Alternative Technologies™

Logic for Serious Database Folks Series

by David McGoveran, Alternative Technologies

LEGAL NOTICE AND LICENSE

IMPORTANT CONTENT NOTICE

It is anticipated that many readers of this document will have some familiarity with the writings of other authors, especially but not limited to E. F. Codd ("EFC"), C. J. Date ("CJD"), and H. Darwen ("HD"), or familiarity with Date and Darwen's The Third Manifesto ("TTM") and related writings. Material in this series will not comply with all of TTM and definitions therein, nor with any other interpretation of the work of EFC. Readers should not assume otherwise of an independent work. I will, from time to time, refer to specific passages in the works of EFC, CJD and HD for purposes of commentary or to illustrate some issue. When I disagree with a some construct by EFC, CJD, or HD and am aware of it, I will say so and give a justification.

CONTACT BY REVIEWERS

Reviewers are encouraged to convey their comments and criticisms to me directly via email addressed to **mcgoveran@AlternativeTech.com**., and including "REVIEW" in the subject line. Please provide document title, revision date, and page when referencing any passage. I will do my best to respond in a timely manner and will try to answer specific questions if there is a reasonably short answer.

Alternative Technologies

Can All Relations Be Updated?

(or Can Any Relation Be Updated?)

Logic for Serious Database Folks Series

by David McGoveran, Alternative Technologies

"Grant me the wisdom to know that which I can change, and that which I cannot."– A Variation on the Serenity Prayer

Preface

This is an extract of a draft chapter in my forthcoming book, tentatively titled Logic for Serious Database Folks. Because that book is still evolving (extensive rewrites) and progressing, I cannot even tell the reader which chapter this will be. I have decided to publish it, in part because Fabian Pascal asked me to do so, and in part because it demonstrates why a deep understanding of formal systems such as set theory and logic is essential for those who wish to understand, explain, use, improve, or criticize the RM in a defensible way. My "hidden" agenda, of course, is to entice you to read more of the book as it becomes available. This article was originally written in 2012 and revised February 2015 when I noticed a number of typographical errors and glosses. I cannot guarantee that I have corrected all of those problems, hence I still consider it a draft.

Introduction

The title of this chapter may sound a bit facetious. After all, relations (or at least their SQL counterparts) are updated all the time in practice. But perhaps that just means we have a practical procedure. It does not necessarily mean we have a proper theoretical foundation for that practice. So we must ask, in all seriousness, what does it mean for a relation to be updated? A relation is a set. What can it possibly mean to "update" a set? What, exactly, did Codd mean that a relation was "time-varying"?

And if a relation is one of a collection of related relations, then those relationships are formally expressed through integrity constraints (i.e., multi-relation dependencies), preferably as a well-formed forumula (wff) in first order predicate logic.[1] Clearly, we need to understand all those relationships in order to know how what would constitute a permissible "update." And even if we do understand them, it is often incorrect to "update" a single relation – changing one relation forces changes in other relations. And again, we've various practical procedures built into commercial DBMSs for handling such requirements, from compensating actions like "cascade delete" to transactions. Even so, programmers sometimes find those built-in procedures inadequate. Something is amiss, but what?[2]

The purposes of this chapter include: (1) Clarifying the history of POOD and the errors in prior art thinking that led to it, (2) emphasizing the crucial role that set membership functions - formulated in FOPL and constructed in a very specific, formal manner can and should play in relation updating (to wit, they set act as selectors on the powerset of the relation's universe of possible members), (3) that treating updates in terms of programming variables and adding those to the relational algebra is a horrible error, and (4) that updates are properly understood as a database operation regardless of whether the treatment is consistent with set theory/FOPL/relational algebra or any formal language that supports programming variables.

Emphasizing and elaborating on the role of set membership functions can play is crucial. If you don't embrace set membership functions and their expression in FOPL then there is no hope of understanding POOD and no hope of understanding relation updating, let alone solving view updating. Set membership functions - the *intensional definition* of a set - are fundamental to set theory, but it seems that many have been taught only the notion of an extensional definition (defining the set by exhibition of its members).

The Lessons of History

As my introduction has suggested, when Codd introduced the term "time-varying" it should have been a warning that the concept of updating relations was not as simple as it appeared. Quite possibly the only reason it was allowed to pass without a more formal explanation was that his audience was comprised significantly of those with more computing experience than knowledge of the foundations of set theory (a knowledge that Codd certainly possessed). As we will see, the term was conceptually misleading. It also glossed over a rather technical equivalent in set theory augmented with a certain non-set theoretic (i.e., non-*relational*) element.

---

[1] One reviewer interpreted this to mean that any perceived or possible relationship had to be expressed as an integrity constraint. To the contrary, I do not consider accidental or incidental relationships among specific occurrences of data value to be necessarily meaningful and certainly not defining of a relationship.
[2] One reviewer immediately replied "TTM's multiple assignment". However, while I consider the concept of multiple assignment an elegant one, it is limited (it cannot substitute for a group of interdependent updates in which an order is required to obtain a desired result – as for example occur in interactive transactions). Furthermore, in light of a viable general algorithm for view/relation updating, it is redundant in exactly those cases in which update of a view would suffice. This method is discussed in another chapter.

Another indication that something was amiss with the concept of updatable relations was the development of compensating actions for referential integrity constraints. Compensating actions to maintain referential integrity include, for example, cascade (delete/update), restrict, and set null. The very fact that these actions must be selected by database designers rather than being automatically determined by the DBMS based on relational theory or logic, strongly suggests that they are *ad hoc* practical procedures – *pragma*. In a word, they are *works-around*. If our understanding of "updates" were complete, works-around should not be necessary.
Last but not least, early efforts to realize the promise of data independence (both physical and logical) – the particular goal that motivated Codd in developing much of the relational model – ran headlong into the view updating problem. While theory suggested that views should behave exactly like so-called base relations with respect to updates, research quickly led to several conclusions: (1) views and base relations were actually different and required different treatments for updates, (2) only a few types of views could be updated – meaning that the update to the view could be propagated to the base relations from which the view derived, and (3) a general view updating algorithm was impossible. Note that the first conclusion leads us to characterize the issue as the "view updating problem" rather than the "relation updating problem". The schism is deep. Again, something is surely amiss.

These issues were addressed in 1994. Since the first publication of the three papers on relation updating and POOD some eighteen years ago[3], refinements to and variations of the explanation of (and algorithms for) view updating have been published by C. J. Date[4]. In addition, I've published my own refinements and explanations, as well as a unified algorithm for updating all relations (not just derived relations).[5] Unfortunately, various perceived problems with both the early approach and the later refinements have been raised. In reviewing those perceived problems, it seems that they arise from misunderstandings of some fundamentals, stemming in part from the fact that Date's explanations are not consistent with the intent (and definitions) of the work from which they were derived.

When Date and I first began exploring the view updating problem in 1993, the state of the art was a set of requirements for any solution (i.e., a set of view updating rules, or bettter, an algorithm). A general consensus in the research community, as evidenced by publications including those of Ted Codd and Chris Date, was that many views were inherently non-updatable. Date was writing the 6th edition of his well-known text *An Introduction to Database Systems*, had stated a set of requirements, and then set about developing examples. One

---

[3] C. J. Date and D. McGoveran, Database Programming and Design, June-August, 1994
[4] See, for example, the books of C. J. Date (which present several variations on view updating and do not offer a satisfactory definition of relation predicate and, in consequence, cannot properly explain POOD). The recent text *View Updating and Relational Theory*, acknowledges my own work (which I deeply appreciate) and purports to explain it, but the explanation is unfortunately incorrect. Chris and I were unable to correct the errors before publication.
[5] See, for example, U. S. Patents 7,263,512 and 7,620,664.

requirement was that the view updating rules, when applied to each side of a set theoretic tautology wherein each side represented a view definition in terms of base relations, yield the same values of the base relations after the view update. He noticed that the requirements seem to imply an inconsistency when applied to certain set theoretic tautologies. To be more precise, when Date's view updating rules were applied to the left hand side of the tautology and independently to the right hand side of the tautology, the resulting updates to base relations were different.

In other words, Date's view updating rules did not preserve the logical equivalence that set theoretic tautologies guaranteed! To see the implications of this result, imagine that you define a view using a particular relational expression. Fortunate you, you happen to have a DBMS that implemented Date's view updating rules and they worked – updates to your new view propagated updates to the base relations (the relations on which the view was defined) exactly the way you expected. For whatever reason, suppose you redefined the view using a relational expression that set theory guaranteed was equivalent to the original defining expression – it was just a different way of deriving the same view. Unfortunately, now Date's view updating rules cause something different to happen to the base relations when you update the view the same way you had previously! Something was terribly wrong with Date's view updating rules or the way in which they were being implemented in Date's approach to view updating.

Date contacted me to see what my take was on this situation. I recognized that the problem was closely related to one I had solved in the late 1980’s while working for a Wall Street client: Namely, we needed to merge two or more databases having different schemas, update the result, and then be able to reverse the merge so as preserve the updates. In other words, for this to work, the merge and all updates had to be set transformations each of which had an *inverse* transformation.

I then translated this general mathematical requirement into a set of three database schema requirements or properties,[6] including the idea that each base relation in the schema had to be semantically orthogonal to the other. This became known as POOD – The Principle of Orthogonal Design (for databases). The semantic orthogonality property holds in a database design (a particular schema) if and only if, ignoring multi-relation constraints, there is no "overlap" or redundancy between the intended membership of any two relations in the schema. If this is true, then we can add (or remove) tuples to (from) one without that operation in and of

[6] Formally stated, this requirement on the database design has three parts: semantic orthogonality, expressive completeness, and representational minimality. Attempts to state the first of these led to the Principle of Orthogonal Design or “POOD”. The other two parts – subsequently named the Principle of (Expressive) Completeness and the Principle of (Representational) Minimality, respectively – a have been largely ignored despite many attempts to emphasize their importance. In short, the Principle of (Expressive) Completeness requires that the set of base relations suffice to express every fact about the intended application, and the Principle of (Representational) Minimality requires that every fact about the intended application, however complex, can be expressed in at most one way. See the chapters on database design for a detailed explanation.

itself implying tuples need to be added (or removed) to the other for database consistency, except as required by some multi-relation constraint (e.g., a foreign key). This notion is analogous to having orthogonal axes in a coordinate system: The variables such axes represent are independent – we can alter the value on one such variable without altering any of the others.

In the same way, the relations in a semantically orthogonal schema are "update independent" in terms of their membership functions. Again, excluding multi-relation constraints, the membership function of a relation must imply that an update to that relation would not also be applicable to (or cause a change to) a second relation in a manner consistent with that second relations's membership function. Put in terms of logic, the membership functions of the base relations must be axiomatically independent under the rules of inference.

To reiterate, the problem Date had discovered regarding requirements for any successful set of view updating rules was that, no matter what set of view updating rules he tried, those rules gave a different result when applied to one side of a set tautology versus the other side of that tautology (the sets, of course, being relations). I saw that his example entailed base relations that were not semantically orthogonal – two of the base relations had "overlapping meanings" so that an update to one logically implied an update to the other if consistency was to be maintained.

When I explained my reversible merge solution to Date, he realized that I had the core of a general solution to view updating. Together, Date and I applied this thinking to the view updating problem. Along the way, I realized that a declarative approach required a declarative specification of the "meaning" of each relation. Date forced me to formalize this idea, which led pointing out that a relation, being a set, must have a membership function. Recall that a set membership function acts as a filter or constraint on set membership.

The set membership function is both a necessary and sufficient condition for set membership. It should *not*, however, be conceived of as merely a "member-at-a-time test independent of required relationships among of the members of the set. (For example, a membership function might require that the cardinality of the set be restricted to some value or range of values.) Given candidate individuals (e.g., a tuple) for membership in a set (e.g., a relation), *all* and *only* those individuals that satisfy the set membership function are members of the set. For consistency with RM and set theory, membership functions need to be expressed in first order predicate logic and as *n*-place predicates, each place corresponding to an attribute of an *n*-ary relation. In discussion with Date, I called this expression of a membership function applied to the specific type of set we call a relation, the relation's predicate. Eventually in the course of our correspondence, we began to call it simply the "*relation predicate*."

Important Note: Throughout this chapter, whenever the phrase "membership function" is applied to a relation, it is to be understood as a synonym for the expression of that membership function in the language of first order predicate logic. Specifically, the membership function for an n-ary relation is an n-place first order predicate.

Together, and over a period of a few months[7], we arrived at a solution to view updating. That solution was published in the industry technical magazine *Database Programming and Design* the following calendar year. It was with some chagrin that only later would I realize that the method was seen as being specific to view updating and not to relation updating in general. My personal intent had been for us to have introduced a new way of understanding *relation updating*. This chapter is, very specifically, about relation updating.

Aside: Database design, relation predicates, POOD, and other related concepts are examined in detail elsewhere in this book.

We Can't Solve If We Don't Understand

At present, I've identified three prevalent types of misunderstandings about the RM that have been the source of enormous mischief. The most prevalent type consists of failures to adhere to the declarative, set-orientation of the RM. This particular misunderstanding will be the one of most interest to us in the present chapter and is discussed below.

The second most prevalent type involves failures to provide the DBMS with sufficient information to perform requested operations as intended. Many data modeling issues fall within this category, as does improper reliance on object (i.e., relations and attributes) names or queries that select the intended data set by accident. For example, every time-varying relation can be understood in a way consistent with set semantics as a time ordered sequence of relations of the same name. In effect, a reference by relation name selects the one relation from this sequence that corresponds to the current time as the *default* relation for the state of the containing database at that time. I'll discuss this understanding of a time-varying relation in the present chapter a bit more formally, as well as another very important example of this type of misunderstanding.

The third type pertains to misunderstandings concerning the meaning and applicability of the RM. For example, I often hear phrases such as "non-relational data" that are alleged to be beyond the applicability of the RM.[8] This characterization of data demonstrates a fundamental lack of

[7] At the time, Chris did not have email. As a result, our written communication – necessary not just oral because of the technical nature of our topic – had to proceed by facsimile. This slowed us down, in part because it meant we could not make direct comments and edits to the other's communication. In consequence, I still have many, though not all of those original faxes showing how the concepts slowly took shape.

[8] There is no inherent property of data that would make it either relational or non-relational. The RM tells us how to represent data and relationships logically using the relational abstraction, no matter what data structures may be used to (physically) store that data. To do so one must understand the logical and semantic relationships among the data elements, the result of an investigative and analytic task that may be impractical for certain applications. Given the limited choices for physical data structures in commercial SQL DBMSs, and the lack of ability to transparently

understanding of the RM. While it is certainly true that such poor understanding of the RM would prevent any cogent attack on the problem of relation updates, let alone development of a solution to the problem, it is somewhat tangential to our present topic and I won't discuss it further here (short of the last footnote). Early chapters in this book will have already clarified the issues, and I presume you would not have reached the present chapter with such misunderstandings lingering.

Set Semantics

Failure to understand that *set semantics* are inherent in the RM leads to many problems. An inability to understand the solution to relation updating I originally proposed with Date is not the least of them. By set semantics I mean those characteristics of set theory that place a limitation on its possible applications. Set semantics is a requirement imposed by the limitations of the formal language of set theory (the expressive power of which we have discussed elsewhere in this book). For one specific example, note that the RM requires that relational operators always act on *sets* of tuples, never tuples. We never specify a particular tuple (i.e., row) per se, but rather a set defined so as to contain that particular tuple. While the effect of an update may be that exactly one particular tuple seems to be altered of all the tuples in the relation, that effect – however desirable and useful – is merely artifact from the set theoretic perspective inherent in the RM. Another limitation imposed by set semantics is the inability to express the concept of a computer variable to which values can be assigned (or "updated").[9] I'll examine this limitation in more detail below.

The most prevalent misunderstandings are due in large part to the subtle influence of physical, procedural thinking on interpretations of the relational model ("RM"). Such influences can be subtle, causing researchers as well as the non-technical user to think of the RM with improper expectations. The RM is a logical, set-oriented data model. The RM is not a storage model or even a prescription for physical data structures[10] (i.e., the RM does not prescribe any particular physical data structure for its physical implementation). The structural aspects of the RM – domains, attributes, tuples, relations, databases, and the various constaints that interrelate them are logical abstractions *only*.

It has been the task of relational theorists to show that these logical abstractions suffice as a way of losslessly representing the semantic content and relationships among the elements of virtually

map between the relational representation and those physical, stored representations, business and IT requirements may be more demanding than the SQL DBMS can meet. Again, using the available commercial SQL DBMSs may be impractical given time and budget constraints in a business project. "Non-relational" data is then a misnomer – it is the SQL DBMS that is non-relational with respect to relational support for application and business requirements.

[9] Please note that I am not saying that programmatic variables cannot be expressed in logic, only that they cannot be expressed in elementary set theory or first order predicate logic. Other, more expressively powerful, systems are required. Unfortunately, such formal systems do violence to the relational model and its intent.

[10] All data structures are physical, so this is a little redundant. RM is purely logical (abstract) so the corresponding concept is *data relationships*. Logical "structures" are merely pictorial representations of relationships.

every known physical data structure.[11] We can think of that logical abstraction as a sufficiently powerful, standardized way of representing information or even knowledge – a *canonical* representation. Once we have a canonical representation, the transformation to and from any specific physical data structures can be completely hidden from users and user-written applications, providing physical data independence. Consistent with set semantics, we will not know how many instances of some physical data structure (e.g., records, key-value pairs, documents, nodes, lists, and so on) are involved when we logically manipulate the data. The RM's query language must therefore be declarative – addressing only logical abstract relationships – and not imperative (i.e., procedural – for example, involving loops and physical structure "navigation"). Procedural handling of physical data structures is then handled "under the covers" and completely hidden from users. That means it can be changed or optimized without requiring changes to application level or user level commands.

Even the most committed relational researchers have innocently perceived non-existent relational clothing on naked physical concepts, and then pondered the resulting anomalies. For example, the view updating problem arose in large part because *row-level* updates were accepted as relational[12], rather than as physical record processing characterized in relational words. Indeed, almost all of the attempts to address view updating prior to 1994, some proposals published long after 1994, and virtually all the implementations in commercial products past and present have understood the problem of view updating to be the same as propagating changes of specific rows in views to the underlying base relations. They *should* have understood it to be the problem of how to propagate a declarative and logical change to the relevant *sets* (i.e., the relations) instead of the *members* of those sets (i.e., individual tuples).

In this chapter, I discuss certain consequences of the above-mentioned misunderstandings, specifically as they relate to updating relations. I develop and restate the generalized relation update algorithm, which applies uniformly to any combination of derived and stored relations. In consequence, we will see that the answer to the title question is both "yes" and "no": We can achieve the goal of relational updates, but only in a manner that provides the DBMS with the power to perform that update.

Relational Updates

The concept of an updatable relation is a little odd. A relation is a set. Relational theory, and so the RM, is a particular expression of standard finite set theory and first order predicate logic. These two formal systems are compatible in the sense that standard finite set theory, its axioms and its theorems can be expressed using the language of first order predicate logic. The

[11] This does not mean that every computational algorithm on those physical data structures has an abstract relational counterpart, nor should they. The relational algebra is intentionally less powerful than a computationally complete imperative language (see the chapters that explain first order predicate logic, higher ordered languages, and computational complete languages, and requirements for declarative database languages).
[12] Row-level updates are most emphatically are not consistent with the RM, as I will explain – again – below.

semantics (i.e., the valid interpretations of the theory) do not include any concept of an object than can change, nor can any symbol in the object language be given a time-varying instantiation. Yet the RM would be of little practical use if we had no notion of updating a relation. So how do we resolve this apparent inconsistency? The answer requires understanding that the notion of an updatable relation is a simplifying gloss that hides considerable set theoretic detail. Unfortunately, the informal terminology in common use – and introduced by Codd – suggests formal expressibility (expressive power of the formal language) of which relational theory *per se* is simply incapable. We must be careful not to be led down the primrose path suggested by the terminology and import non-first order languages into the RM.[13]

Note: I understand that the discussion which follows can be "dense" for readers unfamiliar with symbolic arguments to follow. As time permits, I intend to provide Venn diagrams, incorporating them as figures in a future version of this chapter. In the meantime, I urge the reader who has difficulty following the symbolic presentation to sketch such diagrams for themselves as they read, and to refer back to them as needed.

Every relation $\{\boldsymbol{R}\}$[14], being a set, has a membership function. When applied to the Universe of Discourse $\{\boldsymbol{U}\}$, that membership function selects or identifies a homogenous set of tuples in $\{\boldsymbol{U}\}$ in that satisfy they the membership function. This set, which we will denote as $\{\boldsymbol{R_U}\}$[15], is the maximal collection[16] of tuples that could possibly satisfy the membership function given the scope and restrictions of the intended application[17]. Please note that this is not a "universal relation" but the "Universe of Discourse restricted to relation $\{\boldsymbol{R}\}$." In the usual understanding of a relation, it is standard to say – following Codd – the relation is a time-varying set, meaning that we use the same name to designate every member of a time-ordered sequence of sets and treating evaluation of $\{\boldsymbol{R}\}$ as being a function of time.

Let's denote a particular instance of this time-ordered sequence as $\{\boldsymbol{R_t}\}$ with **t** being an ordered index. Each $\{\boldsymbol{R_t}\}$ (meaning the set of tuples both individually and collectively) must satisfy the

---

[13] See the detailed discussion of expressibility, completeness, decidability, and consistency provided in other chapters.

[14] Note that I am deliberately using set notation $\{\boldsymbol{R}\}$ - rather than just a symbol $\boldsymbol{R}$ – to emphasize the set character of every relation $\{\boldsymbol{R}\}$. Also, note that I am not using $\{\boldsymbol{R}\}$ to denote what is commonly called the "state" of a relation.

[15] Note also that I am avoiding a theory-laden functional notation such as $\{\boldsymbol{R(_U)}\}$which would imply a language more powerful than that of simple set theory and an abstract object $\{\boldsymbol{R(_U)}\}$that is not a simple set.

[16] Herein, I use the terms "collection" and "set" somewhat differently. I will use the term "set" when the members have a defining set of properties and a relationship among those properties which is, at least in principle, computably verifiable. I use the term "collection" in a more general way, to denote a union of the members of such sets and possible other, less well-defined members. Note, however, that I do not use the term "collection" to denote a set of sets.

[17] Please note that I am not using the term "application" as a computer program or even a set of programs, but as a generic name for the intended interpretation and particular use of RM.

membership function of $\{\boldsymbol{R_U}\}$. Of crucial importance, however, is that the membership function for $\{\boldsymbol{R_t}\}$ differs from the membership function for $\{\boldsymbol{R_U}\}$ by the addition of an formal conjunct which I call the "assertion predicate". and, additionally, the instantiated predicate representing that membership function as applied to any tuple of the set must have been asserted to be "true"[18] for time ***t*** by a suitably authorized user of the DBMS managing the database containing $\{\boldsymbol{R_t}\}$. Each $\{\boldsymbol{R_t}\}$ is a subset of $\{\boldsymbol{R_U}\}$. (Can you now explain why?)

Thus, there is a complement of $\{\boldsymbol{R_t}\}$ relative to $\{\boldsymbol{R_U}\}$, denoted by $\{\boldsymbol{R_U}\}\backslash\{\boldsymbol{R_t}\}$. For lack of a better term, we will call this the *state complement* of $\{\boldsymbol{R_t}\}$. The *state complement* of a relation is extremely important to a proper understanding of updating relations, as we will see below. Indeed, they must be considered carefully whenever set tautologies involving set difference or complement become a part of our reasoning. A similar notion may be defined for a database if we think of it as a so-called "time-varying set". Note that this conception requires some care if we are to cover the prospect of relations and domains being added and droped, or new constraints being identified, a complication I will not address here.

Anytime someone uses such terminology, remember that it brings a ton of unexplained baggage into the formal discussion. If $\{\boldsymbol{R}\}$is said to denote a time-varying relation, it is really denoting an ordered collection of relations $\{\{\boldsymbol{R_1}\},\{\boldsymbol{R_2}\},\ldots,\{\boldsymbol{R_t}\},\ldots\{\boldsymbol{R_N}\}\}$ that have the same membership function, but distinct memberships (or "values") distinguished solely by which assertion predicates are evaluated as "true". DBMS software typically sees to it that the "current" relation $\{\boldsymbol{R_t}\}$ in the collection is accessed at any given time, giving the appearance that $\{\boldsymbol{R}\}$ is a variable. (Why? Because the named $\{\boldsymbol{R}\}$ seems to be able to change values over time just as a programming variable would.) However, this is done "under the covers" so that the relational algebra (the formal system) never deals with such relations as variables.

Most DBMSs offer no ability to access any other member of the collection $\{\boldsymbol{R_t}\}$ and, if they do, the relational algebra must *in effect* treat them as multiple, distinct relations, not multiple "values" of a *single* "time-varying" relation. That some so-called temporal database management systems (or some theoretical approach suggested in the literature) suggests otherwise is a continuation of corrupt semantics and lack of clarity regarding the metamathematical properties of the language(s) so created.

That said, note that nothing in the expressive power of a formal first order theory like that underlying the RM gives us the right to treat the ordered collection $\{\{\boldsymbol{R_1}\},\{\boldsymbol{R_2}\},\ldots,\{\boldsymbol{R_t}\},\ldots\{\boldsymbol{R_N}\}\}$ as a formal object in its own right: It is not a variable like a variable in a programming language (what Date and Darwen would call a "relvar") to which one can assign a value – such concepts require at least second order languages if not even more powerful languages like the lambda calculus. And such languages destroy any expectations of well-behavedness users of relational

[18] By "true" we mean, of course, that that user is making an *assertion* that the tuple represents a state of affairs that is correct with respect to the application. It does not mean that it the tuple represents the "real world", whatever you consider that to be.

theory might otherwise expect or require.[19] At best, a symbol or name in relational theory designates a value. The symbol or name does *not* identify an address of some conceptual container, as it might in an imperative programming language. (Just to be clear, I am not opposed to the languages having programmatic variables or to their conceptual use in understanding how databases are managed: I am opposed to formalizing such concepts and introducing them into the formal language of RM.)

The value identified by a relational symbol must not be and cannot be "altered" during any deduction or other use of the relational algebra, <u>including</u> to represent updates. The semantics of the language simply do not support it. Instead, an update of a relation is actually the generation of a new logical specification or expression for the resulting relation based upon the specification of the initial relation. Evaluation (which *might* be said to occur in the relational calculus, but certainly not in the relational algebra) then tells us which tuples belong to the resulting relation. Using an operation that is not part of the relational algebra we use renaming to call the resulting relation by the same designation as the initial relation. A typical DBMS discards the initial relation through an "under the covers" mechanism. All of these operation can be given expression in a powerful language like the lambda calculus, but that would defeat at several of the intended benefits of the RM. (See footnote 19 above.)

Nonetheless, it is incorrect to say that RA does not have the expressive ability to convey some model of what we call an update. Even if you insist on thinking in terms of relvars, the new value to be assigned to the relvar is that resulting from evaluation of a relational expression. Recasting this in terms of an expression in set semantics yields a set mapping – an equivalennce relationship: The relational expression on the right hand side of the mapping is equivalent (but *not* assigned) to the relation on the left hand side. Note that the relational expression might include "constants." For example, a set of specific tuples might be incorporated in the expression, thereby defining a relation extensionally.

<u>Aside</u>: As discussed elsewhere, the key concept that must be understood in order to discuss relation updatability is that of a relation predicate – i.e., set membership functions as applied to relations. It is a crucial task of database design to formalize the membership function or relation predicate ***PR*** of every relation $\{\boldsymbol{R_U}\}$ in a schema. Notice that ***PR*** is the membership function $\{\boldsymbol{R_U}\}$, which is the union of all *permissible* relations $\{\boldsymbol{R_t}\}$. For every $\{\boldsymbol{R_t}\}$, there exists a relation predicate $\boldsymbol{PR_t}$ which is just ***PR*** with an additional conjunct $\boldsymbol{A_t}$ which I call the *assertion predicate*. Conceptually, every tuple belonging to $\{\boldsymbol{R_t}\}$ is one for which the corresponding $\boldsymbol{A_t}$ (tuple) has been set to a logical constant representing "true" by a suitably authorized user of the DBMS (as, for example, when a user "inserts" a tuple into $\{\boldsymbol{R_t}\}$). Alternatively, every tuple that

[19] I explore these issues in considerable detail in elsewhere this book. However, to reiterate, the value of the RM lies strongly in its ability to offer a declarative query language. Declarative query languages must be decidable and consistent, and preferably expressively complete. Programmatic variables require at a least second order languages and second (and higher) order languages (like functional programming) are not both decidable and consistent.

does not belong to $\{\boldsymbol{R_t}\}$ is one for which $\boldsymbol{A_t}$ (tuple) has been set to a logical constant representing "false" by a suitably authorized user of the DBMS (as, for example, when a user "deletes" a tuple from $\{\boldsymbol{R_t}\}$). Please note that a relation predicate contains all the schema information that Date has traditionally identified as a "header" and, inasmuch as it also captures all relevant constraints, is much more. In particular, it captures the relation's *semantics*.

Let's begin the analysis of relation updatability with a little agreement about how I promise to use terminology. Whenever I use the term "update" in lower case, I will mean *any* conceptual modification to a database value, whether those modification results in the (1) deletions of sets of tuples ("tuple delections"), (2) insertions of sets of tuples ("tuple insertions"), or (3) some combination of tuple deletions and tuple insertions that effectively results in "tuple updates" (i.e., changes to non-key attribute values in tuples selected by some immutable key attribute or attributes). An arbitary combination of any of the foregoing is also possible, though we see no reason to call out these as separate kinds of *individual* updates since the foregoing suffice for expressive completeness of update possibilities.

By contrast with the general term "update", I will also use the terms "UPDATE", "INSERT" and "DELETE" though probably not as you might expect. In particular, when I use the term "INSERT" in upper case, I will mean a modification of a set that specifically results in tuple insertions (i.e., the "addition" by union with the initial database of a set of zero or more tuples selected from the complement of the initial database relative to its Universe of Discourse, that set of zero or more tuples satisfying some logical expression[20]) and no other kind of modification. When I use the term "DELETE" in upper case, I will mean a modification of a set that specifically results in tuple deletions (i.e., the "removal" by set difference of a subset of the initial database of a set of zero or more tuples and satisfying some logical expression) and no other kind of modification. When I use the term "UPDATE" in upper case, I will mean a modification of a set that specifically results in tuple updates (i.e., changes to non-key attribute values in zero or more tuples selected by logical expression).[21] Unless I explicitly state an expression incorporating any of the terms "UPDATE", "INSERT" or "DELETE" is, for example, a "pseudo-SQL" example, I specifically do *not* mean the terms as used in any particular query language such as SQL.

We begin this analysis of relational updates by asserting a RM ***update policy*** (not a definition) to guide the analysis:

---

[20] I use the term logical expression here rather than relational expression because I want it to be compatible with and combinable with relation predicates as necessary. In principle, I could express all relation predicates and expressions in the relational algebra, but the deductive apparatus is then a bit more cumbersome.

[21] Note that such an UPDATE operation has an important property. Designate the initial set by $\{\boldsymbol{S_i}\}$ and the final set – after UPDATE of non-key attributes $\{\boldsymbol{a}\}$ – as $\{\boldsymbol{S_f}\}$. Then, the relational projection of $\{\boldsymbol{S_i}\}$ over key attribute(s) $\boldsymbol{k}$ must be identical to the relational projection of $\{\boldsymbol{S_f}\}$ over key attribute(s) $\boldsymbol{k}$: $\boldsymbol{\Pi_k}\{\boldsymbol{S_i}\} = \boldsymbol{\Pi_k}\{\boldsymbol{S_f}\}$.

Update Policy 1:

Relations can only be updated as a consequence of updating a database comprising those relations.

On its face, this assertion may seem overly constraining and perhaps even disabling. However, I will argue that exactly the opposite is true: This policy is enabling. Furthermore, it is a consequence of the RM and one that will lead us to a general mechanism for "modifying" relations. But first, here is an expanded version of the update policy that provides a little better explanation of what I mean by an update:

Update Policy 1 (Expanded Version):

An update is an operation on an initial database that has as its result one or more sets (in combination thereby constituting a new database), each of those sets:

(1) having the properties of a relation,
(2) being derived by applying set operations to sets in the initial database, and
(3) selected by a logical expression which expresses the membership functions of those sets.

Exactly one such operation, which I will call "update", need be defined. I will define this operation in such a way that tuple deletions and tuple insertions are special cases of tuple updates. Mathematically, these special cases are "degenerate".

To Know A Thing, You Must Also Know What It Is Not

In the analysis that follows, we will be discussing certain properties of sets. Ultimately, we will specialize those sets to relations. All of these sets, then, are discrete and have finite cardinality. We use the notation $\{\boldsymbol{Z}\}$ to denote a set called "$\boldsymbol{Z}$". For purposes of simplifying the exposition, we assume (and later insist) that every set has a well-defined membership function and that every set $\{\boldsymbol{Z}\}$ is a proper subset of some specified and unique collection $\{\boldsymbol{Z_U}\}$, denoting the Universe of $\{\boldsymbol{Z}\}$ (a set) for the discussion or analysis at hand. Then $\{\boldsymbol{Z}\}$ and its relative complement $\{\boldsymbol{Z_U}\}\backslash\{\boldsymbol{Z}\}$ are not functionally dependent in the sense that no tuple(s) contained in the relative complement $\{\boldsymbol{Z_U}\}\backslash\{\boldsymbol{Z}\}$ are computably determined by any set of tuples in $\{\boldsymbol{Z}\}$ and vice-versa. That is, $\{\boldsymbol{Z_U}\}$ can be partitioned into two parts, one labeled $\{\boldsymbol{Z}\}$ and one labeled $\{\boldsymbol{Z_U}\}\backslash\{\boldsymbol{Z}\}$), but there is no algorithm by which we can compute the members of $\{\boldsymbol{Z}\}$ from knowledge of the members of its relative complement $\{\boldsymbol{Z_U}\}\backslash\{\boldsymbol{Z}\}$ alone (without knowledge of $\{\boldsymbol{Z_U}\}$). Likewise, there is no algorithm by which we can compute the members of $\{\boldsymbol{Z_U}\}\backslash\{\boldsymbol{Z}\}$ from knowledge of the members of $\{\boldsymbol{Z}\}$ alone. In other words, we need to have specified at least two of the three sets $\{\boldsymbol{Z}\}$, $\{\boldsymbol{Z_U}\}\backslash\{\boldsymbol{Z}\}$, and $\{\boldsymbol{Z_U}\}$ to be able to evaluate any set theoretic expressions involving them. All too often, both relational theorists, practicing database designers, and query writers fail to observe this stricture, resulting in ambiguous expressions in the relational algebra.[22]

Set Transformations Are Relationships Between Sets, Not Set "Constructors"

In set theory and therefore in a relational database, updates are set transformations. That is to say, what we informally call an update is actually a way of deriving a new set from an existing set. The new set does not replace the old set: Concepts such as "replacement" or "assignment" lie outside standard set theory. Put another way, an update is realized in set theory as a mapping between two sets which most practitioners tend to think of, rather informally and imprecisely, as the input set and the output set, or even more commonly as the initial state of the relation and the final state of the relation.[23]

We can thus characterize an update as follows. Let each $\{\boldsymbol{DB}_\mathrm{i}\}$ and $\{\boldsymbol{DB}_\mathrm{j}\}$ be a heterogeneous collection of tuples, where $\boldsymbol{i}$ and $\boldsymbol{j}$ are used as arbitrary indices of such collections. Let $\{\boldsymbol{X}_\mathrm{i}\}$ and $\{\boldsymbol{Y}_\mathrm{j}\}$ represent subcollections of $\{\boldsymbol{DB}_\mathrm{i}\}$ and $\{\boldsymbol{DB}_\mathrm{j}\}$ respectively, and $\{\boldsymbol{DB}_\mathrm{i}\}\backslash\{\boldsymbol{X}_\mathrm{i}\}$ represent the subcollection that is the complement of $\{\boldsymbol{X}_\mathrm{i}\}$ *relative to* $\{\boldsymbol{DB}_\mathrm{i}\}$ and $\{\boldsymbol{DB}_\mathrm{j}\}\backslash\{\boldsymbol{Y}_\mathrm{j}\}$ represent the subcollection that is the complement of $\{\boldsymbol{Y}_\mathrm{j}\}$ *relative to* $\{\boldsymbol{DB}_\mathrm{j}\}$. Further, let $\{\boldsymbol{DB}_\mathrm{i}\}$ and $\{\boldsymbol{DB}_\mathrm{j}\}$ each be proper subsets of a Universe $\{\boldsymbol{U}\}$.

We are now in a position to formally characterize an update of a database as a transformation. Specifically, we have:

Definition: A *database update* is a transformation $\boldsymbol{T}$ that acts on database $\{\boldsymbol{DB}_1\}$ to produce $\{\boldsymbol{DB}_2\}$, symbolically – $\{\boldsymbol{DB}_1\} =\!\boldsymbol{T}\!\!=> \{\boldsymbol{DB}_2\}$ – if and only if there exists subcollections $\{\boldsymbol{X}_1\}$ and $\{\boldsymbol{Y}_2\}$ such that:

$\{\boldsymbol{X}_1\} =\!\boldsymbol{T}\!\!=> \{\boldsymbol{Y}_2\}$ (i.e., $\boldsymbol{T}$ acts on $\{\boldsymbol{X}_1\}$ to produce $\{\boldsymbol{Y}_2\}$)

and

$\{\boldsymbol{DB}_1\}\backslash\{\boldsymbol{X}_1\} \equiv \{\boldsymbol{DB}_2\}\backslash\{\boldsymbol{Y}_2\}$ (i.e., the relative complement remains unchanged)

That is, there must some subcollection $\{\boldsymbol{X}_1\}$ of $\{\boldsymbol{DB}_1\}$ and some subcollection $\{\boldsymbol{Y}_2\}$ of $\{\boldsymbol{DB}_2\}$ such that the mapping $=\!\boldsymbol{T}\!\!=>$ is a mapping between those subcollections and the portions of $\{\boldsymbol{DB}_1\}$ not mapped by $=\!\boldsymbol{T}\!\!=>$ to $\{\boldsymbol{DB}_2\}$ must be also be in $\{\boldsymbol{DB}_2\}$ (i.e., unchanged).[24]

The transformation $\boldsymbol{T}(\{\boldsymbol{X}_1\})$ is a logical expression that converts the logical expression $(\{\boldsymbol{X}_1\})$ which selects or identifies $\{\boldsymbol{X}_1\}$ – this being a subcollection of relations (and associated attributes) contained in $\{\boldsymbol{DB}_1\}$ – and produces a second logical expression $(\{\boldsymbol{Y}_2\})$ which selects

[22] SQL's "grammar" and nulls aggravate the situation, leading to unintended and sometimes incomprehensible results.
[23] Such understanding is informal because it is not expressible in set theory and imprecise because it does not uniquely specify the set-to-set mapping involved in the "update".
[24] For those more familiar with mathematics, we say they are *invariant* (i.e., unchanged) under the mapping $\boldsymbol{T}$.

or identifies a resulting subcollection of relations $\{\boldsymbol{Y}_2\}$ contained in $\{\boldsymbol{DB}_2\}$. To put it another way, the transformation $\boldsymbol{T}()$ creates the membership functions for $\{\boldsymbol{Y}_2\}$ from the membership functions for $\{\boldsymbol{X}_1\}$.

The language of set theory provides no means to express concepts like creating a set, destroying a set, or altering a set. Set theoretic language is not powerful enough for this purpose and so establishes a hard limit on set theoretic semantics.[25] Instead, it describes how sets are – or are not – related to each other. A set, possibly empty, can be derived from other sets by establishing a mapping relationship between them. In order to be consistent with the semantics of set theory, we must think of $\{\boldsymbol{DB}_1\}$ and $\{\boldsymbol{DB}_2\}$ as both pre-existing and post-existing (at least in principle) so that – to emphasize the point – $\boldsymbol{T}$ expresses a mapping relationship between them.

The subcollection of relations $\{\boldsymbol{Y}_2\}$ may be specified in terms of $\{\boldsymbol{X}_1\}$[26] in any computable form including (1) denotation of an explicit set of relations (i.e., values that serve as constants), (2) relational expressions possibly incorporating computable functions (i.e., derived relations), or (3) some combination of the foregoing. In effect, the specification for $\{\boldsymbol{Y}_2\}$ is also a logical expression, possibly derived from the logical expression for $\{\boldsymbol{X}_1\}$. $\boldsymbol{T}$ tells us how $\{\boldsymbol{DB}_1\}$ and $\{\boldsymbol{DB}_2\}$ are related, and how to compute $\{\boldsymbol{DB}_2\}$ from $\{\boldsymbol{DB}_1\}$.

A Database Can Be Analyzed (Decomposed) Into Relations

Each homogeneous set of tuples that satisfy the first logical expression is a relation.[27] By a homogeneous set of tuples I mean a set of tuples having the same set of attributes and the same set of constraints insofar as the DBMS is able to determine from the information available to it.[28] By way of illustration, suppose we have a simple database ***D*** comprising tuples of various degree but all of them have the attribute ***A***. Then, a logical expression ***P(A)*** that refers only to values of the attribute ***A*** (i.e., ***P(A)*** is a 1-place predicate in ***A***) is then *potentially* satisfied by every tuple in ***D***:[29] No matter what other attributes are included in a tuple, if ***P(A)*** does not refer to them, they

[25] Recall that the semantics of a formal system is a characterization of the valid models of the formal system – those interpretations such that the formal system faithfully reflects the objects, operations, and relationships of that model. Clearly, if the formal system's language cannot express a concept, then no model requiring that concept can be a valid model of the formal system.

[26] Note that we have defined $\{\boldsymbol{X}_1\}$ such that it includes every portion of $\{\boldsymbol{DB}_1\}$ on which $\boldsymbol{T}$ depends. Similarly comments apply for $\{\boldsymbol{Y}_2\}$ so long as self-reference is excluded.

[27] Of course, logical expressions can be very complex containing bound predicate variables with each having a particular scope. For purposes of this discussion, I will ignore this complication, assuming that the reader understands that each bound predicate variable represents a different set of tuples that are independent of other such sets unless the logical expression includes some specific relationship among them.

[28] For all theoretical and practical purposes, any alleged constraint not expressed in a form computable by the DBMS and not made known to the DBMS *does not exist*. It has no place in any analysis or discussion of how the DBMS does or should behave.

[29] In the general case, we can compose a logical expression that is an n-place predicate that is satisfied by or *selects* any set of n-tuples in D. This is a consequence of Leibniz Law.

can have no affect on the truth or falsity of ***P(A)*** when it is evaluated for any tuple (with its specific value of ***A***) in ***D.***

A logical predicate given in first order predicate logic[30] and serving as the membership function of the set must be *the only way* in which relations in a relational database are considered to have been identified to the DBMS. Convenience of exposition (and usage) may dictate the use of a symbol or “name“ in place of the predicate that defines the universe of permissible values of a relation (i.e., its membership function). However, such pragmatic symbolism should never be used in computation (definition, selection, or updates) because two distinct relation symbols ("relvars" if you must) may not represent distinct predicates.[31] Even worse, two uses of the same relation symbol, one bound by a logical quantifier and the other not, must be evaluated differently – they must not be equated. The relationship of two occurrences of relation symbols can only be understood by examining, comparing, and combining the corresponding predicates (that define them) in the context of the logical expressions in which they appear. The logical predicate is then an *effective procedure* for identifying zero or more relations in a database (which is a heterogeneous collection of tuples).

Now let's continue developing our model of database updates. For convenience in this section, I will transition to functional notation in place of the mapping notation used earlier. As no doubt the reader knows, functional notation permitts nested expressions (composition) and the LHS is understood as the result of evaluating the RHS. Let $\{\boldsymbol{DB}_j\} = \boldsymbol{T}(\{\boldsymbol{DB}_i\})$. Then, for every ***T*** there exists a subcollection $\{\boldsymbol{S}\}$ of $\{\boldsymbol{DB}_i\}$ such that:

$$\{\boldsymbol{DB}_i\} \text{ MINUS } \{\boldsymbol{S}\} \text{ UNION } \boldsymbol{T}(\{\boldsymbol{S}\}) \equiv \boldsymbol{T}(\{\boldsymbol{DB}_i\}) \qquad (1)$$

<u>*Aside*</u>: Constraints need not be "checked" as a separate operation in this update model.[32] They are incorporated into the appropriate logical expressions for each set or collection. In this way, there is no issue of "failing" a constraint during an update. Rather, the constraint may be understood as restricting the membership of the set such that the remaining members will be "successful" under the update. For example, suppose that a transformation ***T*** is applied to a database $\{\boldsymbol{DB}\}$

---

[30] We explain elsewhere in the book from which this extract was taken that the expressive power of at least first order predicate logic is required, that second order leads to dire consequences, and that evaluation of first order predicate expressions is always limited to finite formulas with variables ranging over finite, enumerable domains. These facts guarantee properties of the formal system that are not only desirable, but necessary for the DBMS to function without *ad hoc* human intervention.

[31] If desired, we could certainly design a DBMS to enforce a "one name per predicate" rule or warn that it is being violated. However, enforcing such a rule might not be desirable – after all, synonyms can be useful. For example, we might give a relation two textual names such as SUPPLIERS – possibly more natural for inventory management – and VENDORS – possibly more natural purchasing.

[32] For the purpose of a proposal such as that of DavidMartinenghi's dissertation ("Advanced Techniques for Efficient Data Integrity Checking", 2005), the combined constraints and updates might be checked in advance for the purpose of improving efficiency (a physical issue).

consisting of a homogeneous collection of tuples – i.e., $\{\boldsymbol{DB}\}$ consists of a single relation $\{\boldsymbol{R}\}$ (ignoring its generalizations, projections, and restrictions for now). If a contradiction exists between the expression $\boldsymbol{T}$ and the defining predicate for $\{\boldsymbol{R}\}$, then the set of tuples selected will be empty and $\boldsymbol{T}$ will have no effect. No "error" can occur. Matters are a bit more complex is $\{\boldsymbol{DB}\}$ has a so-called "transition constraint" or if it consists of two relations $\{\boldsymbol{R1}\}$ and $\{\boldsymbol{R2}\}$ with a constraint among them. In such cases, the proposed value of $\boldsymbol{T}(\{\boldsymbol{DB}\})$ must, in general, be computed before such constaints can be checked.[33]

Make special note that a logical expression applied to $\{\boldsymbol{DB}\}$ selects _all_ those homogeneous sets of tuples that satisfy it. For example, assuming there is no difference between the predicates for relations S (SNO, CITY, STATUS) and SC (SNO, CITY) except for conjuncts involving STATUS, there is no way to construct a logical expression that distinguishes between these tuples without referencing STATUS.

Consider what happens in expression (1) above when $\{\boldsymbol{DB}_\mathrm{i}\}$ MINUS $\{\boldsymbol{S}\} \equiv \{\boldsymbol{DB}_\mathrm{i}\}$, that is $\{\boldsymbol{S}\}$ is not contained in $\{\boldsymbol{DB}_\mathrm{i}\}$. Then, we have:

$$\{\boldsymbol{DB}_\mathrm{i}\} \text{ UNION } \boldsymbol{T}(\{\boldsymbol{S}\}) = \boldsymbol{T}(\{\boldsymbol{DB}_\mathrm{i}\})$$

In other words, our generalized update degenerates into an INSERT.

Similarly, when $\{\boldsymbol{DB}_\mathrm{i}\}$ UNION $\boldsymbol{T}(\{\boldsymbol{S}\}) = \{\boldsymbol{DB}_\mathrm{i}\}$ in expression (1), we have:

$$\{\boldsymbol{DB}_\mathrm{i}\} \text{ MINUS } \{\boldsymbol{S}\} = \boldsymbol{T}(\{\boldsymbol{DB}_\mathrm{i}\})$$

In other words, the update degenerates into a DELETE. This occurs when $\boldsymbol{T}(\{\boldsymbol{S}\})$ is singular and maps alls tuples in $\{\boldsymbol{S}\}$ to the empty set. In practice, we usually indicate this by simply leaving out the subexpression UNION $\boldsymbol{T}(\{\boldsymbol{S}\})$.

From the above, we can see that the update expression (1) implements an UPDATE when both $\{\boldsymbol{DB}_\mathrm{i}\}$ MINUS $\{\boldsymbol{S}\}$ and UNION $\boldsymbol{T}(\{\boldsymbol{S}\})$ are non-empty. Further, the update an UPDATE degenerates into a DELETE or an INSERT when $\{\boldsymbol{S}\}$ or $\boldsymbol{T}(\{S\})$ are empty, respectively. Note that, although the UPDATE has a delete phase and an insert phase, this does *not* imply that

[33] Such deferred or "end of transaction" checking of constraints is necessary, in part, because it does not appear to be possible to provide a constraint checking algorithm that will predict in polynomial complexity whether or not an arbitrary constraint (first order formula) involving more than one relation will be satisfied by the database. Instead it may be less complex to simply perform the evaluation (i.e., compute the resulting set(s) and then check whether or not the constraint expression is satisfied). Deferred checking also avoids the complexities of interdependencies that might cause certain operations to be non-commutative or non-distributive such that the result depends on operation order. That said, an approximately general solution (e.g., Martineghi, 2005) is still worthy for those cases in which it will work as long as the non-applicable cases can be definitively identified in advance and deferred then used.

UPDATE is a DELETE followed by an INSERT.[34]

Now let's consider an example, namely updating a database $\{\boldsymbol{DB}_1\}$ containing the relations SS (SNO, CITY, STATUS) and SC (SNO, CITY). The update algorithm for transformation $\mathcal{E}$ is conceptually as follows (recalling the definitions of $\{\boldsymbol{DB}_1\}$ and $\{\boldsymbol{S}\}$ given above):

```
Select all tuples from $\{\boldsymbol{DB}_1\}$ consistent with $\mathcal{E}$
Partition those tuples into sets of homogeneous tuples
For each set $\{\mathbf{R}\}$ of homogenous tuples
    Logically combine the defining predicate for $\mathbf{R}$ into $\mathcal{E}$
    Compute $\{\boldsymbol{S}\}$ and $\mathcal{E}(\{\boldsymbol{S}\})$
    Compute $\mathcal{E}(\{\boldsymbol{DB}_1\})$
End
Compute the restriction of $\mathcal{E}(\{\boldsymbol{DB}_1\})$ [call this $\boldsymbol{C}(\mathcal{E}(\{\boldsymbol{DB}_1\}))$] under the
    conjunct of relevant multi-relation constraints $\boldsymbol{C}$.
If $\boldsymbol{C}(\mathcal{E}(\{\boldsymbol{DB}_1\})) == \{0\}$, then $\boldsymbol{DB}_2 = \boldsymbol{DB}_1$, else $\boldsymbol{DB}_2 = \mathcal{E}(\{\boldsymbol{DB}_1\})$
```

Points Arising:

Now lets consider certain points regarding the update of SS and SC in $\boldsymbol{DB}$.

(1) Note that either $\mathcal{E}(\{\boldsymbol{DB}\})$ or $C(\mathcal{E}(\{\boldsymbol{DB}\}))$ may be the empty set.

(2) By using the defining predicates for tuples instead of relation names SS and SC, if $\mathcal{E}$ references tuples containing the attribute SNO, then it references both SS.SNO and SC.SNO.

(3) The tuples of $\{\boldsymbol{S}\}$ and $\mathcal{E}(\{\boldsymbol{S}\})$ are different for each relation – thus there is no concern about any attempt to insert a tuple (i.e., a single-tuple set) into relation SS that does not have a STATUS value during an UPDATE.

(4) There are no "compensatory actions", nor are any necessary. In effect, we are treating the database $\{\boldsymbol{DB}_1\}$ as if it were a view and letting the update propagate to the contained base relations as appropriate.[35]

[34] I leave it as an exercise to explore the various ways in which equating UPDATE with DELETE followed by INSERT is flawed. Hint: How does the INSERT make certain that the set of keys for the inserted tuples are the same as those for the set of keys for the previously deleted tuples without reference to the before image and the after image of each operation?

[35] For those unfamiliar with my view updating algorithm, I suggest reading US Patent 7,263,512, available at www.AlternativeTech.com/ATpubs_dir.html. In due course, a chapter will be written on the subject consistent with the style of the rest of this book.

(5) The issue of views versus base relations is irrelevant, since any reference to a view can be "unwound" so as to be a reference to the "base" relations on which that view is defined. If constraints are applied to views, in which case they must be checked at the properly level of deriving those views in $\mathcal{E}(\{\boldsymbol{DB}\})$.

(6) As long as a transaction contains only relational algebra statements that are commutative (order independent), then there exists a view that can take the place of that transaction. Developing an algorithm for translating such transactions into updates on views is a planned project, and should not be too difficult.

I have not attempted an explanation using so-called "relational assignment" (assigning a value to a symbol denoting relation) in this presentation for two reasons. First, I do not know how to do so without considerable effort. I simply don't think about the problem that way. Second, I have considerable doubt as to whether it can be done with consistent semantics for the assignment operation. *Relational assignment* in the sense used by Codd must be subject to the same concerns raised above regarding *updatable relations*. In other words, it is a gloss – what is sometimes called "syntactic sugar" – on the more detailed set theoretic relationships. In any case, what is needed is what might be called *database assignment* since the database is the appropriate level for an update that spans or potentially affects multiple relations.

Aside 1: Once again, set theory does not have assignment or "variables" (a symbol is merely a designation a fixed set), or any notion of set that can change or morph or be "updated". Introducing these notions requires a powerful language capable of constructs like "the set containing only those sets that do not contain themselves." For these and other reasons, I have used set theoretic semantics and not attempted a relational assignment formulation. Of course, I am open to and even eager to see such a formulation if anyone thinks it can be done without simultaneously destroying – for example – the decidability of the relational algebra[36], decidability being a necessary property of a declarative query language.

Aside 2: I see no reason to be concerned with whether the user is or is not aware of something. Users may be surprised by hidden data or constraints, just as they will be if they do not understand logical expressions for the data and the defining predicates (membership functions) for relations. To avoid user surprise, fully inform the user – and the DBMS.

Care must be taken when translating a virtual relation into a stored relation. The virtual relation definition must be converted into a stored relation with multi-relation constraints to the stored relations on which the virtual relation depends (including those from which it derives its tuples).

[36] I am, of course, referring to the relational algebra as applied to a specific application on a real computer, which is always restricted to expressions of finite length and sets of finite cardinality.

For example, if a projection view is converted into a stored relation, the existentially quantified clause in the defining predicate is translated into an equivalence relationship. This constraint effectively prevents any insert to *either* relation unless values for all attributes that have been projected away are included in the insert expression. These may be included either implicitly (i.e., via defaults, computed values, or lookup values) or explicitly (e.g., in pseudo SQL: 'WHERE EXISTS (SS.STATUS) AND SS.STATUS = value').

Conclusions

If the title question is interpreted as requiring update of relations by identifying the relation in question solely name, the answer is "No." Relations *per se* cannot be updated in the general case if they are differentiated merely by name[37] – the DBMS must have more information. Relations are sets and, as such, must be differentiated by their membership functions. I've detailed how to construct membership functions in practice in the chapters on database design.

If we want to maintain the integrity of a relation, the tuples that make up that relation can only be "updated" by applying a set transformation to the relation that contains them and expresses their interrelationship. By analogy, the relations that make up a *database* can only be "updated" by applying a set transformation to the database that contains them and expresses their interrelationship. Thus, all relations can be "updated" as a consequence of updating the database that contains them.

If we understand updatability to mean that an algorithm exists that (a) produces the desired, logically consistent, and intuitive (assuming knowledge of the database is complete) result, and (b) never produces an "error" condition, then the answer is "Yes".

In concluding, note that a database update (i.e., a database transformation) may be the identity map when applied to some databases[38]. Given our understanding of updates, it is clear that this is not an error: It is possible that a particular transformation, applied to a some databases, might conceptually be equivalent to "altering" or "affecting" exactly zero tuples. It is wrong to think that such updates have "failed". Nonetheless, familiarity with programming and procedural operations on physical data structures often seduces one into thinking in terms such as "errors" and "failures". Only by careful adherence to set semantics can we avoid such misunderstandings. Logical operations either do or do not affect the values in the database. The only results that should be considered errors when using a truly relational DBMS are those that indicate system failures – such as arithmetic overflows, stack overflows, network failures, and disk crashes.

Acknowledgements: I would like to thank Fabian Pascal and Erwin Smout for their detailed reviews and input to this article. The position expressed, errors and all, are entirely my own.

[37] This is not to say that, at the level of domains, abstract concepts in the object language might need to be differentiated by using distinct symbols. I examine this possibility elsewhere.
[38] To use other terminology, the transformation may map some database "states" to themselves.