arXiv:1202.0474v2 [cs.DB] 5 Feb 2012

Relational Semantics for Databases and Predicate Calculus*

Philip Kelly† M.H. van Emden‡

Abstract

The relational data model requires a theory of relations in which tuples are not only many-sorted, but can also have indexes that are not necessarily numerical. In this paper we develop such a theory and define operations on relations that are adequate for database use. The operations are similar to those of Codd's relational algebra, but differ in being based on a mathematically adequate theory of relations. The semantics of predicate calculus, being oriented toward the concept of satisfiability, is not suitable for relational databases. We develop an alternative semantics that assigns relations as meaning to formulas with free variables. This semantics makes the classical predicate calculus suitable as a query language for relational databases.

1 Introduction

Relational databases started with Codd's idea of queries in a declarative language to be translated to operations on data that were justified by a machine-independent semantics. Such a utopian vision was actually not unrealistic: predicate calculus is a declarative language that appeals to a user's intuition and this language has a machine-independent semantics in terms of an implementable data-structure, namely, relations.

Somehow this promising beginning led to SQL. There were probably several things that went wrong along the way. In this paper we address one of these. In analyzing what went wrong one should first acknowledge the fundamental rightness of Codd's idea. Next one should examine whether Codd had a sufficient understanding of what mathematics has to say about relations and about algebra. In this paper we address ourselves to the first part: relations.

What Codd found in mathematics was that an n-ary relation is a subset of a Cartesian product $S_0 \times \cdots \times S_{n-1}$. It may well be true that the required more general version is nowhere to be found in mathematics. If this is the case, then it is because *for the purposes of mathematicians*, nothing more is needed. We need to realize that the purposes of database theory are not necessarily those of mathematicians. This does not mean that, as the descent into SQL might suggest, mathematics itself is inadequate, only that within mathematics developments are needed that have been overlooked so far. What needs to be done in database theory is to use *mathematical methods* to define relations in such a way as to suit the needs of our theory, irrespective of whether such a definition has been sanctioned by prior use in mathematics. This is the purpose of the present paper.

According to the relational database model, data are organized by means of relations. This is a promising point of departure, provided relations are suitably defined, as just discussed. Another doctrine of the relational database model is that the querying of a database should be based on an algebra in the mathematical sense. As far as the present paper is concerned, this is a premature commitment. Once a suitable notion of relation is defined, it is natural to enquire what *operations* on relations can yield the relations that are answers to queries. As we show in this paper, such operations can be defined set-theoretically, without any algebra in the mathematical sense and without commitment to a language in which to express queries.

Although the operations we define are not designed to constitute an algebra in the mathematical sense, they do fit together in a natural way. This is proved by the fact they allow a natural translation, in the

*Report number DCS-343-IR, Department of Computer Science, University of Victoria, Canada.
†Department of Computer Science, University of Victoria, Canada.
‡Department of Computer Science, University of Victoria, Canada.

sense of compositional semantics, from predicate calculus queries and definitions to relational expressions. Our operations include projection and join, but defined in a new way to fit our new definition of relation. They also include a new operation, which we call *filtering*. This natural family needs a name; we propose Elementary Theory of Relations (ETR). The "elementary" is needed because of its elementary nature and to distinguish it from such non-elementary mathematics as found in works such as [7].

Overview of this paper Relations are made up of tuples, and tuples are, in their full generality, *functions*. Thus to get relations right, we need to get tuples right. To get tuples right, we do as much as possible with functions. This gets us all the way up to Cartesian products. The needed review of terminology and notation for functions is in Section 2. In Section 3 we review Codd's original formulation of the relational data model. The deficiencies noted here motivate our definition of relations given in Section 4. This serves as basis for what we consider the rational reconstruction of the relational data model presented in Section 5. The operations for querying are worked out in Sections 6 and 7. In Section 8 we show how the operations take part in the semantics of predicate calculus. In this way predicate calculus becomes, once again, a strong candidate for a database query language. In Section **??** we outline what needs to be done.

2 Set-theoretical terminology

Loosely speaking, a relation is a set of tuples. Of course the tuples in such a set have to have something common if they are to constitute a valid relation. What precisely they have to have in common comes down to the precise notion of "tuple". To get this right, we use the definition of Halmos [12], which views a tuple ("family" in his terminology) as a function. In this section we show which of the various definitions and notations for functions and tuples we use.

$\mathcal{N}$ is the set of natural numbers; $\iota(n)$ is the set $\{0, \ldots, n-1\}$ of the first n natural numbers. We write $\subset$ for the subset relation between sets. We write $\langle a, b\rangle$ for the ordered pair with elements a and b.

Definition 1 (function) *A* function *is a triple consisting of a set that is its* source*, a non-empty set that is its* target*, and a* mapping *that associates with every element of the source an element of the target*[1]*. If $s \in S$ is associated by the map with $t \in T$, then we can write $s \mapsto t$.*

The set of all functions with source S and target T is denoted $S \to T$. This set is often referred to as the type *of the functions belonging to it. Thus we write $f \in S \to T$ rather than the usual $f : S \to T$.*

Let f be a function in $S \to T$ and let S' be a subset of S. $f(S')$ is defined to be $\{f(x) \mid x \in S'\}$. In particular, $f(S)$ is the set of elements of T that are a value of f.

Example 1 *Let $c \in \{\clubsuit, \diamondsuit, \heartsuit, \spadesuit\} \to \{black, white, red\}$ be a function with a mapping such that $\clubsuit \mapsto$ black, $\diamondsuit \mapsto$ red, $\heartsuit \mapsto$ red, and $\spadesuit \mapsto$ black. One may write the mapping of c more compactly as*

$$c = \begin{array}{c|c|c|c} \spadesuit & \diamondsuit & \heartsuit & \clubsuit \\ \hline black & red & red & black \end{array},$$

where the order of the columns is immaterial: only the pairing of the source and target elements matters.

Example 2 *For finite S and non-empty finite T we have that*

$$|S \to T| = |T|^{|S|} \tag{1}$$

where $|X|$ is the number of elements in finite set X. Note that $|\{\} \to T| = 1$, as there is one function of type $\{\} \to T$ for any non-empty finite T. Equation (1) can be extended to cardinal arithmetic for infinite S or T.

Definition 2 (restriction) *Let f be a function in $S \to T$ and let S' be a set. $f \downarrow S'$ is* the restriction of f to S'*. It has $S \cap S'$ as source, T as target, and its mapping is $x \mapsto f(x)$.*

Definition 3 (insertion) *If $S' \subset S$ then $i \in S' \to S$ is* the insertion of S' with respect to S *if it has the map $x \mapsto x$.*

[1] Most authors use "domain" for "source" and "co-domain" for "target". We avoid "domain" because of its other uses in computer science.

Definition 4 (function sum) *Functions $f_0 \in S_0 \to T_0$ and $f_1 \in S_1 \to T_1$ are* summable *iff for all $x \in S_0 \cap S_1$, if any, we have $f_0(x) = f_1(x)$. If this is the case, then $f_0 + f_1$, the* sum of f_0 and f_1, *is the function in $(S_0 \cup S_1) \to (T_0 \cup T_1)$ with map $x \mapsto f_0(x)$ if $x \in S_0$ and $x \mapsto f_1(x)$ if $x \in S_1$.*

Example 3 *Let $f_0 \in \{a, b\} \to \{0, 1\}$ such that $a \mapsto 0$ and $b \mapsto 1$. Let $f_1 \in \{b, c\} \to \{0, 1\}$ such that $b \mapsto 1$ and $c \mapsto 0$. Then f_0 and f_1 are summable and $(f_0 + f_1) \in \{a, b, c\} \to \{0, 1\}$ such that $a \mapsto 0$, $b \mapsto 1$, and $c \mapsto 0$.*

Definition 5 (function composition) *Let $f \in S \to T$ and $g \in T' \to U$ with $T' \subset T$. The* composition *$g \circ f$ of f and g is the function in $S \to U$ that has as map $x \mapsto g(f(x))$.*

For examples, see Figure 1.

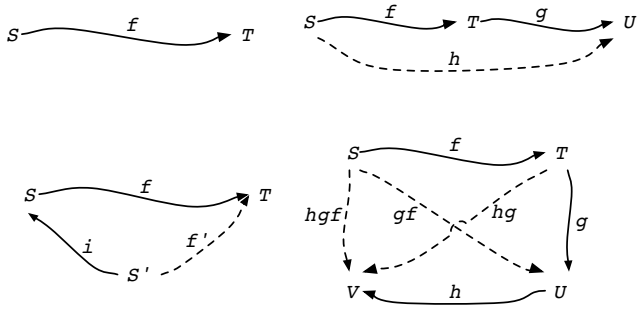


Figure 1: Top left: $f \in S \to T$. Top right: $h = g \circ f$. Bottom left: $S' \subset S$ and $i \in S' \to S$ is the insertion function of S' with respect to S. The diagram asserts $f' = f \circ i$. Bottom right: $gf = g \circ f$, $hg = h \circ g$, and $hgf = h \circ (g \circ f) = (h \circ g) \circ f$.

If T is a set of disjoint sets, then there is, for each element x of $\cup T$, a unique element of T to which x belongs. This observation suggests the following definition.

Definition 6 (sorting function) *Let T be a set of disjoint sets. The* sorting function *σ_T of T is the function in $\cup T \to T$ that maps each element x of $\cup T$ to the unique set in T to which x belongs.*

The subscript in σ_T is often dropped, hopefully only when the corresponding set of disjoint sets is clear from the context.

Definition 7 *If $f \in S \to T$ is such that $x \neq y$ implies $f(x) \neq f(y)$, then f is said to be* injective. *If $f(S) = T$, then f is said to be* surjective. *If f is injective and surjective, then it is said to be* bijective.

Lemma 1 *If $f \in S \to T$ is bijective, then a function $g \in T \to S$ exists with map $t \mapsto s$ iff $t = f(s)$. g is bijective and is called the* inverse *f^{-1} of f.*

2.1 Tuples

A tuple is a function. The only thing that is special about tuples is their terminology. Let $t \in I \to T$ be a tuple (i.e. a function) that maps every $i \in I$ to $t(i) \in T$. The source is called "index set". $t(i)$ is called the tuple's *component indexed* by i; $t(i)$ is often written as t_i.

Definition 8 (empty tuple) *A tuple of type $I \to T$ is empty when I is empty.*

For every non-empty T there is a unique empty tuple of type $\{\} \to T$.

Definition 9 (subtuple, sequence) *If t is a tuple with index set I and if I' is a subset of I, then $t \downarrow I'$ is the* subtuple *of t defined by I'. If $I = \iota(n)$ for some natural number n, then t is called a* sequence *and $t \downarrow I'$ is a* sub-sequence *of t.*

The indexes of a tuple need not be numbers. Even if they are, they need not be contiguous numbers. Even if they are, the least of them need not be 0. But if the index set of a tuple t is $\iota(n)$ for some natural number n, then we have the benefit of a concise notation: $t = [t_0, \ldots, t_{n-1}]$[2].

Consider tuples in $\iota(3) \to \iota(3)$. Then $[2,0,1]$ is a bijection, so that $[2,0,1]^{-1}$ exists. In fact, we have $[2,0,1]^{-1} = [1,2,0]$.

Example 4 *$t = [d,b,a,c]$ implies that the source of t is $\iota(4)$, and that $t_0 = d$, $t_1 = b$, $t_2 = a$, and $t_3 = c$. We can say the following about $t' = t \downarrow \{2,3\}$: $t'_2 = a$, $t'_3 = c$, while t'_0 and t'_1 are not defined. Note that t', though a subtuple, is not a sequence.*

We often consider tuples of which the elements have different types, that is, belong to different sets. The typing of such tuples can be succinctly characterized as follows.

Definition 10 (signature, sorting of tuples) *Let I be a set of indexes, T a set of disjoint sets, τ a tuple in $I \to T$, and t a tuple in $I \to \cup T$. We say that t is* sorted by *(or* has signature*) τ iff $\tau = \sigma \circ t$, where σ is the sorting function of T.*

That is, t is sorted by τ if we have $t(i) \in \tau(i)$ for all $i \in I$: the type of each element of t is determined by τ. See Figure 2.

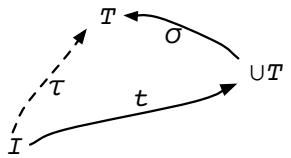


Figure 2: $\tau \in I \to T$ is the signature of $t \in I \to \cup T$ because $\tau = \sigma \circ t$.

Let $\mathcal{B} = \{\mathtt{t}, \mathtt{f}\}$. Let $T = \{\mathcal{B}, \mathcal{N}\}$ and $\tau = [\mathcal{B}, \mathcal{N}, \mathcal{N}]$. Then $[\mathtt{t}, 4, 5]$ has signature τ, while $[\mathtt{t}, \mathtt{t}, 5]$ does not.

2.2 Pattern matching

We want to express mathematically a certain way in which one tuple "matches" another, like $[a,a,c]$ matching $[x,x,z]$ because we can transform $[x,x,z]$ to $[a,a,c]$ by substituting a for x and c for z in $[x,x,z]$.

For example $[a,a,a]$ matches $[x,x,z]$, but not the other way around. While $[a,a,c]$ does match $[x,x,z]$, $[a,b,c]$ does not. Also, both $[a,a,c]$ and $[a,b,c]$ match $[x,y,z]$. It will be useful for later to note that $[x,y,z]$ matches *every* triple in $[0,1,2] \to \{a,b,c\}$.

We can make these intuitions precise in the following way.

Definition 11 (compatible, matching, pattern, substitution) *Let T be a set of disjoint sets. Let t be a tuple typed by $\tau \in I \to T$ and let $p \in I \to X$ be a tuple typed by $\varphi \in X \to T$. The typing of p is* compatible with *τ in the sense that $\tau = \varphi \circ p$. We define a binary relation between t and p by the condition that there exists an $s \in X \to \cup T$ such that $t = s \circ p$. In this context we say that t* matches *pattern p with matching substitution s.*

Let us see how this works out for one of the examples just mentioned. We have $I = \{0,1,2\}$, $T = \{\{a,b,c\}\}$, $X = \{x,y,z\}$. Thus $[a,a,c] \in I \to \cup T$ matches $[x,x,z] \in I \to X$ because there exists $s \in X \to \cup T$ such that $[a,a,c] = s \circ [x,x,z]$. Several such values for s exist: $s = \begin{array}{c|c|c} x & y & z \\ \hline a & a & c \end{array}$, $s = \begin{array}{c|c|c} x & y & z \\ \hline a & b & c \end{array}$, and $s = \begin{array}{c|c|c} x & y & z \\ \hline a & c & c \end{array}$. But no $s \in X \to \cup T$ exists such that $[a,b,c] = s \circ [x,x,z]$. So $[a,b,c]$ does not match pattern $[x,x,z]$.

[2] We reserve the angle brackets $\langle$ and $\rangle$ for pairs that are not necessarily related to sequences with $\iota(2)$ as index set.

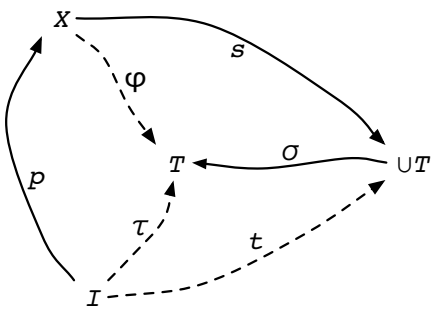


Figure 3: To Definition 11.

In Figure 3 we have $t = s \circ p$: t matches p with substitution s. Are there patterns p that match every tuple t? In fact, if p is a bijection, then p (which is a function) has an inverse p^{-1} and we have $t \circ p^{-1} = s \circ p \circ p^{-1} = s$. This gives the matching substitution with which an arbitrary t matches p.

2.3 Cartesian products

In mathematics a Cartesian product is often a set of tuples $[x_0, \ldots, x_{n-1}]$ where the elements of the tuples belong to a set D. Such a Cartesian product is denoted D^n. Expressed more formally, D^n is the set of tuples that have in common the signature $\tau \in I \to T$, where $I = \iota(n)$ and $T = \{D\}$. In mathematics one sometimes generalizes the notion of Cartesian product to $D_0 \times \cdots \times D_{n-1}$, which is the set of tuples with the common signature $\tau \in I \to T$, where $I = \iota(n)$ and $T = \{D_0, \ldots, D_{n-1}\}$ and $i \mapsto D_i$ for all $i \in \iota(n)$.

The two examples suggest a further generalization: to allow the index set I to be an arbitrary set. This generalization is rarely, if ever, needed in mathematics, but is useful in data management[3].

Definition 12 (Cartesian product) *Let I be a set of indexes, T a set of disjoint sets, and τ a tuple in $I \to T$. The* Cartesian product on τ, *denoted* $\textsc{cart}(\tau)$, *is the set of all tuples that have signature τ.*

3 The relational model according to Codd

We have now all the ingredients for an adequate definition of "relation". Before proceeding to such a definition we review in this section Codd's original one, from section 1.3 ("A Relational View of Data") of [3].

> *The term* relation *is used here in its accepted mathematical sense. Given sets $S_1, \ldots, S_n$ (not necessarily distinct), R is a relation on these n sets if it is a set of n-tuples each of which has its first element from S_1, its second element from S_2, and so on. We shall refer to S_j as the jth* domain *of R.*

In other words, mathematically speaking, a relation is a subset of a Cartesian product of the form $S_1 \times \cdots \times S_n$. Note that S_j, not j, is the domain. Thus S_i and S_j may be the same set, even though $i \neq j$. The above quote continues with:

> *For expository reasons, we shall frequently make use of an array representation of relations, but it must be remembered that this particular representation is not an essential part of the relational view being expounded. An array which represents an n-ary relation R has the following properties:*
>
> *1. Each row represents an n-tuple of R.*

[3] Often referred to as "data modeling", which is suspect: what one models is the *world*; data is what the model is made of.

2. *The ordering of rows is immaterial.*
3. *All rows are distinct.*
4. *The ordering of columns is significant — it corresponds to the ordering $S_1, \ldots, S_n$ of the domains on which R is defined.*
5. *The significance of each column is partially conveyed by labeling it with the name of the corresponding domain.*

Codd gives as example of such an array the one shown in Figure 4. He observes that this example does not illustrate why the order of the columns matters. For that he introduces the one in Figure 5.

supply	(supplier	part	project	quantity)
	1	2	5	17
	...	...	...	...

Figure 4: Codd's first example: domains all different.

component	(part	part	quantity)
	1	5	9
	...	...	...

Figure 5: Codd's second example: domains not all different.

He explains Figure 5 as follows.

> ... two columns may have identical headings (indicating identical domains) but possess distinct meanings with respect to the relation.

We can take it that in Figure 5 we have $n = 3$, $S_1 = S_2 =$ part, and $S_3 =$ quantity. As $S_1, \ldots, S_n$ need not all be different, columns can only identified by $\{1, \ldots, n\}$.

Codd goes on to point out that in practice n can be as large as thirty and that users of such a relation find it difficult to refer to each column by the correct choice among the integers $1, \ldots, 30$. According to Codd, the solution is as follows.

> *Accordingly, we propose that users deal, not with relations, which are domain-ordered, but with* relationships*, which are their domain-unordered counterparts. To accomplish this, domains must be uniquely identifiable at least within any given relation, without using position. Thus where there are two or more identical domains, we require in each case that the domain name be qualified by a distinctive* role name*, which serves to identify the role played by that domain in the given relation.*

Apparently

- "Role names" are non-numerical identifications of columns.
- A database is a collection of relations and relationships. The former in case all of $S_1, \ldots, S_n$ are different; the latter if not.
- Relations are defined mathematically as subsets of $S_1 \times \cdots \times S_n$. Relationships are something else. Their mathematical definition is not given.

However, later on page 380 of [3], we find

> *... it is proposed that most users should interact with a relational model of the data consisting of a collection of time-varying relationships (rather than relations).*

That is clear enough: no longer any use for the mathematically defined relations. But the next sentence is

Each user need not know more about any relationship than its name together with the names of its domains (role qualified whenever necessary).

Apparently role qualification is not always deemed necessary. We interpret this as the proposal to have data in the form of a relation when the degree is small and the domains are all different. If the degree is not small or if there are repeated domains, then the data are supposed to be in the form of a "relationship".

We consider such dual treatment of data unsatisfactory. Our proposal is to have all domains "role-qualified" so that all data are in the form of what Codd calls "relationships". Of course relationships are not subsets of $S_1 \times \cdots \times S_n$. Codd did not say what they were, mathematically. We propose to refer to "relationships" as "relations", to be defined mathematically in the next section.

4 A mathematical definition of relation

In modeling data, Codd wanted that "the term *relation* be used in its accepted mathematical sense". The problem with this is that this accepted sense arose out of typically mathematical concerns. It is not to be expected that this sense is adequate when one shifts the area of inquiry from pure mathematics to data management.

However, mathematics is ready to help out once one has decided on a suitable extra-mathematical application, and the relational representation of data is one such application. All that is needed is to drop the notion that the tuples constituting relations have to be indexed numerically. In fact, the indexes can be the "role names" of which Codd noticed that they would be useful when there are multiply occurring domains or when the tuples have many elements.

For this reason we have defined a Cartesian product to consist of tuples with index sets that are not necessarily numerical. This is the needed generalization; we can, as usual, define a relation as a subset of a Cartesian product, but then in the sense of Definition 12. As this Cartesian product may have any signature, that signature must be included in the definition of relation. Hence:

Definition 13 (relation) *A* relation *is a pair $\langle \tau, E \rangle$ where τ, the* signature *of the relation, is a tuple of type $I \to T$, where T is a set of disjoint sets, and E, the* extent *of the relation, is a subset of* $\textsc{cart}(\tau)$.

As all tuples in the extent of a relation have the same signature, the tabular notation of Example 1 is convenient: every tuple can be a row under the same headings. If the index set has fewer elements than the extent, then it may be preferable to transpose the table, as is done in the following example.

Example 5 *Consider the relation $\langle \tau, E \rangle$ where*

$$\tau \in \{divisor, dividend, quotient\} \to \{\iota(6)\}$$

and $E = \{t \mid t \text{ is typed by } \tau \text{ and } t_{dividend} = t_{divisor} \times t_{quotient}\}$, that is

$E =$

dividend	0	1	2	2	3	3	4	4	4	5	5
divisor	0	1	2	1	3	1	4	2	1	5	1
quotient	0	1	1	2	1	3	1	2	4	1	5

.

Example 6 *The difficulty with Codd's example in Figure 4 is that the column headings have to serve both as indexes and as domains. Let us adopt Codd's names for the domains. Let us introduce role names and collect these into the index set $I = \{sup, prt, pct, qty\}$.*

Then we define supply $= \langle \tau, E \rangle$ as relation suitable to represent Codd's data with $\tau \in I \to T$. Here $T = \{supplier, part, project, quantity\}$, where we assume that the elements of T are disjoint sets. The relation is now $\langle \tau, E \rangle$, where

$$\tau = \begin{array}{c|c|c|c} sup & prt & pct & qty \\ \hline supplier & part & project & quantity \end{array},$$

and

$$E = \begin{array}{c|c|c|c} sup & prt & pct & qty \\ \hline sup1 & prt2 & pct5 & 17 \\ \cdots & \cdots & \cdots & \cdots \end{array}.$$

Example 7 *The difficulty with Codd's example in Figure 5 is that the column headings cannot serve both as indexes and as domain names because of the repeated domain. Let us adopt Codd's names for the domains. Let us introduce role names and collect these into the index set* $I = \{prt, assembly, qty\}$.

Then we define component $= \langle \tau, E \rangle$ *as relation suitable to represent Codd's data with* $\tau \in I \to T$. *Here* $T = \{part, quantity\}$ *where we assume that the elements of* T *are disjoint sets. The relation is now* $\langle \tau, E \rangle$, *where*

$$\tau = \begin{array}{c|c|c} prt & assembly & qty \\ \hline part & part & quantity \end{array},$$

and

$$E = \begin{array}{c|c|c} prt & assembly & qty \\ \hline prt1 & assy5 & 9 \\ \dots & \dots & \dots \end{array}.$$

Example 8 *In a relation* $\langle \tau, E \rangle$ *we can have that* τ *is* $\{\} \to T$. *As there is only one tuple of type* τ *(the empty one) there are only two possibilities for* E *as subset of* $\{\} \to T$.

5 A reconstruction of the relational model according to set theory

Let us now see what the relational format for data would look like to someone who knows some set theory and who was only told the general idea of [3] without the details as worked out by Codd. We first look at how data are stored, then how they are queried.

5.1 Using base relations as repositories for data

Let us start with one way of characterizing a database, relational or not. The information in a database describes various aspects of things like parts of a gadget, books in a library, and so on. The things may be concrete objects with a certain degree of permanence. They may also be transient states of affairs, such as transactions, events, or employees in a company. We refer to these various things as *items*.

There is no limit to the information one can collect about an item as it exists in the world. Hence one performs an act of abstraction by deciding on a set of *attributes* that describe the item and one determines what is the *value* of each attribute for this particular item. A consequence of this abstraction is that it cannot distinguish between items for which all attributes have the same value. The set of attributes has to be comprehensive enough that such identification does not matter for the purpose of the database.

The foregoing is summarized in the first of the following points. The remaining points constitute a reconstruction of the main ideas of a relational database as suggested by the first point.

1. A database is a description of a world populated by *items*. For each item, the database lists the *values* of the applicable *attributes*.
2. The database presupposes a set of attributes, and for each attribute, a set of allowable values for this attribute. Such sets of admissible values are called *domains*. Let I be the set of attributes and let T be a set of domains, which we assume to be mutually disjoint. As each attribute has a uniquely determined domain, this information is expressed by a function, say τ, that is of type $I \to T$.
3. The description of each item is an association of a value with each attribute that is applicable to the item. If $I' \subset I$ is the set of attributes of the item, then I' is the index set of the tuple describing it. The signature of this tuple is $\tau \downarrow I'$.
4. Let $I_0, \dots, I_{n-1}$ be the index sets (not necessarily different) of the tuples occurring in the database. Then $\tau \downarrow I_0, \dots, \tau \downarrow I_{n-1}$ are the signatures occurring in the database. For all i, let E_i be the set of tuples that are in the database and have signature $\tau \downarrow I_i$. Then the database consists of the relations
$$\langle \tau \downarrow I_0, E_0 \rangle, \dots, \langle \tau \downarrow I_{n-1}, E_{n-1} \rangle.$$

5. The life cycle of a database includes a design phase followed by a usage phase. In the design phase I, T, and $\tau \in I \to T$ is determined, as well as the subsets $I_0, \ldots, I_{n-1}$ of I. This is the database *scheme*. In the usage phase the extents $E_0, \ldots, E_{n-1}$ are added and modified. With the extents added, we have a database *instance*.

5.2 Using computed relations as answers to queries

We may need rearrangements of the information in the base relations. This need can be met by using the relation resulting from evaluating an expression in terms of base relations and relational operations. We regard such an expression as a *query* and its value as the *answer* to the query.

6 Operations on relations

6.1 Boolean operations

Among relations of the same signature, certain operations are defined that mirror the boolean set operations. Of these, intersection is useful as an auxiliary to the definition of relational join; see Definition 18. We also list the cognates of some of the most common set operations.

Definition 14

$$\begin{array}{lrcl}
\textit{intersection:} & \langle \tau, E_1 \rangle \cap \langle \tau, E_2 \rangle & \stackrel{def}{=} & \langle \tau, E_1 \cap E_2 \rangle \\
\textit{union:} & \langle \tau, E_1 \rangle \cup \langle \tau, E_2 \rangle & \stackrel{def}{=} & \langle \tau, E_1 \cup E_2 \rangle \\
\textit{difference:} & \langle \tau, E_1 \rangle \setminus \langle \tau, E_2 \rangle & \stackrel{def}{=} & \langle \tau, E_1 \setminus E_2 \rangle \\
\textit{complement:} & \langle \tau, E \rangle^C & \stackrel{def}{=} & \langle \tau, \textsc{cart}(\tau) \setminus E \rangle
\end{array}$$

6.2 Projection

If every tuple of a relation is transformed in the same way into the subtuple (see Definition 9) defined by a subset of the index set, then the result is a relation.

Definition 15 (projection) *Let τ be in $I \to T$, where T is a set of disjoint sets and I is an index set. Let J be a subset of I. The* projection on J *of the relation $\langle \tau, E \rangle$ is written $\pi_J(\langle \tau, E \rangle)$ and is defined to be the relation $\langle \tau \downarrow J, \{t \downarrow J \mid t \in E\} \rangle$.*

6.3 Filtering

Suppose we have a relation $r = \langle \tau, E \rangle$ with E a set of tuples of signature τ where $\tau \in I \to T$ and T a disjoint set of sets. Consider now a pattern X that is typed compatibly with the tuples of r; that is, $p \in I \to X$ and $\varphi \in X \to T$ such that $\tau = \varphi \circ p$. Here X is the set containing the components of the patterns. We can think of them as variables or, better, as *indeterminates*. What matters more than their name is that they are what is substituted when a pattern p matches a tuple t from r. The condition for such matching is the one in Definition 11.

In the setting of Figure 3 one can consider the set of elements of E that match the pattern p. The corresponding substitutions are a set of tuples with the same signature, hence can be the extent of a suitably defined relation. This relation is the result of "filtering" r with p. This result is denoted $r : p$.

Definition 16 (filtering) *Let T be a set of mutually disjoint sets and let $\sigma \in \cup T \to T$ be the function that maps an element to the set which it belongs. Let $\tau \in I \to T$ determine the types associated with the indexes and let $p \in I \to X$ determine the variables associated with the indexes. Let $s \in X \to \cup T$ substitute domain elements for variables and let $\phi \in X \to T$ determine the types of the variables. Then the* filtering : *of a relation by p is defined by*

$$\begin{array}{rcl}
\langle \tau, E \rangle : p & = & \langle \varphi, E' \rangle \textit{ where} \\
E' & = & \{s \in X \to \cup T \mid \exists t \in E \textit{ such that } t = s \circ p\}
\end{array}$$

The case where p is a bijection is interesting, as it gives rise to the following equalities:
For all $E \subset \text{CART}(\tau)$

$$(\langle \tau, E\rangle : p) : p^{-1} = \langle \tau, E\rangle$$

and, for all $E \subset \text{CART}(\varphi)$

$$(\langle \varphi, E\rangle : p^{-1}) : p = \langle \varphi, E\rangle.$$

The relations $\langle \tau, E\rangle$ and $\langle \varphi, E'\rangle$ can be thought of as renamings of each other, with p and p^{-1} as renaming schemes.

If p is not a bijection, then there is potentially more going in $\langle \tau, E\rangle : p$ than a renaming, so it would be a mistake to call it that. What is it that happens in $\langle \tau, E\rangle : p$ with an arbitrary p? We think that "filtering" is a good name.

6.4 Join

Given a relation r, one may be interested in relations that have r as projection. There is a largest such, which is the "cylinder" on r. This gives the idea; to get the signatures right, see the following definition.

Definition 17 (cylinder) *For $i \in \{0,1\}$, let τ_i be in $I_i \to T_i$, such that τ_0 and τ_1 are summable; T_i is a set of disjoint sets. Let $\langle \tau_0, E_0\rangle$ be a relation. The* cylinder with respect to τ_1 on $\langle \tau_0, E_0\rangle$ *is written as $\pi_{\tau_1}^{-1}(\langle \tau_0, E_0\rangle)$ and is defined to be relation*

$$\langle \tau_0 + \tau_1, \{t \in \text{CART}(\tau_0 + \tau_1) \mid t \downarrow I_0 \in E_0\}\rangle.$$

The notation π^{-1} suggests some kind of inverse of projection. It is suggested by facts such as $\pi_{I_0}(\pi_{\tau_1}^{-1}(\langle \tau_0, E_0\rangle)) = \langle \tau_0, E_0\rangle$.

Example 9 *Let $I_0 = \{a, b, c\}$ and $I_1 = \{b, c, d\}$. Let τ_0 be in $I_0 \to \{\{0,1\}\}$ and τ_1 be in $I_1 \to \{\{0,1\}\}$. Consider the relations $\langle \tau_0, E_0\rangle$ and $\langle \tau_1, E_1\rangle$, where $E_0 =$*

a	b	c
0	0	1
1	1	0

and $E_1 =$

b	c	d
0	1	0
1	1	1

. We have $\pi_{\tau_1}^{-1}(\langle \tau_0, E_0\rangle) = \langle \tau_0 + \tau_1, C_0\rangle$ and $\pi_{\tau_0}^{-1}(\langle \tau_1, E_1\rangle) = \langle \tau_0 + \tau_1, C_1\rangle$ where $C_0 =$

a	b	c	d
0	0	1	0
0	0	1	1
1	1	0	0
1	1	0	1

and $C_1 =$

a	b	c	d
0	0	1	0
1	0	1	0
0	1	1	1
1	1	1	1

.

Definition 18 (join) *For $i \in \{0,1\}$ let there be relations $\langle \tau_i, E_i\rangle$, with $\tau_i \in I_i \to T_i$ and T_i a set of disjoint sets. If τ_0 and τ_1 are summable, then the* join *of $\langle \tau_0, E_0\rangle$ and $\langle \tau_1, E_1\rangle$ is written as $\langle \tau_0, E_0\rangle \bowtie \langle \tau_1, E_1\rangle$ and defined to be $\pi_{\tau_1}^{-1}(\langle \tau_0, E_0\rangle) \cap \pi_{\tau_1}^{-1}(\langle \tau_1, E_1\rangle)$.*

The intersection in this definition is defined because of the assumed summability of τ_0 and τ_1. The signature of $\langle \tau_0, E_0\rangle \bowtie \langle \tau_1, E_1\rangle$ is $\tau_0 + \tau_1$.

Example 10 *Let τ_0, τ_1, E_0, and E_1 be as in Example 9. Then we have $\langle \tau_0, E_0\rangle \bowtie \langle \tau_1, E_1\rangle = \langle \tau_0 + \tau_1, E\rangle$ where $E =$*

a	b	c	d
0	0	1	0

.

Example 11 *Consider the relations shown in the tables in Figure 6. Each table consists of a line of headings followed by the line entries of the tables. The line entries represent the tuples of the relations. Each table has three such lines.*

The following abbreviations are used. In the SUPPLIERS *table,* SID *for supplier ID,* SNAME *for supplier name, and* CITY *for supplier city. In the* PARTS *table,* PID *for part ID,* PNAME *for part name, and* PQTY *for part quantity on hand. In the* PROJECTS *table,* RID *for project ID and* RQTY *for part quantity required.*

We want to know the part names and cities in which there is a supplier with a sufficient quantity on hand for at least one of the projects.

SUPPLIERS =

SID	SNAME	CITY
321	LEE	TULSA
322	POE	TAOS
323	RAY	TULSA

PARTS =

PID	PNAME	SID	PQTY
213	HOSE	322	13
214	TUBE	321	6
215	SHIM	322	18

PROJECTS =

RID	PID	RQTY
132	215	2
133	214	11
134	213	18

Figure 6: The relations for Example 11.

Let the tables in Figure 6 be the relations SUPPLIERS $= \langle \tau_0, E_0 \rangle$, PARTS $= \langle \tau_1, E_1 \rangle$, and PROJECTS $= \langle \tau_2, E_2 \rangle$. In addition, there is a relation in the query for which there is no table, namely the less-than-or-equal relation. Mathematically, there is no reason to treat it differently from the relations stored in tables. Hence, we also include it as LEQ $= \langle \tau_3, E_3 \rangle$.

The index sets of τ_0, τ_1, τ_2 are the sets of the column headings of the tables for SUPPLIERS, PARTS, and PROJECTS, respectively:

- for τ_0 the index set is $\{\text{SID}, \text{SNAME}, \text{CITY}\}$,
- for τ_1 it is $\{\text{PID}, \text{PNAME}, \text{SID}, \text{PQTY}\}$,
- for τ_2 it is $\{\text{RID}, \text{PID}, \text{RQTY}\}$,
- for τ_3 it is $\{\text{RQTY}, \text{PQTY}\}$.

The extents E_0, E_1, and E_2 are as described in Figure 6. Moreover, $E_3 = \{t \in \text{CART}(\tau_3) \mid t_{\text{RQTY}} \leq t_{\text{PQTY}}\}$.

Definition 19 (relational product) *For $i \in \{0, 1\}$ let there be relations $\langle \tau_i, E_i \rangle$, with $\tau_i \in I_i \to T_i$ and T_i a set of disjoint sets. If I_0 and I_1 are disjoint, then the* relational product *of $\langle \tau_0, E_0 \rangle$ and $\langle \tau_1, E_1 \rangle$ is defined and it is defined to be equal to $\langle \tau_0, E_0 \rangle \bowtie \langle \tau_1, E_1 \rangle$.*

The word "join" is borrowed from relational databases [4, 1], where "natural join" denotes a similar operation. Definition 19 shows that an operation reminiscent of Codd's relational product is a special case of join.

The following lemma consists of assorted equalities selected to build intuition.

Lemma 2

$$\begin{aligned} \langle \tau, E_0 \rangle \bowtie \langle \tau, E_1 \rangle &= \langle \tau, E_0 \cap E_1 \rangle \\ \pi_{\tau_1}^{-1}(\langle \tau_0, E_0 \rangle) &= \langle \tau_0, E_0 \rangle \bowtie \langle \tau_1, \text{CART}(\tau_1) \rangle \\ \langle \tau_0, E_0 \rangle \bowtie \langle \tau_1, \text{CART}(\tau_1) \rangle &= (\langle \tau_0, \text{CART}(\tau_0) \rangle \bowtie \langle \tau_1, E_1 \rangle) \end{aligned}$$

7 Queries

Certain queries can be expressed as projections of joins. In such simple queries each relation is referred to only once. When a query needs to refer to the same relation more than once, SQL uses the renaming operation. In this section we show how filtering can take the place of renaming.

7.1 Projections of joins as queries

The SQL query in Figure 7 can be understood as specifying a relation defined in terms of given relations understood as in Definition 13. In this query, projection and join can be understood according to Definitions 15 and 18, respectively.

```
SELECT PNAME, CITY
FROM SUPPLIERS, PARTS, PROJECTS
WHERE PARTS.PID = PROJECTS.PID
      AND SUPPLIERS.SID = PARTS.SID
      AND RQTY <= PQTY
```

Figure 7: An SQL query to the relations in Figure 6.

PC =

PARENT	CHILD
MARY	JOHN
JOHN	ALAN
MARY	JOAN

Figure 8: Relation for Example 12.

Consider the expression

$$\pi_{\{\text{PNAME,CITY}\}}(\text{SUPPLIERS} \bowtie \text{PARTS} \bowtie \text{PROJECTS} \bowtie \text{LEQ}). \tag{2}$$

For the relations to be joinable, the signatures τ_0, τ_1, τ_2, and τ_3 have to be summable. That is, any elements common to their source sets have to have the same value. For example, the source sets of τ_0 and τ_1 have SID in common. They both map SID to its domain, which is the set of supplier IDs. Therefore τ_0 and τ_1 are summable; hence SUPPLIERS $\bowtie$ PARTS is defined (Definition 18). Similarly with the other joins in the expression (2), which has as value the relation described by the SQL query in Figure 6.

The query in Example 11 uses relations as they are are given. Let us now consider queries joining relations *derived from* given relations. Such derivations can be effected with filtering.

7.2 Queries requiring filtering

Example 12 *In Figure 8 we show a table specifying a relation consisting of tuples of two components in which one is a parent of the other. It is required to identify pairs of persons who are in the grandparent relation.*

What distinguishes this query from the one in Example 11 is that the relations do not occur in the join as given, but are derived from the given relation. On the basis of the derived relations we create one in which the pairs are in the grandparent relation. In one of the SQL dialects this would be:

```
SELECT PC0.PARENT, PC1.CHILD
FROM   PC AS PC0, PC AS PC1
WHERE  PC0.CHILD = PC1.PARENT
```

In this query, the derived tables are obtained via the linguistic device of first renaming PC to PC_0 and then to PC_1. The SQL compiler translates such a query behind the scenes to suitable relational operations.

Let us now return to Example 12 to see how filtering is used here. Suppose we have a set $X = \{x, y, z\}$ and an index set $I = \{\text{PARENT}, \text{CHILD}\}$. Let p and q both be in $I \to X$, $p =$

PARENT	CHILD
x	y

, and $q =$

PARENT	CHILD
y	z

.

The relation

$$\pi_{\{x,y\}}(\text{PC} : p \ \bowtie \ \text{PC} : q)$$

is the equivalent of the relation resulting from the SQL query. Here PC : p and PC : q denote the results of filtering PC with respect to p and q, respectively.

In this example, we have followed database usage in making the index set $I = \{\text{PARENT}, \text{CHILD}\}$ non-numerical. If we set $I = \iota(2)$, then we can write $[x, y]$ for p and $[y, z]$ for q. The query then becomes

$$\pi_{\{x,z\}}(\text{PC} : [x, y] \ \bowtie \ \text{PC} : [y, z]), \tag{3}$$

This bears a resemblance to the formula of predicate calculus

$$\exists y.\ pc(x, y) \wedge pc(y, z). \tag{4}$$

In the next section we will use the ETR operations for semantics of predicate calculus.

In this paper we do not consider any query language; we only discuss relational operations and how they can be used to obtain the computed relations that we want as results from queries, independently of how they might be formulated in a query language.

Example 13 *Consider the following instance of Figure 3.* $I = \iota(2)$, $X = \{x, z\}$, *and* $T = \{R\}$. *Because* T *has a single element, there is only one possibility for* τ *and* φ, *while* σ *is pre-ordained by* T. *Let us define*

$$prod = \langle \tau, M \rangle \text{ with } M = \{[u, v, w] \in \mathcal{R}^3 \mid u \times v = w\}$$

and

$$sq = \langle \varphi, S \rangle \text{ with } S = \{s \in \{x, z\} \to \mathcal{R} \mid s_x^2 = s_z\}$$

prod is the ternary product relation; sq is the binary squaring relation. In Figure 3, t *is a tuple in* M *and* s *is a tuple in* S.

The fact that, and the way in which, squaring is a special case of multiplication can be expressed as

$$sq = prod : [x, x, z]$$

8 ETR as semantics for predicate calculus

Let us recapitulate Codd's plan for databases:

1. data in the form of relations (the base relations), and
2. queries, formulated in predicate calculus, that define answers in the form of relations and that are translated to machine-executable operations on base relations

As for (1), we argued that Codd's understanding of the mathematics of relations was not adequate; we provided a theory that answers to the needs of databases. As for (2), again Codd gave a good start. Unfortunately, it seems that the necessary follow-up has failed to appear. We address part (2) of Codd's plan in this section.

The attraction of predicate calculus is that it is a formalism which is close to natural language and has a mathematically defined semantics. Early in the history of predicate calculus this semantics existed only in the form of an inference system. A formula was considered true or false according to whether the corresponding truth value was derivable in the inference system. This was valuable because the inference rules were all deemed sound. But could they be *proved* to be so? And could they prove everything that could reasonably considered to be true? These questions, the soundness and the completeness of the inference system, could not be answered without a *declarative* semantics of logic. Here "declarative" means independence of any procedure, such as an inference system. It is this declarativity that makes predicate calculus potentially valuable for database: we want a correctness criterion for an implementation of the query language that does not refer to some other mechanism.

In the 1930s Tarski proposed a declarative semantics for predicate calculus. The proposal was successful because it corresponded to intuition and because the intuition-inspired inference systems of the day turned out to be sound and complete with respect to it. Thus these inference systems constituted *procedural semantics*, as computer scientists were to call it later. It must have been the fact that predicate calculus has both a declarative and a procedural semantics that recommended this formalism to Codd.

Codd's idea can be summarized as: predicate calculus as a declarative query language for the user with an implementation based on a sound and complete inference system. At the time this was proposed and implemented by Green [9].

However, the inference systems for predicate calculus are not suitable for implementation of information retrieval on the scale of databases. If the difficulty needs to be summarized in a few words one could say that the inference system operates on a tuple at a time. At the scale of a database one needs to operate on meanings of predicate symbols (relations) as a whole.

This suggests a third semantics for predicate calculus: in terms of relations and operations on them. That is, in terms of a relational algebra. The pioneers in this direction were Halmos [13] and, again,

Tarski [16, 17]. The applicability Tarski's relational algebra to databases was noted by Imielinski and Lipski [15], but this does not seem to have had an impact in the database world.

Codd saw the need for a relational algebra as counterpart for a declarative query language. He proposed the relational calculus, a drastic modification of predicate calculus. He defined a relational algebra corresponding to it. In this way he obtained a mathematical basis for the implementation of databases. Because of the tenuous connection between relational calculus and predicate calculus, the declarative nature of the query language has been lost. Thus it was left to others to restore to databases the dual semantics that makes predicate calculus so valuable: a declarative semantics with an operational semantics that is sound and complete.

So far we have only used ETR for a mathematical definition of relations and operations on them. We proceed to show how ETR can be used to provide an operational semantics for predicate calculus. What makes our semantics valuable for databases is that it is defined in terms of the operations on relations as defined in ETR. At the same time it applies (see Theorems 1, 2, 3, 4) to predicate calculus of which the declarative semantics has been established equivalent to the proof theory of logic in the 1930s. An alternative route would have been to show that the declarative semantics of predicate calculus can be adapted to Codd's relational calculus. We have declined to take this route; neither has it, to our knowledge, been taken by others.

A disadvantage for database use of predicate calculus is that its tuples are restricted to numerical indexes and that there is a single domain. In our use of ETR as algebraic counterpart of predicate calculus we restrict ourselves to this special case. But we note that versions of predicate calculus have been developed to accommodate multiple domains, though still restricted to numerical indexes; see "many-sorted logic" [6, 8]. ETR has full generality of indexing as well as many-sortedness. To apply it, as we do here, to the semantics of conventional predicate calculus, does not use the full capability of ETR.

Satisfaction semantics of predicate calculus Conventional semantics is primarily concerned with the justification of inference systems. The use of predicate calculus for the definition of functions or relations is secondary, if considered at all. As a result, conventional semantics centres around the concept of *satisfaction*: under what conditions is a formula satisfied by a given interpretation of the predicate symbols and constant symbols under a given assignment of domain elements to the variables.

For the sake of brevity we assume the absence of function symbols, in harmony with their absence in conventional relational database theory. Our language of logical formulas is determined by a set P of *predicate symbols*, a set V of *variables*, and a set of *constant symbols*.

A *term* is a variable or a constant symbol.

A *formula* can be

- an *atom* (or *atomic formula*) is an expression of the form $p(t_0, \ldots, t_{k-1})$, where $p \in P$ is a predicate symbol and $t_0, \ldots, t_{k-1}$ are terms.
- a *conjunction* is an expression of the form $F_0 \wedge \cdots \wedge F_{n-1}$ where F_i is a formula, $i \in \{0, \ldots, n-1\}$.
- an *existential quantification* is an expression of the form $\exists x.F$ where x is a variable and F is a formula.
- a *negation* is an expression of the form $\neg F$, where F is a formula.

An *interpretation* for the language consists of a set $\mathbf{D}$ called the *domain* of the interpretation, a function $\mathcal{P}$ that maps every k-ary predicate symbol in P to a subset of $\mathbf{D}^k$, and a function $\mathcal{F}$ that maps every constant symbol to an element of $\mathbf{D}$.

An interpretation assigns the meaning $M(t)$ to a term t and determines whether a variable-free formula is true (in that interpretation). We refer to M as "meaning function" or as "interpretation".

- $M(t) = \mathcal{F}(t)$ if t is a constant symbol
- A variable-free atom $p(c_0, \ldots, c_{k-1})$ is satisfied by an interpretation iff $[M(c_0), \ldots, M(c_{k-1})] \in \mathcal{P}(p)$.
- A *conjunction* $A_0 \wedge \cdots \wedge A_{n-1}$ of variable-free atoms is satisfied by M if A_i is satisfied by M, $i \in \{0, \ldots, n-1\}$.
- A variable-free formula $\neg F$ is satisfied by M if F is not satisfied by M.

We now consider meanings of formulas that contain variables. Let $\mathcal{A}$ be an *assignment*, which is a function in $V \to \mathbf{D}$, assigning an individual in $\mathbf{D}$ to every variable. In other words, $\mathcal{A}$ is a tuple of elements of $\mathbf{D}$ indexed by V. As meanings of terms with free variables depend on $\mathcal{A}$ we write $M_{\mathcal{A}}$ for the function mapping a term to a domain element. $M_{\mathcal{A}}$ is defined as follows.

- $M_{\mathcal{A}}(t) = \mathcal{A}(t)$ if t is a variable.
- $M_{\mathcal{A}}(c) = M(c)$ if c is a constant symbol.
- $p(t_0, \ldots, t_{k-1})$ is satisfied by M and $\mathcal{A}$ iff $[M_{\mathcal{A}}(t_0), \ldots, M_{\mathcal{A}}(t_{k-1})] \in \mathcal{P}(p)$.

Now that satisfaction of atoms is defined, we can continue with:

- $F_0 \wedge \cdots \wedge F_{n-1}$ is satisfied iff the formulas F_i are satisfied, for $i = 0, \ldots, n-1$.
- If F is a formula, then $\exists x.F$ is satisfied by M and $\mathcal{A}$ iff there is a $d \in \mathbf{D}$ such that F is satisfied with M and $\mathcal{A}_{x|d}$ where $\mathcal{A}_{x|d}$ is an assignment that maps x to d and maps the other variables according to $\mathcal{A}$.
- $\neg F$ is satisfied iff formula F is not satisfied.

If S is a formula without free variables (a *sentence*) that is satisfied by no M, then S is said to be *unsatisfiable*; if S is satisfied by all M, then it is said to be *valid*.

Denotation semantics So far satisfaction semantics, which is standard in treatments of predicate logic. The purpose of these treatments is to establish results like the soundness and completeness of inference systems or to characterize the nature of logical implication. The assigning of meanings to formulas with free variables plays a subordinate role in standard treatments of predicate logic.

One of Codd's requirements was that formulas with free variables (used to represent queries) be given as meaning a relation (the set of answers). This can be done with a generalization beyond the satisfaction semantics summarized above. This generalization comes down to a generalization of the mapping M that extends its applicability from closed formulas to open ones. We call this generalization *denotation semantics*.

Let us suppose that F is a formula and that X is the set of its free variables. We now consider

$$\{t \in (X \to \mathbf{D}) \mid \exists \mathcal{A}.\ t = \mathcal{A} \downarrow X \text{ and } F \text{ is satisfied by } M \text{ and } \mathcal{A}\ \}.$$

As $\mathcal{A}$ is a tuple in $V \to D$, $\mathcal{A} \downarrow X$ is a tuple in $X \to D$. Hence this set is the extent of a relation with $X \to D$ as signature. This set of tuples is independent of $\mathcal{A}$, so it is entirely determined by F and M. Accordingly, it can be used in the definition of a meaning function $\mathcal{M}$ to be applicable to open formulas F, as follows.

Definition 20 *Let F be a formula with X as set of free variables.*

$$\mathcal{M}(F) = \langle X \to \boldsymbol{D}, \{t \mid \exists \mathcal{A}.\ t = \mathcal{A} \downarrow X \text{ and } F \text{ is satisfied by } M \text{ and } \mathcal{A}\ \}\rangle.$$

Note that this is a relation with tuples indexed by variables; the relations that are the meanings of predicate symbols are indexed numerically, as forced by the syntax of predicate calculus.

Definition 20 is a generalization of M because it applies to closed formulas as well. If X is empty (we have a closed formula F), then $\mathcal{M}(F)$ is a set of 0-tuples. As there is only one 0-tuple, there are only two sets of 0-tuples. Apparently a closed formula F is unsatisfiable iff $\mathcal{M}(F) = \{\}$.

An example from Codd As an application of Definition 20 we consider Codd's [3] (page 383) definition of the "natural" join between binary integer-indexed relations R and S as

$$R * S = \{(a, b, c) : R(a, b) \wedge S(b, c)\}. \tag{5}$$

This demonstrates a common confusion. The expression (a, b, c) refers to a triple of elements of the domain; therefore a, b, and c are metalanguage names for semantic entities. By its form $R(a, b) \wedge S(b, c)$ suggests a formula of logic, a syntactic entity. Here a, b, and c refer to variables of the formal logic.

It is not difficult to explain how this confusion came about. What is meant by Equation 5 is the informal mathematical statement

"$R * S$ is the set of triples (a, b, c) such that (a, b) in R and (b, c) in S."

There is no formal logic here: R and S are names of relations, a, b, and c are names of domain elements. Such a statement is often abbreviated to a shorthand such as

$$R * S = \{(a, b, c) : (a, b) \in R \text{ and } (b, c) \in S\}, \tag{6}$$

which is not confusing, because it is still clear that it is entirely informal mathematics.

Usually such fine distinctions as we just made here do not matter. However, when embarking on a project like the relational calculus the distinctions become crucial. We demonstrate the denotation semantics developed in this section by its definition of the "natural" join between two relations. Let P contain the predicate symbols r and s. For the variables, let V be $\{x, y, z, \ldots\}$. With this syntax we can make formulas such as $r(x, y) \wedge s(y, z)$.

On the semantic side, we'll let domain $\mathbf{D}$ be $\{a, b, c\}$ and consider the binary integer-indexed relations

$$\rho = \langle \iota(2) \to \mathbf{D}, \{[a, c], [c, b], [b, a], [b, b]\}\rangle$$

and

$$\sigma = \langle \iota(2) \to \mathbf{D}, \{[a, b], [b, c], [c, a]\}\rangle.$$

Assignments map variables to domain elements, so $\mathcal{A}$ could be a mapping with $x \mapsto b$, $y \mapsto a$, and $z \mapsto b$. It doesn't matter how $\mathcal{A}$ maps the other infinitely many variables.

Let's assume that the interpretation M maps r to ρ and s to σ. With this interpretation we define a binary operation $*$ on binary integer-indexed relations as

$$\begin{aligned}
& \rho * \sigma \\
= \; & \langle \{x, y, z\} \to \mathbf{D}, \{t \mid \exists \mathcal{A}.\ t = \mathcal{A} \downarrow \{x, y, z\} \text{ and } r(x, y) \wedge s(y, z) \text{ is satisfied by } M \text{ and } \mathcal{A}\}\rangle : [x, y, z]^{-1} \\
= \; & \langle \iota(3) \to \mathbf{D}, \{[a, c, a], [c, b, c], [b, a, b], [b, b, c]\}\rangle
\end{aligned}$$

From Definition 20 it is clear that, by taking the place of the F in that definition, only $r(x, y) \wedge s(y, z)$ is part of the formal language of predicate calculus.

Properties of Definition 20 Our claim that Definition 20 provides a semantics for the open formulas of predicate calculus is based on the following theorems. In these theorems we assume that the interpretation M has $\mathbf{D}$ as domain.

Theorem 1 *Consider the atomic formula $q(p_0, \ldots, p_{n-1})$. Assume that X is the set of variables in $[p_0, \ldots, p_{n-1}]$. Then we have*

$$\mathcal{M}(q(p_0, \ldots, p_{n-1})) = \langle X \to \boldsymbol{D}, M(q) : [p_0, \ldots, p_{n-1}]\rangle.$$

Proof. According to Definition 20 the left-hand side has $X \to \mathbf{D}$ as signature. So both sides have the same signature. It remains to prove that the extents of the relations are equal.

Let $\mathcal{A}$ be such that $a = \mathcal{A} \downarrow X$.

a in the extent of the left-hand side	$\Leftrightarrow$ (Definition 20)
$q(p_0, \ldots, p_{k-1})$ is satisfied by M with $\mathcal{A}$	$\Leftrightarrow$ (use satisfaction)
$[a(p_0), \ldots, a(p_{k-1})] \in M(q)$	$\Leftrightarrow$ (use $t = a \circ p$)
$[t_0, \ldots, t_{k-1}] \in M(q)$	$\Leftrightarrow$ (definition of :)
$a \in$ extent of $M(q) : [p_0, \ldots, p_{k-1}]$.	

Theorem 2 *Let X be the set of variables in the conjunction $B_0 \wedge \cdots \wedge B_{n-1}$ of atomic formulas. Then we have*

$$\mathcal{M}(B_0 \wedge \cdots \wedge B_{n-1}) = \langle X \to \boldsymbol{D}, \mathcal{M}(B_0) \bowtie \cdots \bowtie \mathcal{M}(B_{n-1})\rangle$$

Proof

Both sides have the same signature. It remains to prove that the extents are equal.

$a \in X \to D$ in the extent of the left-hand side
$\Leftrightarrow$ (Definition 20)
$B_0 \wedge \cdots \wedge B_{n-1}$ satisfied by M and $\mathcal{A}$ such that $a = \mathcal{A} \downarrow X$, where X is the set of variables of $B_0 \wedge \cdots \wedge B_{n-1}$
$\Leftrightarrow$ (definition of $\wedge$)
B_i satisfied by M and $\mathcal{A}$ such that $a_i = \mathcal{A} \downarrow X_i$, where X_i is the set of variables of B_i, for $i = 0, \ldots, n-1$
$\Leftrightarrow$ (Definition 20)
a_i in extent of $\mathcal{M}(B_i)$ for $i = 0, \ldots, n-1$ and signature of $\mathcal{M}(B_i)$ is $\tau_i = X_i \to D$ and $\tau_0, \ldots, \tau_{n-1}$ are summable
$\Leftrightarrow$
a in extent of $\pi^{-1}_{X_0}\mathcal{M}(B_0) \cap \cdots \cap \pi^{-1}_{X_{n-1}}\mathcal{M}(B_{n-1})$
$\Leftrightarrow$ (definition of join)
a in extent of $\mathcal{M}(B_0) \bowtie \cdots \bowtie \mathcal{M}(B_{n-1})$.

Theorem 3 *Let F be a formula with X as its set of free variables, and $Y = \{y_0, \ldots, y_{k-1}\}$ a subset of X. Then we have*

$$\mathcal{M}(\exists y_0 \ldots y_{k-1}.F) = \langle (X \backslash Y) \to \boldsymbol{D}, \pi_{X \backslash Y}(\mathcal{M}(F)) \rangle$$

Proof Both sides have the same signature. It remains to prove that the extents are equal.

We take $\exists y_0 \ldots y_{k-1}.F$ to be a shorthand for $\exists y_0.\exists y_1 \ldots . F$.
$a \in X \backslash \{y\} \to D$ in the extent of the left-hand side
$\Leftrightarrow$ (Definition 20)
$\exists y.F$ is satisfied by M and $\mathcal{A}$ such that $\mathcal{A} \downarrow X \backslash \{y\} = a$
$\Leftrightarrow$ (definition of satisfaction, $\exists$ case)
$\exists d \in D$ such that F is satisfied by M and $\mathcal{A}$ such $\mathcal{A} \downarrow X = a^d$ where a^d such that $a^d(y) = d$ and $a^d(v) = a(v)$ for all $v \in V$ where v is not y
$\Leftrightarrow$ (definition of $\mathcal{M}$)
a^d in extent of $\mathcal{M}(F)$
$\Leftrightarrow$ (definition of π)
a in extent of $\pi_{X \backslash \{y\}}\mathcal{M}(F)$.

Theorem 4 *Let F be a formula. Then we have*

$$\mathcal{M}(\neg F) = \mathcal{M}(F)^C$$

where the complementation of the relation $\mathcal{M}(F)$ is according to definition 14.

Proof.
Both sides have the same signature. It remains to prove that the extents are equal.

$a \in X \to D$ in extent of left-hand side	$\Leftrightarrow$ (Definition 20)
$\neg F$ satisfied by M and $\mathcal{A}$ such that $\mathcal{A} \downarrow X = a$	$\Leftrightarrow$ (definition of $\neg$)
F not satisfied by M and $\mathcal{A}$ such that $\mathcal{A} \downarrow X = a$	$\Leftrightarrow$ (Definition 20)
a not in extent of $\mathcal{M}(F)$	$\Leftrightarrow$ (definition of complementation)
$a \in \mathcal{M}(F)^C$.	

9 Related work

Hall, Hitchcock, and Todd [11] noted the confusion of Codd and subsequent authors resulting in "relations" as distinct from "relationships". They defined relations with non-numerical indexes as the correct remedy.

The semantics given here applies only to first-order predicate calculus without function symbols. This restriction is lifted in [5]. But there the semantics applies not to the classical syntax of predicate calculus, as presented here, but to the Horn-clause subset of the clausal syntax.

Clark [2] seems to be the first to explicitly define a denotation semantics for atomic formulas.

10 Conclusions

We see three main contributions of this paper. The first is to develop ETR, an elementary theory of relations adequate for expressing relational databases, queries, and answers. This theory is entirely in terms of elementary set theory, the same set theory that underlies all mathematics. ETR should be compared with Codd's relational algebra, created solely for the purpose of supporting relational calculus.

The second contribution is to show that the special-purpose relational calculus is not necessary. We show that queries can be formulated in predicate calculus, the same logical calculus that is the formal logic preferred for the formalization of mathematical theories since well before the advent of databases.

The third contribution arises as a by-product of the first two. Its significance is independent of database theory: a generalization of the traditional satisfaction semantics of predicate calculus. Denotation semantics, as we call this generalization, is suited to applications elsewhere in computer science, as it makes predicate calculus into a powerful and flexible tool for defining new relations in terms of existing ones. The traditional satisfaction semantics remains of course satisfactory for the traditional purpose of formal logic: to analyse formalized theories for validity and to investigate the soundness and completeness of proof procedures.

11 Acknowledgements

We benefited from discussions with Areski Nait-Abdallah. The research was supported by the University of Victoria and by the Natural Science and Engineering Research Council of Canada.

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases: The Logical Level*. Addison-Wesley, 1994.

[2] Keith L. Clark. Logic-programming schemes and their implementations. In Jean-Louis Lassez and Gordon Plotkin, editors, *Computational Logic*, pages 487–541. MIT Press, 1991.

[3] E.F. Codd. A relational model of data for large shared data banks. *Comm. ACM*, pages 377 – 387, 1970.

[4] E.F. Codd. Relational completeness of data base sublanguages. In Randall Rustin, editor, *Data Base Systems*, number 6 in Courant Computer Science Symposia. Prentice Hall, 1972.

[5] M.H. van Emden. Compositional Semantics for the Procedural Interpretation of Logic. In *Proc. Intern. Conf. on Logic Programming*, pages 315–329, 2007.

[6] H.B. Enderton. *A Mathematical Introduction to Logic*. Fletcher and Sons, Ltd, 1972.

[7] Roland Fraïssé. *Theory of Relations, Revised Edition*. North Holland, 2000.

[8] Jean Gallier. *Logic for Computer Science*. Harper and Row, 1986.

[9] Cordell Green. The application of theorem-proving to question-answering systems. Technical Note No. 8, Artificial Intelligence Group, Stanford Research Institute, June 1969.

[10] Andrzej Grzegorczyk. *An Outline of Mathematical Logic: Fundamental Results and Notions Explained with All Details*. D. Reidel, 1974.

[11] Patrick Hall, Peter Hitchcock, and Stephen Todd. An algebra of relations for machine computation. In *Proc. Annual Symp. on Principles of Programming Languages*, pages 225–232, 1975.

[12] Paul R. Halmos. *Naive Set Theory*. D. Van Nostrand, 1960.

[13] Paul R. Halmos. *Algebraic Logic*. Chelsea, 1962.

[14] Leon Henkin, J. Donald Monk, and Alfred Tarski. *Cylindric Algebras*. North Holland, 1971.

[15] T. Imielinski and W. Lipski. The relational model of data and cylindric algebras. *Journal of Computer and System Sciences*, 28:80–102, 1984.

[16] A. Tarski. A representation theorem for cylindric algebras. *Bull. Amer. Math. Soc.*, 58:65 – 66, 1952.

[17] A. Tarski and F.B. Thompson. Some general properties of cylindric algebras. *Bull. Amer. Math. Soc.*, 58:65 – 65, 1952.