Relations with Stored and Inherited Attributes

Witold Litwin¹

(March 2016)²

Abstract. The universally applied Codd’s relational model has two constructs: a stored relation, with stored attributes only and a view, only with inherited ones. In 1992, we have proposed a construct with both types of attributes. Examples showed the new construct attractive. No one jumped however on the idea. We now revisit our proposal. We show that the relational schemes using our construct may be more faithful to reality. They may also spare the logical navigation through joins and complex value expressions to queries. Both annoyances are unavoidable at present, unless dedicated views hid them. But managing views is cumbersome on its own. Our construct appears the first practical solution to this decades’ old dilemma. Better late than never, existing DBSs should easily accommodate our proposal, with almost no storage and processing overhead.

1. Introduction

The universally applied Codd’s (relational) model for a Database (Management) System (DBS), [C69] & [C70] has two basic constructs: a stored relation and a view. Both are named finite relations with atomic attributes only, in 1st Normal Form (1NF) thus. A Stored Relation, (SR), called also base one, or simply relation or a (relational) table, has stored (base) attributes (columns) only. A view, also called Inherited Relation (IR), has only the inherited attributes. These get values from SRs or from views through a stored expression of some data manipulation language (DML), usually a stored SQL query. In 1992, we proposed an additional construct. It was also a 1NF relation, but with both stored and the inherited attributes, [LKR92]. Examples showed the construct attractive. The idea seemed promising also for OODBs, à la mode in those times.

No one followed the lead however. Below, we revisit our proposal thoroughly. We call our construct Stored and Inherited Relation, (SIR). We qualify of SIR-model the data model resulting from our proposal. A view in SIR-model may thus also inherit from an SIR. We refer to Codd’s model as to Stored Relation or View model, (SRV-model). We believe the reader familiar with the SRV-model and SQL in particular.

We show that SIR-model adds useful capabilities to those of the SRV-model. It preserves all the latter by definition. We propose SQL extensions for SIRs. Next, we restate the relational scheme design rules to include SIRs. It will appear, perhaps surprisingly, that a relational DB may advantageously consist of SIRs only. We show the implementation of the SIR-model over an existing DBS rather easy and with negligible storage and processing overhead. We hope the model entering the practice, “better late than never”.

Next section details the SIR-model. We discuss the basic concepts and the SQL extensions. We show that conceptual schemes with SIRs may be more faithful to the reality. The ER-model becomes rather useless as the add-on for the relational design. Queries may become free from the customary logical navigation through inter-relational joins. SIRs appear the first generally practical solution to this decades’ old annoyance. Likewise, SIRs may spare to queries complex aggregate expressions, another old exasperation. Both troubles are unavoidable for SRV-model at present, unless dedicated views hid them. Managing multiple views is however enough cumbersome on its own to be rarely practiced.

Section 3 restates the relational schema design rules. We first generalize the NFs other than 1NF to SIRs. Next, we restate the Heath’s and Fagin’s theorems. The restated theorems create lossless decomposition into projections that are SIRs instead of into the usual ones in SRV-model. Our

¹ Université Paris Dauphine - PSL, LAMSADE, Paris, France, witold.litwin@dauphine.fr
² Working Version. Last update January 5, 2017
approach avoids then usually the logical navigation over the projections that the current decomposition creates necessarily. More precisely, for the restated Heath’s theorem, the decomposition is totally logical navigation free, while some need for it, likely occasional in practice, remains for the restated Fagin’s one.

Section 4 discusses the implementation of SIRs over existing DBSs. Section 5 draws the conclusions and overviews the future work.

2. SIR-Model

2.1 Overview

Figure 1 shows the SIR-model versus the SRV-model. As already discussed, the SR construct is the same for both models. An SRV-model view, i.e., an IR, is also a view for the SIR-model. The inverse is not true, as also stated. Next, as both constructs in SRV-model, an SIR is a finite subset of a Cartesian product of atomic attributes (columns) over some domains, subject to any algebraic or predicative operations, and aggregate or scalar functions, applying to 1NF relations. As we said however, unlike an SR or a view, some attributes of an SIR are the stored ones and some are inherited. A stored attribute (SA) has its values stored somewhere in the DB. The values of an inherited attribute, (IA), are calculated. As usual for an SR or a view, an SIR has a scheme. That one defines every SA and IA. We suppose every SA declared as usual, e.g., as in an SQL dialect. Each IA in contrast is calculated by a specific scheme that we call inheritance expression (IE).

An IE selects one or several IAs, e.g., in the Select clause for an IE defined using an SQL expression. These may be inherited from stored relations, views or SIRs. An SIR scheme may contain one or multiple IEs. Each IE defines a disjoin subset of IAs for the SIR. Every SA and IA in an SIR should have a unique proper name, i.e., without any prefix, like the usual relation name from which an IA to disambiguate could come from. Each IE resembles a view scheme. As the latter, every IE selects some IAs and determines tuples of IA values using some relational selection expression, e.g., an SQL Select statement. As usual, we basically consider these values not materialized. Unlike for view tuple however, each tuple calculated through an IE contributes with its IAs and their values as sub-tuple to some SIR tuple, Figure 2. At the figure, each tuple of, e.g., $IE_1$, contributes to some tuple with all the SAs and with the sub-tuples of all the others $IE$s there as well. SAs are collectively named B at the figure for reasons we explain soon. A green rectangle represents each sub-tuple.

More precisely, for every SIR tuple $t$, every IE calculates at most one tuple of IAs values to become a sub-tuple of $t$. For this purpose, every IE contains a specific equijoin clause. This one matches attributes in relations the tuples the IE creates inherit their values from, with some SAs or IAs in the SIR other than the IAs of the IE. Let $R$ be an SIR, $I$ an IE and $R’ = R / I$. Every attribute of $R’$ belongs also to $R$. The clause may use therefore $R$ name to disambiguate an $R’$ attribute name. We call the join recursive, since it defines $R$ by referring to $R$. For each $R’$ tuple $t’$, at most one $I$ tuple, say $t$, should match the recursive join. Tuple $t$ becomes the sub-tuple of the $R$ tuple with the matching $t’$. If
for some \( t' \), \( I \) does not find any matching \( t \), \( I \) sets the sub-tuple to the null tuple, i.e., with the null as every IA selected by \( I \). In other words, for every \( t' \), the recursive join clause either joins \( t' \) with \( t \) or with the null tuple.

Figure 2 also shows the result when a null sub-tuple results. E.g., the bottom tuple contains the null \( IE_1 \) sub-tuple, as well as null \( IE_2 \) sub-tuple besides. If this may happen, the recursive join should be a half outer join preserving \( R' \). Otherwise it can be an inner join. Actually, as users often despise outer joins, one may let them using the inner join sometimes instead of the formally necessary outer one. We’ll show that an SIR-enabled DBS may indeed easily rewrite the former to the latter. For the same reason, one may leave even the entire clause implicit, as we’ll show as well.

Next, let us write an SIR \( R \) as \( R \left( B, V \right) \) with \( B \) containing all the SAs, and \( V \) all the IAs. \( V \) consists thus of the columns labelled with IEs at Figure 2. For every \( R \), we require \( B \) to be a stored relation for SRV-model, i.e., without duplicates. We call \( B \) stored relation of \( R \) or base of \( R \). Furthermore, as any relation, any SIR has one or more keys. We require the primary key of its base among these keys. We recall that for any relation, all attributes are functionally dependent on every key. For any key value of any \( R \), we have thus for each SA and each IA, at most one (atomic) value, respectively stored or inherited. The rationale is, again, the requirement of 1NF for any SIR. Finally, we always have \( \text{card}(R) = \text{card}(B) \). All this is also the rationale for the requirement we made above on the sub-tuple calculation. Namely, that for every SIR tuple, every IE creates, through its selection expression with the recursive join, at most one tuple as the sub-tuple.

![Figure 2. An SIR with multiple IEs. Stored values are grey, inherited ones are green. IEs actually bring respectively three, four and as many tuples as there are stored ones. V unions all the IAs.](image)

We also require any IE to bear a name, unique within its SIR. This name may be explicit or implicit as it will appear. It may serve as the reference to the IE, as we’ll show. If needed, we disambiguate IE names as usual for SQL, e.g., as we disambiguate attribute names.

We suppose SIR-model constructs defined as usual through some DDL (Data Definition Language). Operationally, we consider from now some SQL-like DDL. We suppose the statements of this DDL extend the similar ones of some popular SQL dialect, we call kernel for the DDL for SIR-model. E.g., Create Table for an SIR extends the kernel Create Table with capability to defines IEs. We show a sample extended Create Table in next section and come back to the whole matter after.

### 2.2 Motivating Example

We reuse the biblical Codd’s Supplier-Part relational DB, originally illustrating his proposal, as popularized by C.J. Date, [D4]. It is often named S-P DB in short. The example restates S-P using SIRs. We refer to the original as to S-P1. We call the restated scheme S-P2.

Example 1. S-P1 models an enterprise with some Suppliers, Parts and Supplies. A supply contains some quantity of a part shipped by some supplier. Besides, some suppliers may be supplying
anything currently, as well as parts may be not supplied at present. S-P1 conceptual schema consists of three stored relations named S, P, SP. This scheme is optimal for the so-called relational design rules, using NFs in SRV-model. SP respects also the referential integrity implicitly, i.e., by default. There cannot be thus a supply declared by a tuple in SP, but with the supplier or the part not in S and P. In practice, the referential integrity of SP would need to be today explicitly enforced however, by the well-known Foreign Key clauses. An application may indeed choose not to enforce it, or only partly.

**S-P2 Scheme**

<table>
<thead>
<tr>
<th>Table S</th>
<th>Table P</th>
<th>Table SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>S# Char, SNAME Char, STATUS Char, CITY Char;</td>
<td>P# Char, PNAME Char, COLOR Char, WEIGHT Char, CITY Char;</td>
<td>S# Char, P# Char, QTY Int,</td>
</tr>
<tr>
<td>I_S (Select SNAME, STATUS, S.CITY As SCITY From S, SP Where SP.S# = S.S#);</td>
<td>I_P (Select PNAME, COLOR, WEIGHT, P.CITY As PCITY From P, SP Where SP.P# = P.P#);</td>
<td></td>
</tr>
</tbody>
</table>

**S-P2 Content**

<table>
<thead>
<tr>
<th>Table S</th>
<th>Table P</th>
</tr>
</thead>
<tbody>
<tr>
<td>S#</td>
<td>SNAME</td>
</tr>
<tr>
<td>S1</td>
<td>Smith</td>
</tr>
<tr>
<td>S2</td>
<td>Jones</td>
</tr>
<tr>
<td>S3</td>
<td>Blake</td>
</tr>
<tr>
<td>S4</td>
<td>Clark</td>
</tr>
<tr>
<td>S5</td>
<td>Adams</td>
</tr>
<tr>
<td>S1</td>
<td>P1</td>
</tr>
<tr>
<td>S1</td>
<td>P2</td>
</tr>
<tr>
<td>S1</td>
<td>P3</td>
</tr>
<tr>
<td>S1</td>
<td>P4</td>
</tr>
<tr>
<td>S1</td>
<td>P5</td>
</tr>
<tr>
<td>S1</td>
<td>P6</td>
</tr>
<tr>
<td>S1</td>
<td>P1</td>
</tr>
<tr>
<td>S1</td>
<td>P2</td>
</tr>
<tr>
<td>S3</td>
<td>P2</td>
</tr>
<tr>
<td>S4</td>
<td>P2</td>
</tr>
<tr>
<td>S4</td>
<td>P4</td>
</tr>
<tr>
<td>S4</td>
<td>P5</td>
</tr>
</tbody>
</table>

1. Figure 3 shows the S-P2 scheme. It is also optimal in SIR-model, as it will appear. The referential integrity is again implicit. Figure 4 shows example extensions of S-P2 relations, i.e., a possible content of each. The schemes and the contents of S and P relations are the original ones. We use self-explaining statements to define every relation. We underline the primary key attributes, as usual. The S, P relations have SAs, but not IAs. They were stored relations in S-P1 and remain so in S-P2. SP
keeps the original SAs with their values, i.e., these of S-P1.SP that are (S#, P#, Qty) to recall. These SAs form the base B of SP. B has its original primary key, (S#, P#). SP however has now also IAs. Also, B key is SP primary one as well. SP became an SIR. All the IA values of SP form V in our generic notation. V values are in Italics at the figure. The choice of IAs means that the Database Administrator (DBA) considers every property of a supplier or of a part, as also, ipso facto, of any supply. We examine the rationale for it later.

The SQL Select statements in SP-2 scheme express two IEs. We named them I_S and I_P. I_S inherits all and only attributes of S, except for S#. Its tuples inherit values from the tuples of these attributes in S. Likewise I_P inherits all and only attributes of P, except for P#. Both have renamed their source attributes CITY into proper names unique in SP, as required for any SIRs. The clause SP.S# = S.S# is the recursive (equijoin) clause for I_S. Here, SP.S# is the attribute outside all these defined by I_S. For the first tuple of SP, with SAs S# = S1 and P# = P1, the join clause matches I_S tuple (Smith, 20, London), since S.S# = S1 for this tuple. S1 does not match S# value in any other I_S tuple, as required for every IE. The obvious reason, for the same result for every SP.S# value besides, is that S# is the key of S. The selected I_S tuple becomes the sub-tuple created by I_S for the first SP tuple.

Likewise, I_P creates, for the first SP tuple, the sub-tuple (Nut, Red, 12, London). Together with the B sub-tuple (S1, P1, 300) and the I_S sub-tuple above, all these sub-tuples form the (entire) SP tuple. Similar calculation determines every other SP tuples. Notice that IAs defined by I_S as well as these of I_P are functionally dependent on the key (S#, P#), as we required for IAs of any SIR. Notice also, that there cannot be an SP tuple with a null I_S or I_P sub-tuple, because of the referential integrity constraint. That is why, both recursive join clauses could safely be inner joins.

Comparison of equivalent queries to S-P1 and S-P2 easily shows how the latter scheme should often spare the logical navigation we spoke about, unavoidable with the former. Consider the client wishing to see names, IDs and quantities of parts supplied by the supplier Smith. Such queries are among the most frequent to countless DBs modelled upon S-P1 since decades. The two queries could be:

(Q1) Select  P#, PNAME, QTY From SP Where SNAME = ‘Smith’ ; /* Query to S-P2

(Q2) Select  P.P#, PNAME, QTY From S, SP, P Where SNAME = ‘Smith’ And S.S# = SP.S# And SP.P# = P.P# ; /* Equivalent query to S-P1 with the inter-relational joins

The query to S-P2 is about as simple as it could be. The query to S-P1 is more procedural. It requires the equijoin clauses between SP and S and P. Such clauses are commonly called logical navigation, [MUV84]. Clients usually dislike the logical navigation, a headache for many, as various forums show. Especially, since the SQL standard advocates these clauses even rather in From clause with explicit Join verb and parenthesis-based algebraic notation. The rationale is the notational compatibility with the outer joins. We show right below the latter notation for half outer joins. We leave to the interested reader the similar rewriting of (Q2).

2. Suppose now that the referential integrity towards S and P is not mandatory for SP. A client wishes therefore to insert into SP at Figure 4, the tuple showing that supplier S7 supplies part P10 in quantity 200, without yet S# = ‘S7’ in S and P# = ‘P10’ in P. Consider, as we do, an insert into an SIR instantiating some SAs being allowed only if one may also calculate then all the IA values then. The insert of the tuple with ‘S7’ as S#, P1 as P# and 200 as QTY would trigger then the evaluation of I_S and of I_P. The former would not produce any matching tuple for S# = S7. Its recursive join clause being an inner one, it would not produce the null I_S tuple neither. Formally, for the tuple under consideration as R’ tuple, i.e., the tuple of SP without IAs of I_S, the I_S recursive inner join would simply not generate any tuple as output for SP. This would violate our requirements on an IE.

The result for I_P would be similar. For any of both reasons, the insert would not go through. In conclusion, neither I_S nor I_P can be IEs for SP anymore. As stated generally in the previous section, recursive half outer joins are now, at least formally, necessary, e.g.,
I_S1 (Select SNAME, STATUS, S.CITY As SCITY, From S Right Join SP on SP.S# = S.S#);
I_P1 (Select PNAME, COLOR, WEIGHT, P.CITY As PCITY From SP Left Join P on SP.P# = P.P#);

I_S1 and I_P1 would now produce in SP the tuple consisting of three sub-tuples \((S7, P10, 200), (null As SNAME...null as SCITY), (null As PNAME...null as PCITY)\). Both IEs apply of course to case (1) above as well. However, as we already said, provided the DBS able to rewrite a recursive inner join into the required half outer one, one could allow I_S and I_P for the current case as well. Users usually prefer inner joins, since outer ones have a well-earned reputation of being tricky if not awkward to use, [DD91]. Perhaps also since, e.g., that the outer joins processing in MsAccess is bugged since the earliest to the current version.

3. The STATUS attribute of S is originally a stored integer. Imagine that it is calculated, e.g., as the total quantity delivered by a supplier divided by hundred and rounded. Thus, the supplier of 100 - 199 parts has status 1 etc. A supplier that does not supply any part for the time being has null as status. If STATUS value changes, while being stored, the client must execute an Update statement. Also, the value would be inaccurate in the meantime. Modern clients may find the issue moderately practical. For S being an SIR, the DBA may rather make STATUS an IA through the following IE. No need for Update queries and the value read by any queries would be always up to date. The IE’s name is STATUS implicitly. We allow this option for any IE inheriting a single attribute.

\[
\text{(Select Int (SUM(QTY) / 100) AS STATUS FROM S LEFT JOIN SP ON S.S# = SP.S# GROUP BY SP.S#).}
\]

Under SRV-model, S can only be a stored relation. If STATUS is to be always up to date, it should be dynamically calculated in a query or the client has to add a dedicated view to S-P1 (conceptual) scheme. The simplest one could be:

Create View Status (S#, Int(SUM(QTY)/100) AS STATUS FROM SP GROUP BY S# ).

However, the immediate practical advantage of S as the SIR is that the simplest query Select * from S would continue to deliver STATUS. With the Status view, the navigational clause S left join SP On S.S# = SP.S# would be necessary. Alternatively, Status should be more procedural. It should inherit also all the other attributes of S and should include the same outer join clause. Each solution of an additional view brings some havoc with respect to our single SIR S. SIR-model clearly alleviates here another fundamental limitation of the “traditional” relational model. We come back to this issue in the next section.

4. Consider finally as a conceptual property of every part, wished to be an attribute of P thus, the list called SUPPLIES, with S#, SNAME and QTY extracted from the full description of each supply of the part in SP. The extracted tuples should be sorted in descending order on QTY. If a part is not supplied for the time being, SUPPLIES should be null. For stored P, such attribute is simply impossible, a situation frustrating perhaps also for some clients. For SIR P, the following IE in P scheme would do. The LIST aggregate function casts for each supplier all the selected tuples into a single Char type value (a string thus), [L3].

\[
\text{SUPPLIES (Select LIST (S#, SNAME, QTY) As SUPPLIES From S, SP where S.S# = SP.S# Group By SP.P# Order By Qty Dsc, SNAME Asc)}
\]

The result, e.g., for P1 would be P tuple:

\((P1, \text{Nut...London}, (S1, Smith, 300 ; S2, Jones, 300)\]

Notice that without the LIST aggregate, SUPPLIES would not be an IE for P. It would inherit indeed more than one tuple for some P tuples, e.g., for that with P1. Observe, finally, that the simplest SQL query Select * From P is now equivalent to that requiring joins between S, P and SP at present, i.e., in S-P1. The latter is more procedural and navigational thus. Actually, without SUPPLIES, the equivalent query would be more involved even with S-P2.@
2.3 DDL Statements for SIRs

Create Table for SIRs defines SAs and IAs basically as above illustrated. We suppose the syntax for SAs is that of the kernel Create Table. Create Table for SIRs may in particular define a stored (only) table, e.g., S and P at the above figures. As one says, who can do more can do less. Also, as mentioned for S-P1, we suppose the Foreign Key clauses possible for SIRs as well.

For views, we suppose basically the reuse as is of the kernel’s Create View statement. The rationale is the absence of the recursive join clause in a view. Reuse of Create View seems easier to implement than a view definition option for Create Table for an SIR. We come back to this issue in Section 4.

The other usual SQL DDL statements are, we recall, Alter Table, Drop Table and Create Index. With respect to the generalization to SIRs of Alter Table, we suppose that it “inherits” from its kernel dialect the clauses Add, Alter or Drop with all their capabilities, to operate on any SAs. We add the extension specific to SIRs, namely that the Alter clause applies also to the IEs. That alteration creates a new IE or drops an existing one. It may transform an SIR into a stored relation or vice versa. Then, Drop Table simply drops as usual the definition and eventually the content of an SIR. The operation should not of course violate the referential integrity. It might thus be required as usual to cascade to other SIRs or get aborted if a violation would result. Furthermore, as for any kernel we are aware of, if a view scheme should get altered, one should use Drop View followed by Create View statement. But, for SIR-model specifically, a view scheme should also have potential to evolve into an SIR, i.e., get some SAs. We suppose for such case the Drop View followed by Create Table. Finally, we suppose Create Index statement the one of the kernel one, applying naturally to the stored or materialized attributes in an SIR only.

Example 2.

1. Suppose that in S-P2 WEIGHT is expressed implicitly in pounds, as apparently in S-P1 originally. Alter P by appending IA WEIGTH_KG converting it to kilograms and IA WEIGTH_T converting it further to tons.

Alter Table P Add (Select X.WEIGHT / 2.1 As WEIGTH_KG From P X, P Where X.WEIGHT = P.WEIGHT), (Select X.WEIGHT_KG / 1000 As WEIGTH_T From P X, P Where X.WEIGHT_KG = P.WEIGHT_KG);

The example defines an SIR that inherits from itself only. We’ll qualify these of self-inheriting. The statement creates two IEs. Each one creates one IA. For convenience, each IA name is then implicitly the name of the IE. Notice that WEIGTH_T IE inherits from WEIGTH_KG values in P. One has to always evaluate thus the latter IE when a query to WEIGTH_T values comes in. We come back to this problem while discussing the implementation of SIRs in Section 4. Finally, while for the sake of the example, we used two IEs, a single one could do, say I_W as follows:

Alter Table P Add I_W (Select WEIGHT / 2.1 As WEIGTH_KG, WEIGHT_KG / 1000 As WEIGTH_T From P X, P Where X.WEIGHT_KG = P.WEIGHT_KG);

Main SQL dialects in fact already allow for both IAs, as so called virtual or computed etc. ones. We come back to this practice in next section. For compatibility, e.g., with SQL Server, the following simpler syntax may do for a self-inheriting SIR as well:

Alter Table P Add I_W (WEIGTH_KG WEIGHT / 2.1, WEIGTH_T WEIGHT_KG / 1000);

2. We alter S by replacing STATUS with the inherited one we discussed.

Alter Table S Drop STATUS Add (STATUS Select Int(SUM(QTY)/100) FROM SP, S GROUP BY S# WHERE S.S# = SP.S#);

Notice that with the altered P and S as above discussed, S-P2 has no more stored relations, only three SIRs. We refer to this DB as S-P3 in what follows.
3. We wish SP to implicitly inherit also eventual alterations of P, adding or dropping attributes there. I_P does not do it. We suppose therefore that the usual SQL (Klein’s) operator ‘*/’ in the dialect used, supports also the syntax */A or */(A1,…,An). The Select list that ‘*/’ generates, does not contain the attributes after ‘/’. We alter the SP scheme in Figure 3 as follows.

Alter Table SP Drop I-P Add I_P_ALL (Select */P.P# From P, SP Where SP.P# = P.P#);

Notice that this Alter refers to the IE I-P by its name (only). Now, if P is altered as in (1) above, SP inherits its new IAs automatically. As usual for ‘*/’ operator, the ‘*/’ may be thus more convenient than an explicit Select list, especially if the latter is long. However, the automatic inheritance may be not for some applications, as widely known.@

2.4 Data Manipulation

Any SIR is a 1NF relation, by definition. The relational algebra operators of SRV-model operate on 1NF relations as defined by their mathematical model, Figure 1. Whether an attribute is a SA or an IA is immaterial to the operators. Each applies thus as is to SIRs as well. One may project, select or join thus any SIRs. The same holds for any SQL Select statements. Including these with value expressions, scalar and aggregate functions, the special clauses: Top k, Group By, Order By...

In short, SIRs do not require extensions to any current DML statements. It may nevertheless be practical to extend the ‘*/’ semantics as discussed in (3) above. For a modification of an SIR, i.e., the SQL Insert, Update or Delete statement, each continues to act for an SA and for an IA as it would act on stored relation or a view with such an attribute. The statement concerning an IA may thus succeed or fail, depending on whether a view update propagates. More precisely, an Insert statement creates as usual any SA value. As in our motivating example, we may require the Insert to commit only if it was also able to successfully calculate all the IAs. The statement may also insert the value of an IA, provided the update may propagate to the source SAs. Create Table may or may not allow it for the referential integrity paths. If the modification cannot propagate, the entire statement should fail, e.g., if an IA is inherited through a value expression or a scalar or aggregate function. Same occurs for Update and Delete statements. The latter deletes as usual physically from the DB all the selected values of the SAs. It also deletes from the SIR the values of all the selected IAs, although conceptually only, of course.

Example 3. The simplest for SP SQL Select statement Select * From SP would show all the SP values, of all SAs and of all IAs in Figure 4. Supposing MS Access SQL as the kernel dialect, would make the statement Insert SP (select ‘S4’ as S#, P4 as P#, 100 as QTY); adding the tuple with these (stored) values and with all its IA values of I_S and of I_P virtually. The statement Update SP set QTY = 250 where S# = ‘S1’ and P# = ‘P1’; should normally succeed, updating one stored value. However, the statement: Update SP set QTY = 250, CITY = ‘Paris’ where S# = ‘S1’ and P# = ‘P1’; may succeed iff a change to CITY value propagates to S. To authorize it requires some thinking. The side-effect would be the city change for every other supply by S1, what may surprise. Next, an update of STATUS of SP may succeed, provided it propagates upward to S. But if it is the IA STATUS above defined, any update to must fail. Finally, the statement Delete SP Where S# = ‘S1’; would erase as usual physically from the DB all the values of the stored attributes in the selected tuples in Figure 4. Conceptually, it would also delete all their inherited values.

2.5 Utility of SIRs

SIRs enhance the usability of the relational model. In the nutshell, an SIR adds to the stored relation it contains, its base thus, the capabilities provided only at the cost of additional view schemes in in SRV-model. The comparison of S-P1 and of S-P3 illustrates the point. As we said already, S-P1 is a template for countless DBs. The benefit from SIRs generalizes accordingly.

For S-P1, with its three relational schemes, a query including any of IAs in SP of S-P3, would require the logical navigation, despised by many clients as well-known. Likewise, only queries could provide the values of S.STATUS or of SUPPLIES as defined for S-P3. These queries would be also rather
procedural, being disliked by many as well. Finally, under strict SRV-model, i.e., without virtual attributes, only queries could calculate WEIGHT_KG and WEIGHT_T. The repetition of the value expressions in such queries is also likely to annoy many.

The only way at present avoiding these annoyances is the creation in S-P1 of additional schemes that define views where all the attributes of S, P and of SP in S-P3 become IAs. A practical solution should contain at least three views, say respectively S', defining IAs having all the values of all the attributes of S in S-P3, then P' similarly for P and, finally, SP' for SP. For the same queries as to S-P3, one has to manage now however six relation schemes at least. This is more complex than some schemes in S-P3.

More formally, to any SRV-model DB scheme with some stored relations $R_i$; $i = 1...N$; may correspond the SIR-model DB with relations $R'_i$; $i = 1...N$, where for each $R_i$, either $R_i = R'_i$, or $R_i$ generalizes to the SIR $R'_i$ with $R_i$ as base. These SIRs spare the logical navigation, otherwise necessary, unless the DB has some additional view schemes. They may also avoid complex queries, necessary if there are no additional dedicated views as well. The logical navigation or more complex queries versus dedicated view schemes is decades’ old dilemma amounting to the proverbial choice between Scylla and Charybdis, [M4]. SIRs appear the first generally practical solution to it. In particular, since they embed two already popular view-saver practices. The one of virtual attributes, amounting to simple arithmetic value expressions in self-inheriting SIRs only at present, we already hinted to. The other, sometimes qualified of implicit joins automatically adds joins to queries.

However, popular DBSs, e.g. SQL Server or MsAccess, limit the practice to QBE interface, although they generate SQL internally, as we discuss soon. SIRs corrects finally conceptual modelling pitfalls of the SRV-model, well-known since Codd’s proposal. These led to the popular Entity-Relationship model, rather useless for DBs with SIRs.

As we finally mentioned, the SRV-model is a strict sub-model of the SIR-model. The trivial condition to stay with the current model, under a hypothetical SIR-model enabled DBS, is simply to refrain from SIRs. Switching to the SIR-model is safe thus. No loss of any current capabilities of a relational DB may result from.

Example 4. Figure 3 and Figure 4 illustrate that if a DBS uses the SIR-model and if for any reasons, we wish S-P1 DB only, it suffices to drop I_S and I_P from S-P2 scheme. Any queries to S-P1 under SIR-model, amount then to these under SRV-model only.

We now justify our claims.

**Conceptual modelling**

For the conceptual modelling, the known goal for a relational DB is to possibly (1) model in the DB scheme all the properties of the “real” object wished for the scheme by most users, (2) have them all possibly in a single stored relation scheme. That is why we use n-ary relations and not, e.g., the once popular binary ones [A74]. A stored relation is not intended however for the calculated values. Those are supposed in views or dynamically calculated in queries. A view in turn cannot have stored attributes. The SRV-model cannot thus fulfill the goal for objects with properties of both types.

Example 5. In S-P1, if STATUS is no more a stored attribute, it cannot be in the stored relation S. A query or some view, say Status, must provide its values instead. No such constraint for S in S-P2.

Another facet of this trouble concerns SP relation in S-P1. It is useful to remind that the relational model illustrated with S-P1 was an instant hit. Most folks attracted to, had no problems with acceptance of S and P schemes. One can easily imagine practical use of these data, despite the inherent simplicity of 1NF. In contrast, many did not swallow SP scheme. It’s hard to imagine an actual supply characterized by the three SP attributes only. At least the names seem a practical must.

Some could think that the obvious way-out was simply to add all the other attributes of S and P as stored ones to SP. Codd has pointed out however already in his original article that it would not be a
wise idea. The so-called today anomalies, initially called strong redundancies, would follow, because of violations of the NFs. We recall that, for one, an important storage overhead could appear. Next, the same value could need many re-inserts, with evident risk of error.

A user could be nevertheless happier with additional attributes despite the anomalies. However, Codd conjectured that most users should be rather unhappy then. According to earlier work, e.g., on the Codasyl model, by definition, the conceptual scheme should be the one most acceptable for the commonwealth of DB users. Codd postulated that the relational conceptual scheme should be therefore that formally minimizing the storage for the set of stored relations free of anomalies, [C69]. Usually, a single relation cannot fulfill this goal. SP for S-P1, but also S and P, seemed, after all, the choice the most conform to these criteria, as widely known. The conceptual modelling insufficiencies for some users should be compensated somehow again by additional dedicated view(s), of S, P and of SP in S-P1 in particular.

But this constitutes the other of the discussed limitations of the SRV-model. The model clearly annoys likely frequent commonwealths of DB users needing conceptual attributes that cannot be in stored relations, as creating anomalies or being inherited only. SP as an SIR in S-P2 avoids these troubles for SP in S-P1 that is the best choice under SRV-model. This would be the case of S and P as well, if extended to SIRs in S-P3, by the inherited, but conceptual attributes in the examples. All three relations would be conform to the Codd’s postulate, provided one restates it as applying to the SIR bases only. In sum, for Codd, for unmotivated reasons, each set of stored attributes to minimize the DB storage for, was a stored relation. The SIR-model shares Codd’s goal. But, a stored attribute, may be accompanied by those not-impacting the storage, i.e., the IAs. The SIRs become the “base”, in the sense of basic, relations for the relational conceptual modelling, not the Codd’s stored (only) ones.

Finally, it’s worth recalling that a heated debate followed the S-P1 scheme proposal almost immediately. It pointed out the limitations we discuss. The popular Entity-Relationship model, (ER), resulted from, [C76]. The ER diagram was recommended as the actual conceptual scheme of a relational DB. This one should be modelled further by the (stored) relation schemes, constituting a kind of internal logical scheme. For S-P1, S & P relations were modelling ER-entities. SP was modelling in contrast “only” the ER-relationship between. The three attribute could suffice then. However, convincing folks that a box of parts they face is “only” a relationship seemed not obvious. Likewise, the question whether a marriage is an entity or a relationship never got a unanimous answer. Again, SIRs avoid such esoteric troubles. There is no more need for the ER-model altogether.

**Logical Navigation, Complex Queries and Value Expressions**

Our next claim has two reasons, occurring in probably any relational DB at present. First, the SRV-model relational design principles that we recalled and also address more in depth in next section, often lead to queries to several relations at once. The mandatory logical navigation through natural inter-relational join clauses results from. It annoys most users since decades, [MUV84]. Likewise, as we pointed out in short already, many queries need results of complex value expressions. Possibly with aggregate functions, GROUP BY, subqueries in Select list or in the Where clause etc. Many users are simply unable to formulate such queries. Additional view schemes, shielding the logical navigation and complex calculus are the only fix for these troubles at present. Managing more schemes is however also an obvious hassle. SIRs may avoid both facets of the dilemma.

Example 6. For our calculated STATUS, the basic SRV-model solution is to create S without it and calculate STATUS in every query needing it. Some S-P1 clients could then nevertheless find the value expression for the STATUS too complex for their taste. Under SRV-model, the only fix is an additional view with all the attributes of S and the STATUS calculation, hiding then its complexity. The penalty is two relation schemes to manage, where one, i.e., S, is then useless for the queries. The SIR S in SP-2 with the inherited STATUS avoids the whole trouble.
The queries (Q1) and (Q2) from Example 1 illustrated how SIR-model may avoid the logical navigation. The only fix hiding the latter under SRV-model would be an additional view, e.g., say V2 with the scheme:

Create View V2 As Select P.P#, PNAME, QTY From S, SP, P Where S.S# = SP.S# And SP.P# = P.P# ;@

More generally, it is easy to see that for any SIR R = (B, V), there is under an SRV-model a couple (B, V'), where B is the stored relation B [R] and V' is the view with all the attributes of R. Each B may then basically be the stored conceptual relation under SRV-model. V' inherits all the attributes of B as they are. It also inherits all the attributes of V, through a formula combining all the IEs of R. It is quite easy to see that such a formula always exists. It may be however pretty complex for several IEs. A hierarchy of views may be then the practical necessity. To avoid the logical navigation and complex queries, SRV-model may thus at least double the number of relational schemes sufficient for same queries under SIR-model. In the same time, fewer schemes of SIR-model do not imply any sensitive performance overhead, as it will appear. All this is a clear advantage of SIR-model as tool for the conceptual modelling.

Virtual Attributes

The next claim was that SIRs generalize in fact some popular practices already beyond SRV-model, as originally defined. The first one is a view-saver usually called *computed, dynamic or virtual* attributes or columns. The concept appeared in 80ties. Major DBSs, Sybase first, picked it up rapidly and still use it. They do it without however, regretfully perhaps, of some research results, [LV86]. Virtual attributes are not stored, but only inherited through simple arithmetic calculations over stored or other virtual attributes in the same relation. A stored relation altered with virtual attribute declarations becomes a self-inheriting SIR.

The IAs WEIGHT_KG and WEIGHT_T in Example 2 are virtual attributes. Self-inheriting SIRs, limited to basic arithmetic value expressions, are thus in fact already widely applied. Virtual attributes are an add-on to SRV-model, as they create SIRs. Strict observance of that model would require the dedicating view with such attributes. Such views would be always computationally sufficient. The concept is thus only a view-saver, intended to enhance the usability. The conjecture appears true. As said, major DBSs propose the concept for decades now. Our examples have shown that SIRs in general provide for by far more extensive calculus capabilities, including the inheritance from multiple relations. These appeared helpful at least for S-P1 becoming our S-P3. S-P1 being the template today for countless actual DBs, we expect similar magnification of these capabilities of SIRs.

Implicit Joins

Research proposed different ways to avoid the logical navigation. The *universal* relation idea we recall in next section was the basis for one group of proposals. The *implicit joins*, sometimes called now also *automatic*, were an alternate [L85], [LSW91]. The universal relation, despite strong excitement, [M04], did not make to popular DBSs. The implicit joins did it, e.g., in SQL Server & MsAccess. Again, the industrial versions limited the research results. The 2016 MsAccess version seems the most extensive up to now. Strange enough, the two MS systems use implicit joins only for the QBE interface. The graphical queries with implicit joins translate to SQL, with the joins added. In a QBE query graph, the implicit joins are directed or undirected arcs. They pop up once one selects the query relations, represented as the graph nodes. Alternatively, through the definition of so-called *sub-tables*, the implicit joins help 4GL forms, called *data sheets*. We detail these terms soon.

The query arcs are derived from directed or undirected arcs, called ambiguously *relations* between tables in a specific diagram of the DB scheme and of views, termed Relationships. The arcs are optionally manually dragged between the diagram nodes that are boxes representing the actual relations, called tables. These may be stored tables or views. One may declare the referential integrity when appropriate and the type of join to be implicit in queries. This can be an inner equijoin (default) or a half outer-join, translated to left or right in SQL. MsAccess may also automatically
propose for the query arcs that are not appearing in the Relationships diagram, provided the DBA permission. In fact this was the initial purpose of implicit joins. In practice, the join attributes, must share the name then and one must be a primary key. The MsAccess automatic join is always an inner equijoin. The attributes involved may be composite. The SQL query generated from can be strange then however, especially for outer joins. The reasons are perhaps clear for Microsoft.

If an arc primary-foreign key exists between two tables, then the table with the foreign one may also automatically become a sub-table, we just spoke about. One can also declare a sub-table more generally, manually among the so-called properties of its super-table. The sub-table is chosen by name and by declaration of an arbitrary atomic attributes per table as implicit join attributes, to select sub-tuple(s) of each super-tuple. Assuming the super-table at the left, the semantic is the implicit join is that of the left equijoin. In this way, e.g., one may declare S a sub-table of SP. MsAccess then automatically chooses SP.S# and S.S# for implicit left join. For unknown reasons, a table may have only one sub-table. If there are several arcs, as it would be for SP, and no manual declaration, one of the arcs is mysteriously preferred. Creation of sub-tables does not avoid the logical navigation in ad-hoc queries. It only let the sub-table tuples to be visible either in as a sub-form of the 4GL form of the super-table, or in the specific view of the super-table, called data sheet view, we mentioned. In the data sheet of SP, for instance, there would be one line for every supply. Right under each such line, one could also see through the implicit join, an on-demand line with all the data of the supplier in S.

The declarations of sub-tables and the arcs of the relationship diagram, spares at least partly the logical navigation. The diagram avoids preexisting views of all these tables, perhaps even the universal one that would be the only way toward the goal under SRV-model. The implicit joins in general are intended as view-savers, e.g., of join clauses between S and SP or SP and P. The implicit joins generated by the MsAccess arcs share the intention. The SIR-model aims at similar capabilities as we showed, but, as for virtual attributes, exceeds potentially the current limitations. E.g. through its IEs, SP can be trivially dealt with as having two sub-tables, S and P, what is impossible for MsAccess at present. Likewise these IEs have the view-saving capabilities for complex values expressions that we discussed. The implicit joins were not even intended for. Summing up, the SIR-model usefully generalizes also that popular practice. Revealing a single umbrella for both discussed practices, what we claimed as well.

The umbrella brings an additional worth mentioning practical advantage on its own. Our examples showed that if there is a choice for an SIR, say R again, multiple IEs should be usually preferable. To avoid the discussed troubles without SIRs, i.e., under the SRV-model, one may create for each IEs a somehow equivalent partial view. This one should have as Create View scheme the IE with the Select clause augmented with the recursive join attributes of R. At the end, one must combine all the partial views into a final one, equivalent to a full view of R (actually, we use this whole approach for SIRs implementation over an existing DBS, discussed later). Using an SIR instead, (the umbrella), one first avoids the partial views through simpler expressions. These are in addition implicitly integrated. Avoiding perhaps the fancy naming conventions on the views, we spoke about, hinting to the common purpose. Most advantageously, the umbrella totally avoids the task of the final view, since the combination of IEs is always implicit. That task, when manual, should usually be boring and error prone, at best. At worst, it could have an unfortunate end altogether, nesting perhaps too many views for DBS operational capabilities.

3. SIR-model Schema Design

The relational scheme design rules have been studied for the SRV-model only. The overall goal was to avoid the anomalies. We now extend these rules to SIRs, for the same goal. We continue with S-P2 as the motivating example. We first restate the NFs. Next, we restate the Heath’s and Fagin’s theorems. The restated theorems generate the same lossless decompositions, but with SIRs instead of the original relations. The benefit is the total avoidance of the logical navigation, necessarily.
generated by the original decompositions, for the restated Heath’s theorem and partial for the Fagin’s decomposition.

3.1 Normal Forms

The basic design rule for a relational DB scheme under SRV-model is the respect of the normal forms (NFs). We recall that these are 1-3NF, BCNF, 4-5NF. Any relation in 5NF is in 4NF that is in BCNF etc. Every relation in SRV-model is by default in 1NF we also recall. Next, relations in 4NF that would not be in 5NF are rare, what makes BCNF and 4NF the most useful in practice. E.g., SP (Sh, P#, QTY) in S-P1 is in BCNF, while SP’ (Sh, SNAME, P#, QTY) with stored attribute SNAME would not be. We’ll give examples of 4NF later. Each NF eliminates some of anomalies we already signaled. E.g., SP’ would need to store SNAME redundantly. Also, SNAME update could erroneously create two different names for same supplier. This could contradict S, where SNAME is anyhow already. Using SP instead, avoids the trouble.

First, recall now that any SIR is in 1NF by definition. Hence no need to restate this NF. The other forms have to be restated for SIRs. Observe in this context that the above anomalies of SP’ would not exist for a view SP’. We therefore state that an SIR R (B, V) is in iNF or BCNF, iff B is in iNF or BCNF. Actually, since R can have null values that were not in the original Codd’s model, we implicitly consider as usual today that NFs apply to relations possibly with as well, e.g., as formally in [JS90].

Example 7. SP in S-P2 is in (extended) BCNF and 4NF, as well as in 5NF even. Indeed, the projection SP [Sh, P#, QTY] on all and only stored attributes conforms to these NFs. Same happens, trivially, for S and P in S-P2. However, as mentioned, the stored relation SP’ (Sh, SNAME, P#, QTY) would not be in BCNF. But, an SIR SP’ with IA SNAME in turn, would be. More generally thus, if, for any reasons, SNAME or any other IA in SP in S-P2 was rather a stored attribute, SP would cease to be in BCNF etc.@

3.2 Schema Design

We recall that at present, i.e. for a SRV-model DB, this process aims on a relational DB the (conceptual) scheme with possibly least number of relations free of anomalies. Usually, it means that every relation has to be proven as in 4NF or as at least in BCNF. The former need occurs if a relation present a (non-trivial) multivalued dependency (MVD). The latter, by far more frequent, characterizes schemes with the functional dependencies (FDs) only. The least number of relations means the grouping of all attributes functionally dependent on the same one(s) into possibly one relation, with the latter as the primary key. Possibly means here the respect of a myriad of other less or more fuzzy criteria, e.g., not “too many” null values for some attributes.

Designing a scheme is furthermore usually a many-steps process. Ideally, we start with the attempt of a single universal stored relation, say U, for the entire DB. U avoids the logical navigation entirely, as all the attributes are in. Unfortunately, chances for U in 4NF are zilch in practice. We usually perform then a decomposition of U into projections, i.e. we suppose that the DB consists of these projections instead. The decomposition must be lossless, producing the projections whose equijoin equals the decomposed relation. Any projection may end up proven in 4NF or proven in BCNF and free of any MVDs. It is then in 4NF thus as well. Or, a projection may not end up so. We decompose any such projections again. We continue, until all projections are anomaly-free.

As known, the two most used decomposition theorems are Heath’s and Fagin’s ones. The former may help with annoying FDs. The latter removes MVDs. Each theorem decomposes a relation into two projections. The resulting scheme has the least possible number of normalized relations for the DB, i.e., is of the smallest size and the optimal one in this sense. Actually, as only a few seemingly know, in presence of both MDs and FDs, Fagin’s theorem must serve first. Otherwise a sub-optimal decomposition may result, meaning the scheme with more stored values than otherwise needed in a scheme nevertheless optimal in the sense we just defined. Even otherwise, there may be several decompositions that are all optimal in the discussed sense. So-called independent projections are
preferable. Their known advantage is the preservation of the FD-cover. Rissanen’s theorem testing the independence of the chosen projections may help.

We now generalize these principles to the SIR-model, i.e., U and the projections may be SIRs. Such schemes were out of scope of the original methodology, of course. In other words, even U may contain IAs, e.g. the aggregate ones we showed. For FDs and MVDs used for the decompositions, we nevertheless originally assimilate all these IAs to SAs. We apply to the projections the restated NFs. Then, in contrast, we consider any IA again as is. For the Heath’s and Fagin’s theorems rested for SIRs, our goal is that the decomposition of an SIR, say R, again, is not only lossless, but also at least one of the projections inherits some, possibly all, attributes of R. The result aimed on is that the lossless decomposition possibly does not cost us the logical navigation through the projections, unlike for the original theorems. We leave for the future eventual restatements of many other rules aimed on best schemes, e.g., the Rissanen’s theorem.

The major gain that will appear below is that, for the same size optimal schemes for a DB, the one using SIRs effectively spares the discussed logical navigation. More precisely, as we’ll show the optimal SIR scheme will be always as follows:

(a) The projections resulting from original Heath’s and Fagin theorems applied to stored relations become bases of SIRs resulting from the restated theorems or remain the same.

(b) In the absence of MVDs, no restated decomposition creates the logical navigation through the projections.

(c) Otherwise, a restated decomposition removing an MVD still spares or at least reduces the logical navigation for some queries addressing the projections, but not to all.

(d) The latter result should concern most of real-life queries.

Indeed, first, the Heath’s theorem states, we recall, that for any stored relation ABC (A, B, C) and an FD A -> B, the decomposition AB (A, B) and AC (A, C) is lossless. That is: ABC (A, B, C) = AB (A, B) Join AC (A, C). In practice, we may have several choices for A,B and C. As every decompositions doubles A, for stored relation ABC, we better choose A with fewest attributes. Also, it is wise making it then the primary key of AB so AB becomes in 3NF at least, as B does not depend on any proper subset of A. Also wisely, for reasons previously invoked, we hunt for the largest B. We restate the theorem for an ABC being a stored relation or an SIR, as the decomposition into AB (A, B) and ACB (A, B, C), where B’ is the IE: B’ (select B from AB where AB.B = ABC.B). This decomposition is also into two schemes and clearly lossless. But, while AC was a stored relation, ABC is an SIR with AC as base. This decomposition is possible only for the SIR-model. Unlike the original one, while it conserves AC as a stored relation of ABC, it also preserves the original attributes A, B, C together. It avoids thus, as promised, the logical navigation to queries selecting B and C.

Next, the Fagin’s theorem also states that in presence of MVD A --> B | C in the presumably stored relation ABC (A, B, C), its decomposition into AB (A, B) and AC (A, C) is lossless. Now, suppose B’ being a (perhaps empty) subset of B such that A -> B’ and let C’ be a (perhaps empty) subset of C, where A -> C’. Actually, we may about always expect either B’ or C’ non-empty, but not both, as in the example that follows. We restate the theorem as follows. Suppose ABC a stored relation or an SIR. The decomposition creates ABC’ (A, B, C’) and ACB’ (A, B’, C) where (i) C’ is the IE C’ (select C’ from AC where AB. A = AC.A) defining C’ thus and (ii) B’ defines B’ as B’ (select B’ from AB where AB.A = AC.A). C’ and B’ avoid the logical navigation for any query to B and C’ or to B’ and C in the projections. Only a query to B/B’ and C/C’ still needs it. We thus do not avoid completely the logical navigation that the decomposition creates. But we limit it to fewer queries. Furthermore, as it will appear the remaining queries should usually have the logical navigation through the final optimal scheme of the DB partly limited. Notice that the restated theorem again conserves each original stored projection as is or as the base of one of the SIR projections.
On these bases, the generic schema generation algorithm for SIRs is quite analogous to that for the stored relations only. More precisely, U remains the starting point, except that it may have IAs upfront. From there, we perform the same, wisely chosen, successive decompositions eliminating MVDs and “annoying”, i.e., anomaly creating, FDs. However, at each step, we now use a restated theorem instead. If we face both dependencies, the restated Fagin’s theorem again should work first. We naturally end up with the same stored relations, hence the same size scheme, but also with lesser need for the logical navigation, as claim (b) states. If there are no MVDs, we remove the discussed logical navigation need entirely, as claim (a) states. Finally, the rationale for claim (c) is that in a real-life DB, MVDs are at most rare with respect to annoying FDs. Also, B’ or C’ should usually have several attributes, unlike B/B’ or C/C’. Even for a decomposed MVD, most queries to the projections should then normally be navigation free as well.

The following example illustrates all the debated points.

Example 8. The biblical S-P1 scheme results from Heath’s theorem only. Similar schemes are countless in practice, as widely known. Our scheme in Example 1 would need the restated Heath’s theorem only. To illustrate also the restated Fagin’s one, we modernize S-P. Each supplier may have email addresses for contact about any of its supplies. Each address is the value of the new stored attribute EMAIL. Every address is for only one supplier. We redesign the S-P scheme under SIR-model accordingly. We call the result S-P4.

We start optimistically with the universal relation U, [MUV84]. In short notation we have:

\[ U (EMAIL, S\#, SNAME, SCITY, STATUS, P\#, PNAME, COLOR, WEIGHT, PCITY, QTY). \]

Notice the necessarily different names for the supplier and part cities, unlike in S and P of S-P1 or S-P2. U is potentially the optimal stored relation for S-P4, unless proven otherwise. What’s easy, since EMAIL already introduces the MVD: S\# \( \rightarrow \) EMAIL \( | \) (SNAME, CITY, STATUS, P\#...QTY). U is not in 4NF thus. Regretfully, U cannot be the optimal S-P scheme. We have to decompose it. We have MVDs and obviously FDs. We start with the restated Fagin’s theorem. The decomposition creates the following relations:

\[ SE (S\#, EMAIL, I\_SP (select SNAME, SCITY As CITY, STATUS from SP, SE Where SE.S\# = SP.S\#)), SP (S\#: SNAME, SCITY, STATUS, P\#...QTY). \]

SE is now an SIR where I\_SP is the IE denoted above as CI. Its base (S\#, EMAIL) would be the stored relation for the original Fagin’s decomposition. SP is the same for both decomposition. We now have thus C’ = (SNAME, CITY, STATUS) and B’ = \( \emptyset \). SE is in the (restated) BCNF. It would not be if any of its IAs, e.g., SNAME, was a stored attribute. The IAs of SE spare the logical navigation to any queries to EMAIL and to any of IAs in I\_SP. Otherwise, these queries would navigate over SE and SP. In contrast queries selecting emails and an attribute in SP that was not inherited in SE would still need to navigate, i.e. would require the SE join SP clause. We come back to these queries later on, showing that practical ones should require lesser navigation anyway, backing up our earlier claim.

SE has no more MVDs, hence it is also in 4NF. SP has no more MVDs neither. But, is not in (restated) BCNF (hence neither in 4NF). The restated Heath’s theorem applies. For all the already discussed reasons, we choose the decomposition:

\[ S (S\#, SNAME, CITY, STATUS), SP (S\#, P\#, PNAME...CITY, QTY, I\_S (Select*/S\# From S, SP Where S.S\# = SP.S\#)). \]

In the projections, we could conveniently rename PCITY and SCITY to simply CITY. The projection SP is again an SIR, with I\_S being B’. The related supposedly stored attributes of the decomposed SP, remain thus preserved in the projection SP, in the form of being as inherited from S. Notice that this does not change anything for SE scheme. S remains the stored relation, as for the original decomposition. It is in BCNF. SP however still isn’t in restated BCNF. Its projection on the stored
attributes indeed isn’t in BCNF in SRV-model, given the FD : P# \rightarrow PNAME, COLOR, WEIGHT, PCITY. We thus apply the restated Heath’s theorem again. One gets SP decomposed to:

\[
P (P#, PNAME, COLOR, WEIGHT, CITY) \text{ and } SP (P#, S#, QTY, I_S (Select*/S# From S, SP Where S.S# = SP.S#), I_P (Select */P# From P, SP Where P.P# = SP.P#)).
\]

Now S-P4 has every relation in BCNF, hence in 4NF, as there are no more MVDs. The optimal scheme is as follows. We underlined the primary key stored attributes.

S (S#, SNAME, CITY, STATUS),
P (P#, PNAME, COLOR, WEIGHT, CITY),
SE (S#, EMAIL, I_SP (select SNAME, SCITY As CITY, STATUS from SP, SE Where SE.S# = SP.S#)),
SP (P#, S#, QTY, I_S (Select*/S.S# *From S, SP Where S.S# = SP.S#), I_P (Select*/P.P# From P, SP Where P.P# = SP.P#)).

Notice that SP scheme is that of S-P2 from Example 1, except for the use of ‘*/’ clause instead of the original lists. That is why the S-P2 scheme is the optimal one as well. Because of IEs, most practical queries is now clearly logical navigation free. However the already signaled queries to SE and SP are not. Some of these queries, e.g., select every P# supplied by supplier with given EMAIL, seem without practical interest. Clients in practice need also names. Then, the restated decomposition still reduces the logical navigation by two joins otherwise necessary, i.e., SE with S to get SNAME and SP with P to get PNAME. We may thus reasonably expect at least some logical navigation spared for practical queries to SE and SP together and for most of such queries to projections of a decomposed MVD in general.

Also, if we did not start decomposing U with the Fagin’s theorem, but with Heath’s one, the result would be the sub-optimal we spoke about. Indeed, the first decomposition of SPE could use the FD : EMAIL \rightarrow S#, leading to:

\[
SE (S#, EMAIL), SP’ (EMAIL, SNAME...P#... I_SE (Select S# From SE, SP Where SE.EMAIL = SP’.EMAIL))
\]

SE is again in BCNF. But now, SP’ is also free from any MVD, hence we do not need Fagin’s decomposition for it neither. However, SP’ isn’t (yet) in restated BCNF. Through successive restated Heath’s theorem decompositions, the final scheme for S-P would be:

\[
S’ (EMAIL, SNAME, CITY, STATUS, I_SE (Select S# From SE Where SE.EMAIL = SP’.EMAIL)),
SE (S#, EMAIL), P (P#, PNAME, COLOR, WEIGHT)
\]

\[
SP’ (P#, EMAIL, QTY, I_SE (Select S# From SE, SP Where SE.EMAIL = SP’.EMAIL), I’_S (Select*/S.EMAIL, From S’, SP’ Where S’.EMAIL = SP’.EMAIL), I_P (Select */P.P# From P, SP’ Where P.P# = SP’.P#))
\]

Now, if a supplier had \( m \) email addresses on the average, \( S’ \) and SP’ would have each \( m \) time more stored values on the average than, respectively, S and SP. We have more stored values than before, i.e., a sub-optimal result, as predicted.

Finally, suppose for S that we calculate STATUS as in Example 1. The only change to S would be the IE defining STATUS:

\[
S (S#, SNAME, CITY, (Select INT(SUM(QTY)/100) As STATUS FROM S, SP GROUP BY S# WHERE S.S# = SP.S#)). @
\]

4. Implementing SIRs

The most practical way towards the SIR-model enabled DBS, is to have SIRs transparently managed by an existing (kernel) SQL DBS. We spoke about abundantly. One way is to create a layer managing the SIRs, say SIR-layer that calls internally the services of the kernel, Figure 5. The SIR-layer would be the only client interface. The DDL and DML statements for SIRs, i.e., at SIR-layer, should extend to
SIRs those of the kernel SQL. The SIR-layer should parse accordingly the former into latter. It should pass the result to the kernel, reformatting perhaps the results.

The SIR-layer should determine from Create Table submitted what type of relation results from. If there is no IE, it will be a stored relation. The statement should be the one of the kernel SQL hence SIR-layer should send it to the kernel as is. Same should occur for Create View at SIR-layer. Generally, the SIR-layer should pass any statement addressing only stored (only) relations or views as is to the kernel.

Creation of an SIR, say R(B, V) is clearly more involved. SIR-layer must represent R for the DBS as B and view(s) somehow representing V. The easiest seems to have within the kernel the full view of R, i.e. as if one could execute there “Create View R’ As Select * From R ;“, renaming then R’ to R. SIR-layer would pass then every query to SIR R to the kernel for the execution as is using the view R instead.

However, the kernel cannot have the full view of any SIR R directly, as SIRs do not exist there. Below, we consider that SIR-layer implements therefore the full view of R by creating in DBS (i) the stored relation B and (ii) a sequence of view(s) representing V from IEs. The view R is the last and the only if there is only one IE. It is defined indirectly from B and the other views. Each of these views results from the previous one and from one IE. This recursive processing follows up the basic semantics outlined in Section 2.1, i.e., without eventual default options for convenience. We now define this processing formally through pseudo SQL notation.

Let Select A From F Where W denote a selection expression for an IE where A denotes the IAs of I and F and W the content of the From and Where clause respectively. Then for any IE I in relation R with R’ defined as in Section 2.1, the following pseudo SQL expression defines the full view of R, say R:

(V1) Create View R As Select R’.*, A From F Where W ;

(V1) suffices if I is the unique in R. This is however only a specific case of the general one we expect, of multiple IEs. We therefore define R more generally, as resulting from the following sequence of view definitions.

Let I_1,…,I_n be the IEs defining R, numbered so to be evaluated in order. The order should let each IE to find all the attributes of its recursive join in the view resulting only previous IEs. For instance, for our example of WEIGHT_KG and of WEIGHT_T, the view with WEIGHT_KG would be created first. Let it be R_0 = B, operationally named R_B in the kernel. Next, let R_1,…,R_n be the views produced each respectively by the evaluations of I_1,…,I_n, as follows:

Create View R_i As Select R_i‐1.*, A_i From F_i Where W_i ; /* i = 1,…n

We create the views R_i in order. R_n is then R.

Indeed, view R_1 is the join of B and of IAs of I_1 through the recursive join clause. B serves there as R’. Likewise R_2 joins all the tuples of R_1, serving as R’ in turn, with the matching values of IAs of I_2 or a null I_2 tuple. Etc.

Practically, to process a Create Table R statement in this way, the SIR-layer first replaces every eventual inner join in the recursive join put there for convenience, with the appropriate left requests. Likewise, it completes the eventual IEs formulated using implicit clauses. Then, the SIR-layer requests from the kernel to create the stored relation R_B. Next, it creates the view R_1 named, say, R_1, then R_2 as R_2 etc. until the view R_n named R. Alternatively, SIR layer could create a single view defining R, using the imbricated left join expression combining all other views R_i and R_B. This strategy could however backfire. Numerous imbrications could exceed the operational limit of the DBS.

Example 9.1. Consider R = SP from Example 1.3, with IEs I_S and I_P thus. We have R_0 = (S#, P#, QTY). Let it be also I_1 = I_S with its IAs (SNAME, STATUS, S.CITY) thus and I_2 = I_P defining (PNAME, COLOR,
WEIGHT, P.CITY) for SP we recall. We could choose the other way, i.e., \(I_1 = I_P\). For our choice nevertheless, when Create Table SP comes in to SIR-layer; this generates for the kernel DBS:

Create Table SP_B . . . from all and only stored attributes of SP, in \(R_0\) above.

Create View SP_1 As select SP_B.*, SNAME, STATUS, S.CITY From SP_S Left Join S On SP.S# = S.S# ;

Create View SP As select SP_1.*, PNAME, COLOR, WEIGHT, P.CITY From SP_1 Left Join P On SP.P# = P.P# ;

The SIR-layer obviously should submit the three statements to the kernel as an atomic transaction, i.e., within Begin ...Commit brackets. From now on, SIR-layer passes every incoming select query to SIR SP to DBS as the query to view SP. It passes all the SA values resulting from an Insert to SP_B. IAs there go to the views if they may propagate to the stored relations or directly to those. Etc.

Figure 5 illustrates a possible implementation of our S-P3 DB from Example 2. Recall that it has only SIRs in its scheme. Its SP relation is an SIR with its usual SA and IAs inherited from S through IEs I_S and I_P in Example 1. Relation S is also an SIR, with S-P1 attributes of S, except for STATUS inherited from SP in Example 1.4 and SUPPLIES from Example 1.6. Finally, P is a self-inheriting SIR with its usual SAs, but also with IAs WEIGHT_KG and WEIGHT_T, calculated in Example 2.

The SIR-layer at the figure shows the three SIRs as rectangles. Each size reflects the number of tuples and the number of attribute values per tuple as seen by the client. This perception corresponds to Figure 4, augmented with the IAs proper to S-P3. The lower part shows under the same convention the stored relations and views possibly implementing S-P3 over some current DBS. We suppose the views created in the order from the bottom towards the top of the figure. The view names for P stress the mandatory order on IEs, since WEIGHT_T needs WEIGHT_KG. The view names for SP illustrate the IE sequence in the above example. The sequence of IEs for S is also arbitrary, could be the other way around. SIR-layer would generate all the schemes at the DBS layer using the rules just stated. Each rectangle length is thus the same as for the SIR-layer rectangle. But it is not so for the width, of the stored relations especially.

As the figure shows, for the three SIR schemes, the SIR-layer would generate nine relational schemes in DBS. Three would be the stored relation schemes. Hence, they would define the relational conceptual scheme of S-P in the SRV-model. The views should be there to help the queries to S-P with the logical navigation and value expressions. Without SIRs, the user or DBA wishing simpler queries would need to basically create and compose the six manually. The virtual attributes could
spare both views for P and for P only. The currently available implicit join capabilities would spare nothing. As said, the SIR-layer could alternatively generate only a single view per every SIR, hence perhaps by far more complex. It would lead to the minimum of two schemes per SIR in an actual DBS, or even only one if SIR-layer generates the virtual attributes. But, the single-view strategy could end up a bad idea for multiple IEs, as pointed out. Finally, to appreciate the non-procedurality gain our sample SIRs may bring, perhaps formulate to S-P1, assumed in SRV-model, the query equivalent to the following simple one to S-P3: Select SNAME, SUPPLIES, PNAME From SP Where STATUS > 2 And Weight_T > 1

The Alter Table and Drop Table statements for an SIR, say R again, also require more processing than their kernel counterparts. For the former, the kernel may in particular create/drop views R_i when R one gets/loses IAs. Likewise, to process the latter, it must drop R_B and all the views R_i. We skip further tedious, details.

Performance wise, the kernel storage for SIR R is in practice the one for R_B. The storage for the views is negligible provided these are not materialized, as we supposed. We have also shown that the optimal scheme with SIRs has the same size and the same stored attributes as the optimal one for SRV-model. Hence, the storage for the values of these SIRs within the kernel is the same. The optimal DB with SIRs for some application should cost thus negligibly more than the optimal DB with the stored relations only for the same goal. Finally, for the above processing scheme, the query parsing overhead by the SIR-layer appears negligible as well. Altogether, perhaps surprisingly, the enticing capabilities of SIRs should come practically without operational overhead.

5. Conclusion

A relation with stored and inherited attributes, i.e., a stored and inherited relation, (SIR), as we called it, appears a useful construct for a relational DB. An SIR may be free of anomalies of a stored relation with the same attributes. In addition, through its inherited attributes, an SIR scheme may be more accurate as a data (conceptual) model. SIRs alleviate in this way the well-known limitations of stored relations, the dark side of the normalization. The popular ER model, proposed precisely because of those limitations, becomes rather useless.

The major practical gain should be typically less procedural queries. The SIR-layer appears from this stance simply a higher level interface to the relational DBs. At first, through the reduced logical navigation for the optimal conceptual scheme of the same size as for the stored relations only. Next, SIRs may avoid also the views often necessary at present for user’s comfort, hiding complex value expressions. The inherited attributes of an SIR generalize for both goals the already popular practices of virtual attributes and of implicit joins. Finally, the implementation of the SIRs using an existing DBS looks rather easy and without operational overhead in practice. Better late than never, the existing DBSs should get improved accordingly.

The design rules for SIRs based on restated NFs and Heath’s and Fagin’s theorems appear about as easy as the current rules. However, the decompositions based on these two theorems exclusively, are only the tip of the iceberg of known proposals. Future work could adapt those proposals to SIRs as well. Perhaps, by starting with the rules for the independent projections already mentioned. Next, one could look upon the lossless decompositions using outer joins, [JS90]. We also mentioned the formal analysis of the well-formed IEs.

With respect to that subject, one may further observe also that all three constructs, i.e., SRs, IRs and SIRs root in a common 1NF construct. One could call it relation with stored or inherited attributes, or stored or inherited relation, say (SoIR) in short. As the name hints, a SoIR may be a stored one, or a view or an SIR. The construct may look like a theoretical brainstorming only at present. Observe however that our Create Table for an SIR is in fact that one of a SoIR, except for a view. Our design rules apply to SIRs, but to SoIRs as well. Summing up, whether one sees our work as on SIRs or on specific SoIRs is just a matter of taste.
SoIR construct roots itself in a still more general 1NF construct that one may call relation with stored or inherited attribute values. The idea is of attributes possibly mixing stored and inherited values. A stored value of such an attribute whenever present, overrides the eventually inherited value. This may be practical, as outlined in [LKR92]. For instance if color of a part in P is green and must be so, while an S-P2 user of a supply involving the part rather sees it as light green, then an update overriding the color in SP could bring the global happiness. Future work could explore that issue as well.

All things considered, the future work should nevertheless start with the implementation of SIRs over a popular DBS, e.g., MySQL, along the lines we defined. Whatever DBS one chooses, the result should be a win-win deal. Finally, most of major DBSs are now interoperable multidatabase systems, [LA86]. SIRs with multibase IEs seem attractive as well.

References

[C70] Codd, E., F. A Relational Model of Data for Large Shared Data Banks. CACM, 13,6,1970.