\mathsf{i}:s,j:t,v:w}\mathsf{Entry}_A) \times M.$$

This produces an associative table of sort $C[(\mathsf{i}, j, k, \ell, s, t), (v, u, w, m)]$. We then compute the following filters over C.

$$F = \mathsf{Filter}_{\text{kernel}}(\mathsf{Filter}_{\text{neighbors}}(C)).$$

We note that F has sort $F[(\mathsf{i}, j, k, \ell, s, t), (v, u, w, m)]$ (just like C). We also note that for every (i, j) the tuple $(\mathsf{i}, \mathsf{j}, \mathsf{k}, \mathsf{l}, \mathsf{s}, \mathsf{t})$ is a key in F if and only if it satisfies the conditions defining the multiset in Equation (5.1). Thus to compute what we need, there only remains multiplying and adding, which is done in the following expression

$$R = \textstyle\bigtriangledown_{+}^{\mathsf{i},j}\ (\mathsf{Map}_{v^\star = w \cdot u}F).$$

Thus R is of sort $[(\mathsf{i}, j), (v^\star)]$ and is such that $(\mathsf{i}, \mathsf{j}, \mathsf{v})$ is in R if and only if $\mathsf{v} = (A * K)_{\mathsf{ij}}$.

□

It is worth remarking that Hutchison et al. [19] showed that for every fixed kernel K, the query $(A*K)$ is expressible in Lara. However, the Lara expression they construct depends on the values of K, and hence their construction does not show that in general convolution is expressible in Lara. Our construction is stronger, as we show that there exists a *fixed* Lara$(\Omega_{(<,\{+,\times\})})$ expression that takes A and K as input and produces $(A * K)$ as output.

5.2 Can LARA with comparisons express the inverse?

We believe that LARA($\Omega_{(<,\{+,\times\})}$) cannot express INV. However, this seems quite challenging to prove. First, the tool we used for showing that INV is not expressible in LARA($\Omega_{(=,\mathsf{All})}$), namely, locality, is no longer valid in this setting. In fact, queries expressible in LARA($\Omega_{(<,\{+,\times\})}$) are not necessarily local.

Proposition 5.2 LARA($\Omega_{(<,\{+,\times\})}$) *can express non-local queries.*

By the discussion in the proof of Proposition 5.1, we can use arbitrary predicates $+$ and $\times$ over Keys to define extension functions. With this observation we can prove that, given a relation $A[K]$ that contains all values in $[n]$, one can construct a LARA($\Omega_{(<,\{+,\times\})}$) expression that defines a relation $\mathsf{BIT}[K,V]$ that contains all pairs (x,i) such that $x \in [n]$ and the i-th bit of the binary expansion of x is 1. With BIT we can mimic the proof of Proposition 8.22 in [25] to show that LARA($\Omega_{(<,\{+,\times\})}$) can express a non-local query. In particular, we can express the transitive closure of a "small portion" of an input graph.

PROOF OF PROPOSITION 5.2. We explain how to use LARA($\Omega_{(<,\{+,\times\})}$) expressions to mimic the proof of Proposition 8.22 in [25]. Assume a schema $\{A[K_1,\emptyset], E[K_1,K_2,\emptyset], P[K_1,\emptyset]\}$ and consider the family F of LARA databases defined as follows. $D \in F$ if and only if all the following holds.

- All keys in E^D and P^D are also in A^D.
- E^D is a disjoint union of a chain and zero or more cycles, that is

$$E^D = \{(a^0_0,a^0_1),\ldots,(a^0_{k_0-1},a^0_{k_0})\} \cup \bigcup_{\mathrm{i}=1}^{K}\{(a^{\mathrm{i}}_0,a^{\mathrm{i}}_1),\ldots,(a^{\mathrm{i}}_{k_{\mathrm{i}}-1},a^{\mathrm{i}}_{k_{\mathrm{i}}}),(a^{\mathrm{i}}_{k_{\mathrm{i}}},a^{\mathrm{i}}_0)\}$$

  with $K \geq 1$ and all the a^j_{i}s different elements in Keys.
- P^D contains an initial segment of the chain in E^D and may contain some of the cycles in E^D, that is, there exists a $K < k_0$ and a set $L \subseteq [\ell]$ such that

$$P^D = \{(a^0_0,a^0_1),\ldots,(a^0_{K-1},a^0_K)\} \cup \bigcup_{j\in L}\{(a^j_0,a^j_1),\ldots,(a^j_{k_j-1},a^j_{k_{\mathrm{i}}}),(a^{\mathrm{i}}_{k_j},a^j_0)\}$$

- $|P^D| \leq \log|A^D|$.
- For every $a \in P^D$ and $b \in A^D \setminus P^D$ it holds that $a < b$.

We prove that there exists a LARA($\Omega_{(<,\{+,\times\})}$) expression $\mathrm{e}[K_1,K_2,V_1,V_2]$ such that for every $D \in F$ it holds that (a,b) is a key e^D if and only if (a,b) is in the transitive closure of E^D restricted to elements in P^D.

We will also make use of the following property of FO($<,\{+,\times\}$). It is known [25] that there exists an FO($<,\{+,\times\}$) formula $BIT(x,y)$ such that $BIT(a,b)$ holds if and only if the bth bit in the binary expansion of a is 1. Similarly as in the proof of Proposition 5.1 and by using $BIT(x,y)$ we can produce an extension function $\mathrm{bit} : [K_1,\emptyset] \mapsto [K_2,\emptyset]$ such that $\mathrm{bit}(k)$

is a table containing all the keys i such that the ith bit of the binary expansion of k is 1. Let $\mathsf{BIT}[K_1, K_2, \emptyset]$ be defined by the expression $\mathsf{BIT} = \mathsf{Ext}_{\text{bit}} A$.

Now assume that $P^D = \{n_1, n_2, \ldots, n_\ell\}$ with $\ell = |P^D|$. By the properties of D, we can assume that n_1 is the minimum value in $<$, and that $n_{\text{i}+1}$ is the successor of n_i in $<$. Moreover, we also know that $\ell \leq \log |A^D|$. All this implies that every set S such that $S \subseteq P^D$ can be represented as an element $\text{code}(S) \in A^D$ as follows. Let $I_S \subseteq [\ell]$ be such that $S = \{n_\text{i} \mid \text{i} \in I_S\}$. Then $\text{code}(S)$ is the element in A^D such that its binary expansion has a 1 exactly at positions I_S, and has a 0 in every other position. Thus, to check that an element $x \in A^D$ is in a set $S \subseteq P^D$, it is enough to check that $BIT(\text{code}(S), x)$ holds or similarly, that $(\text{code}(S), x) \in \mathsf{BIT}^D$.

The above observations allow us to simulate an existential second order quantification over subsets of P as a (first order) quantification over elements in A as follows. Let $\psi := \exists S \varphi$ be a second order formula with φ an FO formula that can also mention atoms of the form $S(x)$. We then can rewrite ψ as $\psi' := \exists s \varphi'$ such that every atom $S(x)$ in φ is replaced by $BIT(s, x)$. It is not difficult to use this property and the special form of E in the family F to construct a first order formula $\text{reach}(x, y)$ of the form $\exists s\ \varphi(x, y, s)$ over vocabulary $\{BIT, E\}$ that is true for a tuple (a, b) if and only if b is reachable from a by following a path in E restricted to the set P (see Proposition 8.22 in [25]).

Now, from $\text{reach}(x, y)$ one can produce a new FO formula $\text{reach}'(x, y)$ over $\{\mathsf{BIT}, E, A\}$, replacing every occurrence of $BIT(x, y)$ by $\mathsf{BIT}(x, y)$, and adding a conjunction with $A(x)$ for every variable x mentioned, ensuring that the resulting formula is a safe formula.

We note that the obtained formula $\text{reach}'(x, y)$ is a standard FO formula (over $\{\mathsf{BIT}, E, A\}$). Then by Theorem 3.3 we know that there exists a LARA expression $\text{e}[(K_1, K_2)]$ such that for every database D' over schema $\{\mathsf{BIT}, E, A\}$ it holds that $(x, y) \in \text{e}^{D'}$ if and only if $D' \models \text{reach}'(x, y)$. Finally, given that BIT can be defined in $\text{LARA}(\Omega_{(<,\{+,\times\})})$ we find that the transitive closure of E restricted to P can also be defined in $\text{LARA}(\Omega_{(<,\{+,\times\})})$. □

This implies that one would have to apply techniques more specifically tailored for the logic, such as *Ehrenfeucht-Fraïssé* games, to show that INV is not expressible in $\text{LARA}(\Omega_{(<,\{+,\times\})})$. Unfortunately, it is often combinatorially difficult to apply such techniques in the presence of built-in predicates, e.g., a linear order, on the domain; cf., [13, 37, 16]. So far, we have not managed to succeed in this regard.

On the other hand, we can show that INV is not expressible in a natural restriction of $\text{LARA}(\Omega_{(<,\{+,\times\})})$ under complexity-theoretic assumptions. To start with, INV is complete for the complexity class DET, which contains all those problems that are logspace reducible to computing the *determinant* of a matrix. It is known that LOGSPACE $\subseteq$ DET, where LOGSPACE is the class of functions computable in logarithmic space, and this inclusion is believed to be proper [10].

In turn, most of the aggregate operators used in practical applications, including standard ones such as SUM, AVG, MIN, MAX, and COUNT, can be computed in LOGSPACE (see, e.g., [9]). Combining this with well-known results on the complexity of computing relational algebra and arithmetic operations, we obtain that the fragment $\text{LARA}_{\text{st}}(\Omega_{(<,\{+,\times\})})$

of $\textsc{Lara}(\Omega_{(<,\{+,\times\})})$ that only mentions the standard aggregate operators above, and whose formulas defining extension functions are *safe*, can be evaluated in Logspace in *data complexity*, i.e., assuming formulas to be fixed.

Proposition 5.3 *Let* $\mathrm{e}[\bar{K},\bar{V}]$ *be a fixed expression of* $\textsc{Lara}_{\mathrm{st}}(\Omega_{(<,\{+,\times\})})$. *There is a* Logspace *procedure that takes as input a* Lara *database* D *and computes* e^D.

Proof of Proposition 5.3. We first explain when an extension function of sort $(\bar{K},\bar{V}) \mapsto (\bar{K}',\bar{V}')$ is definable by a *safe* formula in $\mathrm{FO}(<,\{+,\times,\})$. Recall that f is definable in $\mathrm{FO}(<,\{+,\times,\})$ when there is a FO formula $\phi(\bar{x},\bar{x}',\bar{\mathrm{i}},\bar{\mathrm{i}}')$ that only mentions atomic formulas of the form $x = y$ and $x < y$, for x, y key-variables, and $\mathrm{i} + j = k$ and $\mathrm{i}j = k$, for i, j, k value-variables or constants of the form $0_\oplus$, such that for each tuple $(\bar{k},\bar{v}) \in A$ we have that $f(\bar{k},\bar{v})$ is precisely the set of tuples $(\bar{k}',\bar{v}')$ such that $\phi(\bar{k},\bar{v},\bar{k}',\bar{v}')$ holds. The safe fragment of this logic is obtained by restricting all negated formulas to be *guarded* by $\bar{x}$ and $\bar{\mathrm{i}}$, i.e., forcing them to be of the form $\neg\psi(\bar{x},\bar{\mathrm{i}})$, and all formulas of the form $x = y$, $x < y$, $\mathrm{i} + j = k$, and $\mathrm{i}j = k$ to have at most one variable that is not guarded by $\bar{x}$ and $\bar{\mathrm{i}}$, i.e., at most one variable that does not appear in $(\bar{x},\bar{\mathrm{i}})$. It is easy to see that if ϕ is safe, then it defines an extension function; this is because for each tuple $(\bar{k},\bar{v})$ we have that the set of tuples $(\bar{k}',\bar{v}')$ such that $\phi(\bar{k},\bar{v},\bar{k}',\bar{v}')$ holds is finite.

We now prove the proposition. Since the expression is fixed we only need to show that each operation used in the expression can be computed in Logspace. The relational algebra operations of join and union can be computed in Logspace; see, e.g., [1]. Since, in addition, aggregate operators included in $\textsc{Lara}_{\mathrm{st}}(<;+)$ can be computed in Logspace, we obtain that, given associative tables A and B, the results of $A \bowtie_\oplus B$ and $A \leftouterjoin_\oplus B$ can be computed in Logspace.

Let us consider now the case of $\mathsf{Ext}_f\, A$, for f an extension function of sort $(\bar{K},\bar{V}) \mapsto (\bar{K}',\bar{V}')$ definable in the safe fragment of $\mathrm{FO}(<,\{+,\textsc{half}\})$. Then f is expressible as a safe FO formula $\phi(\bar{x},\bar{x}',\bar{\mathrm{i}},\bar{\mathrm{i}}')$ that only allow atomic formulas of the form $x = y$ and $x < y$, for x, y key-variables, and $\mathrm{i} + j = k$ and $\textsc{half}(\mathrm{i}) = j$, for i, j, k value-variables or constants of the form $0_\oplus$. We need to show that for each tuple $(\bar{k},\bar{v}) \in A$ we can compute the set of tuples $(\bar{k}',\bar{v}')$ such that $\phi(\bar{k},\bar{v},\bar{k}',\bar{v}')$ holds in Logspace. But this is easy to see, due to the safety condition and the fact that relational algebra operations and the arithmetic computations of the form $x = y$, $x < y$, $\mathrm{i} + j = k$, and $\textsc{half}(\mathrm{i}) = j$ can be computed in Logspace when at most one of the variables in the expression is not guarded. □

Hence, proving Inv to be expressible in the language $\textsc{Lara}_{\mathrm{st}}(\Omega_{(<,\{+,\times\})})$ would imply the surprising result that Logspace = Det.

Conclusion and Future Work

We believe that the work on query languages for analytics systems that integrate relational and statistical functionalities provides interesting perspectives for database theory. In this paper we focused on the LARA language, which has been designed to become the core algebraic language for such systems, and carried out a systematic study of its expressive power in terms of logics and concepts traditionally studied in the database theory literature.

As we have observed, expressing interesting ML operators in LARA requires the addition of complex features, such as arithmetic predicates on the numerical sort and built-in predicates on the domain. The presence of such features complicates the study of the expressive power of the languages, as some known techniques no longer hold, e.g., genericity and locality, while others become combinatorially difficult to apply, e.g., Ehrenfeucht-Fraïssé games. In addition, the presence of a built-in linear order might turn the logic capable of characterizing some parallel complexity classes, and thus inexpressibility results could be as hard to prove as some longstanding conjectures in complexity theory.

A possible way to overcome these problems might be to not look at languages in their full generality, but only at extensions of the tame fragment $\text{LARA}(\Omega_{(=,\mathsf{All})})$ with some of the most sophisticated operators. For instance, what if we extend $\text{LARA}(\Omega_{(=,\mathsf{All})})$ directly with an operator that computes Convolution? Is it possible to prove that the resulting language ($\text{LARA}(\Omega_{(=,\mathsf{All})})$ + Convolution) cannot express matrix inverse Inv? Proving that Einstein summation can be expressed in LARA and even in the key-generic fragment $\text{LARA}(\Omega_{(=,\mathsf{All})})$ provides additional evidence in favor of LARA as a good language for tensor-based reasoning.

Another interesting line of work corresponds to identifying which kind of operations need to be added to LARA in order to be able to express in a natural way recursive operations such as matrix inverse. One would like to do this in a general yet minimalistic way, as adding too much recursive expressive power to the language might render it impractical. It would be useful to start then by identifying the most important recursive operations one needs to perform on associative tables, and then abstract from them the minimal primitives that the language needs to possess for expressing such operations. We are also interested in studying the expressiveness of LARA when strings as a data type are allowed in the language, as most database systems must rely not only on numerical calculations, but also on string manipulation.

There is a commendable trend in contemporary database research to leverage, and considerably extend, techniques from database query processing and optimization, to support large-scale linear algebra computations. In principle, data scientists could then work directly

in SQL or related languages. Still, some users will prefer to continue using the matrix sub languages they are more familiar with. Supporting these languages is also important so that existing code need not be rewritten. From the perspective of database theory, it then becomes relevant to understand the expressive power of these languages as well as possible.

Also, we have not covered all standard constructs from linear algebra. For instance, it may be worthwhile to extend our framework with the operation of putting matrices in upper triangular form, with the Gram-Schmidt procedure (which is now partly hidden in the eigen operation), and with the singular value decomposition. When moving to more and more powerful and complicated languages, however, it becomes less clear at what point we should simply move all the way to full SQL, or extensions of SQL with recursion.

Bibliography

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1):1:1–1:39, February 2008.

[3] Sercan Ömer Arik, Mike Chrzanowski, Adam Coates, Greg Diamos, Andrew Gibiansky, Yongguo Kang, Xian Li, John Miller, Jonathan Raiman, Shubho Sengupta, and Mohammad Shoeybi. Deep voice: Real-time neural text-to-speech. *CoRR*, abs/1702.07825, 2017.

[4] Matthias Boehm, Michael W. Dusenberry, Deron Eriksson, Alexandre V. Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick R. Reiss, Prithviraj Sen, Arvind C. Surve, and Shirish Tatikonda. Systemml: Declarative machine learning on spark. *Proc. VLDB Endow.*, 9(13):1425–1436, September 2016.

[5] Robert Brijder, Floris Geerts, Jan Van den Bussche, and Timmy Weerwag. On the expressive power of query languages for matrices. *CoRR*, abs/1709.08359, 2017.

[6] Robert Brijder, Floris Geerts, Jan Van den Bussche, and Timmy Weerwag. On the expressive power of query languages for matrices. In *ICDT*, pages 10:1–10:17, 2018.

[7] Mücahit Büyükyılmaz and Ali Çıbıkdiken. Voice gender recognition using deep learning. 12 2016.

[8] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970.

[9] Mariano P. Consens and Alberto O. Mendelzon. Low complexity aggregation in graphlog and datalog. *Theor. Comput. Sci.*, 116(1):95–116, 1993.

[10] Stephen A. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64(1-3):2–21, 1985.

[11] Martín Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[12] Serge Abiteboul et al. Research directions for principles of data management (dagstuhl

perspectives workshop 16151). *Dagstuhl Manifestos*, 7(1):1–29, 2018.

[13] Ronald Fagin, Larry J. Stockmeyer, and Moshe Y. Vardi. On monadic NP vs. monadic co-np. *Inf. Comput.*, 120(1):78–92, 1995.

[14] William M. Farmer. Chiron: A multi-paradigm logic. 2007.

[15] Floris Geerts. On the expressive power of linear algebra on graphs. In *22nd International Conference on Database Theory, ICDT 2019, March 26-28, 2019, Lisbon, Portugal*, pages 7:1–7:19, 2019.

[16] Martin Grohe and Thomas Schwentick. Locality of order-invariant first-order formulas. In *MFCS*, pages 437–445, 1998.

[17] Lauri Hella, Leonid Libkin, Juha Nurmonen, and Limsoon Wong. Logics with aggregate operators. *J. ACM*, 48(4):880–907, July 2001.

[18] Stephan Hoyer, Joe Hamman, and xarray developers. xarray development roadmap. Technical report, 2018. Available at `http://xarray.pydata.org/en/stable/roadmap.html`, retrieved on March 2019.

[19] Dylan Hutchison, Bill Howe, and Dan Suciu. Lara: A key-value algebra underlying arrays and relations. *CoRR*, abs/1604.03607, 2016.

[20] Dylan Hutchison, Bill Howe, and Dan Suciu. Laradb: A minimalist kernel for linear and relational algebra computation. In *Proceedings of BeyondMR@SIGMOD 2017*, pages 2:1–2:10, 2017.

[21] Neil Immerman. *Descriptive complexity*. Graduate texts in computer science. Springer, 1999.

[22] Andreas Kunft, Alexander Alexandrov, Asterios Katsifodimos, and Volker Markl. Bridging the gap: towards optimization across linear and relational algebra. In *Proceedings of BeyondMR@SIGMOD 2016*, page 1, 2016.

[23] Olivier Lezoray, Christophe Charrier, Hubert Cardot, and Sébastien Lefèvre. Machine learning in image processing - special issue editorial. *EURASIP J. Adv. Sig. Proc.*, 2008, 01 2008.

[24] Leonid Libkin. Expressive power of SQL. *Theor. Comput. Sci.*, 296(3):379–404, 2003.

[25] Leonid Libkin. *Elements of Finite Model Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[26] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[27] Hung Q. Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. In-database

factorized learning. In *AMW*, 2017.

[28] Travis Oliphant. NumPy: A guide to NumPy. USA: Trelgol Publishing, 2006–.

[29] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NIPS-W*, 2017.

[30] Hendrik Purwins, Bo Li, Tuomas Virtanen, Jan Schlüter, Shuo-Yiin Chang, and Tara N. Sainath. Deep learning for audio signal processing. *CoRR*, abs/1905.00078, 2019.

[31] Christopher Ré, Divy Agrawal, Magdalena Balazinska, Michael Cafarella, Michael Jordan, Tim Kraska, and Raghu Ramakrishnan. Machine learning and databases: The sound of things to come or a cacophony of hype? In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 283–284, New York, NY, USA, 2015. ACM.

[32] Alexander M. Rush. Tensor considered harmful. Technical report, Harvard NLP Blog, 2019. Available at `http://nlp.seas.harvard.edu/NamedTensor`, retrieved on March 2019.

[33] Alexander M. Rush. Tensor considered harmful pt. 2. Technical report, Harvard NLP Blog, 2019. Available at `http://nlp.seas.harvard.edu/NamedTensor2`, retrieved on March 2019.

[34] Florin Rusu and Yu Cheng. A survey on array storage, query languages, and systems. *CoRR*, abs/1302.0103, 2013.

[35] Abdollah Safari, Rachel MacKay Altman, and Thomas M. Loughin. Display advertising: Estimating conversion probability efficiently. *arXiv e-prints*, page arXiv:1710.08583, Oct 2017.

[36] Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. Learning linear regression models over factorized joins. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 3–18, New York, NY, USA, 2016. ACM.

[37] Thomas Schwentick. On winning Ehrenfeucht games and monadic NP. *Ann. Pure Appl. Logic*, 79(1):61–92, 1996.

[38] Tomer Shay. Most popular databases in 2018 according to stackoverflow survey. 2018.

[39] Michael Stonebraker. The Case for Polystores, 2015.

[40] Yi Tay, Luu Anh Tuan, and Siu Cheung Hui. Recurrently controlled recurrent networks. *CoRR*, abs/1811.09786, 2018.

[41] Yi Tay, Luu Anh Tuan, Siu Cheung Hui, and Jian Su. Densely connected attention propagation for reading comprehension. *CoRR*, abs/1811.04210, 2018.

[42] Aäron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew W. Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *CoRR*, abs/1609.03499, 2016.

[43] Richard Zhang, Phillip Isola, and Alexei A. Efros. Colorful image colorization. *CoRR*, abs/1603.08511, 2016.