Expressive Power of an Algebra For Data Mining

TOON CALDERS
University of Antwerp, Belgium
and
LAKS V.S. LAKSHMANAN and RAYMOND T. NG
University of British Columbia, Canada
and
JAN PAREDAENS
University of Antwerp, Belgium

The relational data model has simple and clear foundations on which significant theoretical and systems research has flourished. By contrast, most research on data mining has focused on algorithmic issues. A major open question is "what's an appropriate foundation for data mining, which can accommodate disparate mining tasks." We address this problem by presenting a database model and an algebra for data mining. The database model is based on the 3W-model introduced by Johnson et al. [2000]. This model relied on black box mining operators. A main contribution of this paper is to open up these black boxes, by using generic operators in a data mining algebra. Two key operators in this algebra are *regionize*, which creates regions (or models) from data tuples, and a restricted form of looping called *mining loop*. Then, the resulting data mining algebra $\mathcal{MA}$ is studied and properties concerning expressive power and complexity are established. We present results in three directions: (1) expressiveness of the mining algebra; (2) relations with alternative frameworks, and (3) interactions between regionize and mining loop.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems; H.2.4 [**Database Management**]: Systems; I.2.6 [**Artificial Intelligence**]: Learning-Knowledge Acquisition

General Terms: Languages, Theory

Additional Key Words and Phrases: Algebra, Expressive power, Data Mining

1. INTRODUCTION

The mainstay of data mining research has largely concerned itself with algorithmic issues. The initial series of papers focused on efficient algorithms for individual

This is a preliminary release of an article accepted by ACM Transactions on Database Systems. The definitive version is currently in production at ACM and, when released, will supersede this version.
Contact author: Toon Calders. Address: Middelheimlaan 1, 2020 Antwerpen, Belgium. Tel.: (+32)32653264. Fax: (+32)32653777.
e-mail addresses: {toon.calders,jan.paredaens}@ua.ac.be, {rng,laks}@cs.ubc.edu
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 2006 ACM 0362-5915/2006/0300-0001 $5.00

mining tasks such as mining frequent itemsets, correlations, decision trees, and clusterings [Hand et al. 2001]. Subsequently, researchers realized the importance of setting mining in the larger context of knowledge discovery from databases (KDD) involving other components. One key component is how to integrate mining with the underlying database systems [Sarawagi et al. 1998; Netz et al. 2001; Chaudhuri et al. 2002]. As a further step, a few data mining "query languages" have been proposed [Han et al. 1996; Meo et al. 1996; Imielinski and Virmani 1999], based on ad hoc extensions to SQL. However, conspicuously absent from the above picture is a uniform model and algebraic framework for supporting data mining. By contrast, the relational model *started* with clean foundations and algebraic and logical frameworks. Substantial research then followed on both theoretical and systems fronts, leading to an extremely successful technology and science.

So how could an algebraic framework impact and further the field of data mining today? Firstly, it is widely agreed (e.g., see [Imielinski and Mannila 1996; Boulicaut et al. 1999]) that it is important to treat results of mining on a par with data objects and manipulate them further. Secondly, there are natural and useful mining computations which can only be expressed as combinations and compositions of known mining tasks. For example, an analyst might find a collection of frequent itemsets bought. He may further analyze these sets using a decision tree to determine under what situations such frequent co-purchases are made. See Section 2 for more detailed examples. However, in the current state of affairs where each mining task is identified with specific algorithms, such "compositions" are not easy. Besides, since each mining task is treated as a "black box", there is little scope for optimization. Thirdly, while a wealth of knowledge is available about the expressive power of database query languages, the field of data mining is in its infancy as far as such foundational pursuits are concerned, where we can ask questions like "what is (or is not) data mining?" and hope to answer them.

In this paper, we present a database model and an algebraic framework for data mining. The framework allows both data and the results of mining to be manipulated as first class citizens. The framework presented here is based on the 3W model [Johnson et al. 2000]. While an overview of the extended 3W model will be presented in the next section, it suffices to say that the 3W model from Johnson et al. [2000] relied on black box mining operators. A first contribution of this paper is to extend the 3W model by "opening up" these black boxes, by using generic operators in a data mining algebra.

Then, the resulting data mining algebra $\mathcal{MA}$ is studied and properties concerning the expressive power and complexity are established. We present results in three directions: (1) expressiveness of the mining algebra; (2) relations with alternative frameworks, and (3) interactions between the new generic operators.

As a preview, our contributions are as follows:

(1) The 3W-model proposed by Johnson et al. [2000] is extended and further refined.

(2) Examples, such as frequent itemset mining and decision trees show the usefulness and expressiveness of the model. In particular, the framework allows to treat the results of data mining (i.e., regions or patterns) on a par with data objects and allows them to be manipulated further via algebraic operations.

(3) Theoretical results about properties of the algebra are given. This study shows that our model allows to reason about integration and combination of different data mining problems at a formal level.

The organization of the paper is as follows. In Section 2, we give several motivating examples of the significance of an algebraic framework for data mining. In section 3, we introduce the three worlds of the 3W-model. Each of the three worlds and its function is discussed in detail. In Section 4, the algebra operations that form the bridges between the three worlds are introduced. In Section 5, the data mining algebra is formally studied. Section 6 discusses related work. In Section 7, we discuss the advantages and the limitations of the algebra, as well as optimization opportunities. Section 8 concludes the paper.

2. MOTIVATING EXAMPLES

All previous studies are almost always geared toward one mining task at a time. In real data mining applications, KDD is rarely a one-shot activity. Rather, it is a *multi-step process* involving different mining operations, data partitioning, aggregation, and data transformations. Thus, previous work fails to address the fundamental need of supporting KDD as a multi-step process. In the following examples, we illustrate several multi-step scenarios, extracted from real mining applications, which call for the ability to manipulate (e.g., analyze, query, transform) the results of mining tasks, making the output of one mining operation the input to another.

EXAMPLE 1. **Associations and Decision Trees** Suppose an analyst analyzes the sales data of a chain store to determine which items were co-purchased with a certain promotional item p, generating a collection of frequent sets. As part of his exploration, he decides to roll up this collection of frequent sets from specific items (e.g., specific brands of meat products) to kinds of items (e.g., the general class of meat). He then wishes to determine the "circumstances" (e.g., `location`, `time`, etc.) under which the frequent co-purchases were made. He does so by constructing a decision tree. The decision tree, when combined with frequent sets, might reveal interesting patterns such as "in northern New Jersey, meat products (not dairy products) are often bought together with p, whereas in southern New Jersey, dairy products (not meat products) are often bought together with p." This example illustrates interesting observations/patterns that can *only* be discovered by freely combining the outcomes of different mining tasks.

EXAMPLE 2. **Stacking Decision Trees** Suppose T_1 is a decision tree that classifies customers in New Jersey into the categories of `highRisk` and `lowRisk` for credit rating. Let T_2 be a decision tree that predicts under what conditions people in New Jersey live in cities vs. the countryside. The analyst may want to combine the two decision trees so as to be able to predict under what conditions people have a certain credit rating and tend to live in a certain neighborhood.[1] One option is to take a cross product between T_1 and T_2. An alternative is to "stack" T_2 below T_1, i.e. each leaf of T_1 is further classified on the basis of T_2 (see Figure 1). Such a classification may be further analyzed, e.g., used as a basis of a group-by.

[1]The training data that led to the two trees may be presently unavailable to the analyst, or doing the combined classification from scratch may take too long for his purpose.

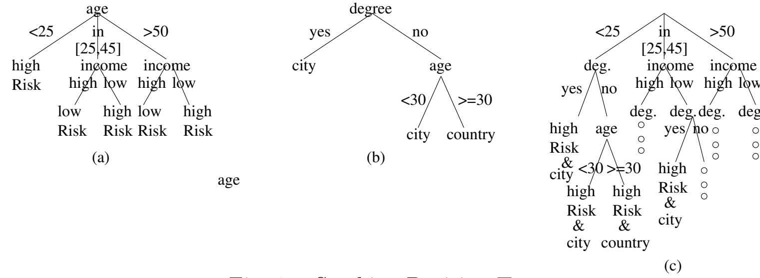


Fig. 1. Stacking Decision Trees

EXAMPLE 3. **Computing Special Regions** Consider a sales data warehouse with measures like `revenue`, `profit`, and dimensions such as `part`, `time`, `location`. The dimensions may have associated hierarchies. For example, a "region" such as `location` = 'quebec/montreal' may be a child of the region `location` = 'quebec'. Suppose the analyst wants to find the minimal regions which satisfy some aggregate property $\mathcal{P}$, e.g., $\mathcal{P} \equiv$ "the total sales exceeds $100,000", where minimality means children of the region do not satisfy $\mathcal{P}$. The analyst might similarly want to find regions whose sales are significantly different from their siblings.

A key aspect exhibited by the above examples is that the data set/space is split by data mining/analysis operations into (possibly overlapping) subsets or "regions". This observation naturally leads to the development of the 3W model.

3. THE 3W MODEL

"3W" stands for the "Three Worlds" for data mining: the D(ata)-world, the I(ntensional)-world, and the E(xtensional)-world. Ordinary data are manipulated in the D-world, regions representing results of mining manipulated in the I-world, and extensional representations of regions manipulated in the E-world. Below we give an overview of these three worlds. See [Johnson et al. 2000] for more details on the original proposal.

Before we start the description of the 3W-model as used in this paper, to avoid confusion for those familiar with the original 3W-proposal [Johnson et al. 2000], we would like to remark that the 3W-model proposed here has some subtle, yet important differences with the model proposed by Johnson et al. [2000]. First of all, the algebraic framework we propose here will contain very important new operators to manipulate regions: the *mining loop*, the grouping and the ungrouping operators. These new operators enable us to open up the black box mining operators proposed by Johnson et al. [2000]. Secondly, for the sake of theoretical evaluation, the model has been simplified as much as possible, without losing the essence. For example, in [Johnson et al. 2000], there was a large emphasis on the attribute domains being *hierarchically organized*. In practical systems, such an ordering of the domains will be a key feature that cannot be neglected. For the theory developed here, however, such an hierarchical ordering of the domains is immaterial, as it can be simulated easily by adding relations that express the hierarchy. E.g., the hierarchical ordering of cities into states can easily be simulated with an explicit relation `City_State`. Another difference is that in [Johnson et al. 2000], so-called *region identifiers* were used to refer to regions. Here, however, we have opted not to explicitly model

region identifiers, but to consider them merely as implementation details.

To illustrate the 3W-model, we will consider the following use case as a running example throughout this and the next section.

EXAMPLE 4. *Let two relations are given:* `Training(shp, col, odor, pois)`*, and* `Testing(shp, col, odor, pois)`*. Based on the contents of* `Training`*, a decision tree $\mathcal{T}$ is constructed that predicts the value of* `pois` *based on the values in* `shp`*,* `col`*, and* `odor`*. Then, the correctness of the tree $\mathcal{T}$ is evaluated on the relation* `Testing`*. The tuples that are misclassified in* `Testing` *are selected and stored in* `Misses(shp, col, odor, pois)`*. In this example, the relations* `Testing`*,* `Training`*, and* `Misses` *live in the D-world. The tree is constructed in the I-world. For the evaluation of the tree on the relation* `Testing` *and the subsequent construction of the relation* `Misses`*, the tree $\mathcal{T}$ and the relation* `Training` *are combined in the E-world.*

3.1 The D-World

In the D-world, data are represented as nested relations. We assume standard notions of the relational database model here. In particular, we assume an infinite set of attribute names $\mathcal{A}$, and two basic attribute types – categorical with domain $\mathcal{U}$, for some countable set $\mathcal{U}$, and rational with domain $\mathbf{Q}$. Every attribute $A \in \mathcal{A}$ has a domain $dom(A)$, which is either $\mathcal{U}$ or $\mathbf{Q}$. *Composed attributes* are recursively defined as follows: every attribute is a composed attribute, and if $B_1 \ldots, B_k$ are composed attributes, then $\{B_1 \ldots, B_k\}$ is a composed attribute. A schema in the D-world is a finite set of composed attributes. Tuples, relations, and domain are introduced in the usual (recursive) way; that is:

(1) A tuple t over schema $\{B_1, \ldots, B_k\}$ is a function from $\{B_1, \ldots, B_k\}$ to $\cup_{i=1}^{k} B_i$ such that $t(B_i) \in dom(B_i)$, $i = 1 \ldots k$.
(2) A relation R over schema $\{B_1, \ldots, B_k\}$ is a finite set of tuples over $\{B_1, \ldots, B_k\}$.
(3) The function dom is extended to composed attributes: $dom(\{B_1, \ldots, B_k\})$ is the set of all relations over the schema $\{B_1, \ldots, B_k\}$.

The set consisting of all attributes and composed attributes is denoted $\overline{\mathcal{A}}$. A D-world database is a finite set of D-world relations.

For notation convenience, we use the following conventions: $D(A_1, \ldots, A_n)$ denotes a relation D over the attributes $A_1, \ldots, A_n$. $(a_1, \ldots, a_n) \in D(A_1 \ldots A_n)$ denotes the tuple t that maps A_i to a_i, $i = 1 \ldots n$. Finally, we will use $t[A_1, \ldots, A_k]$ to denote the restriction of a tuple t to the attributes $A_1 \ldots, A_k$.

The algebra for the D-world is the nested relational algebra extended with aggregation (with the standard set of aggregate functions SUM, COUNT, AVG, MIN, MAX) and arithmetic. We have chosen for the *nested* relational model because nesting makes it easier to express more complex structures without adding too much expressive power. E.g., Paradaens and Van Gucht [1998] show that any query from a flat relation to a flat relation expressible in nested relational algebra is also expressible without nesting. Furthermore, for many data mining problems, the input or the output are given as a collection of sets. For example, for frequent itemset mining, the input consists of a database of sets, the result of a clustering is a partitioning of the tuples into disjoint sets, etc.

The D-world algebra hence consists of the following algebraic operations:

(1) The traditional relational operators: $\pi_{\langle attributes \rangle}$ (projection), $\sigma_{condition}$ (selection), $\cup$ (union), $\setminus$ (set difference), $\times$ (Cartesian product), and $\rho_{attr \to attr}$ (renaming).

(2) $Nest_{\langle attributes \rangle}$ (nesting), and $Unnest_{attribute}$ (unnesting). Let $D(A, B)$ be a relation. $Nest_B\ D$ is the following relation over the schema $\{A, \{B\}\}$:

$$\{(a, [b]) \mid [b] = \{b \mid (a, b) \in D\}, [b] \neq \{\}\} \ .$$

For unnest, let $D'(A, \{B\})$ be a relation over $\{A, \{B\}\}$. Then, $Unnest_B\ D'$ is the following relation over the schema $\{A, B\}$: $\{(a, b) \mid \exists s \in D' : s(A) = a \ \wedge \ b \in s(\{B\})\}$.

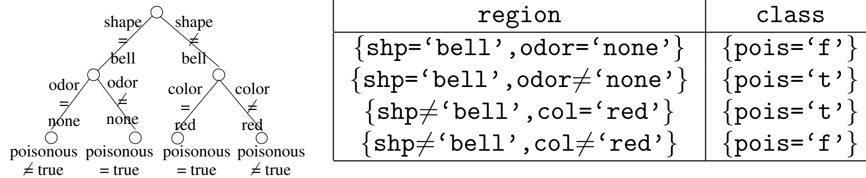
(3) $\Gamma^{Aggregate \texttt{ as } attribute}_{\langle grouping\ attributes \rangle}$ (aggregation): for example, let $D(A, B, C)$ be a relation, with $dom(C) = \mathbf{Q}$. $\Gamma^{\texttt{SUM}(C) \texttt{ as } S}_{\langle A,B \rangle}\ D$ is the following relation over $\{A, B, S\}$: $\{(a, b, s) \mid \exists v : (a, b, v) \in D \ \wedge \ s = \sum\{t[C] \mid t \in D, t[A, B] = (a, b)\}\}$. Notice that the grouping attributes can be the empty list, indicating that only one tuple (s) is constructed, where s is the sum of all values in C.

(4) $Calc\ _{expr \texttt{ as } attribute}$ (arithmetic): for example, let D be a relation over the schema $\{A, B\}$, then, $Calc\ _{\texttt{A+B as S}}\ D$ is the relation over $\{A, B, S\}$ that consists of a tuple $(a, b, a + b)$ for every tuple (a, b) of D. Here, *expr* is any arithmetic expression over attributes of D.

EXAMPLE 5. *In the D-world we can manipulate our relations* `Training` *and* `Testing` *with the nested relational algebra with aggregation and arithmetic. For example, we can count the number of instances in each class in the training set with* $\Gamma^{\texttt{COUNT}(*) \texttt{ as Nr}}_{\langle \texttt{pois} \rangle}$ `Training`*, where* `*` *denotes all attributes in the schema of* `Training`*. The ratio of correctly classified examples versus the total number of examples can be calculated as follows:*

$$Calc_{\texttt{(t-m)/t as ratio}} \left(\Gamma^{\texttt{COUNT}(*) \texttt{ as t}}_{\langle \rangle}\ \texttt{Testing} \times \Gamma^{\texttt{COUNT}(*) \texttt{ as m}}_{\langle \rangle}\ \texttt{Misses} \right)$$

3.2 The I-World



region	class
{shp='bell',odor='none'}	{pois='f'}
{shp='bell',odor≠'none'}	{pois='t'}
{shp≠'bell',col='red'}	{pois='t'}
{shp≠'bell',col≠'red'}	{pois='f'}

Fig. 2. Example decision tree

Objects in the I-world correspond to regions, defined by a set of constraints. A region is defined as a finite set of basic constraints over attributes. A *basic* constraint over the set of attributes $\mathcal{A}$ is an expression of one of three forms:

(1) $\alpha_1 A_1 + \ldots + \alpha_k A_k\ \theta\ \beta$, with α_i's and β rational numbers, θ is $<$, $\leq$, $=$, or $\neq$, and all A_i's have domain $\mathbf{Q}$; or,

(2) $A \theta u$, with $dom(A) = \mathcal{U}$, $u \in \mathcal{U}$, and θ is $=$ or $\neq$; or,

(3) $\{x_1, \ldots, x_k\}\theta\{B\}$, with $dom(B) = dom(\{A_1, \ldots, A_k\})$, and x_i is either A_i, or an element of $dom(A_i)$. θ is either $\in$ or $\notin$.

The semantics of a region $reg = \{C_1, \ldots, C_n\}$ is the set of all data tuples that satisfy the conjunction $\bigwedge_{i=1\ldots n} C_i$.

The set of all regions is denoted $\mathcal{REG}$.

The equality on regions is defined *syntactically*. That is, two regions $\{C_1, \ldots, C_n\}$ and $\{C'_1, \ldots, C'_{n'}\}$ are considered the same if and only if both sets contain exactly the same expressions. Hence, even though the regions $\{A \geq 5, A \leq B\}$ and $\{A \geq 5, A \leq B, B \geq 5\}$ have the same semantics (since $(A \geq 5) \wedge (A \leq B) \wedge (B \geq 5)$ is equivalent to $(A \geq 5) \wedge (A \leq B)$), they are considered to be different. The reason for defining the equality to be syntactic is twofold; firstly, *semantic* equivalence will be expressible with the selection predicate, as substantiated later. Secondly, we will often consider a region as being defined by a set of constraints, that can be composed and decomposed. In this perspective, considering two regions to be equivalent if they are semantically the same, would result in misleading interpretations. This effect becomes especially important in the E-world, where we combine regions with data, but at the same time we do not want to allow manipulations of the regions, as such operations belong to the I-world.

We assume an infinite countable set of *region description attributes* $\mathcal{RDA}$. The function *dom* is extended to $\mathcal{RDA}$ as follows: for all $R \in \mathcal{RDA}$, $dom(R) = \mathcal{REG}$. A schema in the I-world is a finite subset $\mathcal{RDA}$. An I-world tuple t over an I-world schema $\{R_1, \ldots, R_m\}$ is a function from $\{R_1, \ldots, R_m\}$ to $\mathcal{REG}$, an I-world relation $\mathcal{R}$ over a schema $\{R_1, \ldots, R_m\}$ is a finite set of tuples over that schema, and an I-world database is a finite set of I-world relations.

For manipulating regions, the *dimension algebra* allows the following operators:

(1) *Relational operators:* ρ (rename), π (project), $\times$ (Cartesian product), $\setminus$ (minus), $\cup$ (union), and σ (select), which have their usual meaning. That is, $\mathcal{REG}$ is treated as an ordinary domain, with equality defined syntactically. Furthermore, on the domain $\mathcal{REG}$, the predicate $\preceq$ is defined as follows: $\{C_1, \ldots, C_n\} \preceq \{C'_1, \ldots, C'_{n'}\}$, if and only if $\bigwedge_{i=1\ldots n'} C'_i$ is a logical consequence of $\bigwedge_{i=1\ldots n} C_i$. Thus, the set of *points* described by $\{C_1, \ldots, C_n\}$ is included in that described by $\{C'_1, \ldots, C'_{n'}\}$. For example, $\{A \leq 5, B \geq 3\} \preceq \{A \leq 7, B \geq 3\}$. We will use $r_1 \prec r_2$ to denote $r_1 \preceq r_2 \ \wedge \ r_1 \neq r_2$.

The relational operators in the I-world allow for comparing regions and forming complex relations between them. For example, let $\mathcal{R}(R)$ be an I-world relation. The following query constructs a relation over (R_1, R_2) consisting of tuples (r_1, r_2), such that $r_1 \prec r_2$, and for every other region r in $\mathcal{R}$, such that $r_1 \preceq r \preceq r_2$, r is either r_1 or r_2. Put otherwise, r_2 is a direct ancestor of r_1 according to $\preceq$ when restricted to the regions in $\mathcal{R}(R)$.

$$\sigma_{R_1 \prec R_2}(\rho_{R \to R_1}\mathcal{R} \times \rho_{R \to R_2}\mathcal{R}) \setminus \pi_{R_1,R_2}(\sigma_{R_1 \prec R \prec R_2}(\rho_{R \to R_1}\mathcal{R} \times \mathcal{R} \times \rho_{R \to R_2}\mathcal{R}))$$

E.g., below, a relation, and the result of this query applied to it have been

given:

R
$\{A<3\}$
$\{A<4\}$
$\{A<4, B<5\}$
$\{A<4, B<3\}$

$\rightarrow$

R_1	R_2
$\{A<3\}$	$\{A<4\}$
$\{A<4, B<5\}$	$\{A<4\}$
$\{A<4, B<3\}$	$\{A<4, B<5\}$

(2) *Grouping and Ungrouping:* Besides being able to express complex relationships between the regions, it is also important to be able to decompose and recombine the basic constraints to form new regions. This functionality is provided by the following *group* and *ungroup* operators.

Although these operations are relatively simple, they allow to express many complex operations on constraints. Ungrouping allows for decomposing a constraint into its basic building blocks, while grouping does the opposite, that is, constructing more complex regions by combining together sets of constraints.

Grouping $G_R(\mathcal{R})$, for an I-world relation $\mathcal{R}$, and an $\mathcal{RDA}$ attribute R of $\mathcal{R}$, does the following: partitions $\mathcal{R}$ on the basis of equality on all attributes except R; for each block of tuples $\{t_1, ..., t_k\}$ in the partition, output one tuple t, such that $t[R'] = t_i[R']$, for all attributes R' of $\mathcal{R}$ except R, where i is any one of $1, ..., n$, and $t[R] = \bigcup_{1 \leq j \leq n} t_j[R]$. E.g., consider the following I-world relation $\mathcal{R}$ over the attributes R_1, R_2:

R_1	R_2
$\{A<5, B<4\}$	$\{\}$
$\{B<3\}$	$\{\}$
$\{A<5\}$	$\{C>5, D+E>4\}$
$\{A<5, B>3\}$	$\{C>5, D+E>4\}$

.

The result of the operation $G_{R_1}(\mathcal{R})$ is the following relation $\mathcal{R}'$, in which the sets in the R_1-attribute have been aggregated by taking the union:

R_1	R_2
$\{A<5, B<3, B<4\}$	$\{\}$
$\{A<5, B>3\}$	$\{C>5, D+E>4\}$

.

Notice that grouping an attribute is very similar to the nesting operator of the nested relational algebra. The only difference is that grouping does not pack elements in a set, but instead takes the union. The reason for this difference is that in the I-world all regions are defined by sets of constrains, and not sets of sets.

Ungrouping $U_R(\mathcal{R})$, for an I-world relation $\mathcal{R}$ and an $\mathcal{RDA}$ attribute R, replaces each tuple $t \in \mathcal{R}$ with $t[R] = \{C_1, \ldots, C_n\}$ by n tuples $t_1, \ldots, t_n$ with $t_i[R] = \{C_i\}$, and $t_i[R'] = t[R']$, for all attributes R' except R. E.g., consider again the I-world relation $\mathcal{R}'$ given above. The result of the operation $U_{R_1}(\mathcal{R}')$ is the

following relation:

R_1	R_2
$\{A<5\}$	$\{\}$
$\{B<3\}$	$\{\}$
$\{B<4\}$	$\{\}$
$\{A<5\}$	$\{C>5, D+E>4\}$
$\{B>3\}$	$\{C>5, D+E>4\}$

.

Ungrouping an attribute is very similar to unnesting; the only difference being that ungrouping does not unpack a set, but instead partitions it into its singleton subsets. The reason for this difference is again that in the I-world all regions need to stay sets of constrains.

With grouping and ungrouping and the relational operators together, we can implement various set-operations at the syntactic level. For example, given a binary relation in the I-world, we can construct for every tuple (r_1, r_2), the region that corresponds to the union of the sets of constraints, that is $r_1 \cup r_2$. We can also express the intersection, and the minus. We now illustrate the minus. Let $\mathcal{R}(R_1, R_2)$ be an I-world relation. We show how for each tuple (r_1, r_2) we calculate the tuple $(r_1,\ r_2,\ r_1 \setminus r_2)$.

$$\mathit{Diff}(\mathcal{R}) := G_{\texttt{Diff}}(U_{\texttt{Diff}}\pi_{R_1,R_2,R_1\ as\ \texttt{Diff}}\ \mathcal{R} \setminus\ U_{\texttt{Diff}}\pi_{R_1,R_2,R_2\ as\ \texttt{Diff}}\ \mathcal{R})$$

E.g., the result of $\mathit{Diff}(\mathcal{R})$, with $\mathcal{R}$ the following relation:

R_1	R_2
$\{A<5, B<3, B<4\}$	$\{A<5\}$
$\{A<5, B>3\}$	$\{B>4\}$

,

is the relation

R_1	R_2	Diff
$\{A<5, B<3, B<4\}$	$\{A<5\}$	$\{B<3, B<4\}$
$\{A<5, B>3\}$	$\{B>4\}$	$\{A<5, B>3\}$

,

Unlike in the D-world, no arithmetic and aggregation are allowed in the I-world, since these operations would create non-$\mathcal{RDA}$ attributes.

Example 6. *In Figure 2, an example decision tree is given. In the root of the tree, the condition* `shp='bell'` *is tested. Tuples satisfying* `shp='bell'` *go to the left branch, the other tuples follow the right branch. As such, the instance space can be split into two regions: the region defined by* `shp='bell'` *and the region defined by* `shp≠'bell'`*. Also in Figure 2, a possible translation of a decision tree as a set of regions can be seen.*

3.3 The E-World

While a region in the I-world is represented by the defining set of constraints, a region in the E-world is represented by an explicit enumeration of the data tuples contained in the region. Aggregations by region of data tuples will be performed here. The relations in the E-world will be formed by combining a D-world relation with an I-world relation. The resulting E-world relation will contain both region

$\mathcal{T}$

region	class
{shp='bell',odor='none'}	{pois='f'}
{shp='bell',odor≠'none'}	{pois='t'}
{shp≠'bell',col='red'}	{pois='t'}
{shp≠'bell',col≠'red'}	{pois='f'}

Testing

shp	col	odor	pois
bell	red	none	f
bell	yellow	none	f
flat	yellow	anise	t
flat	green	none	f

Cl

region	class	shape	color	odor	poison
{shp='bell',odor='none'}	{pois='f'}	bell	red	none	f
{shp='bell',odor='none'}	{pois='f'}	bell	yellow	none	f
{shp≠'bell',col≠'red'}	{pois='f'}	flat	green	none	f

Fig. 3. The E-world relation `Cl` formed by populating $\mathcal{T}$ and `Testing`

description attributes and regular D-world attributes. Subsequently, in the E-world, the data can be aggregated and selected, and then the results can be transferred back to their respective worlds.

Hence, an E-world schema is the union of a D-world schema and an I-world schema. Let $\{R_1, \ldots, R_m, B_1, \ldots, B_n\}$ be an E-world schema, with $R_1, \ldots, R_m \in \mathcal{RDA}$, and $B_1, \ldots, B_n \in \overline{\mathcal{A}}$. An E-world relation over $\{R_1, \ldots, R_m, B_1, \ldots, B_n\}$ is a finite set of tuples over $\{R_1, \ldots, R_m, B_1, \ldots, B_n\}$, where a tuple again is defined as a function on the attributes that respects the domains.

The algebra in the E-world is very restricted. This is because in the E-world we want to maintain the strong connection between the regions and the tuples. Therefore, operators that can break this connection, such as projection and Cartesian product are disallowed. The E-world algebra thus consists of only the following operators:

(1) $\Gamma^{\mathit{Aggregate}\ \texttt{as}\ \mathit{attribute}}_{\langle \mathit{grouping\ attributes} \rangle}$ (aggregation) and $\mathit{Calc}\ _{\mathit{expr}\ \texttt{as}\ \mathit{attribute}}$ (arithmetic), as defined in the D-world. Since E-world relations also have region description attributes, the grouping attributes can be regions as well. Again two regions are considered the same if and only if they are syntactically equal; that is, they are exact the same set of basic constraints. The new attributes that hold the result of the aggregation and arithmetic, must be attributes from $\mathcal{A}$ with domain $\mathbf{Q}$.

(2) $\sigma_{condition}$ (selection). Selection is defined as for the relational algebra. Again, $\mathcal{RDA}$ attributes are interpreted syntactically.

EXAMPLE 7. *We continue our running example. The D-world contains the relation* `Testing`*, and the I-world the relation* $\mathcal{T}$(`region`, `class`) *that is given in Figure 2, that represents a decision tree. In the E-world these two relations can be combined to form the relation* `Cl`(`region`, `class`, `shp`, `col`, `odor`, `pois`). *For every tuple* $(\{r_1, \ldots, r_n\}, \{c\})$ *of* $\mathcal{T}$*, and every tuple* (s, c, o, p) *of* `Testing` *that satisfies all constraints* $r_1, \ldots, r_n, c$*,* `Cl` *contains the tuple* $(\{r_1, \ldots, r_n\}, \{c\}, s, c, o, p)$*. In Figure 3, an example of combining* $\mathcal{T}$ *and* `Testing` *is given. The correctly classified examples of* `Testing` *can be obtained by projection on the D-world attributes of* `Cl`*. We call the relation consisting of the correct examples* `Correct`*. Finally, the*

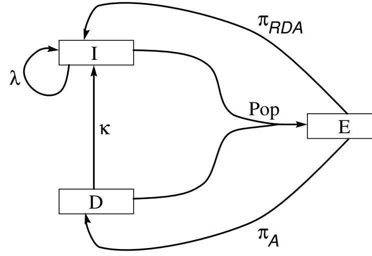


Fig. 4. Bridges between the worlds

relation `Misses` *can be defined as* `Misses` $=$ `Testing` $\setminus$ `Correct`*.*

3.4 3W-Database

A database in each world is a set of relations in that world. A *3W-database* is now defined as a triplet $(\mathbf{D}, \mathbf{I}, \mathbf{E})$ where $\mathbf{D}$, $\mathbf{I}$, and $\mathbf{E}$ are respectively D-, I-, and E-world databases.

4. MINING AND OTHER BRIDGING OPERATIONS

Till now, we have described intra-world manipulations. Next, we describe how to move from one world to another, which we call bridging operations. The most important bridging operations are those performing data mining. Specifically, a mine operation $\mu()$ takes data from the D-world to construct a set of regions in the I-world. In [Johnson et al. 2000], each data mining task was represented by a black box operator from the D-world to the I-world, e.g., $\mu_1()$ for frequent sets, $\mu_2()$ for decision tree, etc. In this paper, we study the alternative of replacing these black box operators by generic operators introduced in the next subsection: the *regionize* κ and the *mining loop* λ operators. Figure 4 summarizes the three worlds and the bridging functions. There is no edge labelled μ, as it is replaced by the two new operators – loop and regionize.

4.1 Bridges From and To the E-world

An important bridging operation is the populate operation, which associates with each region, the set of data tuples contained in it. More precisely, $Pop(D, \mathcal{R})$, upon inputs D a relation in the D-world, and $\mathcal{R}$ an I-world relation, yields a relation E in the E-world with schema $schema(D) \cup schema(\mathcal{R})$. E consists of the concatenation of a tuple $t_d \in D$ and a tuple $t_r \in \mathcal{R}$, whenever t_d satisfies every constraint associated with t_r.

Furthermore, to go from the E-world to the I-world, we use $\pi_{\mathcal{RDA}}(E)$ to retain only the $\mathcal{RDA}$ attributes. To go from the E-world to the D-world, we use $\pi_{\mathcal{A}}(E)$ to retain only the non-$\mathcal{RDA}$ attributes.

EXAMPLE 8. *In Figure 3, an example of the populate operation is given. Starting from the I-world relation $\mathcal{T}$ and the D-world relation* `Testing`*,* $Pop(\texttt{Testing}, \mathcal{T})$ *creates an E-world relation, which is called* `Cl` *in Figure 3.*

To go from E to I or D, we can use projection. $\pi_{\mathcal{RDA}}$ `Cl` *creates a new relation in the I-world over the attributes* `region` *and* `class` *that consists of two tuples (the*

first two tuples of `Cl` *collapse on the* $\mathcal{RDA}$*-attributes).*

$\pi_{\mathcal{A}}$ `Cl` *creates the D-world relation* `Correct` *as explained in Example 7.*

4.2 The Mining Operations

4.2.1 *The Basic Regionize Operator.* The regionize bridging operator is used to create regions from data. The format of this operator is: $\kappa_{template \texttt{ as } attr}(D)$, where $D(B_1, \ldots, B_n)$ is a D-world relation, and *template* is a constraint over the attributes $B_1, \ldots, B_n$, and over the terms $val(B_1), \ldots, val(B_n)$. *template* gives a description of how to compute a constraint from a tuple over the schema $\{B_1, \ldots, B_n\}$. The result of $\kappa_{template \texttt{ as } R}(D)$ is the following I-world relation $\mathcal{R}$ over schema $\{R\}$: for each tuple $(b_1, \ldots, b_n)$ of D, $\mathcal{R}$ will contain the tuple $(\{template[b_1, \ldots, b_n]\})$, where $\{template[b_1, \ldots, b_n]\}$ is the template with each occurrence of $val(B_i)$ replaced with b_i. The resulting expression must be a (syntactically) valid constraint.

Example 9. *Let* $D(A, B, C)$ *be the relation with tuples* $(2, 2, 4)$ *and* $(5, 1, 3)$*. The expression* $\kappa_{B+C \leq val(B)+val(C) \text{ as } R}(D)$ *creates the I-world relation over the schema* $\{R\}$ *with tuples* $(\{B + C \leq 6\})$*, and* $(\{B + C \leq 4\})$*.*

The definition is κ is simple. However, when combined with I-world manipulations, it can be quite powerful. Notice, e.g., that κ is not restricted to constraints over the attributes of one relation in the D-world. Indeed; we can take the Cartesian product of two relations, effectively allowing the construction of constraints over pairs of tuples, or we can use the aggregation and arithmetic operations in the D-world to first construct new values, ... The expressive power of such combinations is clearly very large. For instance, we can express regions such as convex hulls and minimum bounding rectangles.

4.2.2 *The Mining Loop Operator.* The mining loop operator is inspired by the viewpoint that most data mining algorithms start out with an initial set of regions, possibly empty. This set of regions is then iteratively refined until a termination condition is satisfied. For example, in frequent itemset computation, the Apriori algorithm [Agrawal and Srikant 1994] begins with candidate singleton itemsets, each of which is represented as a region. These regions are refined to retain only the frequent ones. New candidate regions are then generated based on the frequent regions. This loop continues until no more candidates can be generated. This example motivates the following definition of a *mining loop*.

$\mathcal{R} := Seed$;
While($\Delta\mathcal{R}$) **loop**
  $\mathcal{R} := Expr(\mathcal{R})$;
End loop
Return $\mathcal{R}$;

The set *Seed* can be any I-world relation. In fact, *Seed* can be any algebraic expression returning a set of regions. In this way we can easily compose different data mining algorithms, by setting the output of one algorithm as the seed of another. *Expr* is an algebraic expression that returns an I-world relation. In the sequel, we will often use the following (more concise) notation $\lambda_{Expr[\mathcal{X}]}(\mathcal{S}eed)$ to denote the mining loop. $\mathcal{X}$ is a dedicated n-ary I-world relation symbol. *Expr* is

an arbitrary algebraic expression over the I-world. The result of λ is defined as the fixpoint of the loop, that is: let $\mathcal{X}_0 := \mathcal{S}eed$, and $\mathcal{X}_i := Expr(\mathcal{X}_{i-1})$, for all $i \geq 1$. The result of $\lambda_{Expr[\mathcal{X}]}(\mathcal{S}eed)$ is the first $\mathcal{X}_i$ in the sequence $\langle \mathcal{X}_0, \mathcal{X}_1, \ldots \rangle$ such that $\mathcal{X}_i = \mathcal{X}_{i+1}$. If no such fixpoint exists, the result of the loop is undefined.

The loop as defined here, only updates the value of one variable $\mathcal{X}$ between the different iterations, and stops when a fixpoint for $\mathcal{X}$ has been reached. It can be proven though, that the extensions where more than one variable is allowed, or the condition (testing the non-emptiness of) $\Delta\mathcal{X}$ can be any comparison between expressions, do have the exact same expressive power as the loop defined here. For example, let $\mathcal{S}$ and $\mathcal{R}_1, \mathcal{R}_2$ be I-world relations. Consider for example the following *extended* loop construction that will be used in the next subsection:

(1) $\mathcal{S} := Seed$;
(3) **While**($\mathcal{S} \neq \{\}$) **loop**
(4) $\quad\mathcal{R}_1 := Expr_1(\mathcal{S})$;
(5) $\quad\mathcal{R}_2 := Expr_2(\mathcal{R}_1, \mathcal{S})$;
(6) $\quad\mathcal{S} := Expr_3(\mathcal{R}_2)$;
(7) **End loop**
(8) **Return** $\mathcal{R}_2$;

It can be shown that such an *extended* loop can be expressed by the regular mining loop. For notational convenience we will often use the extended version of the loop. For details on how an extended loop can be expressed with a regular data mining loop, we refer to Appendix A.

4.3 A Full Example: Frequent Itemset Mining

In the following, we give an example of the operations introduced so far. Specifically, we show how to perform the frequent itemset [Agrawal et al. 1993] computation. In the next subsection, we also give another example showing how to perform decision tree construction.

To begin, the input is a D-world relation $D(\{\texttt{Product}\})$ that stores transactions as follows: $\{(\{a,b\}), (\{b,c\}), (\{a,b,c\})\}$.

The domain of `Product` is, e.g., the set of all products of a store, and the domain of the nested attribute $\{\texttt{Product}\}$ is consequently all sets of products.

Let the minimal support threshold be 2. Hence, we are interested in all sets of items I that are a subset of at least two sets in D. The itemsets will be represented as regions in the I-world. The region associated with itemset I will describe all transactions that contain it. That is, the itemset $\{a,b\}$ will be represented by the region $\{a \in \{\texttt{Product}\}, b \in \{\texttt{Product}\}\}$.

We get the set of the items I as follows: $Items := Unnest_{\{Product\}}\ D$. In our example, the result of this operation is the relation $\{(a), (b), (c)\}$ over the schema $\{\texttt{Product}\}$.

The regions associated with the singleton items are formed with κ:

$$\mathcal{C} := \kappa_{\texttt{val(Product)}\in\{\texttt{Product}\}\ \texttt{as Set}}\ Items\ .$$

The resulting relation is

Set
$\{a \in \{$Product$\}\}$
$\{b \in \{$Product$\}\}$
$\{c \in \{$Product$\}\}$

Counting the candidates in set $\mathcal{C}$ is done by combining $\mathcal{C}$ and D in the E-world:

$$P \;:=\; Pop(\mathcal{C}, D) \;\;,$$

resulting in

Set	{Product}
$\{a \in \{$Product$\}\}$	$\{a, b\}$
$\{a \in \{$Product$\}\}$	$\{a, b, c\}$
$\{b \in \{$Product$\}\}$	$\{a, b\}$
$\{b \in \{$Product$\}\}$	$\{b, c\}$
$\{b \in \{$Product$\}\}$	$\{a, b, c\}$
$\{c \in \{$Product$\}\}$	$\{b, c\}$
$\{c \in \{$Product$\}\}$	$\{a, b, c\}$

Subsequently, the support of the itemsets is counted with the aggregation function COUNT, and the sets with support higher than 2 are selected

$$E \;:=\; \sigma_{supp \geq 2}\, \Gamma^{\texttt{COUNT(\{Product\}) as supp}}_{\langle \texttt{Set} \rangle}\; P \;\;,$$

resulting in the E-world relation

Set	supp
$\{a \in \{$Product$\}\}$	2
$\{b \in \{$Product$\}\}$	3
$\{c \in \{$Product$\}\}$	2

which is projected back to the I-world by $\mathcal{F} \;:=\; \pi_{\mathcal{RDA}}\, E$. Note that in our running example, there is one $\mathcal{RDA}$-attribute, namely Set. $\pi_{\mathcal{RDA}}$ only retains the these attributes, and hence this expression gives the following I-world relation:

Set
$\{a \in \{$Product$\}\}$
$\{b \in \{$Product$\}\}$
$\{c \in \{$Product$\}\}$

New candidates can be formed by extending the successful candidates with one extra item that is not yet in the set, and then doing the monotonicity check. The first step, extending successful candidates with one extra item, can be done as follows:

$$\begin{aligned} \mathcal{S} \;&:=\; U_{\texttt{Set as Single}}\, \mathcal{F} \\ \mathcal{P}re\mathcal{C} \;&:=\; \rho_{\texttt{U} \longrightarrow \texttt{Set}} \pi_{\texttt{U}}\, Union(\sigma_{\texttt{Set} \not\supseteq \texttt{Single}}\; (\mathcal{F} \times \mathcal{S})) \end{aligned}$$

Union denotes the following expression that, for each tuple $(\{i_1, \ldots, i_n\}, \{i\})$ generates a new tuple $(\{i_1, \ldots, i_n\}, \{i\}, \{i_1, \ldots, i_n, i\})$:

$$Union(\mathcal{X}(\texttt{Set}, \texttt{Single})) \;:=\; G_U\; U_U \left(\begin{array}{l} \;\;\pi_{\texttt{Set,Single,Set as U}}\; \mathcal{X} \\ \cup\; \pi_{\texttt{Set,Single,Single as U}}\; \mathcal{X} \end{array} \right)$$

E.g., *Union* applied to the following relation:

Set	Single
$\{a \in \{\texttt{Product}\}, b \in \{\texttt{Product}\}\}$	$\{c \in \{\texttt{Product}\}\}$

results in the following relation:

Set	Product	U
$\{a \in \{\texttt{Product}\}, b \in \{\texttt{Product}\}\}$	$\{c \in \{\texttt{Product}\}\}$	$\left\{\begin{array}{c} a \in \{\texttt{Product}\}, \\ b \in \{\texttt{Product}\}, \\ c \in \{\texttt{Product}\} \end{array}\right\}$

.

The monotonicity check is then performed for an itemset $\{i_1, \ldots, i_k\}$: for every item i_j, check whether $\{i_1, \ldots, i_k\} \setminus \{i_j\}$ was a frequent set. First we make the relation $\mathcal{PC}$ (parent-child) that contains the tuples $(\{i_1, \ldots, i_n\}, \{i_1, \ldots, i_n\} \setminus \{i_j\})$ for all regions $\{i_1, \ldots, i_n\}$ in $\mathcal{C}$, and $j = 1 \ldots n$:

$$\mathcal{PC} := \pi_{\texttt{Set,C}}\ G_{\texttt{C}}\ \sigma_{\texttt{M}\neq\texttt{C}}\ U_{\texttt{C}}\ U_{\texttt{M}}(\pi_{\texttt{Set,Set as M,I as C}}\mathcal{P}re\mathcal{C})$$

Then we select those itemsets that have a child that was not frequent:

$$\mathcal{IF} := \pi_{\texttt{Set}}(\mathcal{PC} \setminus (\mathcal{P}re\mathcal{C} \times \mathcal{F}))$$

Finally, we get the new candidates by removing the sets in $\mathcal{IF}$: $\mathcal{C} := \mathcal{P}re\mathcal{C} \setminus \mathcal{IF}$. The complete candidate generating expression will be denoted $genCan(\mathcal{F})$.

The same procedure is repeated; the candidates are counted, frequent itemsets are selected, new candidates are generated, until no new candidates can be generated anymore.

These points combined give the following expression:

(1) $Items := Unnest_{\{Product\}} D$;
(2) $\mathcal{C} := \kappa_{\texttt{val(Product)}\in\{\texttt{Product}\}\texttt{ as Set}}\ Items$;
(3) **While**($\mathcal{C} \neq \{\}$) **loop**
(4) $P := Pop(\mathcal{C}, D)$;
(5) $E := \sigma_{supp\geq 2}\ \Gamma^{\texttt{COUNT(\{Product\}) as supp}}_{\langle\texttt{Set}\rangle} P$;
(6) $\mathcal{F} := \mathcal{F} \cup \pi_{\mathcal{RDA}}\ E$;
(7) $\mathcal{C} := genCan(\mathcal{F})$;
(8) **End loop**
(9) **return** $\mathcal{F}$;

4.4 Decision Tree Example

In this subsection we give a detailed description of an expression for constructing a decision tree in $\mathcal{MA}$. The expression is based on the well-known CART algorithm [Breiman et al. 1984] for tree induction. In the CART-algorithm a divide-and-conquer methodology is followed. For numeric data, CART will construct decision trees with binary splits of the form $A < n$, $A \geq n$, where A is an attribute, and n is a number. CART constructs the tree as follows. For a given database, first it is checked whether the labels of the instances are homogeneous. In this case, one node, labelled with this unique class label is returned. If the class label is not homogeneous in the database, then all possible ways to split the database are considered, and the split $A < n$, $A \geq n$ that maximizes a goodness measure, such as,

e.g., information gain, is chosen. CART is then recursively applied to the two parts defined by the split. The resulting trees are attached to the root by respectively an edge labelled $A < n$ and an edge labelled $A \geq n$.

In our example expression, we assume that the input of the problem is a relation $D(A_1, \ldots, A_n, DL)$ in the D-world of the form:

A_1	A_2	...	A_n	DL
a	b	...	c	0
a	d	...	c	1
b	c	...	e	1
...	...	...	...	...

For simplicity, we assume that for all attributes the domain is $\mathbf{Q}$. Each tuple in D is an instance, and DL, the decision label, identifies the class the instance is in.

The complete expression is given in Figure 5. In this section, we show step-by-step how the complete expression is constructed. The output will be an I-world relation $\mathcal{T}$ that holds the regions associated with the leaves of the tree, together with their label. That is, for each leaf in the tree we consider the set of all inequalities on the edges from the root to that leaf. The region associated with the leaf hence consists of this set of inequalities. For example, suppose that there is a leaf in the tree which is reached via edges labelled respectively $A < 3$, $B \geq 5$, and $C < 2$, and having class label "1". For this leaf, the relation $\mathcal{T}$ will contain the tuple

$$(\{A < 3, B \geq 5, C < 2\}, \{DL = 1\})$$

The complete set of all leaves with associated class labels describes the tree. For a more elaborated example, see Figure 3.

Before we give the expression for the decision tree construction, we first introduce two important constructions. The first construction selects one region from a set of regions in the I-world. This construction is important to handle those cases where we have to choose one region. This case occurs, for example, when there are multiple splits that evaluate to the exact same goodness measure. Even though each of the best splits is equally good, one has to be chosen for the correct working of the algorithm. The second construction we treat separately, is the computation of the gini-score for all splits.

4.4.1 *Selecting a Region.* In the decision tree example, we often need to select one region from a relation. This situation occurs, e.g., when we need to select an open node to expand, or when there are different splits that evaluate to the exact same goodness measure. In these situations, for the correct working of the decision tree expression, one of the regions or splits needs to be selected. In this subsection we will give an expression that unambiguously does select one region. Given a relation $\mathcal{S}(Reg)$ consisting of regions of the form

$$\{A_{i_1} \leq v_1, \ldots, A_{i_k} \leq v_k, A_{i_{k+1}} > v_{k+1}, \ldots, A_{i_l} > v_l\} \ ,$$

$SelectRegion(\mathcal{S})$ selects a unique region in $\mathcal{S}$. This selection is done by imposing a total linear order on the regions, and selecting the smallest region in $\mathcal{S}$ w.r.t. this order. Before we discuss the total order on the regions, we first introduce an order on the basic constraints. This *basic order* is then later on extended to regions.

Basic Order: Let $A_{i_1}\theta_1 v$ and $A_{i_2}\theta_2 w$ be two basic constraints. Then, $A_{i_1}\theta_1 v$ comes before $A_{i_2}\theta_2 w$ in the order iff either:

(1) $i_1 < i_2$, or

(2) $A_{i_1} = A_{i_2}$, and $\theta_1 =$"$\leq$", and $\theta_2 =$"$>$", or

(3) $A_{i_1} = A_{i_2}$, and $\theta_1 = \theta_2$, and $\{(A_{i_1}\theta_1 v)\} \prec \{(A_{i_2}\theta_2 w)\}$. (Recall that $\prec$ denotes strict containment of regions)

It is straightforward to create a relation $\mathcal{B}Order(B, A)$ that consists of all pairs $(\{A_{i_1}\theta_1 v\}, \{A_{i_2}\theta_2 w\})$, such that $A_{i_1}\theta_1 v$ comes before $A_{i_2}\theta_2 w$. We construct all expressions of the form $A_i \leq v$ with the following expression:

$$Expr_{A_i,\leq} \;:=\; \kappa_{A_i \leq val(A_i)\ as\ R}\ D\ .$$

Similarly, there are expressions $Expr_{A_i,>}$ that construct all expressions of the form $A_i > v$. The relation $\mathcal{B}Order(B, A)$ can now be formed with the following expression:

$$\begin{aligned}
&\rho_{R\to B} Expr_{A_1,\leq} \times \rho_{R\to A} Expr_{A_1,>} \\
\cup\ &\sigma_{B\prec A}(\rho_{R\to B} Expr_{A_1,\leq} \times \rho_{R\to A} Expr_{A_1,\leq}) \\
\cup\ &\sigma_{B\prec A}(\rho_{R\to B} Expr_{A_1,>} \times \rho_{R\to A} Expr_{A_1,>}) \\
\cup\ &\rho_{R\to B} Expr_{A_1,\leq} \times \rho_{R\to A} Expr_{A_2,\leq} \\
\cup\ &\rho_{R\to B} Expr_{A_1,\leq} \times \rho_{R\to A} Expr_{A_2,>} \\
\cup\ &\ldots \\
\cup\ &\rho_{R\to B} Expr_{A_n,\leq} \times \rho_{R\to A} Expr_{A_n,>} \\
\cup\ &\sigma_{B\prec A}(\rho_{R\to B} Expr_{A_n,\leq} \times \rho_{R\to A} Expr_{A_n,\leq}) \\
\cup\ &\sigma_{B\prec A}(\rho_{R\to B} Expr_{A_n,>} \times \rho_{R\to A} Expr_{A_n,>})
\end{aligned}$$

With the relation $\mathcal{B}Order$ it is easy to express the following query $First(\mathcal{S}(Reg))$, that creates the relation consisting of the tuples $(\{c_1, \ldots, c_k\}, \{c_i\})$, with c_i the first element among $c_1, \ldots, c_k$ in the basic order, and $(\{c_1, \ldots, c_k\})$ in $\mathcal{S}$, over the schema $\{Reg, F\}$:

$$\begin{aligned}
First(\mathcal{S}(Reg)) := (U_F\ \pi_{Reg,Reg\ as\ F}\ \mathcal{S}) \setminus \pi_{Reg,F}\ \sigma_{T=B,F=A} \\
((U_F\ U_T\ \pi_{Reg,Reg\ as\ T,Reg\ as\ F}\ \mathcal{S})\ \times\ \mathcal{B}Order)\ .
\end{aligned}$$

Order on Regions: We extend the total order on the basic constraints to a total order on regions. Because we have a total order on the basic constraints, we can represent the regions as a bit string; the ith bit in the bit string is 1 if and only if the basic constraint with rank i in the total order, is in the region. The total order on the regions that we construct, corresponds to the natural order on the binary numbers represented by the bit strings. In practice, this principle is implemented as follows. A region r_1 comes before another region r_2 iff either $r_2 \setminus r_1$ is empty (that is, every non-zero bit in the bit string of r_2 is also non-zero in r_1) or both $r_1 \setminus r_2$ and $r_2 \setminus r_1$ are nonempty, and the smallest basic constraint in r_2 comes before the smallest basic constraint in r_1 (that is, the first non-zero bit in the bit string for r_1 which is zero in the bit string of r_2 is more significant than the first non-zero bit for r_2 that is zero for r_1).

The following expression $\mathcal{O}rder(\mathcal{S})$ does the following: given a relation $\mathcal{S}(Reg)$, it returns the relation $\mathcal{O}rder(R_1, R_2)$ that consists of all pairs of regions (r_1, r_2)

with r_1, r_2 in $\mathcal{S}$, such that r_1 comes before r_2 in the total order on regions. First we construct all pairs of regions of $\mathcal{S}$.

$$P := \pi_{Reg\ as\ R_1}\ \mathcal{S}\ \times\ \pi_{Reg\ as\ R_2}\ \mathcal{S}$$

Then, for each pair (r_1, r_2), the tuple $(r_1, r_2, r_1 \setminus r_2, r_2 \setminus r_1)$ is formed using the difference operator *Diff* introduced in Section 2.2:

$$D := \mathit{Diff}_{R_2 \setminus R_1\ as\ D_2} \mathit{Diff}_{R_1 \setminus R_2\ as\ D_1}(P)\ .$$

$\mathcal{O}rder(\mathcal{S})$ consists of two parts $\mathcal{O}_1$ and $\mathcal{O}_2$ that are respectively those pairs (r_1, r_2) with $r_2 \setminus r_1$ empty, and those where the first element of $(r_1 \setminus r_2)$ comes before the first element of $(r_2 \setminus r_1)$ (the expression $First_{D_1\ as\ F_1, D_2\ as\ F_2}(D)$, denotes the construction that adds two attributes F_1 and F_2 to D that hold respectively the first basic constraint in D_1 and the first basic constraint in D_2. This construction is similar to $First(\mathcal{S})$):

$$\begin{aligned}
&\mathcal{O}_1 := \pi_{R_1,R_2}\ \sigma_{D_2=\{\}}\ D \\
&\mathcal{O}_2 := \pi_{R_1,R_2}\ \sigma_{B=F_2,A=F_2}(First_{D_1\ as\ F_1,D_2\ as\ F_2}(D) \times \mathcal{B}Order) \\
&\mathcal{O}rder(\mathcal{S}) := \mathcal{O}_1 \cup \mathcal{O}_2
\end{aligned}$$

Finally, we can select the first region of $\mathcal{S}$ in the order; the first region is the one that does not occur as second component in $\mathcal{O}rder$:

$$\mathit{SelectRegion}(\mathcal{S})\ :=\ \mathcal{S} \setminus \pi_{R_2\ as\ Reg}\ \mathcal{O}rder(\mathcal{S})\ .$$

4.4.2 *Computing the GINI-Index.* Let T denote a dataset with examples from n classes. The GINI-index $gini(T)$ of T is defined as $1 - \sum_{i=1}^{n} p_i^2$, where p_i is the relative frequency of the ith class in T. The GINI-index is often used to evaluate the quality of a split in the construction of a decision tree. Consider, e.g., a split that divides the dataset T into two disjoint datasets T_1 and T_2. The gini score of this split is defined to be $\frac{|T_1|}{|T|} gini(T_1) + \frac{|T_2|}{|T|} gini(T_2)$. The split that minimizes the score is considered to be the best one.

In this subsection we show how we can compute the gini-index in $\mathcal{MA}$. We assume the following setting: in the D-world we have the dataset D over attributes $A_1, \ldots, A_n, DL$, and in the I-world we have a relation $\mathcal{S}(Reg)$ consisting of different regions that in fact represent one side of the split. That is, for each region r in $\mathcal{S}$, we will evaluate the gini-index of splitting D into the set of tuples that do satisfy r, and the set of tuples that do not satisfy r.

First we compute the total number of positive examples, and the total number of negative examples:

$$\mathit{Stats}\ :=\ \Gamma_{\langle\rangle}^{\texttt{COUNT}(*)\ as\ npos}\ \sigma_{DL=1}\ D \times \Gamma_{\langle\rangle}^{\texttt{COUNT}(*)\ as\ nneg}\ \sigma_{DL=0}\ D$$

Then we populate the regions in $\mathcal{S}$ with D and the statistics:

$$P\ :=\ \mathit{Pop}(\mathcal{S}, D \times \mathit{Stats})$$

From P we can compute, for every region r, the numbers of positive examples np_1 and np_2, and the numbers of negative examples nq_1 and nq_2, in respectively r and the complement of r:

$$N := \mathit{Calc}_{npos-p_1\ as\ p_2, nneg-q_1\ as\ q_2} \Gamma_{\langle Reg,npos,nneg\rangle}^{\texttt{SUM(DL)}\ as\ p_1, \texttt{SUM(1-DL)}\ as\ q_1}\ P$$

The gini-index for the split based on r is

$$\frac{p_1+q_1}{npos+nneg}\left(1-\frac{p_1^2+q_1^2}{(p_1+q_1)^2}\right)+\frac{p_2+q_2}{npos+nneg}\left(1-\frac{p_2^2+q_2^2}{(p_2+q_2)^2}\right)$$

For notational convenience, we denote this sum *gini*. The expression for computing the gini-index for all splits is: $Gini(\mathcal{S}) := Calc_{gini\ as\ G}\ N$.

The minimal gini score is computed by aggregation. Recall that we did not include Cartesian product in the E-world. This absence of the Cartesian product makes the expression to select the regions with the minimal score a bit trickier. Using the connections between the worlds and D-world algebra, however, we can still express this query. Indeed, we can project the minimal gini score to the D-world, add it with the Cartesian product in the D-world to the relation *Stats*, and repeat the computation above. Of course, it is highly unrealistic to select the region with the minimal gini-score in this way in real situations. This construction, however, is of theoretical interest, as it shows that the Cartesian product in the E-world is not necessary to express the query under construction. For notational convenience we denote the expression selecting the tuples of $Gini(\mathcal{S})$ with minimal gini index as $\sigma_{G=\min(G)}\ Gini(\mathcal{S})$

4.4.3 *Decision Tree Expression.* The complete expression for the construction of a decision is now given in Figure 5. In step (1), the variable $\mathcal{T}$, which will hold the decision tree, is initialized to the empty tree. In (2), the set of open nodes $\mathcal{O}pen$ is set to the relation with one region $\{\}$. This empty region actually represents the constraint "true", which holds for all data points. In steps (3) to (5), variables $\mathcal{S}plits$, $\mathcal{N}egLabel$ and $\mathcal{P}osLabel$ are initialized. They represent respectively the set of all possible ways to split a node in the tree, the negative label, and the positive label. The initialization of these variables occurs before the mining loop, that is, κ is not used inside a mining loop. Then the main loop (6) to (22) is entered. In this loop, the tree will be refined until the set of open nodes does not change anymore. This occurs when either the tree is pure; i.e. $\mathcal{O}pen$ became empty, or, when the leaves of the tree are still impure, but cannot be split anymore. This second situation can happen when the input is inconsistent. The refinement in the loop goes as follows: first, an open node is selected in (7), and removed from $\mathcal{O}$pen in (8). It is checked whether the selected region in $\mathcal{S}$ is pure, by populating it with the input data in (9). If there are no instances of the positive class in the input that fall into the region in $\mathcal{S}$, then the region in $\mathcal{S}$ is purely negative. In that case, $\mathcal{S}$ is annotated with the negative label and added to the tree (lines (10) and (11)). Lines (12) and (13) handle the case if $\mathcal{S}$ is purely positive. If the region in $\mathcal{S}$ is neither purely positive, nor purely negative, it is further split in lines (14) to (20). First, all possible splits of the region in $\mathcal{S}$ are listed in (15), and scored in (16). In (17) and (18), one of the splits that minimizes the gini score is selected. Finally, in (19) and (20), the two regions resulting from splitting $\mathcal{S}$ are added to the set of open nodes. After the loop has ended, in (23), the list of pure regions forming the decision tree is returned.

(1) $\mathcal{T} := \{\}$;
(2) $\mathcal{O}pen := \{\{\}\}$;
(3) $\mathcal{S}plits := \kappa_{A_1<val(A_1)\ as\ RL, A_1\geq val(A_1)\ as\ RR}\ D$
$\cup\ \ldots$
$\cup\ \kappa_{A_n<val(A_n)\ as\ RL, A_n\geq val(A_n)\ as\ RR}\ D$
(4) $\mathcal{N}egLabel := \kappa_{DL=0\ as\ Class} D$;
(5) $\mathcal{P}osLabel := \kappa_{DL=1\ as\ Class} D$;
(6) **While**($\Delta\mathcal{O}pen$) **loop**
Select an open node and check if it is pure
(7) $\mathcal{S} := SelectRegion(\mathcal{O}pen)$;
(8) $\mathcal{O}pen := \mathcal{O}pen \setminus \mathcal{S}$;
(9) $P := Pop(\mathcal{S}, D)$;
(10) **if** ($\pi_{\mathcal{RDA}}\ \sigma_{DL=1} P = \{\}$) **then** *(pure, $DL = 0$)*
(11) $\mathcal{T} := \mathcal{T} \cup \mathcal{S} \times \mathcal{N}egLabel$;
(12) **else if** ($\pi_{\mathcal{RDA}} \sigma_{DL=0} P = \{\}$) **then** *(pure, $DL = 1$)*
(13) $\mathcal{T} := \mathcal{T} \cup \mathcal{S} \times \mathcal{P}osLabel$;
(14) **else** *(impure)*
Consider all splits and select the best
(15) $\mathcal{A}llSplits := Union_{Reg\cup RL\ as\ LReg, Reg\cup RR\ as\ RReg} \mathcal{S} \times \mathcal{S}plits$;
(16) $Score := Gini(\pi_{LReg}\ \mathcal{A}llSplits)$;
(17) $\mathcal{B}est := \pi_{\mathcal{RDA}}\ \sigma_{G=\min(G)}\ Gini(\mathcal{S})$;
(18) $\mathcal{S}electedBest := SelectRegion(\mathcal{B}est)$;
Add the split to the set of open nodes
(19) $\mathcal{B}\mathcal{S}plit := \sigma_{LReg=Reg} \mathcal{A}llSplits \times \mathcal{S}electedBest$;
(20) $\mathcal{O}pen := \mathcal{O}pen \cup \pi_{LReg}\ \mathcal{B}\mathcal{S}plit \cup \pi_{RReg}\ \mathcal{B}\mathcal{S}plit$;
(21) **end if**
(22) **End loop**
(23) **return** $\mathcal{T}$;

Fig. 5. Constructing a Decision Tree in $\mathcal{MA}$.

5. EXPRESSIVENESS OF DATA MINING QUERIES

In this section, we give formal results on the expressive power of the algebraic framework introduced in Sections 2 and 3. In this section we only give proof sketches of the theorems. For the full proofs we refer to the appendices.

The 3W-algebra introduced so far is denoted $\mathcal{MA}$. First we prove that on the one hand the mining algebra $\mathcal{MA}$ as is, is too powerful; it is computational complete. On the other hand, if we disallow looping, the resulting language $\mathcal{DA}$ has the same expressive power as the relational algebra. Therefore, we introduce a restricted form of looping, called the *static* mining loop. The restriction of $\mathcal{MA}$ to expressions with only static loops is denoted *static* $\mathcal{MA}$. It is shown that *static* $\mathcal{MA}$ is strictly less powerful than $\mathcal{MA}$. We also compare our framework with an alternative framework $\mathcal{MA}^{FI}$ without mining loop, but where frequent itemset mining is introduced as a black box operator. The relations between the different languages is studied in this section. The results we prove are summarized in Figure 6.

We begin with a definition of a data mining query. A *data mining query* is a

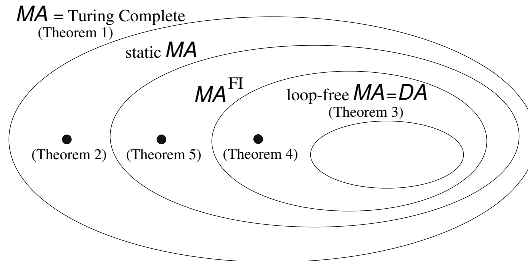


Fig. 6. Summary of the Results of Section 4

computable function that maps a 3W-database $\mathbf{W} = (\mathbf{D}, \mathbf{I}, \mathbf{E})$ to an X-database, where X is one of D, I, or E. We assume that there exist fixed input and output schemas for the query q; i.e., we want to avoid queries where the *schema* of the output depends on the input. Furthermore, following standard practice in data mining, we assume that in an input 3W-database to a data mining query, $\mathbf{I}$ and $\mathbf{E}$ are empty. We call such a query a $D{\rightarrow}X$ data mining query, reflecting that it maps a D-world database to an X-world database.

We will denote the language of all expressions we can form with the algebra operations in Sections 2 and 3 by $\mathcal{MA}$. An $\mathcal{MA}$-*expression* is any legal expression composed of the operators presented so far in this paper. A data mining query q is *expressible in* $\mathcal{MA}$ if there exists an $\mathcal{MA}$-expression $\mathcal{E}$ such that for each 3W-database $\mathbf{W}$, $\mathcal{E}(\mathbf{W}) = q(\mathbf{W})$. To facilitate the calibration of the expressive power of the mining algebras $\mathcal{MA}$, static $\mathcal{MA}$, and $\mathcal{MA}^{FI}$ in relation to known query languages, we mostly consider $D{\rightarrow}D$ data mining queries. We refer to the D-world base query language of nested relational algebra extended with aggregation and arithmetic as $\mathcal{DA}$. We now show that $\mathcal{MA}$-expressions, as such, are too powerful for our purposes.

Before we give the theorem, we first introduce the notion of *genericity*. Because the elements of the unordered set $\mathcal{U}$ only can be compared with $=$, in some situations, it is impossible to distinguish between two elements of $\mathcal{U}$ without mentioning them explicitly in the query. For example, consider the D-world consisting of one relation $R(A, B) = \{(a, b), (a, c)\}$. In this situation, an algebra expression cannot distinguish between b and c without mentioning them explicitly. So, every expression that does not use the constants b and c, either returns both b and c, or neither. In short, if it is impossible to distinguish between two elements in the input, it should not be possible to distinguish between them in the output. This principle is called genericity and is formally defined as follows. We call a query q generic, if, for all input databases D, for all permutations h of $\mathcal{U}$, $q(h(D)) = h(q(D))$. That is, q does not depend on a specific ordering of the elements of $\mathcal{U}$, only on their relation in D.

THEOREM 5.1. *Every generic computable $D{\rightarrow}D$ data mining query is $\mathcal{MA}$-expressible.*

Proof sketch

(1) In the I-world we can simulate increment and decrement of integers, comparison

and looping. It is well-known that these operators are Turing Complete for integer computations; i.e., they are sufficient for expressing every computable function from integers to integers. The simulation is roughly as follows.

Increment: The number n is represented by the following I-world relation $\mathcal{N}r[n]$:

Reg
$\{\texttt{Nr} \leq 0\}$
$\{\texttt{Nr} \leq 1\}$
$\{\texttt{Nr} \leq 2\}$
...
$\{\texttt{Nr} \leq n\}$

Let R be the following D-world relation over schema $\{\texttt{Nr}\}$: $\{(0)\}$. This relation R can easily be constructed using a D-world algebra expression. Let $\mathcal{X}$ be an I-world relation representing a number n. The increment operation on $\mathcal{X}$ is implemented in the algebra as follows: $\mathcal{X} \cup \kappa_{\texttt{Nr}\leq\texttt{val(C)}}\, \pi_{\mathcal{A}}\, \Gamma^{\texttt{COUNT(Reg) as C}}_{\langle\texttt{Nr}\rangle}\, Pop(\mathcal{X}, R)$. That is, first the relation $\mathcal{X}$ is populated with R to form the E-world relation

Reg	Nr
$\{\texttt{Nr} \leq 0\}$	0
$\{\texttt{Nr} \leq 1\}$	0
$\{\texttt{Nr} \leq 2\}$	0
...	...
$\{\texttt{Nr} \leq n\}$	0

Then the number of regions per Nr-value is counted, and this count is projected back into the D-world, resulting in the D-world relation over schema $\{\texttt{Nr}, \texttt{Cnt}\}$: $\{(0, n+1)\}$. Finally, with the regionize operation $\kappa_{\texttt{Nr}\leq\texttt{val(Cnt)}}$, a new region $\{\texttt{Nr} \leq n+1\}$ is constructed, which is added with the union to the I-world relation $\mathcal{X}$. The resulting relation represents the number $n+1$.

Looping: is done with the data mining loop. As illustrated in Appendix A, the data mining loop λ has the same expressive power as extended loops.

Comparison: can be done in the condition of the extended data mining loop. These three operations together form a Turing Complete computation paradigm for integers, assuming that the integers are given as relations $\mathcal{N}r[n]$.

(2) Every D-world relation can be encoded as an integer, on this integer we can do all Turing expressible computations, and the resulting integer can be mapped back to a relation. For the details of this encoding, we refer to the electronic appendix of this paper.

□

COROLLARY 5.2. *The data mining algebra $\mathcal{MA}$ can express any computable D→X query.*

PROOF. In Theorem 5.1, it is shown that every computable D→D-query is $\mathcal{MA}$-expressible. Given the completeness for D→D-queries, for D→I and D→E it suffices to show that there exist encodings of I-world relations in the D-world, and of E-world relations in the D-world, that can be mapped back with the $\mathcal{MA}$-algebra to the encoded relations.

Indeed, let $enc(\mathcal{R})$ denote the encoding of an I-world relation $\mathcal{R}$ in the D-world. Let q be a computable D→I-query. Then, the D→D query $enc \circ q$ (q followed by enc) is computable, and can hence be computed with an $\mathcal{MA}$-expression $expr$. The result of q can then be obtained by mapping the result of $expr$ to the I-world relation it represents.

We illustrate the encoding of an I-world relation with an example. Let $\mathcal{R}$ be an I-world relation over the schema $\{R\}$, $dom(R) = \mathcal{REG}(A, B)$ with A and B attributes over domain $\mathbf{Q}$. Hence, the constraints in the attribute R will be of the form $aA + bB\ \theta\ c$, with a, b, and c rational numbers. We will encode such a constraint as a tuple (a, b, t, c), with t a constant that denotes the comparison operator; e.g. $t = 0$ denotes θ='=', $t = 1$ denotes θ='$\leq$'. The relation D in the D-world encoding $\mathcal{R}$ has schema $\{\{A, B, T, C\}\}$. This encoding can be mapped to $\mathcal{R}$ by an $\mathcal{MA}$-expression.

For the D→E-queries a similar encoding can be used. □

The theorem and its corollary imply that the proposed framework – $\mathcal{MA}$ with κ allowed inside a mining loop – is too powerful. Henceforth, we seek to restrict the class of $\mathcal{MA}$-expressions allowed. We call a mining loop operation $\lambda_{Expr[\mathcal{X}]}(\mathcal{S}eed)$ *static* provided *Expr* does not contain any occurrences of κ and λ. The latter condition disallows nested loops whereas the former one disallows the ability to apply κ to newly constructed data objects (hence the term static). We call an $\mathcal{MA}$-expression $\mathcal{E}$ *static* if all occurrences of λ in it are static. The example in section 3.3 shows that the frequent itemset query can be expressed with static $\mathcal{MA}$-expressions. The following theorem shows that while being sufficiently expressive to capture interesting data mining tasks, static $\mathcal{MA}$-expressions are not computationally complete, making them more suitable for data mining.

Theorem 5.3. *There exist data mining queries expressible in $\mathcal{MA}$ but not in static $\mathcal{MA}$.*

Proof. First we will show a bound on the size of the output of a static $\mathcal{MA}$-expression. Then we will give a computable D→D-query that violates this bound. Since $\mathcal{MA}$ is Turing Complete, this query is expressible in $\mathcal{MA}$, and because the query violates the bound, it cannot be expressed in static $\mathcal{MA}$.

Bound on static queries. Let *Expr* be a static $\mathcal{MA}$-expression containing k loops. Then, there exist polynomials $p_0, \ldots, p_k$ that only depend on *Expr* such that, given input-relations with at most n tuples, the output can contain at most the following number of tuples:

$$p_0(2^{p_1(2^{p_2(2^{\cdots 2^{p_k(n)\ldots)))}})})})$$

That is, there is a constant number of exponentiations. Intuitively, this formula comes from the fact that except for the looping operation in the I-world, all other operations generate at most a polynomial number of tuples in the D-world. Then, the only way to get new basic constraints in the I-world is by κ. Since the number

of different basic constraints in the I-world is limited, only a polynomial number of new basic constraints can be formed. Furthermore, despite the power looping gives, the output of the looping operation is bounded by the number of regions that can be formed with the basic constraints. Because a region is a set of basic constraints, the number of different regions that can be formed with b basic constraints is 2^b. A relation with ℓ attributes in the I-world can thus at most have $2^{\ell b}$ tuples.

We start out with n tuples in the D-world. Without looping we can only create a polynomial number $p_k(n)$ of tuples, and thus, at the moment that the first loop starts, there are at most $p_k(n)$ basic constraints in the I-world. Therefore, the looping creates at most $2^{\ell p_k(n)}$ tuples. This reasoning can now be repeated; the non-looping operators create at most $p_{k-1}(2^{p(n)})$ new basic constraints. The second loop can thus generate $2^{\ell p_{k-1}(2^{p(n)})}$ tuples, and so on.

Query not in static $\mathcal{MA}$. Consider the following query: given a D-world relation D, it creates the following relation over schema $\{A\}$: $\{(i) \mid i = 1 \ldots N\}$, with N the number

$$\left.\begin{matrix} \quad\quad 2 \\ \quad\;\;\vdots \\ \;\;2 \\ 2 \end{matrix}\right\} |D|$$

This function is clearly computable, but cannot be calculated by a static $\mathcal{MA}$-expression, because the number of exponentiations is unbounded. □

The key questions at this point are how expressive static $\mathcal{MA}$-expressions are, and where the expressiveness comes from. To answer these questions, we give the following definition. We call an $\mathcal{MA}$-expression in which there is no occurrence of λ, a *loop-free $\mathcal{MA}$-expression*.

Notice that loop-free $\mathcal{MA}$-expressions are necessarily static. The following theorem says that when we restrict attention to $D{\rightarrow}D$, the class of such data mining queries that can be expressed using $\mathcal{MA}$ are not "new", in that they can already be expressed in $\mathcal{DA}$, i.e., such queries can be expressed without leaving the D-world! Thus, despite the ability to apply κ to construct constraints, compare sets of constraints, group and ungroup sets of constraints, loop-free $\mathcal{MA}$-expressions cannot do anything beyond what nested relational algebra with aggregation and arithmetic can do. The fact remains, of course, that $\mathcal{DA}$ cannot express $D{\rightarrow}X$ mining queries where X is I or E, whereas loop-free $\mathcal{MA}$ can express a whole class of such queries. Call an $\mathcal{MA}$-expression $D{\rightarrow}X$ whenever it expresses a $D{\rightarrow}X$ mining query. As we shall see later, static $\mathcal{MA}$-expressions (involving mining loop) *can* indeed express new $D{\rightarrow}D$ mining queries that *cannot* be expressed in $\mathcal{DA}$.

THEOREM 5.4 LOOP-FREE. *For every loop-free $\mathcal{MA}$-expression that is $D{\rightarrow}D$, there exists an equivalent expression in $\mathcal{DA}$.*

Proof sketch The proof is based on a simulation of the operations in the I- and E-world within the D-world. As is already illustrated in the proof of Corollary 5.2, I-world relations can be represented in the D-world. Instead of interpreting them directly, as κ does, the constraints are evaluated only when they are needed to. Thus, for example, when a union is applied to two regions in the I-world, the corre-

sponding sets of tuples are unioned in the D-world. For most operations it is easy to simulate them. One case, however, requires special attention: **Selection based on containment of regions over rational attributes.** A non-trivial I-world operation to simulate in the D-world is selection based on region containment. Suppose, e.g., that we have a relation $D(\{A, B, T, C\})$ encoding an I-world relation $\mathcal{R}(R)$ with constraints of the form $aA + bB\ \theta\ c$. A tuple $(\{(a_1, b_1, t_1, c_1), \ldots (a_k, b_k, t_k, c_k)\})$ of D encodes the tuple $(\{a_1A + b_1B\ \theta_1\ c_1, \ldots, a_kA + b_kB\ \theta_k\ c_k\})$ of $\mathcal{R}$, with θ_i='=' if $t_i = 0$, θ_i='$\leq$' if $t_i = 1$. In this setting, simulating a selection based on containment of regions, comes down to solving a system of linear inequalities whose coefficients are given as attribute values in the D-world. In Appendix C it is shown that solving such a system of linear inequalities can indeed be expressed with a $\mathcal{DA}$-expression. □

Without the mining loop, all $D{\rightarrow}D$ data mining queries expressible in the mining algebra are already expressible in $\mathcal{DA}$. Indeed, the mining loop brings substantial expressive power to the mining algebra. Having seen the power that mining loop brings to a data mining algebra, we can ask whether there is an alternative to mining loop, that is somehow less powerful. Indeed, there have been several attempts (e.g., [Boulicaut et al. 1999; Netz et al. 2001]) at incorporating specific data mining tasks into a database system, where the whole task (e.g., frequent itemsets, decision trees, etc.) is captured as a "black box" operator. We examine this alternative in the next section, specifically for frequent itemsets.

5.1 Frequent Itemsets as a Black Box Operator

In the following, we define a new class of $\mathcal{MA}$-expressions, which uses this operator, referred to as the FI operator. The FI-operator takes as input a D-world relation $D(\{\texttt{Item}\})$ and is parameterized by a support threshold ts. The output of the FI-operator is then defined as the I-world relation $\mathcal{R}$ consisting of the tuples $(\{i_1 \in \{\texttt{Item}\}, \ldots, i_k \in \{\texttt{Item}\}\})$, $1 \leq k \leq n$, such that the support of the itemset $\{i_1, \ldots, i_k\}$ is at least ts. By $\mathcal{MA}^{FI}$, we mean the language $\mathcal{MA}$, with κ and λ replaced by the FI-operator. $\mathcal{MA}^{FI}$-expressions have the obvious meaning.

A natural question is whether $\mathcal{MA}^{FI}$-expressions have an expressive power beyond $\mathcal{DA}$. Superficially, this question may seem trivial since FI essentially has an ability similar to the powerset operator. However, the collection of frequent sets lives in the I-world, and our focus is on D→D queries! The following example shows that finding frequent itemsets that are maximal w.r.t. the inclusion ordering, can be expressed in $\mathcal{DA}$.

Example 10. Let $D(\{\texttt{Item}\})$ be a transaction database, and I be a maximal frequent itemset for threshold ts. Since I is frequent, I must be present in at least ts transactions, say $\{t_1, \ldots, t_{ts}\}$. As well, the intersection of *any* ts transactions is a frequent itemset. Because I is maximal, I equals every intersection of ts transactions in which it is contained. Therefore, every maximal frequent itemset is the intersection of exactly ts transactions. We can now find all maximal ts-frequent itemsets with the query that takes all ts-intersections, and selects the maximal sets among those. This query is expressible in relational algebra, since ts can be considered as a constant here.

The following theorem establishes nevertheless that $D{\rightarrow}D$ $\mathcal{MA}^{FI}$-expressions

are more expressive than $\mathcal{DA}$.

THEOREM 5.5 FI-EXPRESSIVENESS. *There exists a D→D $\mathcal{MA}^{FI}$-expression for which there does not exist an equivalent expression in $\mathcal{DA}$.*

Proof sketch The transitive closure query TC can be expressed in $\mathcal{MA}^{FI}$. Let $D(A, B)$ be a relation in the D-world, and A and B have domain $\mathcal{U}$. The result of TC on D is the transitive closure of D; i.e., $TC(D)$ contains (a, b) if and only if there exist tuples (a, x_1), (x_1, x_2), $\ldots$, (x_k, b) in D. The expression for TC in $\mathcal{MA}^{FI}$ works as follows: first, nesting is used to create the items for the FI-operator. Each item will represent one pair (a, b) in D. The transaction database will consist of one transaction that contains all edges. Then, FI is applied with threshold equal to 1. Hence, in fact, FI is used to create all subsets of the set of all pairs. From these sets of pairs, those are selected that represent chains $(a, x_1), (x_1, x_2), \ldots, (x_k, b)$ in D. Based on these chains, the pairs (a, b) of $TC(D)$ can be found. For details we refer to Appendix D.

On the other hand, it is well-known that this query cannot be expressed in nested relational algebra with arithmetic and aggregation, as TC is not local, and the nested relational algebra with arithmetic and aggregation can only express local queries [Libkin and Wong 1997]. Also the ability to do nesting does not help, as the flat-flat theorem [Paradaens and Van Gucht 1998] states that every nested relational algebra expression that maps a flat relation to a flat relation can be expressed without nesting. The proof in of Paradaens and Van Gucht [1998] is actually for the nested relational algebra without arithmetic and aggregation, but the simplified proof of the flat-flat theorem of Van den Bussche [2001] can easily be adapted to include arithmetic and aggregation. □

Transitive closure is not a very natural query in the data mining context. One might wonder how the situation is for more natural data mining queries. The answer is as follows. There are a lot of natural queries expressible in $\mathcal{MA}^{FI}$ that are NP-complete. For example: given two transaction databases $\mathbf{T_1}, \mathbf{T_2}$, find all transactions in $\mathbf{T_2}$ that do contain an itemset I such that the support of I in $\mathbf{T_1}$ is strictly smaller than its support in $\mathbf{T}_2$. The *comparative containment* problem, which is known to be NP-complete [Garey and Johnson 1979], can be reduced to this query. If DLOGSPACE $\neq$ NP, the latter problem cannot be expressed by an expression in $\mathcal{DA}$. This is because we can show via a similar reasoning as in [Consens and Mendelzon 1993], that our D-world algebra can be evaluated in deterministic logspace.

5.2 Expressiveness of the Mining Loop Operator

So far, our analysis addresses most of the expressiveness issues concerning $\mathcal{MA}^{FI}$-expressions. The only exception is the question whether there is a difference in expressive power between static $\mathcal{MA}$-expressions and $\mathcal{MA}^{FI}$-expressions. The next theorem shows that using κ and mining loop operators is strictly more expressive than using only the black box operator FI.

THEOREM 5.6. *There exists a static $\mathcal{MA}$-expression for which there is no equivalent $\mathcal{MA}^{FI}$-expression.*

Proof sketch The intuition is as follows. We can use $\mathcal{MA}$-expressions and $\mathcal{MA}^{FI}$ to decide languages of 0 and 1's as follows: the input to the expression is given as a binary relation $D(\texttt{Nr},\texttt{Bit})$ in the D-world. We require that $\texttt{Nr}$ is a key, and the input string then is the string of bits, according to the order in $\texttt{Nr}$. The decision whether or not this string is in the language is then reflected by non-emptiness of the output. The non-emptiness of an $\mathcal{MA}^{FI}$-expression can be decided in PSPACE via a similar pipelining argument as in [Abiteboul and Hillebrand 1995]. On the other hand we can simulate every *exponential* space Turing Machine in the I-world with $\mathcal{MA}$. We can carry over the order on $\texttt{Nr}$ to an order on the subsets of $\{\texttt{Nr} \leq nr \mid nr \in \pi_{\texttt{Nr}} R\}$. This order allows us to represent the work tape now as a unary relation in the I-world. Bit i is 1 if and only if the ith set is in the relation. Therefore, static $\mathcal{MA}$ can be used to decide exponential space languages. It is well-known that there exist languages decidable in exponential space, but not in polynomial space. For more details we refer to Appendix B. □

Together with Theorem 5.4, the above theorem indicates that mining loops lead to additional expressive power. We can now ask the following questions for static $\mathcal{MA}$-expressions: does each application of the mining loop operator leads to additional expressiveness? Does the ability to combine and compose loop operators increase the expressiveness? The following theorem answers this question positively.

THEOREM 5.7. *For each k, there exists a static $\mathcal{MA}$-expression with k loops such that it is not equivalent to any static $\mathcal{MA}$-expression with $k-1$ loops.*

PROOF. Recall from the proof of Theorem 5.3 that for any static $\mathcal{MA}$-expression containing k loops, there exist polynomials $p_0, \ldots, p_k$, such that, given input-relations with at most n tuples, the output can contain at most the following number of tuples:

$$p_0(2^{p_1(2^{p_2(2^{\cdot^{\cdot^{\cdot 2^{p_k(n)\ldots)))}}}})})})$$

We will now show that for every k, there exists a query expressible with k loops, that, given the relation $D(A) = \{(1),(2),(3),\ldots,(n)\}$ in the D-world, outputs the relation $\{(i) \mid i = 1\ldots N\}$, with N the number

$$\left.2^{2^{2^{\cdot^{\cdot^{\cdot 2^{n}}}}}}\right\} k$$

As this query cannot be expressed with $k-1$ loops (it violates the bound), such an expression with k loops shows that there exists a query expressible with k loops but not with $k-1$ loops.

First, the regionize operator is used to create n basic constraints in the I-world with $\kappa_{A \geq val(A)} D$. Then, with a static loop, it is possible to construct all regions

based on the basic constraints in a relation $\mathcal{R}(R)$. That is, suppose that $\mathcal{R}$ is the following I-world relation:[2]

R
$\{\}$
$\{A \geq 1\}$
...
$\{A \geq n\}$

Then there exists an expression using one static loop that outputs the following "powerset" relation $\mathcal{P}$:

R
$\{\}$
$\{A \geq 1\}$
$\{A \geq 1, A \geq 2\}$
...
$\{A \geq 1, A \geq 2, \ldots, A \geq n\}$

Notice that the constraints are totally ordered; we can say that $A \geq a_1$ comes before $A \geq a_2$ if $a_1 \leq a_2$. This order can be carried over to the sets in $\mathcal{P}$. We can for example order the regions lexicographically. Based on this order, a relation $\mathcal{O}$ over attributes $\mathtt{R}_1, \mathtt{R}_2$ is constructed, of pairs (r_1, r_2), with $r_1, r_2 \in \mathcal{P}$, and $r_1 \leq r_2$. This ordering can be constructed in the same static loop as the powerset. Starting from the set $\mathcal{P}$ given above, the relation $\mathcal{O}$ would look like:

$\mathtt{R}_1$	$\mathtt{R}_2$
$\{\}$	$\{\}$
$\{\}$	$\{A \geq 1\}$
$\{\}$	$\{A \geq 1, A \geq 2\}$
...	...
$\{\}$	$\{A \geq n\}$
$\{A \geq 1\}$	$\{A \geq 1\}$
$\{A \geq 1\}$	$\{A \geq 1, A \geq 2\}$
...	...
$\{A \geq 1\}$	$\{A \geq n\}$
...	...
$\{A \geq n\}$	$\{A \geq n\}$

Subsequently, the relation $\mathcal{O}$ is populated with the relation over attribute A that consists of only one tuple, (1). This results in the following E-world relation E:

$$\{(r_1, r_2, 1) \mid (r_1, r_2) \in \mathcal{O}\} \ .$$

Then, the following aggregation is applied: $\Gamma^{\mathtt{COUNT}(*)}_{\langle \mathtt{R}_1 \rangle} E$. The result is a relation with all tuples (r, n), where r is a region of $\mathcal{P}$, and n is the number of successors of r, given the order in $\mathcal{O}$. Since the order is total, for every number i from 1 to 2^n, there is a tuple (r, i) in the answer. Finally, projecting this relation to the D-world

[2]Where $\{\}$ represents the TRUE region.

results in: $\{(i) \mid i = 1 \ldots 2^n\}$. For this construction, only one static loop is needed. To get the relation $\{(i) \mid i = 1 \ldots N\}$, with N the number

$$\left. 2^{2^{\cdots 2^{n}}} \right\} k \ ,$$

it suffices to apply the same expression k times; the first time, the relation with numbers form 1 to 2^n is generated, the second time the numbers 1 to 2^{2^n} is generated, and so on. Hence, there exists an expression with k loops that computes the desired query. □

6. RELATED WORK

Meo et al. [1996] introduce a SQL-like operator for mining association rules. Han et al. [1996] propose a data mining query language extending SQL, which allows various data mining tasks to be specified. In both studies, however, the mining results cannot be explicitly manipulated. Netz et al. [2001] and Chaudhuri et al. [2002] explore the integration of data mining and relational databases (e.g., adding a classification capability to SQL server via the OLE-DB interface). A classification model M can be created and later populated with different data sets to give predictions, via the so-called prediction joins. However, other than browsing and prediction, their framework does not provide any other operation for manipulating M. Furthermore, there is no notion of composing mining operations in their framework.

The inductive database framework presented by Boulicaut et al. [1999] allows to query the theory of the database. As a concrete example, a possible instantiation of this principle in the context of frequent set mining is that there exists a virtual table with frequent itemsets that can be queried [Boulicaut et al. 1999]. Similar in nature is MSQL, developed by Imielinski and Virmani [1999]. However, in both studies, there is no notion of composing mining operations. Furthermore, the mined itemsets merely appear as syntactic objects, i.e., they are not interpreted, as opposed to the spatial constraint objects in our framework. We are not aware of any expressibility results of their frameworks. Geist and Sattler [2002] propose a uniform framework for data mining based on constraint databases. However, they do not provide generic operators capable of capturing various mining tasks, nor address expressive power issues.

Tsur et al. [1998] explore how similar optimizations as in the apriori algorithm can be extended and applied in the larger context of database queries. This study results in the query flock as a generate-and-test model for data mining problems. However, our focus is broader, as we try to integrate data mining queries into database systems, where in [Tsur et al. 1998], the main focus is on optimizing database queries using techniques developed in frequent itemset mining.

Studies on constraint databases such as [Paradaens et al. 1994] have clear relevance to our framework. However, the relevance is solely for manipulating regions *once* they are created from data by means of mining. A main contribution of our paper is a mining algebra and constraint database algebras do not fulfill this need.

Imielinski and Mannila [1996] set out a research agenda for the database and data mining research community challenging it to come up with a framework whereby data mining results can be treated as first-class objects and be subjected to further manipulation, and different mining tasks can be composed. The data mining model and algebra proposed in this paper takes a significant step in addressing the above challenge.

Very related to the spirit of our work, are $\mathcal{LDL}^{++}$ [Wang and Zaniolo 2000] and ATLaS [Zaniolo 2005; Law et al. 2004; Wang and Zaniolo 2003]. $\mathcal{LDL}^{++}$ and ATLaS are extensions of respectively $\mathcal{LDL}$ and SQL that add the ability of defining new *user defined aggregates* (UDAs), making them suitable for data mining. Especially ATLaS is very interesting with respect to our proposal, as it is also based on the principles of relational databases and query languages. The UDAs are defined by giving three expressions: *initialize*, *iterate*, and *terminate*. These expressions are given in SQL, and can use some temporary tables to store results. The SQL expressions in the definition of the aggregation can again, recursively, make use of UDAs. Notice that, even though this is clearly a kind of looping construction, there are important differences with our data mining loop; UDAs iterate over streams of tuples, whereas our looping construction is a fixpoint loop, iteratively refining a set of constraints.

Another difference between our model and ATLaS is that in ATLaS there is no notion of constraints, patterns or regions. Indeed, ATLaS is a query language enabling data mining operations, on top of relational databases. There is, however, no notion of an architecture such as the three worlds we propose. Hence, in ATLaS, the results of mining have to be encoded into the relational model, and subsequent queries of found patterns have to deal with decoding and encoding the found patterns. Also, the ATLaS query language seems to be more procedurally driven, making it less attractive for query optimization. For the expressiveness results, in [Law et al. 2004], it was shown that ATLaS can simulate Turing machines, and hence has similar computational and expressive power as $\mathcal{MA}$. Furthermore, it was shown how frequent itemset mining and prediction methods can be declared in ATLaS. There are no sub-fragments defined of ATLaS that are not Turing complete. Nevertheless, the treatment of data mining operations as manipulations of streams is very interesting, because it inherently implies a pipelined evaluation, avoiding unnecessarily materializing temporary results.

Lastly, there is a huge body of literature on the expressive power of database query languages (e.g., see [Abiteboul et al. 1995] for a survey). However, both the proposal of a comprehensive data mining algebra and a study of expressive power in this context, to the best of our knowledge, are novel.

7. DISCUSSION

In this section we discuss the proposed framework, and try to answer, or at least provide some insight into, the following questions:

—Of the different proposed algebras, what's the most appropriate foundation for data mining, and why?

—What are the limitations of the model we propose? Are there data mining operations that cannot be expressed in our model? For what kinds of data mining

operations is the model best suited?

—What are the opportunities for optimization in the model?

7.1 Appropriate Foundation for Data Mining

Given the properties on the expressive power proven in the paper, we believe that of the proposed algebras, static $\mathcal{MA}$ comes closest to the desired foundation for data mining. First of all, static $\mathcal{MA}$ not being Turing complete should not be considered a problem. The main reason why a Turing complete language is less interesting is optimization; optimizing a query language becomes typically more difficult as the expressive power increases. Therefore, the most interesting situation is to have a query language that can express exactly the intended class of queries, and nothing more. Clearly, a Turing complete language is not desirable in this perspective, as it represents the complete opposite. "Too powerful" is also a problem as one of the goals of our research is to learn about the true nature of data mining. Ideally, the algebraic operations reflect common building blocks of data mining algorithms. A Turing complete language won't give us a lot of insight in this matter.

On the other hand, a non-Turing complete language cannot express all queries. We hypothesize, however, that static MA still includes the practical relevant data mining queries. Unfortunately, it is impossible to justify this thesis, as there does not exist a generally accepted notion of what is a data mining query. We can, however, give some more intuition on why we believe static $\mathcal{MA}$ does not lose too many interesting queries. The true difference between static $\mathcal{MA}$ and the Turing complete $\mathcal{MA}$ is that in static $\mathcal{MA}$, no new constraints can be created *inside* a mining loop. As the mining loop iteratively refines sets of constraints, this restriction, in some sense, corresponds to first fixing the search space (i.e., constructing all needed constraints), and then locating desired regions in this space. Hence, intuitively, static $\mathcal{MA}$ is able to express exactly those queries that ask for all elements in a predefined search-space. This condition seems to be fulfilled for many data mining algorithms; often, mining consists of locating one, some, or all, suitable patterns of a certain fixed pattern type. E.g., in decision tree construction, the search space is limited to the set of decision trees, where the splitting conditions have a predefined format. Similarly, in itemset mining, the itemsets are known in advance. In K-medoids clustering, the potential clusterings are defined by the possible mid-points of the clusters, and so on.

We don't see this paper as the last word on what is mining or even what is the most appropriate algebra for mining. Instead, we see this as the first work that points the right direction and the right kind of framework in which such fundamental questions can be rigorously studied. In this perspective, it is also worth remarking that it is *not* necessary that every (efficient) algorithm someone comes up with for a given operation (e.g., frequent set computation) be expressible in an algebra. This is not the main purpose of an algebra. Instead, what is important is that the algebra covers what are considered to be the core set of operations. It should be expressive and at the same time flexible enough to allow various alternate efficient realizations.

7.2 Limitations of the Algebra

The most important limitation of the algebra is in the I-world, where we restrict the regions to be expressible as sets of linear inequalities. This limitation means that the results of some data mining operations might not be expressible, as they require more complex mathematical objects. Straightforward examples are, e.g., non-linear regression methods, support vector machines, and clustering methods resulting in non-linear regions, etc. Notice, however, that for these applications the *nature* of the results remains the same. For example, for clustering, the goal is to generate groups of similar objects that can be described succinctly, as to summarize the database. These succinct descriptions of clusters, in many cases, can be constraints describing the region of the cluster. E.g., for the K-means clustering algorithm, a cluster with center c is the set of points p for which the following constraints hold: for all other centers of clusters c', the distance between p and c is smaller than or equal to the distance between p and c'. Unfortunately, unless the L_1-norm is being used, this constraint is non-linear.

Therefore, the description of the output of some data mining tasks requires more sophisticated constraints than the ones used in the paper. Nevertheless, in most cases, the theoretical results presented in the paper still hold when more sophisticated constraints are used, because the interpretation of the regions does not change. For example, when the constraints are extended to include polynomial inequalities instead of linear inequalities, only the proof of the equality of loop-free $\mathcal{MA}$ and $\mathcal{DA}$ requires some work. Nevertheless, similar techniques can be used to show that elimination methods as used in the linear case still obtain. From a description point of view, such extensions to ease the expression of data mining tasks requiring sophisticated constraints is an interesting direction for further work. In this paper, however, for reasons of simplicity, we have opted to restrict to linear constraints.

7.3 Optimization Opportunities

In the algebraic framework we propose, we see many opportunities for optimization. First of all, a large part of the data mining algebra is the (nested) relational algebra. Obviously, we can use currently existing optimization techniques for the nested relational algebra for this sub-fragment. There are, however, also optimizations possible that are specially aimed at the mining operators:

(1) For the data mining loop operation, loop fusion techniques [Kennedy and McKinley 1994] might be appropriate, especially when, in practical systems, macros and other “syntactic sugar” is used. For example, suppose that in the context of frequent itemset mining, a user is interested in all itemsets frequent in one relation and infrequent in another one. This user might write this query in the following, declarative, way. First, two mining loops are written, similar to the frequent itemset example given in Section 4.3, one to find all itemsets frequent in the first relation, and one to find all frequent itemsets in the second relation. Then, the final answer is found by taking the difference of the two sets. In this construction, however, in practice it will be far more interesting to fuse the two mining loops into one loop, especially when the number of frequent sets in the first relation is rather small compared to the second relation. Such

situations, where two or more similar mining queries are used, might be quite common, especially in practical systems where often a graphical interface that is between the user and the database system constructs the queries.

(2) For the I-world, spatial indexing techniques such as R-trees [Guttman 1984], can help answering topological queries on regions, such as, e.g., *give all regions that are maximal*. Even though from an expressivity viewpoint such a physical operator is not necessary, some topological queries might turn out to be so prevalent, that a specialized, physically optimized operator is justified; e.g., queries asking for the most specific or most general patterns in a relation might turn out to be very common in data mining operations.

(3) For the populate operation, it is important to be able to quickly locate all regions containing a certain point, and to enumerate all points that fall in a given region. Another consideration is that the result of a populate operation can be very large. Because the constraints that need to be populated by the data are in fact composed of a limited set of basic constraints, techniques from frequent itemset mining will be appropriate in this context. There is a straightforward mapping; if basic constraints are considered as items, the regions can be considered as itemsets. Thus, the storage and querying of regions can be very similar as for itemsets.
Another example of the usefulness of frequent itemset mining techniques is the following: it is not always possible to fully materialize the result of a populate operation, because, in fact, a populate operation implies a join between a relation from the D-world, and a relation in the I-world containing regions. If the goal is, e.g., to count the number of tuples per region, optimizations might include techniques such as condensed representations for frequent itemsets, that only store non-redundant frequency information from which other frequencies can be derived [Mannila and Toivonen 1996; Calders et al. 2006]. Also efficient counting mechanisms like, e.g., FP-trees [Han et al. 2000], can be useful in this respect.

(4) Another technique that might be applicable is that of performing aggregation on streaming data. Using this we can avoid having to first fully materialize the results of a populate operation and can instead aggregate the data being populated, on the fly. Notice that this optimization is very much related to the evaluation of queries in ATLaS [Law et al. 2004].

8. CONCLUSION

In this paper, we present an algebraic foundation for understanding the integration of data mining and relational systems. We propose a data mining algebra that includes two generic operators regionize and mining loop. We analyze the expressive power of our mining algebra, for different combinations of operators.

More concretely, we show the following results. For the relation between regionize and looping, it turns out that on the one hand, when regionize is permitted inside the mining loop, the language is too powerful for data mining, i.e., computationally complete. On the other hand, when a mining loop is not allowed, the resulting language $\mathcal{DA}$ becomes too restricted, i.e., it has the same expressive power as the nested relational algebra with aggregation and arithmetic. A more interesting sit-

uation is when both regionize and loop are allowed, but regionize is only allowed to appear outside of a mining loop. The sub-language of $\mathcal{MA}$ satisfying this constraint is called *static* $\mathcal{MA}$. For the expressiveness, it is shown that static $\mathcal{MA}$ is less expressive than $\mathcal{MA}$, but is still powerful enough to express mining queries such as frequent itemset mining and decision trees. Besides, the algebra provides a natural mechanism for composing mining tasks.

We also compare our approach with alternative frameworks. The most straightforward alternative framework is to add mining operations such as frequent itemset computation as a "black box" operator. The resulting algebraic framework is denoted $\mathcal{MA}^{FI}$. However, we show that this alternative is strictly less expressive than static $\mathcal{MA}$. Hence, we get a hierarchy of languages which can be summarized as follows:

$$\mathcal{DA} \equiv \text{loop-free } \mathcal{MA} \sqsubset \mathcal{MA}^{FI} \sqsubset \text{static } \mathcal{MA} \sqsubset \mathcal{MA}$$

In particular, $\mathcal{MA}$ is too powerful, in that it is Turing complete. At the other extreme, loop-free $\mathcal{MA}$ is too weak: it is no more expressive than $\mathcal{DA}$. Static $\mathcal{MA}$ is less expressive than $\mathcal{MA}$, although it is strictly more expressive than $\mathcal{MA}^{FI}$. The latter cannot express decision trees, e.g., while static $\mathcal{MA}$ can.

In future work, we will compare static $\mathcal{MA}$ with other black box mining operator extensions, such as a decision tree operator. We will evaluate whether static $\mathcal{MA}$ is more expressive than these alternatives. If true, this paves the way for understanding the true nature of integration, as well as for developing unified optimization strategies. Another important direction is to evaluate expressiveness based on $D{\rightarrow}X$ mining queries, where X is I or E. Finally, it would be interesting to pursue the optimization opportunities in the model proposed here.

APPENDIX

A. EXTENDED LOOP

In this appendix we show that the simple loop construction $\lambda_{Expr}(\mathcal{S}eed)$ is powerful enough to express more complicated types of loops. Formally, we will rewrite the following type of loop as a simple loop.

$\mathcal{S}_1 := Seed_1;$
$\quad\ldots$
$\mathcal{S}_n := Seed_n;$
While($C_1 \theta C_2$) **loop**
$\quad\mathcal{R}_1 := Expr_1;$
$\qquad\ldots$
$\quad\mathcal{R}_m := Expr_m;$
End loop
Return *Ret*;

In this loop construction, $Seed_1, \ldots, Seed_n$ are I-world relations or expressions, and $Expr_1, \ldots, Expr_m$, and Ret are algebra expressions that return an I-world relation. $\mathcal{S}_1, \ldots, \mathcal{S}_n$, and $\mathcal{R}_1, \ldots, \mathcal{R}_m$ are dedicated symbols that denote variables. With every variable a fixed schema is associated. C_1 and C_2 are algebra expressions that return relations over the same schema. θ is one of $=$, $\neq$, $\subseteq$. The variables

$\mathcal{R}_1, \ldots, \mathcal{R}_m$, and $\mathcal{S}_1, \ldots \mathcal{S}_n$ can be used in the expressions as if they were regular I-world relations.

The rewriting of the extended loop construction into a simple loop will be done in four steps:

(1) The variables $\mathcal{S}_i$ and $\mathcal{R}_j$ will be replaced by one single variable $\mathcal{X}$;
(2) The group of n assignments before the loop and the group of m assignments inside the loop are both replaced by only one assignment;
(3) The looping condition $C_1 \theta C_2$ is replaced by the simple construction $\Delta\mathcal{X}$;
(4) The expression *Ret* in the return clause is replaced with $\mathcal{X}$.

After these four steps have been completed, the resulting loop is an equivalent, simple loop.

Removing multiple variables. Let $S_i^1, \ldots, S_i^{k_i}$ be the attributes of the relation $\mathcal{S}_i$, for $i = 1 \ldots n$, and let $R_j^1, \ldots, R_j^{l_j}$ be the attributes of the relation $\mathcal{R}_j$, for $j = 1 \ldots m$. Let a be the maximal arity of the $\mathcal{S}_i$'s and $\mathcal{R}_j$'s. That is, $a = \max(\{k_i \mid i = 1 \ldots n\} \cup \{l_j \mid j = 1 \ldots m\})$. We will replace all variables $\mathcal{S}_i$ and $\mathcal{R}_j$ by one variable $\mathcal{X}$ of arity $a + 1$, over the schema $\{ID, X_1 \ldots, X_a\}$. Every tuple of the $\mathcal{S}_i$'s and $\mathcal{R}_j$'s will be represented as a tuple of $\mathcal{X}$. The variables $X_1, \ldots, X_a$ will be used to store the tuple, and the attribute *ID* is used to link the tuples in $\mathcal{X}$ to the correct variable among the $\mathcal{S}_i$'s and $\mathcal{R}_j$'s. For this purpose, the constants $\{Var = i\}$ will be used to denote that the tuple corresponds to the i-th variable in the list $\mathcal{S}_1, \ldots, \mathcal{S}_n, \mathcal{R}_1, \ldots, \mathcal{R}_m$. These constants $\{Var = i\}$ can easily be constructed with our data mining algebra, with the expression

$$V_i =_{def} \kappa_{Var=i \texttt{ as } ID} D ,$$

with D an arbitrary D-world relation. Notice that not all variables have the same arity a. Hence, we need a special constant to pad the tuples in $\mathcal{X}$ representing a tuple of arity less than a. Let $\{X = 0\}$ be this special value, and denote the expression $\kappa_{X=0 \texttt{ as } X_i} D$ generating the relation over schema $\{X_i\}$ with only one tuple ($\{X = 0\}$) by N_i (N of "Null").

EXAMPLE 11. *Suppose that there are two variables: $\mathcal{R}$ with schema $\{\mathtt{R_1}, \mathtt{R_2}\}$ and $\mathcal{S}$ with schema $\{\mathtt{S_1}\}$. Then, a is 2, and the variable $\mathcal{X}$ will have schema $\{\mathtt{ID}, \mathtt{X_1}, \mathtt{X_2}\}$. Suppose that the variable $\mathcal{R}$ and $\mathcal{S}$ are instantiated as follows:*

$\mathcal{R} =$

$\mathtt{R_1}$	$\mathtt{R_2}$
$\{\mathtt{A} = 1\}$	$\{\}$
$\{\mathtt{A} = 1\}$	$\{\mathtt{B} = 1\}$

$\mathcal{S} =$

$\mathtt{S_1}$
$\{\mathtt{A} = 1, \mathtt{B} = 1\}$

This situation corresponds to the following instantiation for the variable $\mathcal{X}$:

ID	$\mathtt{X_1}$	$\mathtt{X_2}$
$\{\mathtt{Var} = 1\}$	$\{\mathtt{A} = 1\}$	$\{\}$
$\{\mathtt{Var} = 1\}$	$\{\mathtt{A} = 1\}$	$\{\mathtt{B} = 1\}$
$\{\mathtt{Var} = 2\}$	$\{\mathtt{A} = 1, \mathtt{B} = 1\}$	$\{\mathtt{X} = 0\}$

The tuples in $\mathcal{X}$ representing the tuples of $\mathcal{S}_i$ are now constructed by the following expression $Encode_i(\mathcal{R}_i)$: $V_i \times \rho_{S_i^1 \to X_1, \ldots, S_i^{k_i} \to X_{k_i}} \mathcal{S}_i \times N_{k_i+1} \times \ldots \times N_a$. A similar

construction is used for the $\mathcal{R}_j$'s. For the other direction, that is, from $\mathcal{X}$ to $\mathcal{S}_i$, the following expression $Extract_i(\mathcal{S}_i)$ can be used:

$$\rho_{X_1 \rightarrow S_i^1, \ldots, X_{k_i} \rightarrow S_i^{k_i}} \pi_{X_1, \ldots, X_{k_i}} \sigma_{ID=\{Var=i\}} \mathcal{X}$$

Again, a similar construction applies for the $\mathcal{R}_j$'s.

With the expressions $Encode(\cdot)$ and $Extract(\cdot)$, we can replace all variables $\mathcal{S}_i$, $\mathcal{R}_j$ by the variable $\mathcal{X}$. Indeed, on the one hand, every occurrence of $\mathcal{S}_i$'s and $\mathcal{R}_j$'s can be replaced with *Extract*-expressions that are only based on $\mathcal{X}$. On the other hand, assignment of the form $\mathcal{S}_i := Expr$ can be replaced by

$$\mathcal{X} := \sigma_{ID \neq \{Var=i\}} \mathcal{X} \cup Encode_i(Expr) \ .$$

Removing Multiple Assignments After the removal of multiple variables, the rewritten loop has the following form:

$\mathcal{X} := Seed_1$;
  ...
$\mathcal{X} := Seed_n$;
While($C_1 \theta C_2$) **loop**
  $\mathcal{X} := Expr_1$;
    ...
  $\mathcal{X} := Expr_m$;
End loop
Return *Ret*;

It is easy to see that both the block of n assignments before the loop and the block of m assignments inside the loop can be rewritten into a single assignment statement. We illustrate the construction for the n assignments before the loop. Let $Seed_j[Expr/\mathcal{X}]$ denote the expression $Seed_j$ in which every occurrence of $\mathcal{X}$ is substituted by the expression $Expr$. We recursively define $SExpr_i$ as follows: $SExpr_1 = Seed_1$, and for all $i = 2 \ldots n$, $SExpr_i$ equals $Seed_i[SExpr_{i-1}/\mathcal{X}]$. Thus, we get $SExpr_2$ by filling in the expression $Seed_1$ in $Seed_2$. Hence, the assignment $\mathcal{X} := SExpr_2$ will result in the same instantiations for $\mathcal{X}$ as the sequence of assignments $\mathcal{X} := Seed_1; \mathcal{X} := Seed_2$. By induction, we can show that the sequence of the first i assignments is equivalent to $\mathcal{X} := SExpr_i$. Therefore, the n assignments can be replaced by the single assignment $\mathcal{X} := SExpr_n$.

For the assignments within the loop, a similar procedure can be applied to replace them by one expression. Hence, after this rewriting step, the loop looks like:

$\mathcal{X} := Seed$;
While($C_1 \theta C_2$) **loop**
  $\mathcal{X} := Expr$;
End loop
Return *Ret*;

Replacing $C_1 \theta C_2$ with $\Delta \mathcal{X}$ For replacing the looping condition, we need a special operator: the "if-then-else". This operator has the following form:

If $(C_1 \theta C_2)$ **Then** $Expr_1$ **Else** $Expr_2$ **Endif**

The semantic of this construction is as follows: if the condition $(C_1 \theta C_2)$ evaluates to true, the result of the complete expression is the result of $Expr_1$, else the result is that of $Expr_2$. We now show that this operator can be expressed in the algebra. First of all: we rewrite $C_1 \subseteq C_2$ as $C_1 \setminus C_2 = \{\}$; $C_1 = C_2$ as $(C_1 \setminus C_2) \cup (C_2 \setminus C_1) = \{\}$, and $C_1 \neq C_2$ is first rewritten as $C_1 = C_2$ and switching the "then" and "else" expressions of the if, and subsequently the equality is rewritten. In this way, we can always get a condition of the form $C = \{\}$. Let $\{R_1, \ldots, R_k\}$ be the schema of the relations returned by $Expr_1$ and $Expr_2$. We rewrite the expression **if** $(C = \{\})$ **then** $Expr_1$ **else** $Expr_2$ as $(Expr_1 \setminus (\pi_{R_1,\ldots,R_k}(C \times Expr_1))) \cup (\pi_{R_1,\ldots,R_k}(C \times Expr_2))$. The equivalence of these two expressions is due to the fact that $\pi_{R_1,\ldots,R_k} C \times Expr_1$ is always $Expr_1$, except in the case that C evaluates to the empty relation, then this expression is empty. Therefore, $Expr_1 \setminus (\pi_{R_1,\ldots,R_k} C \times Expr_1)$ is empty if C is not, and is $Expr_1$ if C is empty. Similarly, $\pi_{R_1,\ldots,R_k} C \times Expr_2$ is empty if C is, and is $Expr_2$ if C is not empty. Hence, the complete expression evaluates to $Expr_1$ if C is empty, and to $Expr_2$ if C is not empty. Therefore, we can express an if-construction in our algebra.

We will assume in our construction that the value of $\mathcal{X}$ changes, as long as the condition $C_1 \theta C_2$ is satisfied. This assumption can easily be made true without changing the result of the loop; indeed, in the original extended loop (that is, before we removed multiple variables), we can add a dummy variable $\mathcal{D}$ over schema $\{X_1\}$, that is instantiated to N_1 before the loop. Furthermore, we add $\mathcal{D} := N_1 \setminus \mathcal{D}$ inside the loop. In this way, the value of $\mathcal{D}$ will iterate between N_1 and the empty relation, and thus never become stable. The result of the loop is not affected by this dummy variable. However, the dummy variable makes sure that the value the unique variable $\mathcal{X}$ after the removal of the multiple variables, will always change between two loop iterations. The reason for this rather technical requirement will become clear after the definition of the rewriting of the looping condition.

With the if-construction we have a tool to replace the condition $C_1 \theta C_2$ by $\Delta \mathcal{X}$ as follows:

$\mathcal{X} := Seed$;
While($\Delta \mathcal{X}$) **loop**
    $\mathcal{X}$:=
        (**If** $(C_1 \theta C_2)$ **Then**
            $Expr$
        Else
            $\mathcal{X}$
        End if);
End loop
Return Ret;

Since we assumed that the value of $\mathcal{X}$ changes as long as the condition $C_1 \theta C_2$ is satisfied, $\mathcal{X}$ remains the same between two iterations if and only if $C_1 \theta C_2$ does no longer hold. In that case the Else-branch of the If-operator is chosen, resulting in the assignment $\mathcal{X} := \mathcal{X}$, thus leaving $\mathcal{X}$ unchanged, which results in an exit of the loop. Therefore, the result of the loop remains the same. Notice that the rewriting did not change the number of assignments within the loop; the if-structure is one expression.

Replacing *Ret* **by** $\mathcal{X}$ This is far the most easy rewriting; instead of returning the expression *Ret*, we let the loop return $\mathcal{X}$, and evaluate the expression *Ret* outside, on the result of the simple loop.

B. $\mathcal{MA}^{FI}$ CAN ACCEPT EXPSPACE LANGUAGES

Let L be a language over the alphabet $\{0,1\}$ in SPACE(2^n), but not in PSPACE. We will reduce deciding L to the non-emptiness problem of an expression in $\mathcal{MA}$. For a string $x \in \{0,1\}^*$, construct the following relation $R(\texttt{Nr},\texttt{Bit})$: R consists of the tuples $(i, x[i])$, $1 \leq i \leq |x|$. Since $L \in$ SPACE(2^n), there exists a deterministic Turing machine M that decides L using at most $2^{|x|}$ space on input x. We assume without loss of generality that M only uses the symbols "0", "1" and blanco on its tapes. We will simulate the working of M in the I-world. We first show how we can encode the tape in the I-world, then how to move the input to the I-world and finally how to simulate the finite control of M.

Work tape We will simulate the work tape with two I-world relations $\mathcal{T}_0(\texttt{Reg})$ and $\mathcal{T}_1(\texttt{Reg})$. First we generate 2^n regions as follows: for each Nr-value in R, we generate a constraint: $\mathcal{C} := \pi_{\texttt{Reg}}\kappa_{\texttt{Nr}\leq val(\texttt{Nr})}\pi_{\texttt{Nr}}R$. Notice that because of the use of $\leq$ in the template, the order $\leq$ on the Nr-values carries over to $\mathcal{S}$ via $\rightarrow$. Thereafter we generate all subsets of $\mathcal{C}$. In this way we have generated 2^n sets. Furthermore, we can carry over the total order on $\pi_{\texttt{Nr}}R$ to $\mathcal{S}$ as follows: a region reg_1 comes before reg_2 if either $reg_1 \subset reg_2$ or reg_1 and reg_2 are incomparable w.r.t. $\subset$ and the smallest constraint (w.r.t. $\prec$) in $reg_1 - reg_2$ comes before the smallest constraint in $reg_2 - reg_1$. This order is well-defined. In fact, if we consider a region as a binary number in which bit i is 1 iff the ith constraint in $\mathcal{C}$ is in Reg, this order coincides with the natural order on the numbers. For convenience we materialize the successor table $\mathcal{SUCC}(\texttt{Reg}_1, \texttt{Reg}_2)$ among the regions in $\mathcal{S}$. It is clear that this table can be constructed with I-world algebra. The work tape is encoded as follows: the i-th symbol s_i on the tape is captured by the i-th region r_i in the order on $\mathcal{S}$; s_i is blanco iff r_i is not in $\mathcal{T}_0$ or $\mathcal{T}_1$; s_i is 0 iff r_i is in $\mathcal{T}_0$; s_i is 1 iff r_i is in $\mathcal{T}_1$. **Move the input to the I-world** We select two sets of Nr-values in R: the ones that index a 0, and the ones that index a 1.

$$\mathcal{ONE} := \kappa_{\texttt{Nr}\leq val(\texttt{Nr})}(\sigma_{\texttt{Bit}=1}R)$$

$\mathcal{ZERO}$ is defined similarly. We then put the input on the tape with a loop that iteratively picks the smallest region w.r.t. $\prec$ in $\mathcal{ONE} \cup \mathcal{ZERO}$, and puts it on the tape. **Finite Control** We assume that M has k states. Since the simulation depends on M, we can treat k as a constant. We generate the set $\{0,\ldots,k-1\}$ as follows.

$$\begin{aligned} States(0) &:= \Gamma_{<>}^{COUNT(*)\ as\ \texttt{State}}\ (R - R) \\ States(n) &:= \left(\Gamma_{<>}^{COUNT(*)\ as\ \texttt{State}}\ States(n-1)\right) \ \cup\ States(n-1) \end{aligned}$$

The set of states then becomes $States(k-1)$. We use $\kappa_{\texttt{State}\leq val(\texttt{State})}$ to get the states in the I-world. Again we make use of $\prec$ to maintain the order in the I-world. This order makes it possible to write expressions that select the i-th state. We will during simulation store the state in the I-relation $\mathcal{STATE}$. The tape-head is stored in $\mathcal{HEAD}$. The tape head is on the i-th symbol if $\mathcal{HEAD}$ contains the i-th region

in the order. It is easy to see that with the if-then-else type of expressions we can simulate a single step of M. Finally, we write a loop that will compute one step per iteration until we reach the accept or reject state. Depending on the state we will then return R or the empty relation.

C. SOLVING SYSTEMS OF INEQUALITIES IN $\mathcal{DA}$

Let $R(A_1, \ldots, A_n, B)$ be a relation over rational numbers. The tuple $(a_1, \ldots, a_n, b)$ of R expresses the linear inequality $a_1x_1 + a_2x_2 + \ldots + a_nx_n \leq b$. We will show an expression $SAT(R)$ in the relational algebra that decides whether the system represented by R is satisfiable.

For solving the satisfiability problem, we will iteratively eliminate variables x_i, using the elimination method of Fourier and Motzkin [Murty 1983; Dantzig 1963]. We will illustrate the elimination method with an example. Consider the following relation together with the system of inequalities represented by it.

A_1	A_2	A_3	A_4	A_5	B
1	3	0	0	0	5
1	0	0	1	−7	3
−1	0	1	0	0	5
0	0	1	0	1	4

$$\begin{aligned} x_1 + 3x_2 &\leq 5 \\ x_1 + x_4 - 7x_5 &\leq 3 \\ -x_1 + x_3 &\leq 5 \\ x_3 + x_5 &\leq 4 \end{aligned}$$

We will rewrite this system into an equivalent system without x_1. This goes as follows: in all inequalities, x_1 can be isolated. That is, for $x_1 + x_4 - 7x_5 \leq 3$, we get: $x_1 \leq 3 - x_4 + 7x_5$, and for $-x_1 + x_3 \leq 5$, we get $x_1 \geq x_3 - 5$, when we isolate x_1. In this way, we get lower and upper bounds on x_1. It is easy to see that $\{x_1 \leq 3 - x_4 + 7x_5, x_1 \geq x_3 - 5\}$ has a solution if and only if there exist x_2, x_3, x_4, x_5 such that $x_3 - 5 \leq 3 - x_4 + 7x_5$, or equivalently, $x_3 + x_4 - 7x_5 \leq 8$, has a solution. The Fourier-Motzkin elimination method is now based on making this combination for every lower and upper bound. Hence, for the given system, we get all lower bounds L on x_1, by selecting those tuples t in R with $t.A_1 < 0$, and all upper bounds U by selecting those with $t.A_1 > 0$. Combining all lower bounds with all upper bounds, can be done by taking the Cartesian product, and then applying the appropriate mathematical operations. The new system without x_1 is then found by unioning together all tuples of R in which A_1 was 0 with the newly constructed inequalities. In this case, this expression (broken down into subexpressions) is:

$$\begin{aligned} L &= \sigma_{A_1<0}\ R \\ U &= \sigma_{A_1>0}\ R \\ N &= \rho_{A_1\to A_1',\ldots,A_5\to A_5',B\to B'} L \times \rho_{A_1\to A_1'',\ldots,A_5\to A_5'',B\to B''}\ U \\ I &= Calc_{(A_1''\cdot A_2' - A_1'\cdot A_2'')\ as\ A_2,\ldots,(A_1''\cdot B' - A_1'\cdot B'')\ as\ B}\ N \\ R' &= \pi_{A_2,\ldots,A_5,B}(\sigma_{A_1=0} R\ \cup\ I) \end{aligned}$$

So, for our example, the resulting relation and system of inequalities are:

A_2	A_3	A_4	A_5	B
3	1	0	0	10
0	1	1	−7	8
0	1	0	1	4

$$\begin{aligned} 3x_2 + x_3 &\leq 10 \\ x_3 + x_4 - 7x_5 &\leq 8 \\ x_3 + x_5 &\leq 4 \end{aligned}$$

This new system does no longer contain variable x_1, and has a solution if and only if the original system has one.

In this way, we can eliminate the variables one by one with one large relational algebra expression. Finally, when all variables are eliminated, a relation S with only the column B is left, with the tuples (b) representing the inequalities $0 \leq b$. Obviously, this system is satisfiable if there is no tuple (b) in S with $b < 0$; that is; when $\sigma_{B<0}\ S$ is empty.

D. TC IS IN $\mathcal{MA}^{FI}$

Let $D(A, B)$ be a relation in the D-world, and A and B have domain $\mathcal{U}$. The result of TC on D is the transitive closure of D; i.e., $TC(D)$ contains (a, b) if and only if there exist tuples (a, x_1), (x_1, x_2), ..., (x_k, b) in D.

In the construction of the TC query, we will represent an edge (a, b) as a constraint $(\{(a, b)\}) \in \{\{A, B\}\}$ in the I-world. We will use the frequent set operator FI with support threshold 1 to construct all subsets of edges in D. Then, those subsets of edges are selected that fulfil the following *path* conditions:

—No two edges have the same start node;

—No two edges have the same end node;

—There exists exactly one node s that has an outgoing edge in the set, but no incoming edge; we call this node s the *starting node*;

—There exists exactly one node e that has an incoming edge in the set, but no outgoing edge; we will call this node e the *ending node*.

If a set of edges fulfils these conditions, then there exists a path from s to e. This can easily be shown by induction on the number of edges in the set of edges S. For $|S| = 1$, there is trivially a path between s and e; namely, the one edge in S connects them. In general, let s' be the node such that (s, s') is the unique outgoing edge from s in S. Then, $S \setminus \{(s, s')\}$ again fulfils the conditions, with as starting and ending nodes respectively the nodes s' and e. By induction there must exist a path between s' and e. Therefore, there is a path between s and e, namely, the edge (s, s') followed by the path from s' to e.

Notice that the other direction holds as well; if there exists a path between two nodes s and e, then this path is a set of edges fulfilling the above conditions. Therefore, the edges in the transitive closure of D are exactly those pairs (s, e) such that there exists a set of edges of D fulfilling the above conditions and having s and e as starting and ending nodes respectively.

To illustrate the construction of the expression for the transitive closure, we will use the following example database: $\mathtt{D} = \{(\mathtt{a}, \mathtt{b}), (\mathtt{a}, \mathtt{c}), (\mathtt{b}, \mathtt{c}), (\mathtt{c}, \mathtt{d})\}$. The transitive closure $TC(D)$ of this relation is $\{(\mathtt{a}, \mathtt{b}), (\mathtt{a}, \mathtt{c}), (\mathtt{a}, \mathtt{d}), (\mathtt{b}, \mathtt{c}), (\mathtt{b}, \mathtt{d}), (\mathtt{c}, \mathtt{d})\}$.

D.1 Constructing the sets of edges

We will nest the relation D twice; once to form the pairs $\{(a, b)\}$, and the second time to form a relation with one tuple over the schema with one attribute $\{\{A, B\}\}$ representing one large transaction $\{(\{(\mathtt{a}, \mathtt{b})\}), (\{(\mathtt{a}, \mathtt{c})\}), (\{(\mathtt{b}, \mathtt{c})\}), (\{(\mathtt{c}, \mathtt{d})\})\}$. The expression to form this transaction database TDB is the following:

$$Nest_{\{A,B\}} \pi_{\{A,B\}} Nest_{A,B}\ \sigma_{A=A',B=B'} (\rho_{A \to A', B \to B'} D \times D)$$

We then apply the frequent itemset operator with threshold 1 to this transaction database: $FI(TDB, 1)$ to get the relation $\mathcal{S}ets$ in the I-world. To make the notations less heavy, we denote the constraint $\{(a, b)\} \in \{\{A, B\}\}$ with $[a, b]$. With this notation, the relation $\mathcal{S}ets$ becomes:

Set
$\{\}$
$\{[\mathtt{a}, \mathtt{b}]\}$
$\{[\mathtt{a}, \mathtt{c}]\}$
...
$\{[\mathtt{c}, \mathtt{d}]\}$
$\{[\mathtt{a}, \mathtt{b}], [\mathtt{a}, \mathtt{c}]\}$
...
$\{[\mathtt{a}, \mathtt{b}], [\mathtt{a}, \mathtt{c}], \ldots, [\mathtt{c}, \mathtt{d}]\}$

D.2 Selecting the sets representing paths

Because it is impossible to unnest the pairs representing the edges in the I-world, it is not easy to select those sets that meet the path conditions. In order to express these conditions, we first construct a helper relation. To make the notations less heavy, we introduce the following notations: $[a, *]$ will denote the region consisting of all constraints $[a, x]$, with x any other vertex. E.g., in the running example, $[a, *]$ denotes the region: $\{\ [\mathtt{a}, \mathtt{a}], [\mathtt{a}, \mathtt{b}], [\mathtt{a}, \mathtt{c}], [\mathtt{a}, \mathtt{d}]\ \}$. Similarly, $[*, a]$ will denote the region consisting of all constraints $[x, a]$, with x any other vertex. In the running example, $[*, a]$ denotes $\{\ [\mathtt{a}, \mathtt{a}], [\mathtt{b}, \mathtt{a}], [\mathtt{c}, \mathtt{a}], [\mathtt{d}, \mathtt{a}]\ \}$. Finally, $[a]$ will denote the set of constraints $[x, y]$ with x or y equal to a, and the other variable any other vertex. That is, in the running example, $[a]$ is: $\{[\mathtt{a}, \mathtt{a}], [\mathtt{a}, \mathtt{b}], [\mathtt{a}, \mathtt{c}], [\mathtt{a}, \mathtt{d}], [\mathtt{b}, \mathtt{a}], [\mathtt{c}, \mathtt{a}], [\mathtt{d}, \mathtt{a}]\}$.

We give an expression that constructs the following relation $\mathcal{E}dges$ in the I-world:

$$\{([x], [x, *], [*, x]) \mid x \textit{ is a vertex}\}$$

That is, for every vertex x in our original relation D, $\mathcal{E}dges$ will contain a tuple $([x], [x, *], [*, x])$. We will first construct three relations, holding respectively $([x])$ for all x, $([x, *])$ for all x, and $([*, x])$ for all x. Then, $\mathcal{E}dges$ is formed by taking the Cartesian product, and exploiting the quality $[x] = [x, *] \cup [*, x]$.

Forming $[x, *]$, $[*, x]$, and $[x]$ First we select all vertices:

$$V := \pi_A\ D \cup \pi_{B\ as\ A}\ D$$

In our running example, V is $\{(\mathtt{a}), (\mathtt{b}), (\mathtt{c}), (\mathtt{d})\}$. Then, all possible pairs are formed,

$$P := \pi_{\{\{A,B\}\}} Nest_{\{A,B\}} \pi_{A',\{A,B\}} Nest_{A,B}\ \sigma_{A'=A,B'=B}(\rho_{A\ as\ A'}\ V \times \rho_{A\ as\ B'}\ V \times V \times \rho_{A\ as\ B}\ V)$$

For our running example, this expression gives:

$\{\{\mathtt{A}, \mathtt{B}\}\}$
$\{\ \{(\mathtt{a}, \mathtt{a})\},\ \{(\mathtt{a}, \mathtt{b})\},\ \{(\mathtt{a}, \mathtt{c})\},\ \{(\mathtt{a}, \mathtt{d})\}\ \}$
$\{\ \{(\mathtt{b}, \mathtt{a})\},\ \{(\mathtt{b}, \mathtt{b})\},\ \{(\mathtt{b}, \mathtt{c})\},\ \{(\mathtt{b}, \mathtt{d})\}\ \}$
$\{\ \{(\mathtt{c}, \mathtt{a})\},\ \{(\mathtt{c}, \mathtt{b})\},\ \{(\mathtt{c}, \mathtt{c})\},\ \{(\mathtt{c}, \mathtt{d})\}\ \}$
$\{\ \{(\mathtt{d}, \mathtt{a})\},\ \{(\mathtt{d}, \mathtt{b})\},\ \{(\mathtt{d}, \mathtt{c})\},\ \{(\mathtt{d}, \mathtt{d})\}\ \}$

Subsequently, we use the frequent set mining operator with threshold 1 to form all subsets of the tuples in P. From these subsets we only keep the maximal ones, thus effectively selecting the sets $[x,*]$:

$$FI(P,1) - \pi_{Reg}\sigma_{Reg' \prec Reg}(FI(P,1) \times \rho_{Reg \rightarrow Reg'} FI(P,1))$$

That is, we select those itemsets S such that there does not exist another itemset S' with $S' \prec S$. Hence, we select the maximal itemsets.

The relations for $[x]$ and $[*,x]$ can be constructed in a similar way.

Combining $[x,*]$, $[*,x]$, and $[x]$ We now combine the relations for $[x,*]$, $[*,x]$, and $[x]$ into one relation, by taking the Cartesian product, and selecting those tuples $([x],[y,*],[*,z])$ with $[x] = [y,*] \cup [*,z]$. In this way we only keep the tuples with $x = y = z$. This selection can be expressed using the *Union* introduced in Section 4.3. This concludes the construction of $\mathcal{E}dges$.

Selecting the sets that fulfil the path conditions We will use the helper relation $\mathcal{E}dges$ to construct the relation $\mathcal{S}E$ over schema $\{Set, Starting, Ending\}$: for every tuple $(\{[x_1,y_1],\ldots,[x_n,y_n]\})$, $\mathcal{S}E$ will contain for $i = 1\ldots n$, the tuples $(\{[x_1,y_1],\ldots,[x_n,y_n]\},[x_i],[y_i])$. Now it is straightforward to select those sets $(\{[x_1,y_1],\ldots,[x_n,y_n]\})$ that fulfil the path conditions. For example, selecting those sets that violate the condition that no two edges can share a starting node is done as follows:

$$\sigma_{Set=Set',Start=Start',End\neq End'}(\mathcal{S}E \times \rho_{Set\rightarrow Set',Start\rightarrow Start',End\rightarrow End'}\mathcal{S}E)$$

For each of the valid sets we can select the starting nodes as follows:

$$\pi_{Set,Start\ as\ Node} - \pi_{Set,End\ as\ Node}\mathcal{S}E$$

In a similar way we can select the ending node. Based on these starting and ending nodes, and the relation $\mathcal{E}dges$, we can construct the relation that consists of the tuples $(S,\{[x,y]\})$, with S a valid set, and x the starting node, and y the ending node. Notice that $\{[x,y]\}$ can be constructed using the equality $\{[x,y]\} = [x,*] \cap [*,y]$. By projecting on the edges, we get the transitive closure $\mathcal{TC}$ in the I-world. Via a populate, and a projection on the D-world attributes, we get the transitive closure in the D-world: $Unnest_{\{\{A,B\}\}}\pi_{\mathcal{A}}Pop(\mathcal{TC}, Pairs)$, where $Pairs$ is the D-world relation holding all pairs of elements in D.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library by visiting the following URL: `http://www.acm.org/pubs/citations/journals/tods/2006-V-N/p1-URLend`.

REFERENCES

Abiteboul, S. and Hillebrand, G. G. 1995. Space usage in functional query languages. In *Proc. ICDT Int. Conf. Database Theory*, G. Gottlob and M. Vardi, Eds. Lecture Notes in Computer Science, vol. 893. Springer-Verlag, London, UK, 439–454.

Abiteboul, S., Hull, R., and Vianu, V. 1995. *Foundations of Databases.* Addison-Wesley, Reading, MA.

Agrawal, A. and Srikant, R. 1994. Fast algorithms for mining association rules. In *Proc. VLDB Int. Conf. Very Large Data Bases.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 487–499.

Agrawal, R., Imielinski, T., and Swami, A. 1993. Mining association rules between sets of items in large databases. In *Proc. ACM SIGMOD Int. Conf. Management of Data*. ACM Press, New York, NY, USA, 207–216.

Boulicaut, J.-F., Klemettinen, M., and Mannila, H. 1999. Modeling KDD processes within the inductive database framework. In *Proc. DaWaK Int. Conf. Data Warehousing and Knowledge Discovery*. Lecture Notes in Computer Science, vol. 1676. Springer-Verlag, London, UK, 293–302.

Breiman, L., Friedman, J., Olshen, R., and Stone, C. 1984. *Classification and Regression Trees*. Wadsworth, Belmont, CA.

Calders, T., Rigotti, C., and Boulicaut, J.-F. 2006. A survey on condensed representations for frequent sets. In *Constraint-based mining and inductive databases*, J.-F. Boulicaut, L. de Raedt, and H. Mannila, Eds. LNCS, vol. 3848. Springer-Verlag, London, UK.

Chaudhuri, S., Narasayya, V. R., and Sarawagi, S. 2002. Efficient evaluation of queries with mining predicates. In *ICDEproc*. IEEE Computer Society, San Jose, CA, 529–540.

Consens, P. and Mendelzon, A. 1993. Low complexity aggregation in graphlog and datalog. *Theoretical Computer Science 116,* 1& 2, 95–116.

Dantzig, G. 1963. *Linear Programming and Extensions*. Princeton University Press, Princeton, NJ.

Garey, M. and Johnson, D. S. 1979. *Computer and Intractability: A Guide to NP-Completeness*. W. H. Freeman, New York.

Geist, I. and Sattler, K. 2002. Towards data mining operators in database systems: Algebra and implementation. In *Proceedings DBFusion International Workshop on Databases, Documents, and Information Fusion*. Vol. 124. CEUR-WS, Karlsruhe, Germany.

Guttman, A. 1984. R-trees: a dynamic index structure for spatial searching. In *Proc. ACM SIGMOD Int. Conf. Management of Data*. ACM Press, New York, NY, USA, 47–57.

Han, J., Fu, Y., Wang, W., Koperski, K., and Zaiane, O. 1996. DMQL: A data mining query language for relational databases. In *Proc. ACM SIGMOD Int. Conf. Management of Data*. ACM Press, New York, NY, USA, 27–33.

Han, J., Pei, J., and Yin, Y. 2000. Mining frequent patterns without candidate generation. In *Proc. ACM SIGMOD Int. Conf. Management of Data*, W. Chen, J. Naughton, and P. Bernstein, Eds. ACM Press, New York, NY, USA, 1–12.

Hand, D., Mannila, H., and Smyth, P. 2001. *Principles of Data Mining*. MIT Press, Cambridge, MA.

Imielinski, T. and Mannila, H. 1996. A database perspective on knowledge discovery. *Comm. of the ACM 39,* 11, 58–64.

Imielinski, T. and Virmani, A. 1999. MSQL: A query language for database mining. *Knowledge Discovery and Data Mining 3,* 4, 373–408.

Johnson, T., Lakshmanan, L. V., and Ng, R. 2000. The 3w model and algebra for unified data mining. In *Proc. VLDB Int. Conf. Very Large Data Bases*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 21–32.

Kennedy, K. and McKinley, K. S. 1994. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*. Springer-Verlag, London, UK, 301–320.

Law, Y.-N., Wang, H., and Zaniolo, C. 2004. Query languages and data models for database sequences and data streams. In *Proc. VLDB Int. Conf. Very Large Data Bases*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 492–503.

Libkin, L. and Wong, L. 1997. On the power of aggregation in relational query languages. In *Proc. DBPL Workshop on Databases and Programming Languages*. Lecture Notes in Computer Science, vol. 1369. Springer-Verlag, London, UK, 260–280.

Mannila, H. and Toivonen, H. 1996. Multiple uses of frequent sets and condensed representations. In *Proc. KDD Int. Conf. Knowledge Discovery in Databases*. ACM Press, New York, NY, USA.

MEO, R., PSAILA, G., AND CERI, S. 1996. A new sql-like operator for mining association rules. In *Proc. VLDB Int. Conf. Very Large Data Bases*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 122–133.

MURTY, K. G. 1983. *Linear Programming*. John Wiley & Sons, New York.

NETZ, A., CHAUDHURI, S., FAYYAD, U. M., AND BERNHARDT, J. 2001. Integrating data mining with sql databases: Ole db for data mining. In *Proc. IEEE ICDE Int. Conf. on Data Engineering*. IEEE Computer Society, Washington, DC, USA, 379–387.

PARADAENS, J., VAN DEN BUSSCHE, J., AND VAN GUCHT, D. 1994. Towards a theory of spatial database queries. In *Proc. PODS Int. Conf. Principles of Database Systems*. ACM Press, New York, NY, USA, 279–288.

PARADAENS, J. AND VAN GUCHT, D. 1998. Possibilities and limitations of using flat operators in nested algebra expressions. In *Proc. PODS Int. Conf. Principles of Database Systems*. ACM Press, New York, NY, USA, 29–38.

SARAWAGI, S., THOMAS, S., AND AGRAWAL, R. 1998. Integrating association rule mining with relational database systems: Alternatives and implications. In *Proc. ACM SIGMOD Int. Conf. Management of Data*. ACM Press, New York, NY, USA, 343–354.

TSUR, D., ULLMAN, J. D., ABITEBOUL, S., CLIFTON, C., MOTWANI, R., NESTOROV, S., AND ROSENTHAL, A. 1998. Query flocks: a generalization of association-rule mining. In *Proc. ACM SIGMOD Int. Conf. Management of Data*. ACM Press, New York, NY, USA, 1–12.

VAN DEN BUSSCHE, J. 2001. Simulation of the nested relational algebra by the flat relational algebra, with an application to the complexity of evaluating powerset algebra expressions. *Theoretical Computer Science 254,* 1-2, 363–377.

WANG, H. AND ZANIOLO, C. 2000. Nonmonotonic reasoning in LDL++. In *Logic-Based Artificial Intelligence*, J. Minker, Ed. Kluwer Academic Publishers, Dordrecht, 523–544.

WANG, H. AND ZANIOLO, C. 2003. ATLaS: A native extension of sql for data mining. In *Proceedings of the Third SIAM International Conference on Data Mining*. SIAM, US.

ZANIOLO, C. 2005. Mining databases and data streams with query languages and rules. In *Proceedings ECML-PKDD 2005 Workshop Knowledge Discovery in Inductive Databases*. LNCS, vol. 3933. Springer-Verlag, London, UK.

Received Month Year; revised Month Year; accepted Month Year

ACKNOWLEDGMENTS

Toon Calders is funded by the Fund for Scientific Research - Flanders (FWO-Vlaanderen) as a post-doctoral fellow. This work has been partially funded by the EU contract IQ FP6-516169. Lakshmanan's research was supported in part by a grant from the Natural Sciences and Engineering Research Council of Canada, a fellowship from the BC Advanced Systems Institute, and a grant from Networks of Centres of Excellence/Institute for Robotics and Intelligent Systems, Phase IV.

THIS DOCUMENT IS THE ONLINE-ONLY APPENDIX TO:

Expressive Power of an Algebra For Data Mining

TOON CALDERS
University of Antwerp, Belgium
and
LAKS V.S. LAKSHMANAN and RAYMOND T. NG
University of British Columbia, Canada
and
JAN PAREDAENS
University of Antwerp, Belgium

ACM Transactions on Database Systems, Vol. V, No. N, July 2006, Pages 1–44.

A. $\mathcal{MA}$ IS TURING COMPLETE FOR D→D QUERIES

In this appendix we proof the following theorem: **Theorem 5.1** Every generic computable $D{\rightarrow}D$ data mining query is $\mathcal{MA}$-expressible. □

The proof will consist of two large parts:

(1) We show that we can encode integers in the I-world, and that the $\mathcal{MA}$-algebra is capable to express every computable query on these integers.
(2) We construct two expressions *enc* and *dec* that respectively encode a D-world as integers in the I-world, and decode the integers in the I-world back to relations in the D-world.

These two parts together make that the $\mathcal{MA}$-algebra is Turing complete for generic D to D queries. Indeed; let q be a query from D to D. Then, $Q = dec \circ q \circ enc$ is a computable query from integers to integers in the I-world. Because the algebra $\mathcal{MA}$ is Turing complete on integers in the I-world, there exists an expression for Q. This expression, preceded by the expression *enc* and followed by *dec* form an expression for q. In the proof special care is needed to handle genericity.

But before we give the full proof, we first introduce some auxiliary expressions that play a crucial role in the proof. We introduce expressions for:

—constructing powersets in the I-world;
—mapping an integer in the D-world to the representation of an integer in the I-world, and vice versa;
—splitting a rational number p/q in the D-world into a triplet of positive integers (s, n, d), with s encoding the sign, n the absolute value of the nominator and q the absolute value of the denominator;

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 2006 ACM 0362-5915/2006/0300-0001 $5.00

—going from an integer to its binary representation and vice versa.

A.1 Auxiliary Expressions

A.1.1 *Constructing powersets in the I-world.* The expression $\mathcal{S}ubs$ expresses the following I→I query: given a relation $\mathcal{S}$ over the schema $\{R\}$, $\mathcal{S}ubs(\mathcal{S})$ will be the I-world relation over schema $\{S\}$ consisting of all non-empty sets of basic constraints that are in regions in $\mathcal{S}$. For example, consider the following relation $\mathcal{S}$:

R
$\{\mathtt{A}=1,\mathtt{B}=1\}$
$\{\mathtt{A}+\mathtt{B}=3,\mathtt{B}=1\}$

The basic constraints in this relation $\mathcal{S}$ are: $\mathtt{A} = 1$, $\mathtt{B} = 1$, $\mathtt{A}+\mathtt{B} = 3$. Hence, $\mathcal{S}ubs(\mathcal{S})$ will be the following relation:

S
$\{\mathtt{A}=1\}$
$\{\mathtt{B}=1\}$
$\{\mathtt{A}+\mathtt{B}=3\}$
$\{\mathtt{A}=1,\mathtt{B}=1\}$
$\{\mathtt{A}=1,\mathtt{A}+\mathtt{B}=3\}$
$\{\mathtt{B}=1,\mathtt{A}+\mathtt{B}=3\}$
$\{\mathtt{A}=1,\mathtt{B}=1,\mathtt{A}+\mathtt{B}=3\}$

The following (static) $\mathcal{MA}$-expression expresses the desired query:

$(\mathcal{X} := U_R(\mathcal{S});$
While($\Delta\mathcal{X}$) **loop**
  $\mathcal{X}{:=}\mathcal{X} \cup \rho_{U\to R}\ \pi_U\ \mathit{Union}(\mathcal{X} \times \rho_{R\to R'}\mathcal{X});$
End loop
Return $\mathcal{X}$;)

Recall that The expression $\mathit{Union}(\mathcal{X}(S_1, S_2))$ used here was introduced in Section 4.3 and produces the following relation over $\{S_1, S_2, U\}$:

$$\{(s_1, s_2, s_1 \cup s_2) \mid (s_1, s_2) \in \mathcal{X}\}\ .$$

A.1.2 *Integers in I- and D-world. Zero* **and** *One* Let D be an arbitrary D-world relation. We give D→D-expressions *Zero* and *One* that respectively return the relations $\{(0)\}$ and $\{(1)\}$ over the schema $\{Nr\}$.

$\mathit{Zero}{:=}\Gamma_{\langle\rangle}^{\mathtt{COUNT(*)\ as\ Nr}}(D \setminus D)$
$\mathit{One}{:=}\Gamma_{\langle\rangle}^{\mathtt{COUNT(*)\ as\ Nr}}\mathit{Zero}$

Integers in the I-world The following I-world relation $\mathcal{N}r[n]$ will be used to denote the number n:

Reg
$\{\mathtt{Nr} \leq 0\}$
$\{\mathtt{Nr} \leq 1\}$
$\{\mathtt{Nr} \leq 2\}$
...
$\{\mathtt{Nr} \leq n\}$

The transformation of I-world relation $\mathcal{N}r[n]$ to the number n in a relation in the D-world and vice versa will be key constructions in the proof of the Turing completeness of $\mathcal{MA}$.

Mapping $\mathcal{N}r[n]$ to n First we give an expression to go from a relation $\mathcal{N}(Nr)$ in the I-world expressing a number n, to the D-world relation $\{(n)\}$. $MapNr_{\mathrm{I}\rightarrow\mathrm{D}}(\mathcal{N})$ is the following expression:

$$\pi_{\mathcal{A}}\Gamma_{\langle\rangle}^{\texttt{COUNT(*) as Nr}} Pop(\mathcal{N}, One);$$

That is, first $\mathcal{N}$ is populated with the D-world relation One. This creates an E-world relation with tuples $(\{Nr \leq i\}, 1)$, for all $i = 1 \ldots n$. The number of tuples in this E-world is then counted, resulting in the number n. Subsequently, this number is projected to the D-world.

Mapping n to $\mathcal{N}r[n]$ Let $MapNr_{\mathrm{D}\rightarrow\mathrm{I}}(D)$ now be the following expression, mapping a D-world relation $D(Nr) = \{n\}$ to the I-world relation $\mathcal{N}r[n]$.

($\mathcal{X} := \kappa_{\texttt{Nr}\leq\texttt{val(Nr) as Reg}}\ Zero$;
While($MapNr_{\mathrm{I}\rightarrow\mathrm{D}} \neq D$) **loop**
  $T := MapNr_{\mathrm{I}\rightarrow\mathrm{D}}(\mathcal{X})$;
  $T' := \rho_{Nr'\rightarrow Nr}\ \pi_{Nr'}\ Calc_{Nr+1 \texttt{ as } Nr'}\ T$;
  $\mathcal{X} := \mathcal{X} \cup \kappa_{Nr\leq val(Nr)\ as\ Reg}T'$;
End loop
Return $\mathcal{X}$;)

The expression $MapNr_{\mathrm{D}\rightarrow\mathrm{I}}(D)$ works as follows: first $\mathcal{X}$ is initialized to $\mathcal{N}r[0]$. In the loop, $\mathcal{X}$ is updated such that after the ith loop, $\mathcal{X}$ is $\mathcal{N}r[i]$. This is done as follows: via $MapNr_{\mathrm{I}\rightarrow\mathrm{D}}$, the number represented by $\mathcal{X}$ is transported to the D-world (in the beginning of the ith loop this number is $i-1$). There, this number is incremented, resulting in i, and the constraint $Nr \leq i$ is constructed and added to $\mathcal{X}$. The loop is repeated until $MapNr_{\mathrm{I}\rightarrow\mathrm{D}}(\mathcal{X})$ equals D; i.e., until $\mathcal{X}$ represents the number stored in D.

Using both $MapNr_{\mathrm{D}\rightarrow\mathrm{I}}$ and $MapNr_{\mathrm{I}\rightarrow\mathrm{D}}$, we can do arithmetic with the encoded numbers in the I-world. Indeed; we can map numbers $\mathcal{N}r[n]$ in the I-world to numbers n in the D-world, do arithmetic on these numbers, and map them back to the I-world. To make notations less heavy, we will use expressions like:

$$\mathcal{X} \ := \ \mathcal{N}r[n] + \mathcal{N}r[m];$$

to actually denote

$$\begin{aligned}\mathcal{X} \ := \ & MapNr_{\mathrm{D}\rightarrow\mathrm{I}}\ \pi_{Nr}\ Calc_{Nr'+Nr''\ as\ Nr} \\ & [\rho_{Nr\rightarrow Nr'}(MapNr_{\mathrm{I}\rightarrow\mathrm{D}}\mathcal{N}r[n]) \times \rho_{Nr\rightarrow Nr''}(MapNr_{\mathrm{I}\rightarrow\mathrm{D}}\mathcal{N}r[m])];\end{aligned}$$

Alternative Representations Notice that it is also possible to use other representations for the integers. For example, we could use the relation $\{(\{Nr = n\})\}$ to represent the relation n. We can go from the representation $\mathcal{N}r[n]$ to this notation in the following way:

$$Map_{(\leq\ to\ =)} := \kappa_{Nr=val(Nr)} MapNr_{I\rightarrow D}\mathcal{N}r[n]$$

The other direction, that is, from $\mathcal{N} = \{(\{Nr = n\})\}$ to $\mathcal{N}r[n]$ will be denoted $Map_{(=\ to\ \leq)}$, and can be expressed as follows:

($\mathcal{X} := MapNr_{D\rightarrow I}\ Zero$;
$\mathcal{Y} := Map_{(\leq\ to\ =)}(\mathcal{X})$;
While($\mathcal{Y} \neq \mathcal{N}$) **loop**
  $\mathcal{X} := \mathcal{X} + 1$;
  $\mathcal{Y} := Map_{(\leq\ to\ =)}(\mathcal{X})$;
End loop
Return $\mathcal{X}$;)

Therefore, in the following we will often assume that we are working with the more convenient form $\{(\{Nr = n\})\}$. To make notations less heavy we will often relax the notation $MapNr(Map_{(=\ to\ \leq)}(\mathcal{N}))$ to simply $MapNr(\mathcal{N})$.

A.1.3 *Splitting a rational number.* Let $D(Nr) = \{(p/q)\}$ be a D-world relation holding a single rational number p/q (p and q are integers). We give expressions that create the I-world relations $\mathcal{S}$, $\mathcal{N}r[|p|]$, and $\mathcal{N}r[|q|]$. $\mathcal{S}$ will be a relation over the schema $\{Sign\}$, holding the sign of p/q. That is, if p/q is positive, $\mathcal{S}$ will be $\{\{Sign = 1\}\}$, otherwise $\mathcal{S}$ will be $\{\{Sign = 0\}\}$. The sign can easily be extracted as follows:

$\mathcal{S}{:=}\kappa_{Sign=val(Nr)}$ (**If** $(\sigma_{Nr\geq 0} D \neq \{\})$ **Then** One **Else** $Zero$);

For notational convenience we introduce the following expression $abs(D)$ that returns the relation $\{|p/q|\}$.

$abs(D)\ \equiv$ (**If** $(\sigma_{Nr\geq 0} D \neq \{\})$ **Then** D **Else** $Calc_{(-Nr)\ as\ Nr}\ D$);

The positive rational number $|p|/|q|$ is now split up into the positive integers $|p|$ and $|q|$ as follows: we iterate over all pairs of positive integers (a, b) in a systematic way until a/b equals $|p|/|q|$. This enumeration of all pairs is possible, because **Q** is countable. For example, the following expression first generates all pairs (a, b) with $a + b = 1$, then with $a + b = 2$, $a + b = 3$, ... The loop is guaranteed to stop, as the pair $(|p|, |q|)$ will be generated at some point in time.

$\mathcal{P} := MapNr_{D\rightarrow I}(Zero)$;
$\mathcal{Q} := MapNr_{D\rightarrow I}(One)$;
$\mathcal{S}um := MapNr_{D\rightarrow I}(One)$;
While($\mathcal{P}/\mathcal{Q} \neq abs(D)$) **loop**
  $\mathcal{S}um := \mathcal{S}um + 1$;
  $\mathcal{P} := MapNr_{D\rightarrow I}(Zero)$;
  While($\mathcal{P}/\mathcal{Q} \neq abs(D)$ and
          $\mathcal{P} \leq \mathcal{S}um - 1$) **loop**
    $\mathcal{Q} := \mathcal{S}um - \mathcal{P}$;
    $\mathcal{P} := \mathcal{P} + 1$;
  End loop
End loop
Return $\mathcal{P}, \mathcal{Q}$;

A.1.4 *Binary Representation.* We now give an expression that maps a positive integer $\mathcal{N}r[n]$ to its binary representation in the relation $\mathcal{B}in(I, B)$. $\mathcal{B}in$ consists of pairs $(\{Index = i\}, \{Bit = b\})$, where b is the ith bit in the binary representation of n.

We can easily find the index of most significant bit of the binary representation of the number n; this index is the largest number i such that 2^i is smaller than or equal to n. Let msb be the following expression, returning the most significant bit of n as the relation $\{(\{Index = i\})\}$.

$\mathcal{I} := MapNr_{D\to I} Zero$;
$\mathcal{P} := MapNr_{D\to I} One$;
While($\mathcal{P} \leq n$) **loop**
  $\mathcal{I} := \mathcal{I} + 1$;
  $\mathcal{P} := \mathcal{P} * 2$;
End loop
Return $\kappa_{Index=val(Nr)\ as\ I}\ MapNr_{I\to D}(\mathcal{I} - 1)$;

We can now get all 1-bits by iteratively applying msb, and subtracting the most significant bit. Hence, we get the following expression:

$\mathcal{N} := n$;
While($\mathcal{N} \neq 0$) **loop**
  $\mathcal{B}in := \mathcal{B}in \cup (msb(\mathcal{N}) \times (\kappa_{Bit=1} R))$;
  $\mathcal{N} := \mathcal{N} - 2^{msb(\mathcal{N})}$;
End loop
Return $\mathcal{B}in$;

We still need to add the 0-bits. This is not too hard; we can just initiate a for-loop over i from 0 to $msb(n)$, and if $(\{Index = i\})$ is not in $\pi_{Index}\ \mathcal{B}in$, we add $(\{Index = i\}, \{Bit = 0\})$ to $\mathcal{B}in$.

For the other direction, we can go through the binary number from the highest index to lowest, and use the following computation method:

$$n \ := \ b_0 + 2 \cdot (b_1 + 2 \cdot (b_2 + \ldots))$$

In this sum, b_i represents the ith bit. Hence, we iteratively pick the largest index i left, add it to twice the result so far, and remove bit i from $\mathcal{B}in$. This procedure stops when $\mathcal{B}in$ becomes empty.

A.2 Computational completeness on Integers

Let Q be a computable function from integers to integers. Since we can loop in the I-world, represent integers and apply arithmetic operations on them, we can express the query Q in the following way: there exists an $\mathcal{MA}$ expression $expr$ such that on input $\mathcal{N}r[n]$, $expr$ returns $\mathcal{N}r[Q(n)]$.

In the following we will show how we can uplift this result to computational completeness for generic and computable D→D queries. Therefore, we show that we can encode every D-world relation as an integer, and decode integers back to D-world relations. Since $\mathcal{MA}$ is computationally complete for integer to integer queries, for any computable query q from D to D, the function that maps the encoding of a D-world to the encoding of the result of q applied to that D-world is expressible in $\mathcal{MA}$. Let Q denote this expression. The expression for q will then be the encoding, followed by Q, followed by the decoding.

A.3 Computational Completeness in General

The proof of computational completeness in general will proceed as follows. Suppose that we want to construct an expression for the generic, computable D$\to$D query q.

(1) First we show how to handle nesting and multiple relations. We reduce the general problem to expressing queries from a single flat relation R to a single flat relation $q(R)$.

(2) Secondly we map such a relation to a relation holding only integers. That is, elements of $\mathbf{Q}$ are split into a triplet of positive integers, and elements of $\mathcal{U}$ are mapped to integers. Due to genericity, however, it is not possible to give a unique mapping from $\mathcal{U}$ to integers. Therefore, we will consider many mappings at the same time.

(3) For each mapping m, we can transform the unique input relation R into a relation with only integers. This table with only integers can be encoded as a single number $e_m(R)$. Similarly, we can decode a number n to a single relation using e_m^{-1}.

(4) For every mapping m we consider the following function from integers to integers: $Q_m = e_m \circ q \circ (e_m)^{-1}$. We will show that this function Q_m does not depend on m. We hence denote this function by Q. Since q is computable, Q will be computable as well.

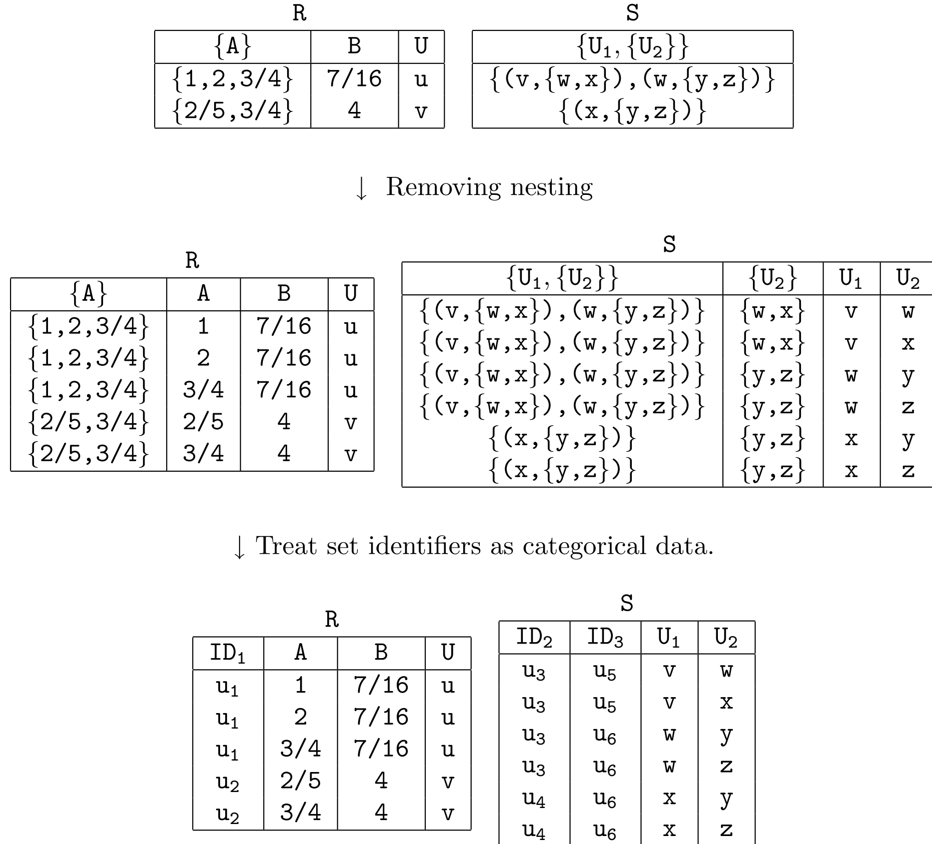
(5) For every mapping m, we will, simultaneously, compute $e_m^{-1}(Q(e_m(R)))$. By definition of Q_m, for every mapping m, the result of $e_m^{-1}(Q(e_m(R)))$ will be $q(R)$. Therefore, for all mappings the same result $q(R)$ will be obtained. Finally, we get $q(R)$ by projecting away the mappings and collapsing all $(m, q(R))$ into $q(R)$.

Simplifying the Problem We will use examples to illustrate the complex constructions used in the encoding. In Figure 7 an example D-world database is given. First we remove nesting as follows: a nested attribute will be treated as if its values are categorical data that play the role of set identifiers. In Figure 7 this principle is illustrated; a nested attribute $\{A\}$ is duplicated, one copy is kept as the set identifier, while the other copy will be unnested. This process is repeated until no nested attributes are left, except for those that are used as set identifiers. The following expression does the unnesting for an attribute $\{A\}$ in a relation R:

$$Unnest_{\{A\}}\sigma_{\{A\}=ID}((\pi_{\{A\}\ as\ ID}R) \times R)$$

In fact, this reduction does not change the nature of the input data; the set identifiers are still nested attributes. For the construction of the encoding, however, the reduction has the important advantage that we can treat these set identifiers as if they are elements of the unordered domain $\mathcal{U}$, without loosing information on the contents of the nested attributes. Therefore, in the construction of the encoding we can assume, without loss of generality, that there are no nested attributes, only attributes with domain $\mathcal{U}$ and $\mathbf{Q}$.

We further reduce the problem of encoding many relations to one relation, by making the Cartesian product of all relations in the input. Hence, in the following



R

{A}	B	U
{1,2,3/4}	7/16	u
{2/5,3/4}	4	v

S

$\{U_1, \{U_2\}\}$
{(v,{w,x}),(w,{y,z})}
{(x,{y,z})}

↓ Removing nesting

R

{A}	A	B	U
{1,2,3/4}	1	7/16	u
{1,2,3/4}	2	7/16	u
{1,2,3/4}	3/4	7/16	u
{2/5,3/4}	2/5	4	v
{2/5,3/4}	3/4	4	v

S

$\{U_1, \{U_2\}\}$	$\{U_2\}$	U_1	U_2
{(v,{w,x}),(w,{y,z})}	{w,x}	v	w
{(v,{w,x}),(w,{y,z})}	{w,x}	v	x
{(v,{w,x}),(w,{y,z})}	{y,z}	w	y
{(v,{w,x}),(w,{y,z})}	{y,z}	w	z
{(x,{y,z})}	{y,z}	x	y
{(x,{y,z})}	{y,z}	x	z

↓ Treat set identifiers as categorical data.

R

ID_1	A	B	U
u_1	1	7/16	u
u_1	2	7/16	u
u_1	3/4	7/16	u
u_2	2/5	4	v
u_2	3/4	4	v

S

ID_2	ID_3	U_1	U_2
u_3	u_5	v	w
u_3	u_5	v	x
u_3	u_6	w	y
u_3	u_6	w	z
u_4	u_6	x	y
u_4	u_6	x	z

Fig. 7. Running Example to illustrate the construction of the encoding. $dom(\mathtt{U_i}) = \mathcal{U}$, $dom(\mathtt{A}) = dom(\mathtt{B}) = \mathbf{Q}$

we will show how to encode a single relation R, over attributes with domain either $\mathcal{U}$, or $\mathbf{Q}$.

Mapping R to an integer $e_m(R)$ For the moment being, we assume that we are working with a fixed mapping. For this mapping, we show how the encoding e_m is constructed. Later on we will show how to construct all possible mappings, and how to deal with all mappings simultaneously.

Suppose that we have a mapping m that maps every element of $\mathcal{U}$ to a positive integer. Then we can transform the relation R into a relation containing only natural numbers. Every element u of $\mathcal{U}$ is replaced by $m(u)$, and every rational number p/q is mapped to a triplet $(s, |p|, |q|)$, with $s = 0$ if p/q is strictly negative, and $s = 1$ otherwise. We fix an order on the attributes of R. Since all attribute values are numeric, and hence ordered, we can order the tuples in an unambiguous way. We will order the tuples in ascending order. The encoding is now as follows. First we transform our relation into one large string of the symbols "0", "1", ",", "(", and ")". The attribute values are written down in binary, and separated by

",". Tuples are enclosed in brackets, and separated by ,. Because the attributes and the tuples are ordered, this string is unique. Furthermore, we encode the symbols "0", "1", ",", "(", and ")" using 3 bits. In this way we get a string of "0" and "1". We add a leading "1" in order not to lose leading "0"'s. The resulting string of "0" and "1" is the number encoding our relation in binary.

EXAMPLE 12. *Let R be the following relation:*

A	B
u	$\frac{1}{5}$
v	$-\frac{3}{4}$

Let m be the mapping $\{u \mapsto 0, v \mapsto 1\}$. First R is transformed into the following relation consisting of only natural numbers:

A	B_s	B_d	B_n
0	1	1	5
1	0	3	4

We fix the order of the attributes as follows: $\mathtt{A} < \mathtt{B_s} < \mathtt{B_d} < \mathtt{B_n}$. The resulting string is the following:

$$(0, 1, 1, 101), (1, 0, 11, 100)$$

We encode 0 *as* 000, 1 *as* 001, (*as* 010,) *as* 011, *and* , *as* 100. *With the leading* 1, *and this encoding, we get the following number in binary encoding the database:*

```
1 010 000 100 001 100 001 100 001 000 001
  011 100 010 001 100 000 100 001 001 100
  001 000 000 011
```

Thus, the integer encoding the above relation under mapping m is

5941608283405117866499

Representing the mapping in the I-world Let m be a mapping from the active domain of the attributes with domain $\mathcal{U}$ to the positive integers. We will store this mapping in the I-world as a tuple in the relation $\mathcal{M}(Map)$. The tuple representing m will be the following:

$$(\{\{U, Nr\} = \{(u, m(u))\} \mid u \in adom_{\mathcal{U}}\}) \ .$$

EXAMPLE 13. *Let m be the following mapping:*

$$\left\{ \begin{array}{ll} \mathtt{u} \mapsto 1 & \mathtt{v} \mapsto 2 \\ \mathtt{w} \mapsto 3 & \mathtt{x} \mapsto 4 \\ \mathtt{y} \mapsto 5 & \mathtt{z} \mapsto 6 \end{array} \right\}$$

This mapping is represented by the following tuple in the I-world:

$\mathcal{M}$

Map
$\left\{ \begin{array}{l} \{U,Nr\} = \{(u,1)\},\ \{U,Nr\} = \{(v,2)\}, \\ \{U,Nr\} = \{(w,3)\},\ \{U,Nr\} = \{(x,4)\}, \\ \{U,Nr\} = \{(y,5)\},\ \{U,Nr\} = \{(z,6)\} \end{array} \right\}$

We have chosen for this rather complicated representation of the mapping, because it is not allowed to nest groupings in the I-world. Hence, it is impossible to first group the pairs $\{u, m(u)\}$ and then group the pairs that are in the mapping. Therefore, we first use nesting in the D-world to form the pairs $\{(u, m(u))\}$ and then use the regionize operator κ to export these pairs to the I-world. There the different mappings are constructed using the powerset construction presented before.

Construction of the Mappings We will construct all possible mappings m *at the same time*. For every mapping m, we encode the relation, resulting in the code $e_m(R)$. First we give an expression that returns the active domain of the unordered elements. That is, we construct an expression that returns all unordered elements that are used in the input. This can be done by simply unnesting nested attributes, projecting on the attributes with unordered domains, and then taking the union. For example, suppose that the input consists of two relations $R(\{U_1\}, A)$ and $S(U_2)$, with U_1 and U_2 the only attributes with domain $\mathcal{U}$, then the active domain query is the following:

$$adom_{\mathcal{U}} \;=\; (\pi_{U_1 \ as \ U}\, Unnest_{\{U_1\}} R) \cup \pi_{S_2 \ as \ U} S$$

We will encode the elements of $\mathcal{U}$ using integers 1 to n, with $n = |adom_{\mathcal{U}}|$. We can easily construct an expression returning the relation $C(Nr) = \{(1), \ldots, (n)\}$ using a similar technique as is used in the proof of Theorem 5.7. We then take the cross-product of $adom_{\mathcal{U}}$ and C to get all possible codes for all elements of $\mathcal{U}$. Subsequently, the pairs (u, k) are nested, and made into constraints in the I-world:

$$\kappa_{\{U,Nr\}=val(\{U,Nr\})} Nest_{\{U,Nr\}}(adom_{\mathcal{U}} \times C)$$

This construction is illustrated with an example in Figure 8.

In this way, we get in the I-world a relation consisting of all encodings of a single element of $\mathcal{U}$. Then we use the powerset-operator introduced in the beginning of this appendix to generate all subsets of codes. From these subsets we select those that represent valid mappings. A subset S of codes represents a valid mapping if:

—for every u in $adom_{\mathcal{U}}$ there is exactly one coding $\{U, Nr\} = \{(u, n)\}$ in S;
—for every u, v, the coding for u is different from the coding for v.

We can easily select those subsets that represent valid codings using the following helper relations:

$$\begin{aligned} &Code_U(Code, U) \;:= \\ &\quad \{(\{\{U, Nr\} = \{(u, n)\}\}, \{U = u\}) \\ &\qquad \mid u \in adom_{\mathcal{U}}, n = 1 \ldots |adom_{\mathcal{U}}|\} \end{aligned}$$

and

$$\begin{aligned} &Code_Nr(Code, Nr) \;:= \\ &\quad \{(\{\{U, Nr\} = \{(u, n)\}\}, \{Nr = n\}) \\ &\qquad \mid u \in adom_{\mathcal{U}}, n = 1 \ldots |adom_{\mathcal{U}}|\} \end{aligned}$$

These relations can be formed easily using the algebra. The connection between $\{\{U, Nr\} = (\{(u, n)\})\}$ and the correct $\{U = u\}$ can be established with a populate operation.

Applying all mappings simultaneously First we transform the whole relation in the D-world to the I-world via regionize operations. For example, let

R

A	B	C	D
u	y	7/16	0
v	w	7/16	1
w	z	7/16	2
x	w	4	3
x	z	4	4

The active domain query for $\mathcal{U}$ in this example is the following:

$$adom_{\mathcal{U}} \;=\; \pi_{A\ as\ U} R \cup \pi_{B\ as\ U} R$$

The resulting relation is:

$$\{(\mathtt{u}), (\mathtt{v}), (\mathtt{w}), (\mathtt{x}), (\mathtt{y}), (\mathtt{z})\}$$

The relation C in this example is thus:

$$\{(\mathtt{1}), \ldots, (\mathtt{6})\},$$

and the set Enc, and the result of the regionize:

$\{\mathtt{U}, \mathtt{Nr}\}$		Map
$\{(\mathtt{u}, \mathtt{1})\}$		$\{\{\mathtt{U}, \mathtt{Nr}\} = \{(\mathtt{u}, \mathtt{1})\}\}$
...		...
$\{(\mathtt{u}, \mathtt{6})\}$		$\{\{\mathtt{U}, \mathtt{Nr}\} = \{(\mathtt{u}, \mathtt{6})\}\}$
$\{(\mathtt{v}, \mathtt{1})\}$		$\{\{\mathtt{U}, \mathtt{Nr}\} = \{(\mathtt{v}, \mathtt{1})\}\}$
...	$\rightarrow$	...
$\{(\mathtt{v}, \mathtt{6})\}$		$\{\{\mathtt{U}, \mathtt{Nr}\} = \{(\mathtt{v}, \mathtt{6})\}\}$
...		...
$\{(\mathtt{z}, \mathtt{1})\}$		$\{\{\mathtt{U}, \mathtt{Nr}\} = \{(\mathtt{z}, \mathtt{1})\}\}$
...		...
$\{(\mathtt{z}, \mathtt{6})\}$		$\{\{\mathtt{U}, \mathtt{Nr}\} = \{(\mathtt{z}, \mathtt{6})\}\}$

Fig. 8. Running Example to illustrate the construction of the encoding. $dom(\mathtt{A}) = dom(\mathtt{B}) = \mathcal{U}$, $dom(\mathtt{C}) = dom(\mathtt{D}) = \mathbf{Q}$

$R(A_1, \ldots, A_n)$ be the relation in the D-world. We transport it to the I-world as follows: first we do n regionize operations, for each attribute separately. Then we combine them with Cartesian products, and reconstruct the tuples by a populate operation:

$$\pi_{\mathcal{RDA}} Pop(\kappa_{A_1 = val(A_1)} R \times \ldots \times \kappa_{A_n = val(A_n)} R, R)$$

Then, for each mapping we replace the values with the corresponding integers; that is, rational numbers are split using the auxiliary expressions, and unordered elements are mapped using the mapping. The result is a relation in the I-world over the schema $\{Map, C_1, \ldots, C_k\}$. With the looping and arithmetic capabilities in the I-world it is now easy to compute the number for each mapping; always select the smallest tuple per mapping, and translate it into a binary number with the auxiliary expressions given in the beginning of the appendix. The binary numbers can easily be concatenated and translated using the codes for "0", "1", ",", "(", and ")".

Computing the Number for Each Mapping Let now Q_m be the following computable function from integers to integers:

$$Q_m(e_m(R)) \;:=\; e_m(q(R)) \;\;,$$

and on all numbers that do not represent a valid encoding, Q_m is the identity.

Let m_1 and m_2 be two mappings. Let Q_1 and Q_2 be the two functions associated with m_1 and m_2, and e_1 and e_2 be the two encodings. By definition,

$$\begin{aligned} q &= (e_1)^{-1} \circ Q_1 \circ e_1 \\ &= (e_2)^{-1} \circ Q_2 \circ e_2 \end{aligned}$$

The following function expresses a permutation on $\mathcal{U}$:

$$e_1^{-1} \circ e_2$$

Therefore,

$$(e_1^{-1} \circ e_2) \circ q = q \circ (e_1^{-1} \circ e_2)$$

and hence,

$$\begin{aligned} q &= (e_1^{-1} \circ e_2) \circ q \circ (e_1^{-1} \circ e_2)^{-1} \\ &= (e_1^{-1} \circ e_2) \circ ((e_2)^{-1} \circ Q_2 \circ e_2) \circ (e_1^{-1} \circ e_2)^{-1} \\ &= e_1^{-1} \circ Q_2 \circ e_1 \end{aligned}$$

Thus,

$$(e_1)^{-1} \circ Q_1 \circ e_1 = e_1^{-1} \circ Q_2 \circ e_1$$

and therefore,

$$Q_1 = Q_2$$

Hence, due to genericity, the function Q_m is the same for all mappings m. We denote this function Q. It is clear that Q is computable, because q is computable. Therefore, there exists an I→I expression that computes Q.

For every mapping m we will do the following: first we encode the D-world as a single number using e_m. Then, we evaluate Q on this number. Finally, we apply the inverse relation $(e_m)^{-1}$ to the result of Q. Because Q_m equals Q, and Q_m was defined in such a way that the result of the sequence $m^{-1} \circ Q_m \circ m$ equals q, for every mapping m the same output relation is computed. Therefore, we can just project away the mapping to get the final result.