Nested Data Models

David Barrett-Lennard

15-Sep-2011

# Abstract

A concept of nested relational databases is discussed, which is similar in nature to object oriented data models, but with a straightforward mapping to a conventional relational representation (usually). The nested form has some advantages which can be formalised with the concept of the unique prime Cartesian factorisation of the extension of the declared constraint on each database.

A constraint on a tuple is a recurring idea. E.g. it's relevant to dbvars of relational databases, tuples in relations, TTM possreps and domains and images of functions.

The motivation is a notion of maximal partitioning of the information in a database into orthogonal parts. This means having as many variables as possible which can be updated independently. There is an emphasis on relation variables (relvars), encompassing both base and derived relvars.

# Possreps

A fundamental purpose of a database is to represent abstract values, such as dates, colours, circles or polygons.

Often a tuple is used for an underlying representation of a value of a given type. For example, a circle can be represented in terms of a tuple that records centre and radius attributes. Chris Date & Hugh Darwen (henceforth referred to as D&D) use the term possrep for this concept (as a reminder that there can be more than one possible representation of a value).

D&D have chosen not to formalise a possrep using the notion of a tuple - probably to avoid confusion between the representation and what is being represented. That is certainly important, but nevertheless it is convenient to think of a possrep as involving an underlying tuple representation in this paper. Note therefore that where D&D speak of possrep components, we might equally call them attributes, which is the terminology used for members of tuples.

For a given data type there is a defined set of allowable values of that type. For a given possrep, this in turn means there is a defined set of possible values of the tuple used to represent the value. This set of tuples (which may be infinite) is the extension of a predicate expressed on the possrep components called the possrep constraint.

The extension of the possrep constraint is a relation but it should not be confused with a relation recorded in a database. Instead we are talking about a relation which represents the constraint and normally has an intensional definition.

# Database schema as possrep of a database type

We consider a relational database to essentially be a database variable (dbvar) which holds a database value (dbvalue) of some database type (dbtype).

We formalise a database schema (dbschema) as a possrep of a dbtype. The representation involves a tuple with relation-valued attributes. We will follow the convention of referring to the components of the possrep as relvars even though it isn’t really appropriate to assume they are variables when one is considering the components of a dbvalue.

# Independently updateable variables

In this paper a variable mean something at run time which holds a value and which can be denoted using an *expression* and typically can be updated with statements in imperative languages.

Let information be recorded in a set of variables, and there are declared constraints on these variables. We say a variable is independently updatable if the set of possible values of that variable (as determined by the constraint) is independent of the values of the other variables.

For the purpose of investigating independently updatable variables we are not interested in alternative ways of changing a variable (e.g. UPDATE versus INSERT versus DELETE on a relvar). Instead we are going down to a finer-grained level so we can talk about variables which can be assigned any value of their type.

In other words we’re tending to (where we can) uphold a principle:

 *To be an updatable variable of type T is to support assignment of any value of type T*.

or

 *To be an updatable variable is to be an independently updatable variable*.

If a variable is not independently updatable we’re tending to ignore it being available as the denoted target for update operations.

# The prime Cartesian factorisation of the extension of a constraint

It has been shown in [1] there is a unique Prime Cartesian Factorisation (PCF) of any non-empty relation. In this paper we are interested in the PCF of the extension of a possrep constraint.

There is a unique PCF of the extension of a given possrep constraint, the effect being to partition the possrep components so that the components within one factor can be updated independently of the other factors - i.e. there are no constraint dependencies between factors.

Note that a unique PCF exists for both finite and infinite non-empty relations, so for example it is applicable to constraints in computational geometry examples involving infinite sets in Euclidean space. Of course that involves *intensional* definitions of sets.

For example it is easy to see that we can factorise

 { (x,y,z) ∈ ℝ3 | x2+y2=1 ∧ |z|<1 }

as

 { (x,y) ∈ ℝ2 | x2+y2=1 } ⨯ { (z) ∈ ℝ | |z|<1 }

The way in which constraints can be expressed is open ended. In the context of mathematical computing, data structures and algorithms in general (not just the trivial finite relations), it would be interesting know whether in general if one splits a representation into two parts with no mutual constraints, can it be assumed this particular division is compatible with the maximal way of splitting the representation into many parts with no mutual constraints? Intuitively it seems plausible but can it be proven? That is the purpose of the unique PCF theorem.

The unique PCF theorem (which is an existence & uniqueness proof) is what allows one to give a formal definition of the unique maximal partition of variables into groups which can be updated independently.

It shows that in general constraints on a tuple no matter how complex can be uniquely written in a normalised form as a conjunction where the extension of each conjunct is a prime Cartesian factor of the extension of the entire constraint. Here we are only talking about a normal form modulo how the conjuncts are expressed. This normal form sounds vaguely like Conjunctive Normal Form (CNF) but it's not the same thing.

It is conjectured that the normal form is very important to high performance dbms implementation technique (it relates to asynchronous updates, locking and concurrency, replication and synchronisation, branching and merging, constraint checking, ...)

We shall abuse terminology and talk about the information recorded in a prime factor, which really means the information recorded by the attributes in that factor. There are some general statements that can be made in this regard. For example, prime factors relate to units of *cohesion*. i.e. there is some sense in which information within a prime factor is mutually interdependent and indivisible, whereas between different prime factors the information is more independent, and provides a basis for supporting independent and concurrent data entry by different users. That means it is a relevant to data representation independence and updateable views. If a representation provides many prime factors then it is expected there are more opportunities for choosing where and what information is visible and/or updatable. This translates into better support for data representation independence.

Prime factors are also relevant to locking and concurrency, and the granularity of atomic updates. For example, if we define atomic updates to be the most primitive updates that are compatible with integrity preservation, then clearly that idea is somehow connected to prime factors. It follows that prime factors might be relevant to scalability of databases. If very large databases don't have prime factors then atomic updates might be complex, take longer to execute, are more difficult to validate and concurrency might be limited. This is particularly relevant to distributed databases - e.g. it might be preferable for prime factors to not be distributed over a network to avoid the need for distributed atomic commit protocols.

If prime factors are indeed an important consideration, it's nice to know that they are defined uniquely and independently of the way integrity constraints happen to be expressed in the schema (e.g. uniqueness constraint, functional dependency, foreign key constraint, or some boolean valued expression making use of the relational algebra). Prime factors provide some basis for comparing alternative data representations.

The prime factors on the database tuple of a relational database represent a fairly coarse partition of a large schema into independent parts, and therefore only appears to have limited utility.

However, later in this paper we discuss the idea that a database can typically be factored into a much larger set of *conditionally existent* prime factors.

If prime factors are small enough, then it is possible that a system with MVCC + one write mutex per factor, has advantages over conventional strict 2PL lock managers with dead-lock detection etc. What's the point in burdening the lock manager with support for concurrent updates within a factor within which values have a bearing on each other - as far as constraint checking is concerned?

# Using the PCF to select appropriate representations

Claim 1: A possrep can be formalised as a non-empty relation where

1. the components of the possrep are exactly the attributes of the relation;
2. the relation is the extension of the possrep constraint (regarded as a predicate parameterised on the attributes); and
3. each element of the relation (i.e. tuple) is assumed to represent a value of the type of the possrep. This is similar in nature to a formal semantics.

Claim 2: Every possrep has a well defined prime factorisation

(obviously since every possrep is formalised by a particular non-empty relation)

Claim 3: All things being equal, possreps with more prime factors tend to be more desirable.

Example:

Consider a type for geometrical points constrained to an annulus

possrep 1: R1 = { (x,y) ∈ ℝ2 | 1 < x2+y2 < 9 }

 Prime factorisation = {R1}

possrep 2: R2 = { (r,t) ∈ ℝ2 | (1<r<3) ∧ (-π≤t<π) }

 Prime factorisation = {R,T}

 where R = { (r) | 1<r<3 } and T = { (t) | -π≤t<π }

An interesting idea is to consider in turn a possrep on each component of a possrep and so on. Without prejudice from type systems, this provides some basis for properly maximising the number of prime factors. As a simple example, a representation of a circle with centre and radius, where the centre can in turn be represented either by (x,y) or (r,t) coordinates.

It may seem odd to have talked about a type of geometrical point constrained to an annulus. It's less odd if you consider that such a constraint may exist on an attribute of a "containing" possrep in which the geometrical point appears. For example we may be interested in circles centred at positions in an annulus, which would suggest a representation using (r,t) for the centre of the circle might be best.

## Example

Consider the ellipse E(CX,CY,A,B) with locus

 { (x,y) ∈ ℝ2 | ((x-CX)/A)2 + ((y-CY)/B)2 = 1 }

and the predicate

 P(cx,cy,a,b) = "the ellipse has centre (cx,cy), major radius a and minor radius b"

In the context of the ellipse E(CX,CY,A,B) the predicate P(cx,cy,a,b) has an extension with a single tuple

 TUP { <cx CX> <cy CY> <a A> <b B> }

The predicate P(cx,cy,a,b) is equivalent to the conjunction

 "the centre of the ellipse has x-coord cx AND

 the centre of the ellipse has y-coord cy AND

 the major radius of the ellipse is a AND

 the minor radius of the ellipse is b"

If you project away cy,a,b then you get a relation with a single tuple

 TUP { <cx CX> }

which is the extension of the predicate

 "there exists cy,a,b such that

 the centre of the ellipse has x-coord cx AND

 the centre of the ellipse has y-coord cy AND

 the major radius of the ellipse is a AND

 the minor radius of the ellipse is b"

which is equivalent to

 "the centre of the ellipse has x-coord cx"

because for example

 "there exists cy such that the centre of the ellipse has y-coord cy"

is true.

# Pseudovariables

D&D use the term *pseudovariable* for a component of a possrep of a variable. For example, given a variable c of type CIRCLE, the expression THE\_R(c) denotes a pseudovariable which is the radius of the circle. They promote the idea that pseudovariables are updatable, irrespective of whether the actual physical representation of the variable matches the possible representation associated with the pseudovariable.

By definition an update to a pseudovariable means that the other components of the possrep are held constant. E.g. while assigning the radius of the circle, the centre is held constant.

This works well if the pseudovariable is associated with a prime Cartesian factor of the possrep relation, because it is true that the pseudovariable can be assigned with any value of its type without a constraint violation.

If not then there are a number of problems that arise

* It breaks one of the hallmarks of a static type checked language - that type checked assignment operations are infallible
* Intermediate states of an atomic update can be visible in the sense that a type constraint violation is raised on an intermediate state, even though the overall update doesn't break type constraints. This leads to many significant problems - e.g. it upsets axiomatic definitions of the language, notions of equivalent update expressions, and complicates and imposes barriers on code optimisation

If pseudovariables are outlawed completely the end result is a pure functional language plus atomic assignment to the database variable (dbvar), which pushes the whole problem of efficient generation of in-place algorithms onto the compiler. This won't be adequate for certain specialised applications like a spatial geometry database with high performance requirements.

Although pure functional languages are great and modern compilers are quite good, it can also be said that in-place algorithms are worthy of study, with many mathematical results of interest. E.g. an interesting fact is that "in-place" quicksort has space complexity O(log n) in the worse case and this depends on tail recursion optimisation.

It is proposed that we reject updatable pseudovariables for which assignment is fallible even though it is assigned with a value of that type.

# Conditionally existent variables

One can speak of variables being created and deleted over time. Expressions sometimes denote a variable and sometimes don't - i.e. whether an expression denotes a variable can change over time.

E.g. in C++

 std::vector<int> L;

 L[3] = 7; // CRASH!!!, L[3] doesn't denote a variable at this point in time

 L.resize(10); // create 10 variables

 L[3] = 7; // ok now L[3] denotes a variable

It is useful to have a separation of concerns between

1. expressions that denote variables
2. operations that efficiently update particular types of variables (such as an increment operation on an integer, or an append operation on a list).

E.g. in C++ modulo assignment is available on int variables so one might use it on an element of L in a statement like this:

 L[3] %= 10;

There is a separation of concerns between

1. the expression L[3] that denotes a variable; and
2. the update operation using %=.

We want the same separation of concerns for updating a relational database, so we’re after expressions that denote variables that can be assigned any value of their type. E.g. maybe something like this appends to a string attribute within a tuple

 append( S WHERE S# = S#('S1') {CITY}, "x")

Prime factorisation of possreps doesn't (on its own) capture the most general notion of separability that is sufficient for pseudovariables to be independently updateable without risk of constraint violations.

For example, consider an implementation of an update operator, which contains a lexical scope within which a pseudovariable has been declared and bound, perhaps to a component of a tuple within a relvar. Within that lexical scope the variable can be passed by reference into update operators that efficiently update the variable in-place. There are many such cases where the variable can be assigned any value of its type without risk of constraint violations.

The prime factor concept appears to be applicable to the special case where both the existence and uniqueness of a specified binding to a pseudovariable that supports infallible assignment is guaranteed. More generally a concept of conditional binding is needed (such as when you want to bind a pseudovariable to the eye colour of Fred Flintstone, but that is conditional on finding the relevant tuple in the database in the first place).

It appears this idea of conditional existence of pseudovariables supporting infallible assignment is related to updatable views. E.g. an updateable view on Fred Flintstone's attributes. A user can be presented with this view and update it as though it's a separate database - i.e. without surprises because of integrity constraints depending on information that isn't visible in that view. Also there are no update ambiguities. It's as though there's a "separable" database nested within a containing database. The existence of the inner database is conditional (and in fact the outer database is able to delete the inner one).

# Nested databases

Consider there is a need to record lots and lots of facts about Fred Flintstone, independently of other things in Bedrock. It makes sense to define an independent schema just for Fred Flintstone (but probably reusable for other characters) using predicates which *implicitly* concern Fred Flintstone, and therefore don't need an identifier for a person! E.g. instead of

 EyeColour(P,C) :- Person P has eye colour C.

the attribute P can be eliminated from the predicate because it is implicit

 EyeColour(C) :- Fred Flintstone has eye colour C.

Note that elimination of identifiers from predicates means that dee and dum might be quite common (i.e. for boolean properties of things).

The person identifier is only needed in the outer or containing database (in order to make statements about that person in relation to other things), and as far as the DBMS is concerned, the identifier can (also) be regarded as a reference to an inner database. The inner database is deleted by the DBMS when it is no longer referenced. Within the inner database the person identifier is irrelevant. This immunises it from updates to the value of the identifier, and also schema changes when the format of the identifier changes. IMO these are striking advantages. For example, in a conventional RDB a change to the format of an identifier can require schema changes to dozens of relvars.

Since we have managed to pull P out of the predicate EyeColour, we have managed to create real cartesian prime factors within the context of a representation of all the information which is exclusively about Fred Flintstone!

Evidently there's a similarity to an OO perspective, because a nested database resembles an object and a surrogate id is like an oid. However I want to emphasise the fact that this connection is superficial. The nested database is founded on logic, and it's obvious that the power of the relational approach is undiminished by nesting given that the more conventional relational representation without nesting can be uniquely derived by simply adding the identifiers back again, as attributes of global predicates (assuming predicates are globally uniquely named).

Minor note: this can lead to multiple identifiers being added back again depending on nesting depth. This upsets unnesting if the nesting depth varies for a given type of nested database, and it means that global identity involves a path into a tree structure of nested namespaces, so the appropriate way to unnest is to record these paths in a single attribute.

todo : it would be useful to understand how the usual concept of functional dependencies and normal forms fits into the picture.

# Using propositions to characterise a value

In a similar fashion to implicitly representing facts about Fred Flintstone without any identifier in the predicates, it's sometimes possible to fully characterise a given value using logic (the motive being to maximise the use of logic in the system). E.g. in the context of a *given* circle we can define predicates

 radius(R) :- the radius of the given circle is R.

 centre(C) :- the centre of the circle is C.

# Concurrency

A significant motive for nested databases is to do with concurrency. If users can update prime factors independently then concurrency is maximised. If the information in a database can be factorised into thousands or millions of (conditional) prime factors, which are each very simple, like a person's eye colour, then it becomes reasonable to support multi-user editing where edits are applied to a local copy of the data without network latency, and even branching and merging of databases is feasible (in a similar fashion to version control systems that support branching and merging of text files). The premise is that users working on separate, long lived tasks, don't tend to edit the same very fine grained prime factors, so the number of conflicts tends to be very small.

# Two meanings of type constraint

The term "type constraint" tends to be overloaded for two very different concepts that should be distinguished:

1. A constraint which is a boolean valued function allowing for a (restricted) set comprehension, and can be used to create new subtypes from existing types (aka specialisation by constraint). E.g. {x ∈ Z | x > 10}
2. An equivalence relation over some set which defines a set of equivalence classes. E.g. this provides a basis for constructing the rationals out of Z×(Z\{0}) using the equivalence relation ~ satisfying (a,b)~(c,d) ⇔ ad=bc.

It doesn’t seem reasonable to think of 2) as a kind of 1). i.e. that equivalence classes be regarded as a subset of the original set. i.e. that equivalence classes be regarded as specialisation by constraint by using a boolean valued expression to obtain a canonical form. It's obscure to think of the rationals as a subtype of Z×(Z\{0}). Indeed if this were the case then why are possreps needed at all? All we need do is specialise tuple-types using set comprehension.

# Information equivalence classes

Integrity constraints are what defeat simple updates. E.g. you might not be able to assert the fact that Acme Inc is a supplier without providing other facts. The facts that need to be provided is determined by the information that needs to be recorded which is determined by the business requirements.

Let S be a database schema which I will take to mean a set of base relvars on which relvar predicates have been defined. Defined means that in any given world situation the extensions of the relvar predicates are determined, hence the dbvalue is determined.

Let

 D = { d | ∃ world situation having dbvalue d recorded }

We can say the schema constraint has been specified "maximally" if its extension equals D. Note that the information to be recorded determines the "maximal" constraint.

Let S1 and S2 be database schemata. Let

 P = { (d1,d2) | ∃ world situation having dbvalue d1 recorded for schema S1 and dbvalue d2 recorded for schema S2 }

Let

 D1 = { d1 | (d1,d2) ∈ P }

 D2 = { d2 | (d1,d2) ∈ P }

Definition: If P is the graph of a function from D1 to D2 then we say the information in S1 *contains* the information in S2.

Definition: If P is the graph of a bijection from D1 to D2 then we say S1 and S2 are *information equivalent* schemas.

The business requirements determine the information to be recorded which in turn determines all the following:

* the information equivalence class
* the "maximal" constraint on each schema within the information equivalence class
* what facts can be asserted independently of other facts

Normalisation is only concerned with identifying a schema within an equivalence class which is more convenient to use in some sense. It doesn't affect what facts can be asserted or retracted independently of other facts.

# Maximal decomposition of information in an information equivalence class

Assume a database value is a tuple with RVAs.

For a database schema S, let D(S) denote the extension of the constraint. D(S) is a non-empty set of tuples with RVAs.

D(S) can be regarded as a relation and therefore we can consider it's prime Cartesian factorisation. Let C(S) denote the bag of integers which are the cardinalities of the prime Cartesian factors of D(S).

The following can be proven:

For any schema S with a finite number N of states there exists a schema S' which is information equivalent to S where C(S') equals the bag of prime number factors of N.

Do we expect that in real database systems N tends to be a large prime number?

I would be surprising if the number of states defined by real-life database schemas (with maximal constraints) tend to be prime numbers. It is difficult to think of no reason to expect it given that the *Prime Number Theorem* says large primes are much less common than large composites, and given that in mathematics/computing it is generally a hard problem to find large prime numbers, despite number theorists thinking about the problem for a long time.

On the contrary this paper conjectures that in practise for real databases with finite types the number of prime factors is very large despite the constraints. Perhaps this isn't apparent because of the tendency to not consider designs having database valued attributes (which goes against Codd's idea of simple attribute types).

# Proof of claim about maximal decompositions

Claim: There exists S' with prime relation factors of D(S') with prime number cardinalities.

The following is an outline of a constructive existential proof which involves relational expressions for each relvar in S' that are defined in terms of the relvars of S:

Let UNION\_OVER <set of relation> mean a union over the given set of relations.

Let JOIN\_OVER <set of relation> mean a join over the given set of relations.

Let d[R] denote the value of relvar R of S in database state d of D(S).

Let F1, ..., Fn be the prime number factors of N = |D(S)|.

Let the N states in D(S) be organised into an n dimensional array of size F1 x F2 x ... x Fn = N. The order doesn't matter.

For j in 1..n and i in 1..Fj,

 Let Bij denote the subset of D(S) corresponding to the set of states in the ith slice of the jth component of the n-dimensional array.

 (a slice has n-1 dimensions)

For j in 1..n and i in 1..Fj,

 Let Tij = UNION\_OVER { JOIN\_OVER { ((R MINUS d[R]) UNION (d[R] MINUS R)){} | R is a relvar of S } | d in Bij }

 (this expression evaluates to DEE if the dbvar of S is in slice Bij, otherwise DUM)

For j in 1..n

 Let S' have relvar Rj' which is defined in terms of the relvars of S using the expression

 Rj' = UNION\_OVER { Tij JOIN REL { TUP { (x i) } } | i in 1..Fj }

 (this expression evaluates to REL { TUP { (x i) } } where the dbvar of S is in slice Bij)

Note that in every state each relvar Rj' of S' has exactly one tuple with one attribute. This follows because any state (i.e. element of D(S)) lies in exactly one slice of the jth coordinate. The UNION over the JOINs picks out exactly one of the TUP { (x i) }. There are Fj slices for the jth component and hence Fj possible values of Rj'.

It is easy to see that a combination of the values of the Rj' corresponds to one of the elements of the n-dimensional array and hence to one of the states in D(S).

# Unrolling a predicate

Here we are using a simple example to illustrate an "unrolling" technique of "partially/fully instantiating" predicates to obtain larger numbers of predicates for a different representation of the same information (i.e. a representation that is information equivalent).

The "unrolling" is just a conceptual technique to find parameterised expressions which denote derived relvars which are suited to be the target of updates.

It should not be taken to mean that actual database schemas will be expressed in the unrolled form.

This example has been chosen to illustrate unrolling to deal with an IND constraint and to relate it to calculations of numbers of states. It is minimal for that purpose. It is not being claimed it is "realistic". See the S&P example for a more complex/realistic example with INDs.

Let T be a finite type where there are n values of type T.

Consider two relvars R1,R2 each having the same single attribute of type T and there's an IND constraint defined: R2 must be a subset of R1.

The number of ways we can have i tuples in R1 is (n choose i).

For i given tuples in R1 the number of subsets of those tuples which can appear in R2 is 2i

Therefore we deduce that the number of database states is SUM { (n choose i) x 2 i | i in 0..n }.

Example with n = 2. We get 9 states:

 (2 choose 0) x 20 + (2 choose 1) x 21 + (2 choose 2) x 22

 = 1x1 + 2x2 + 1x4

 = 1 + 4 + 4

 = 9

For notational convenience let the two values of T be a and b and I'll pretend R1 and R2 are subsets of {a,b}. The following table shows the possible database states:

|  |  |
| --- | --- |
| **R1** | **R2** |
| {} | {} |
| {a} | {} |
| {a} | {a} |
| {b} | {} |
| {b} | {b} |
| {a,b} | {} |
| {a,b} | {a} |
| {a,b} | {b} |
| {a,b} | {a,b} |

Now it turns out the following is true

 SUM { (n choose i) x 2i | i ∈ 0..n } = 3n

(e.g. above n=2 and we have 32 = 9 states. For n=3 we get 33 = 27 states, For n=4 we get 34 = 81 states and so on).

It's not obvious from the formula itself that SUM { (n choose i) x 2i | i in 0..n } simplifies to 3n.

There is an alternative representation of the same information involving n Cartesian factors for the n values of T. These factors can be updated independently.

Each factor has exactly 3 possible states:

1. the value isn't present in either R1 or R2
2. the value is present in R1 but not R2
3. the value is present in both R1 and R2

Note that these propositions treat the value as implied, in exactly the manner discussed previously when refer to nested databases.

When n is large (e.g. 232 if T is a 32 bit integer) we have a massive factorisation of the information into parts that can be updated independently.

# Factorisation for the S&P schema with relvars S, P, SP and the usual keys and INDs declared

The "nesting" is a technique that can be used in a real schema definition to help organise the information in a way that highlights orthogonality.

I'm going to follow the convention of using the term "relvar" for the relations of a database value. One might consider them to be possrep components and a database constraint to be a possrep constraint.

Note that if one puts the D&D schema into 6NF there are 8 relvars. None of them are independently updatable.

## Supplier Database

A SUPPLIER\_DATABASE is a type and represents information about one supplier and has three relvars which can be updated independently, having the following predicates

* the supplier has name <SNAME>
* the supplier has status <STATUS>
* the supplier has location <CITY>

These three relvars have the constraint that they have exactly one tuple.

Note that these predicates explicitly presuppose the existence of the supplier (hence the definite article is used). The predicates are perfectly valid if we restrict ourselves to world situations where the supplier exists.

Note that even though a supplier number (S#) identifies a supplier, it doesn't appear as an attribute in any of the relvars of the SUPPLIER\_DATABASE.

There is a separation of concerns between identifying the supplier and representing facts about the supplier. There could be a hundred relvars for the latter and none of them are concerned with how a supplier is identified. This is handy when the key is composite or there are multiple candidate keys.

## Part Database

A PART\_DATABASE is a type and represents information about one part and has four relvars which can be updated independently, having the following predicates

* the part has name <PNAME>
* the part has color <COLOR>
* the part has weight <WEIGHT>
* the part has location <CITY>

These four relvars have the constraint that they have exactly one tuple.

SUPPLIER\_DATABASE and PART\_DATABASE values can appear within a SUPPLIER\_AND\_PARTS\_DATABASE.

## Supplier and parts database

A SUPPLIER\_AND\_PARTS\_DATABASE is a type and has three relvars:

|  |  |  |
| --- | --- | --- |
| **Relvar name** | **Predicate** | **Key** |
| S | there is a supplier <S#> described by supplier database <SDB> | {S#} |
| P | there is a part <P#> described by part database <PDB> | {P#} |
| SP | there is a supplier <S#> that ships quantity <QTY> of part <P#> | {S# P#} |

(with the appropriate IND constraints, these three relvars only represent one Cartesian factor of the possrep components)

Note that these predicates are existentially quantified in concrete parts and suppliers.

In effect this leads to supplier database variables and part database variables for each concrete supplier and part that exists.

Note that the creation and destruction of the supplier databases and the part databases happens through NSERT/DELETEs on the these three relvars.

### Unrolling of the supplier and parts database

In the S&P database where there are 5 suppliers, 6 parts and 12 shipments one can regard that information as being equivalent to a representation involving 23 independently updatable relvars, using the "unrolling" technique on the supplier and part numbers:

* there is a supplier S1 described by supplier database <SDB>
* ...
* there is a supplier S5 described by supplier database <SDB>
* there is a part P1 described by part database <PDB>
* ...
* there is a part P6 described by part database <PDB>
* supplier S1 ships quantity <QTY> of part P1
* ...
* supplier S4 ships quantity <QTY> of part P5

In this representation, we can incorporate the factors in the nested databases to end up with 5x3 + 6x4 + 12 = 51 variables which can be updated independently.

Can we reasonably say these variables are independently updatable given that their very existence is conditional? In general we can say: *If the variable exists then it is independently updatable*.

This idea is not expected to be controversial. E.g. in an dynamically resizable array the elements are independently updateable if they exist.

# FDs and keys

FDs reflect business requirements. The original S&P schema is information equivalent to the nested schema representation. This implies that the FD’s have been captured!

In the nested schemas there are still FDs and keys. The original S&P database which is 5NF has been decomposed into three simpler distinct database schema which are each in 5NF. Within these simpler schema there are relvars which are independently updateable, even though there are no independently updatable relvars in the original schema.

Most of the original FDs are no longer applicable in the nested schemas. For example the FD {S#}->{CITY} in S is inapplicable because attributes S# and CITY don't even appear in the same database schema (never mind the same relvar).

Note that the FD {S#}->{CITY} essentially means a supplier is located in exactly one city. It doesn't imply that the city cannot be updated independently.

There's a mapping from a set of nested schemas to an information equivalent unnested schema. The FD {S#}->{CITY} is implied by the nested database schema because

a) There is an FD {S#}->{SDB}

b) There is an FD {SDB}->{CITY}

c) FDs are transitive

(where SDB is the attribute name for a supplier database - a database which records information about a single supplier)

b) holds because

 1) there is a declared constraint in the supplier database, that there is exactly one CITY tuple. i.e. The FD {} -> {CITY} holds within the supplier database.

 2) FDs must be adjusted when they are translated between databases. The FD {} -> {CITY} within the supplier database becomes {SBD} -> {CITY} in the "containing" suppliers and parts database.

# Which is more complicated?

One might say the nested schema is adding complexity.

The factorisation of the information into different schema can reduce complexity because they can be designed, specified, documented and understood in isolation. Also having lots of FDs defined implicitly seems an advantage in complicated examples.

There are many things to consider before making a judgement on the relative merits.

# Physical versus logical

The nested versus unnested representation can be discussed at both logical and physical levels of discourse.

It depends on many factors but often a nested physical representation outperforms an unnested representation by orders of magnitude.

# Atomic facts

Amongst other things we have investigated the notion that an independently updatable derived relvar can be associated with the value of a single attribute of a single tuple of a relvar. This provides an insight into defining some notion of "atomic facts".

# Misc

A schema determines what base relvars are defined. It doesn't prescribe what derived relvars can be defined.

The question of what relvars can be updated independently of other relvars is determined by the information equivalence class.

Consider the derived relvar

 VAR S1\_CITY VIRTUAL (S WHERE S# = S#('S1') {CITY} )

The predicate for this is:

 There is a supplier identified by 'S1' located in a city named [CITY]

We have to be careful what we mean by independently updatable. E.g. the assignment

 S1\_CITY := REL { TUP { CITY Athens } }

trips a constraint violation if there is currently no supplier named 'S1' recorded in the database

To speak of independent updatability in this case we need a presupposition of the existence of a supplier named 'S1'. I think of there being conditionally existent relvars. With the presupposition we can speak of the predicate

 The supplier identified by 'S1' is located in a city named [CITY]

I see the nested databases as making these presuppositions explicit (e.g. the database recording all information about a supplier presupposes the supplier).

Note that we can't put the presuppositions into the predicates as conditional statements. Consider:

 If there is a supplier identified by 'S1' then that supplier is located in a city named [CITY]

This isn't a predicate because it doesn't uniquely determine an extension in every world situation.