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 single construct with both types of attributes. Examples showed the 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 to queries the customary logical navigation through join clauses and complex value expressions. Both annoyances are unavoidable at present, unless hidden behind dedicated views. These are cumbersome in turn. Our construct appears the first practical solution to this decades’ old dilemma. Better late than never, existing DBMSs should easily accommodate it, with almost no storage and processing overhead.

1. Introduction

The universally applied Codd’s (relational) model, [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) often called also base one, or simply a relation or a (relational) table, has stored (base) attributes (columns) only. A view, also called Inherited Relation (IR), has only the inherited attributes. Those get values from SRs or from views through a stored expression of some data manipulation language (DML), e.g., a stored SQL query. The two constructs are dichotomic. In 1992, we proposed a non-dichotomic 1NF construct that was a relation with both stored and inherited attributes, [LKR92]. Examples showed the construct attractive. The idea seemed promising also for OODBs, à la mode in these times.

No one followed the lead however. Below, we revisit our proposal thoroughly. We call our construct Stored and Inherited Relation, (SIR). It adds up to the current constructs of SR and of IR. An IR is supposed henceforward to also perhaps inherit from an SIR. We qualify of SIR-model the data model resulting from our proposal. We refer to Codd’s model as to SRV-model (Either Stored Relation or View model). We believe the reader familiar with the SRV-model and SQL in particular.

We show that SIR-model adds useful capabilities to all those of the SRV-model. The latter are naturally preserved. We restate the relational scheme design rules to include SIRs. It will appear, perhaps surprisingly, that a relational DB should often advantageously consist of SIRs only. We show the implementation of the SIR-model over an existing DBMS rather easy and almost without storage or 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 for SIRs. 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 hidden behind dedicated views. These are however also cumbersome, enough to be rarely practiced.

Section 3 continues with the SIR-model specific schema design rules. We restate for SIRs the NFs other than 1NF. We also restate the Heath’s and Fagin’s lossless decomposition theorems. The restated theorems create specific SIRs instead of the usual projections. The decomposition continues to be lossless, but usually is also “logical navigation less” over the projections. It is totally so for the

---

¹ Université Paris Dauphine - PSL, LAMSADE, Paris, France, witold.litwin@dauphine.fr
restated Heath’s decomposition and mostly, in practice, for the restated Fagin’s one. Section 4 discusses the implementation of SIRs over existing DBMSs. Section 5 draws finally the conclusions and overviews the future work.

2. SIR Model

2.1 Schema Definition

Figure 1 illustrates 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 there, is also a view for the SIR-model. The inverse is not true, as also stated. Next, any SIR or a view in SIR-model is a finite subset of a Cartesian product of atomic attributes over some domains, subject to any algebraic or predicative operations, and aggregate or scalar functions, applying to 1NF relations. A stored attribute, (SA), of an SIR has usual stored values. An inherited attribute (IA) results from a stored formula, called from now, inheritance expression (IE) that we detail soon. An inherited value is basically not stored. However, as known, it might, becoming a materialized one.

Let R (S, V) be now some with SIR with S being all the SAs, and V all the IAs. We require the primary key of the projection R[S] to be also a key of R. We thus always have card (R) = card R[S].

![Figure 1 SRV-model versus SIR-model](image)

Also, any key of R[S] is also a key of R, obviously. V’s scheme may result from one or multiple IEs.

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. A single IE defines all the IAs of V through a formula that could be technically of some view, say V’. V’ would be a projection of V on all its IAs, i.e., without perhaps duplicated tuples. Multiple IEs may be more convenient, as we’ll show. Each IE defines then some but not IAs for R, hence for every inherited tuple it creates. Any such tuple augments some tuples already in R. It may happen that a tuple in R does not inherit any value from an IE. This is like if it inherited a null value, i.e., like if it got preserved by a half outer join. For multiple IEs, every IE defines a view, say I, with IAs being a strict subset of V. We require all such subsets to be disjoint. The attributes of V are their union. Finally, I’s should not present any name conflicts. All IA names should be thus different, original names being perhaps renamed.

Below, we define any IE using an SQL Select. As for a view at present, this seems the most convenient. In particular any IE should have a name that may be explicit or implicit as it will appear. The name may serve a DDL (Data Definition Language) statement to refer to the IE, as we’ll show as well. Next, although each IE contributes to define R, we require any IE, identified to “its” I from now on, to refer to R in From clause of I, say as R’, as well as in a specific recursive join clause in Where clause, we’ll discuss soon. These references operationally specify how each inherited tuple augments these already in R. For this purpose, we consider R’ as R with all the SAs and IAs and their values defined by S and all the IEs supposed evaluated prior to I. All R’ tuples are in this way in the Cartesian product formed by From clause of I. Let R’I denote the relation formed from all and only super-tuples of this product matching the Where clause of I, whose attributes are thus R’.* and I.*. Next, let I’ denote the final projection of I on attributes defined in the Select clause of I. Then, we form R as
Select R’. *, I’. * From R’ Left Join R’I. The join denotes here the natural equijoin, i.e., the equijoin on all the common attributes, hence all these of R’. 

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.

To respect the condition of always having card (R) = card R[S], we finally request from every IE that the key of R’ remains the key of R. The Select clause should act then so that the join attributes that are all these of R’ besides, make the final projection selecting for each tuple of R’, only one tuple with from R’I. As wished, all this preserves for R every tuple t’ of R’, while forming from t’, also a single tuple t in R. As wished too, to form t, we augment t’ either with the matching I tuple, also necessarily a single such tuple or a null value for each I attribute otherwise.

We call well-formed an IE or their collection for V if the result fulfils all the above requirements. Otherwise, the IE or the collection is invalid. We show the examples soon. We leave the exhaustive analysis for the future.

We consider finally that the DDL and DML statements for SIR-model use as the kernel some favorite SRV-model’s SQL dialect. SIRs specific clauses expand the kernel. Especially, this concerns the Create Table statement that also has to define IAs. Any SAs are then defined as usual by the (kernel) dialect. The IAs use Select expression(s) of the dialect that could be in Create View under SRV-model, i.e. if all the relations referred to were stored ones or views. For a SIR, the expression however in addition refers to R itself, as already discussed. Actually this reference may be sometimes implicit for convenience, as we’ll show. The following motivating example illustrates all these points. We come back to details of Create Table statement as well as to the remaining other DDL statements for SIRs afterwards.

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 from three stored relations named S, P, SP. This scheme is optimal for the so-called relational design rules, using NFs. SP respects also the referential integrity by default. There cannot be thus a supply declared by a tuple in SP, but with the supplier or the part not yet declared in S and P. However, in practice, the referential integrity of SP would be today enforced only if defined by optional Foreign Key clauses. An application may indeed prefer not to enforce it, or only partly.
1. Figure 3 shows the S-P2 scheme. It is also optimal, as it will appear. The referential integrity is again the default. 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 key attributes, as usual. The S, P relations have SAs, but not IAs, i.e. V is empty. They were stored relations in S-P1 and remain as is in S-P2. SP keeps the original, i.e., from S-P1, SAs with their values. Referring to our generic notation above, for R = SP thus, these SAs form R[S]. SP also keeps the original primary key. SP however also carries now IAs. It became thus an SIR. The IA values of SP form V in our generic notation. The choice of IAs means that the DBA considers every property of a supplier or of a part, as also, ipso facto, of any supply. We examine the rationale behind later. The SQL Select statement in Create Table statement expresses a single IE defining all these IAs. We named the IE as I_SP.

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

Figure 3 S-P2 scheme.

### 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>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table P</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P#</td>
<td>PNAME</td>
</tr>
<tr>
<td>P1</td>
<td>Nut</td>
</tr>
<tr>
<td>P2</td>
<td>Bolt</td>
</tr>
<tr>
<td>P3</td>
<td>Screw</td>
</tr>
<tr>
<td>P4</td>
<td>Screw</td>
</tr>
<tr>
<td>P5</td>
<td>Cam</td>
</tr>
<tr>
<td>P6</td>
<td>Cog</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table SP</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>S#</td>
<td>P#</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>S2</td>
<td>P1</td>
</tr>
<tr>
<td>S2</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>

Figure 4 S-P2 content. IAs are in Italics.

Here, V attributes and values are all and only created by I_SP. V thus have all and only IAs in S and P, except the key attributes S.S# and P.P#. The two CITY attributes are also renamed. All the resulting IAs form I’ above. When I_SP is being evaluated, SP referred to in From clause, R’ above, supposedly contains all and only the values of its SAs, i.e., of (S#, P#, QTY). Here, X = R’V has all and every tuple t
selected by I_SP, but augmented with the R’ values contained in the tuple of R’ x V from which t resulted. E.g. augmented with (S1, P1, 200) for v ∈ V ; v = (Smith, 20…) that 1st line in SP in Figure 4 inherits. The final SP in Figure 4 and Figure 4, i.e., R above, is SP := (S#, P#, QTY) left join (S#, P#, QTY, SNAME…PCITY) AS X on SP.S# = X.S# and SP.P# = X.P# and SP.QTY [SP.S#, SP.P#, SP.QTY, SNAME…PCITY]. The novelty of SP, making it an SIR here, is that it has SAs and IAs brought by I_SP. The final SAs are all and only also in SP as R’. Also, given the current content of S, P and R’, the result has no tuples augmented with null values.

I‐SP preserves the key of SP as R’ for the resulting SP as R, as required. Indeed both relations have (S#, P#) as the key, while S# and P# are respectively the keys of S and P. The From clause produces conceptually the Cartesian product (SP.S#, SP.P#,QTY, S.S#…, P.P#…). For every SP.S# and SP.P# value then, the product may contain at most one tuple matching the recursive join clause in I_SP we spoke about that is here: SP.S# = S.S# and SP.P# = P.P#. The Where clause selects this only tuple. The final join may produce then at most one tuple from that one, given, again, that (S#, P#) is the key of SP as R’. The values of these attributes continue therefore to identify each tuple created. Hence, they remain the key of the resulting SP.

For instance, for the SP tuple with key (S1, P1) in Figure 4 both values being here SAs, the product has only one tuple (S1, P1, 300, S1, Smith, 20, Red, 12, London). The join clauses select this one only. The projection removes the duplication of S1, P1 forming a tuple of V’. The final join produces the SP tuple (S1, P1, 300…London). The first three values are the stored ones, all the others are inherited. This is the 1st SP tuple in Figure 4. The similar evaluation produces all the other tuples there. If there were in addition some stored tuples in SP not matching on join clauses any tuple in the product, these would be concatenated with the adequate null values. Finally, I_SP reveals a well-formed IE.

2. Suppose now that the referential integrity is not desired for SP. In addition to the S‐P2 content at Figure 4, one could wish in SP the tuple showing, e.g., that supplier S7 we have the supplies part P1 in quantity 200, without yet S# = S7 in S. The insert of the SAs into SP, would make I_SP to produce the tuple (S7, P1, 200, null as SNAME…null as PNAME…null as P.CITY). This, despite the tuple with P1 in SP. One may find this result an undesirable side‐effect, as well as vice versa for a hypothetical part P7 supplied by S1. If so, the following IE could do:

I_SP1 (Select SNAME, STATUS, S.CITY, PNAME, COLOR, WEIGHT, P.CITY From (SP Left Join S On SP.S# = S.S#) Left Join P On SP.P# = P.P#)

Here, the 1st left join, on SP and S, produces the relation, say X, inheriting all the values in S when join attributes SP.S# and S.S# match. X furthermore preserves now also any tuple stored in SP that would not match any tuple in S.S#. The 2nd left join preserves every tuple of X. It further expands each either with the tuple of P matching SP.P# value or with nulls. Hence we may have in final SP values inherited only from S, or only from P, or both. Notice that we could commute the left joins.

3. Observe that we have in fact in SP two sources of inheritance: S and P. An IE per source may seem more appropriate, perhaps also simpler, than a single IE for both. For sure, the result is easier to imagine in customary graphical form where IEs would be arcs. The following two IEs could do:

I_S (Select SNAME, STATUS, S.CITY S.CITY As SCITY from S,SP WHERE SP.S# = S.S#) and I_P (Select PNAME, COLOR, WEIGHT, P.CITY As PCITY from P,SP WHERE SP.P# = P.P#).

Each IE is less procedural than I_SP and I_SP1, especially. At least, since each is shorter. It is also likely from the example that most users should rather prefer more, but simple IEs than less, but complex ones. Observe that I_SP and I_P commute. Hence, one may evaluate them in any order. Also, easy to see that each is well¬formed as well as the set they form. We leave as easy exercise to find out the single equivalent IE.

4. The STATUS attribute of S is a stored one, simply an integer. Imagine however that behind, it is in fact somehow calculated, e.g., as the total quantity delivered by a supplier divided by hundred and
rounded. E.g., the supplier supplying 100 - 199 parts total has status 1 etc. If S is an SIR, one may rather define STATUS as an IA, through the following IE. We allow that one, as any IE creating a single attribute, to bear implicitly the attribute name. Here, the implicit name would be STATUS.

(Select Int(SUM(QTY)/100) AS STATUS FROM S, SP GROUP BY S# WHERE S.S# = SP.S#).

Under SRV-model, S can only be a stored relation. Hence, the stored value would need to be updated from time to time, being possibly inaccurate in the meantime. All this does not sound practical. Rather, STATUS should not be then in S, but should be dynamically calculated in a query or should "emigrate" to a dedicated view. 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 join clause on S and Status would be necessary. Alternatively, Status should be more complex, inheriting not only S# and QTY, but also all the other attributes of S and including the same join clause as our IE. Each solution brings some havoc with respect to single SIR S. SIRs clearly alleviate here a fundamental limitation of the ‘traditional’ relational model. We come back to this important issue in the next section.

5. Consider as conceptual property of every supplier, wished an attribute of S so, the list of the supplies it provides, called SUPPLIES, with P#, PNAME and QTY for each supply. The list elements should be sorted in descending order on QTY. If the supplier does not supply anything for the time being, SUPPLIES should be null. The following IE in S scheme defines the related inheritance from SP. The LIST aggregate function casts for each supplier all the selected tuples into a single Char type value (a string thus), [L3].

(Select LIST (P#, PNAME, QTY) As SUPPLIES From S, SP where S.# = SP.S# Group By SP.S# Order By Qty Dsc, PNAME Asc)

The result for S1 would be S tuple:

(S1, Smith...London, (P3, Screw, 400 ; P1, Nut, 300 ;...P6, Cog, 100)),..., (S5..., Athens, null)

Notice that without the aggregation, SUPPLIES would be invalid. It would possibly inherit more than one tuple for some S# values, e.g., for S1. Also, it would not let S# to remain the key of S, hence of its stored part neither. Observe, finally, that the simplest SQL query Select * from S is now equivalent to that requiring basically a half outer join between S and SP at present, i.e., in S-P1. The latter is more procedural thus. Actually, without SUPPLIES, the equivalent query would be more involved even with S-P2.@

As mentioned for S-P1, and following the SRV-model, we suppose also for SIRs that optional Foreign Key clauses may accompany an SIR Create Table specifying the referential integrity details. Also, analogously to views in SRV-model, IEs basically do not cascade upward the updates to IAs. They might however, with additional usual dialect-dependent clauses. Within the limits already heavily studied for view updates.

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. 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 is then easier to implement than the alternative parsing of Create Table with a view definition as an option as well. 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 first statement for SIRs, we suppose that it “inherits” from its kernel dialect the clauses Add, Alter or Drop, operating on stored attributes, as well as all the related capabilities. The
specifically for SIRs supposed extension is that the Alter clause applies also to the IEs. That alteration creates a new expression or drops an existing one. It may transform an SIR into a stored relation or a view, 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 otherwise result. Furthermore, as for any kernel we are aware of, if a view scheme should get altered, one should use Drop View and Create Table statement. In particular, one should apply this procedure to provide a view with SAS, i.e., to alter it to an SIR in our context. Finally, we suppose Create Index statement the one of the kernel one, applying naturally to the stored or materialized attributes only in an SIR.

Example 2.
1. Suppose WEIGHT expressed implicitly in pounds. Alter P by appending IA WEIGHT KG converting it to kilograms and IA WEIGHT T converting it further to tons.

Alter Table P Add (Select WEIGHT / 2.1 As WEIGHT KG From P, P Where X.WEIGHT = P.WEIGHT), (Select WEIGHT KG / 1000 As WEIGHT T From P, P Where X.WEIGHT KG = P.WEIGHT KG);

The example defines SIRs that inherit from themselves. We’ll qualify these naturally of self-inheriting. Each IE again implicitly bears the name of the attribute it creates. Notice that, this time, they are to be evaluated in order. In fact, we used two IEs to illustrate this case. A single IE say I_W could indeed also do:

Alter Table P Add I_W (Select WEIGHT / 2.1 As WEIGHT KG, WEIGHT KG / 1000 As WEIGHT T);

Main SQL dialects would in fact allow defining both attributes as so called virtual or computed etc. attributes. We come back to this practice in next section. For compatibility, the following simpler syntax could do for an IE in a self-inheriting SIR:

Alter Table P Add I_W (WEIGHT / 2.1 As WEIGHT KG, WEIGHT KG / 1000 As WEIGHT T);

For the same reason, the IE(s) could be the other way around: ... WEIGHT KG WEIGHT / 2.1, WEIGHT T AS WEIGHT KG / 1000... .

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

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

Notice that after the alterations to P above and to S here, S-P2 has no stored relations anymore, only SIRs.

3. We wish SP to implicitly inherit also eventual alterations of S or P, adding or dropping attributes there. I_SP 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. Here A designates the attribute(s) that ‘*’ does not generate in its Select list. We alter the SP scheme in Figure 3 as follows.

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

Notice that this Alter refers to the IE I-SP by its name (only). Observe also that with all the Alter statements taken care of, S-P2 would still have its three relations only, but they would all become SIRs. Actually, it would be the optimal scheme for our DB, as it will appear in Section 3.@

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 involved is a SA or an IA is immaterial to these 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, no extension to any DML statement of an SQL dialect is required to accommodate SIRs. An
extension to ‘*’ semantics may nevertheless be practical as we show soon. For a modification of an
SIR, i.e., the SQL Insert, Update or Delete statement, it continue to act for an SA and for an IA as it
would act on stored relation or a view with such an attribute. The effect on a SIR may be then a
combined one. More precisely, an Insert creates as usual any SA value. It might insert a value of an IA
as well, provided the update propagates to the source SAs. The whole insert may therefore
alternatively 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. As for SRV-model, a modification may cascade upward or
downwards along referential integrity paths. A modification of an IA may lead to a physical deletion
elsewhere as well.

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. The Insert statement of the MS Access SQL dialect, Insert SP (select
’S4’ as SH, P4 as P#, 100 as QTY); would add the tuple with these (stored) values and, therefore, with
all those inherited through I_SP or I_S and I_P etc. The statement Update SP set QTY = 250 where
SH = ‘S1’ and P# = ‘P1’; should normally succeed, updating one stored value. However, the statement:
Update SP set QTY = 250, CITY = ‘Paris’ where SH = ‘S1’ and P# = ‘P1’; may succeed iff a change to
CITY value propagates to S. To authorize it requires some thinking. The side-effect that the city
would change also for any other supply by S1 may indeed surprise. Next, an update of STATUS as SA
in S may succeed and as an IA in SP as well, 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
SH = ‘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 ones.

2.5 Utility of SIRs

SIR-model enhances the usability of the relational model. It does it through new capabilities of SIRs,
while keeping all those of the SRV-model. Examples above already hinted to such capabilities. In a
nutshell, an SIR adds to a stored relation the capabilities provided otherwise only by dedicated views.
One avoids then the cumbersome management of those. One also avoids perhaps the navigation
through them and the stored relation. More generally, on the theoretical side, SIRs may avoid the
conceptual modelling pitfalls of the SRV-model, well-known since the relational model was proposed,
but without a satisfactory solution within the model. On the practical side, they may avoid the usual
logical navigation in queries along the primary-foreign key logical access paths, [MUV84]. They may
finally avoid complex values expressions in queries.

Both the logical navigation and the value expressions are often necessary in queries at present,
unless hidden behind dedicated views. As we hinted already, managing views is however also a
hassle, including the inclusion in queries of the logical access paths to these in turn. This decades’ old
dilemma was basically the proverbial one, of choosing between Scylla and Charybdis, [M4]. SIRs
appear the first generally practical solution, i.e., avoiding to queries both the logical navigation and
complex expressions, without dedicated views. The result should be useful. One reason is that these
capabilities of SIRs expand and put under a single umbrella in fact those of two already popular
practices. One practice, called mainly virtual attributes, amounts to self-inheriting SIRs with simple
arithmetic value expressions. The other, sometimes qualified of implicit joins helps with the logical
navigation.

To justify our claims, we focus on S-P1. Countless actual DBs use nevertheless S-P1 as the template.
The benefits we show generalize accordingly. We also 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 DBMS, is simply to refrain from SIRs. Switching to the SIR-model is a safe move 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 DBMS uses the SIR-model and for any reasons, we wish S-P1 DB only, it suffices to drop I_SP from the 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 basic 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, (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 only 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 it instead. No such constraint for S in S-P2, benefiting from the IE STATUS.

Another facet of this trouble concerns the 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 attributes there only. Names at least 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 be a bad 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, for Codd, most users should be unhappy. The result would not be a conceptual scheme therefore. According to already earlier work, e.g., the Codasyl model, by definition, that scheme should indeed be most acceptable for the commonwealth of DB users. Codd postulated that the relational conceptual scheme should be therefore the one formally minimizing the storage for the set of stored relations, [C69]. SP as in 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 selected users should be compensated somehow again by additional dedicated view(s), the basic (universal) one of S, P and of SP in S-P1 in particular. But this would constitute the other of the discussed limitations of the SRV-model. SP as an SIR in S-P2 avoids all these troubles. This would be the case of S and P as well, if extended to SIRs as in the examples. All would be conform to the Codd’s postulate, provided restated as applying to the storage for SIRs. Obviously, this would be in practice equal to the storage for stored attributes only. In a nutshell, for Codd, for unmotivated reasons, each set of stored attributes to minimize the storage for was a stored relation. The SIR-model shares Codd’s storage goal. But, a stored attribute, may be accompanied by attributes not-impacting the storage in its relation, i.e., the IAs. The SIRs are then rather the “base” relations, not the Codd’s stored (only) ones.

Finally, it’s worth recalling that a heated debate followed the SP scheme proposal. 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, using SIRs, avoids such esoteric troubles. There is no more need for the ER-model altogether.

**Logical Navigation and Complex Value Expressions**

Our next claim has two reasons, occurring in probably any relational DB at present. First, the well-known relational design principles that we recall in next section, often lead a query to address several relations. The mandatory logical navigation through inter-relational join clauses, sometimes called *natural*, results from. It annoys most users since decades, [MUV84]. Likewise, a query may need results of a complex value expression. Possibly with aggregate functions, GROUP BY, subqueries in Select list or in the Where clause etc. Many users are simply unable to formulate complex queries. The only fix to both issues at present is the additional views, shielding the logical navigation and complex calculus. Managing additional schemes is however another hassle. Full SIRs avoid both facets of the dilemma.

Example 6. Some S-P1 users could clearly find the value expression for the STATUS too complex for their taste. Under SRV-model, a dedicated additional view must hide it then. The SIR S in SP-2 avoids this trouble. Next, consider a stereotype request, say Q, to either DB, selecting in every supply, the IDs and names of the supplier and part involved, together with the quantity supplied. Let query Q₁ express then Q for S-P1 and query Q₂ for S-P2. These could be basically as follows:

(Q₁) Select S#, SNAME, P#, PNAME, QTY from S,P,SP Where S.S# = SP.S# and P.P# = SP.P#;
(Q₂) Select S#, SNAME, P#, PNAME, QTY from SP;

The joins in Q₁ are unavoidable for any equivalent query to S-P1. They are due to the logical navigation through S, P and SP. In contrast, neither Q₂ as one sees, nor its equivalencies need the navigation. Hence, all spare the join clauses.

**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 DBMSs, Sybase first, picked it up rapidly and still use it, without, regretfully perhaps, however of some research results, [LV86]. Virtual attributes are not stored, but calculated through value expressions, from other attributes in the same stored relation or view. The relation may have then both stored and inherited attributes. It is thus an SIR. More precisely, it is a self-inheriting one, further limited basically to simple arithmetic value expressions only.

The IAs WEIGHT_KG and WEIGHT_T in Example 2 define virtual attributes. Self-inheriting SIRs are thus in fact already widely applied. Full SIRs obviously provide for more complex calculus capabilities, by far, through the inheritance from multiple relations. Examples proved these helpful at least for S-P. That DB is the template for countless actual ones, as widely known.

The virtual attributes are an add-on to SRV-model since they create SIRs in fact. Strict observance of that model would require instead the dedicating view with such attributes. Such views would be always computationally sufficient. The concept is thus only a view-saver, presumed to enhance the usability. The conjecture appears true. As said, major DBMSs propose the concept for decades now. More general than self-inheriting only SIRs should accordingly help usability of numerous actual DBs as well.

**Implicit Joins**

Research proposed several ways to usually avoid the logical navigation. One group of proposals was based on the *universal* relation idea we recall in next section. The *implicit joins*, sometimes called
now also *automatic*, were an alternative idea [L85], [LSW91]. The universal relation, despite strong excitement, [M04], did not make to popular DBMSs. The implicit joins entered, e.g., SQL Server & MsAccess. As for the virtual attributes, the industrial versions limited the research results. The MsAccess version seems the most extensive up to 2016. 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, so called *data sheets* especially. We recall 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 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. Alternatively, the joins may be tried out automatically from the query, provided the DBA permission. In fact this was the initial purpose of implicit joins. In practice today, the join attributes must share the name then and one must be a primary key. The automatic join is always an inner equijoin. The attributes involved may be composite. The SQL query generated from can be often strange then however. 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 that is 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, avoid the logical navigation. They do it without some preexisting view of all these tables, perhaps even the universal one, that would be the only way toward the goal under SRV-model. The implicit joins act as view-savers, like join clauses between $S$ and $SP$ or $SP$ and $P$, as the implicit joins generated by the arcs also do etc. But, in addition, other discussed IEs may offer the virtual attribute did for their goal. The practice is popular with major DBMSs already for decades, despite its limitations. The SIR-model aims at similar capabilities, but, as for virtual attributes, potentially, beyond the current limitations. E.g. through its IEs, $SP$ can be trivially dealt with as having two sub-tables, $S$ and $P$. Likewise these IEs avoid the view-saving complex values expressions avoidance capabilities we discussed. The implicit joins do not provide these. Summing up, the SIR-model usefully generalizes also this popular practice. Revealing finally a single umbrella for both discussed practices, what we claimed as well.

The umbrella role 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 way is to create for each IEs a somehow equivalent partial view. This one should have as Create View scheme the IE augmented with the join attributes with $R$ in the Select clause. At the end, one must combine
all the partial views into a final one, equivalent to a full view of \( R \). Using an SIR instead, (the umbrella), one first avoids the partial views through simpler expressions. Those 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 should usually be boring and error prone, at best. At worst, it could have an unfortunate end altogether, nesting perhaps too many views for DBMS 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. We show lossless decompositions benefiting from SIRs. These decompositions avoid some logical navigation, necessarily introduced by the “classical” ones. It will appear that Heath’s decomposition benefits more from the novelty.

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., \( SP (S#, P#, QTY) \) in S-P1 is in BCNF, while \( SP’ (S#, SNAME, P#, QTY) \) 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 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 \( R(S, V) \) is in iNF or BCNF, iff \( R[S] \) is in iNF or BCNF.

Example 7. \( SP \) in S-P2 is in (extended) BCNF and 4NF, as well as in 5NF even. Indeed, the projection \( SP (S#, P#, QTY) \) 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 \( SP’ (S#, SNAME, P#, QTY) \) with stored attribute SNAME 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 may 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 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 thus 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 again any such projections. We continue, until every projection is anomaly-free.

As know, the two basic decomposition theorems are Heath’s and Fagin’s ones. The former may help with annoying FDs. The latter removes MVDs. 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, i.e., leading to more stored values, may result. Both theorems decompose a relation into two projections. Hence 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. Nevertheless, several lossless decompositions of a relation through these theorems usually exist. Then, 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 assimilate all these IAs nevertheless to SAs. We naturally apply to the projections the restated NFs. These in contrast, consider any IA as is. We’ll now also restate for SIRs the Heath’s and Fagin’s theorems. The goal is that a decomposition of an SIR R is not only lossless, but also at least one of the projections preserves some, possibly all, attributes of R. The result aimed on is that the scheme with the projections possibly as “logical navigation less” as was R. This will appear possible only through the projections being SIRs. We leave for the future the possible restatements of others of many known rules, intended to help with even better schemes, e.g., the Rissanen’s theorem.

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

- In the absence of MVDs, no decomposition introduces the discussed logical navigation.
- Otherwise, a decomposition removing an MVD may still spare the logical navigation to some queries addressing the projections, but not to all.
- For a real-life DB, we may reasonably expect the discussed logical navigation spared to most or even all 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, if we have choice we tend to choose A wisely with fewest attributes. Possibly, we choose A also the primary key of AB, in 3NF at least then. Also wisely, for reasons already invoked, we hunt for the largest possible B. We restate the theorem for an ABC being an SIR, to the decomposition into AB (A, B) and ABC (A, B, C), where ABC.B is an IA defined by the IE: (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. This decomposition is thus possible only for the SIR-model. Unlike the original one, it preserves the attributes A,B,C together, in resulting ABC. 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. Let us now denote as B’ a (perhaps empty) subset of B and as C’ a (perhaps empty) subset of C such that A -> B’ and 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: the decomposition creates AB (A, B’, C) and AC (A, B’, C) where the IE (select C’ from AC where AB. A = AC.A) defines C’ and the IE (select B’ from AB where AB. A = AC.A) defines B’. The result avoids thus the navigation for any query to B and C’ or
to B’ and C in the projections. Only the queries to B/B’ and C/C’ still need it. We thus do not avoid completely the logical navigation that the decomposition creates. But we limit it to fewer queries.

On these bases, the generic schema generation algorithm for SIRs is then 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 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 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-P3.

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 possibly the optimal stored relation for S-P3, unless proven otherwise. What’s easy, since EMAIL already introduces the MVD: \( S# \rightarrow \rightarrow EMAIL \mid (SNAME, CITY, STATUS, P#...QTY) \). U is not in 4NF thus. Regretfully, the optimal S-P scheme cannot thus be U (only). We have to decompose it. We have MVDs and obviously FDs. We start as above indicated with the restated Fagin’s theorem. The decomposition creates two relations:

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

SE is now an SIR, while it would be a stored relation (and SIR) only for the original Fagin’s decomposition. We have \( C’ = (SNAME, CITY, STATUS) \) 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 make many queries navigation free. Otherwise, these queries would need to navigate over SE and SP. For instance, the query that one may expect frequent, selecting every email of a supplier with given name. 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. Such queries, e.g., all emails and all names of parts supplied by a supplier, seem nevertheless here clearly of by far lesser practical interest than those to SE, saved from the navigation, like the cited one.

SE has no more MVDs, hence 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:

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

In the projections, we could by the way more conveniently rename PCITY and SCITY to simply CITY.

The projection SP is again an SIR, with the same attributes as the decomposed one. Some became now however inherited from S. Notice that this does not change anything for SE scheme. S is an SIR and in BCNF, whether thus restated or not. SP however still isn’t. Its projection on the stored
attributes indeed isn’t in SRV-model, given the FD : P# -> PNAME, COLOR, WEIGHT, PCITY. We thus apply the restated Heath’s theorem again to SP. On the It gets decomposed to:

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

Now S-P3 has every SIR in BCNF, hence in 4NF, as there are no more an MVD. The optimal scheme is as follows. We underlined the primary key stored attributes.

\[ S (S#, SNAME, CITY, STATUS), \]
\[ P (P#, PNAME, COLOR, WEIGHT, CITY), \]
\[ SE (S#, EMAIL, \text{ (select SNAME, SCITY As CITY, STATUS from SP, SE Where SE.S# = SP.S#))}, \]
\[ SP (P#, S#, QTY, \text{ (Select */S# *From } S, SP \text{ Where } S.S# = SP.S#), (Select */P.P# \text{ From } P, SP \text{ Where } P.P# = SP.P#)). \]

Notice that SP scheme is that of S-P2 from Example 1. That is why the S-P2 scheme is the optimal one as well. 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 one we spoke about. Indeed, the first decomposition of SPE could use the FD : EMAIL \(\rightarrow\) S#, leading to:

\[ SE (S#, EMAIL), SP (S#, SNAME... \text{ (Select EMAIL From SE Where SE.S# = SP.S#))}) \]

SE is again in BCNF, SP is also free from any MVD, but isn’t (yet) in BCNF. Through successive decompositions of Heath’s theorem, the final scheme for S-P would be:

\[ S (S#, SNAME, CITY, STATUS) \text{ SE (S#, EMAIL) } P (P#, PNAME, COLOR, WEIGHT) \]
\[ SP' (P#, EMAIL, QTY, \text{ (Select */S.EMAIL, P.P# From } S, P \text{ Where } S.EMAIL = SP.EMAIL \text{ AND P.P# = SP.P#}) \]

Now, if a supplier had \(m\) email addresses on the average, SP’ would have \(m\) time more stored values than SP. Clearly, we got a sub-optimal result.

Finally, suppose STATUS calculated in Example 1. The only change would be the IE defining it in S. I.e., we would have:

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

If we had also an IA, say STATUS1 in S, defined as the number of parts supplied, the following IE would trivially define both STATUS and STATUS1:

\[ \text{(Select INT(SUM(QTY)/100) As STATUS, COUNT (*) As STATUS1 FROM SP GROUP BY S# WHERE S.S# = SP.S#).} \]

Observe that we could restate Fagin’s theorem so to avoid completely the logical navigation through AB and AC. The price would be the view (only) SIR, being in fact the additional view: ABC (select A, B, C From AB, AC Where AB.A = AC.A). We do not feel this price worth in practice. Our \textit{credo} is, we recall, the decompositions avoiding the creation of any additional view. However the decompositions with additional views for MVDs only still lead to a minimal scheme. But, for the goal of the full avoidance of the logical navigation creation, i.e., even for MVDs. So, some may feel our current criterion perhaps too stringent, after all.

4. Implementing SIRs

The most tempting way towards the SIR-enabled DBMS, is to have SIRs transparently managed by an existing DBMS with some kernel SQL dialect. We spoke about abundantly. One way is to create a layer managing the SIRs, say SIR-layer that calls internally the services of the DBMS, Figure 5. The DDL and DML statements for SIRs, i.e., at SIR-layer, should extend to SIRs those of the DBMS as we
discussed. The SIR-layer should parse accordingly the former into latter. It should pass the result to DBMS for the execution, reformattting perhaps the returned 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 pass to the DBMS as is. Same should occur for Create View at SIR-layer. Other DDL statements and DML ones addressing only stored (only) relations or views should pass to DBMS as they are as well.

Creation of an SIR, say R(S, V) is clearly more involved. SIR-layer must represent R for the DBMS as S and view(s) somehow taking to the account V. The easiest seems to generate the full view, i.e. Select * From R as if R was a stored relation or a view in the DBMS. Within the DBMS, this view could be called R. SIR-layer would pass then every query to SIR R to DBMS for execution using the view R instead.

However such a view is impossible to declare directly from SIR R, as that one is unknown to DBMS. Below, we consider that SIR-layer implements therefore Create Table R by creating in DBMS (i) the stored relation S 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 S and the other views. Each of those views results the previous one and one IE. This recursive processing follows up the semantics outlined in Section 2.1. We now define it formally through pseudo SQL notation.

Let \( I_1, \ldots, I_n \) be the IEs defining R, to be evaluated in the order \( I_1, I_2, \ldots \). Let it be \( R_0 = S \), operationally named \( R_S \) in the DBMS. Next, let \( R_1, \ldots, R_n \) be the views produced by successive evaluations of \( I_1, I_2, \ldots, I_n \), each as in Section 2.1. The last view \( R_n \) is the view R, as we’ll show. Next, let \( R_j^i \) denote the relation formed from all and only tuples of the Cartesian product of relations in \( I_j \) from clause, matching the Where clause of \( I_j \). These attributes are therefore \( R_{j-1}^i \) and all the attributes of all the other relations named there. Next, let \( R_j^{i'} \) denote the final projection of \( R_j^i \) on attributes defined in the Select clause of \( I_j \). Finally, let \( A_j \) be the attribute(s) of \( R_{j-1} \) and \( B_j \) be the other attribute(s) there that together form the recursive join clause of \( I_j \). We form \( R_j \) as:

\[
\text{Select } R_{j-1}^i \ast, R_j^{i'} \ast \text{ From } R_{j-1} \text{ Left Join } R_j^i \text{ On } A_{j-1} = B_j.
\]

In this way, to process a Create Table R statement, the SIR-layer first request from the DBMS to create a stored relation, say \( R_S \), with the stored attributes S of R. Then, SIR creates the view \( R_1 \) as, say \( R \_1 \), then \( R_2 \), named \( R \_2 \) etc. until the view R. Alternatively, SIR layer could create a single view defining R, using the imbricated left join expression combining all other views \( R_j \) and S. This strategy could be however problematic. Cumulative imbrications could exceed some operational limit of the DBMS.

Example 9.1. Consider \( R = SP \) from Example 1.3, with IEs \( I_S \) and \( I_P \) thus. We have \( R_0 = SP_S = \{S\#, P\#, QTY\} \). Let it be also \( I_1 = I_S \) inheriting (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, SIR-layer generates for the DBMS:

Create Table SP_S .... from all and only stored attributes of SP.

Create View SP_1 As select SP_S.*, 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 three statements should obviously get submitted to DBMS as an atomic transaction, i.e., with Begin Transaction...Commit brackets. SIR-layer then passes every incoming select query to SIR SP to DBMS as the query to view SP. It passes an insert to SP_S. Etc.

Figure 5 illustrates a possible implementation of the following variant of our SIR-based S-P DB, say S-P3. We suppose this one to have SP with its usual SA and IAs with the latter inherited from S.
through IEs I_S and I_P in Example 1. Next, S ignores EMAIL from Example 8, having only its S-P1 attributes, but with STATUS inherited from SP in Example 1.4 and SUPPLIES from Example 1.6. Similarly, P has its usual SAs, but also IAs WEIGHT_KG and WEIGHT_T, calculated through the two IEs in Example 2. The upper part shows the three relations at SIR-layer. Recall from the Alter statement example that all three are SIRs. They result Create Table and may be Alter Table statements we discussed, entered at SIR-layer. Unlike any current DB, S-P1 in particular, S-P3 has thus no stored (only) relations.

The respective dimensions of each rectangle at SIR-layer reflect the size of the content for the user of the represented SIR. That is, the number of tuples and the number of attribute values per tuple. This 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-P over some current DBMS. 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 DBMS 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 DBMS. Three would be the stored relation schemes. Hence, they would also 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 DBMS, 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 benefit the above SIRs bring here, perhaps formulate to S-P1, assumed in SRV-model, the query equivalent to the following simple one to S-P3: Select SNAME, PNAME From SP Where STATUS > 2 And Weight_T > 1@.

The Alter Table and Drop Table at SIR-layer are also clearly more involved than at present. The former may in particular create or drop views, when a stored table gets/loses IAs. It may also drop a view and create one or more new ones, when IEs get added, altered or dropped. We skip further easy, but tedious, details.
Performance wise, the storage for an SIR R is in practice the same as for R_S. 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 under the dialect DBMS is the same. The storage for the views is negligible provided these are not materialized, as we suppose. The optimal DB with SIRs should cost thus negligibly more than the optimal DB with the stored relations only for the same application. Finally, for the above processing scheme, the query parsing overhead by the SIR-layer appears negligible as well. Altogether, perhaps surprisingly, the new capabilities that SIRs bring, come thus practically without operational overhead.

5. Conclusion

The 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 limitation 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 DBMS looks rather easy and without operational overhead in practice. Better late than never, the existing DBMSs 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 a SIR. The construct may look like a theoretical workout only at present. Observe however that our Create Table for a 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 attributes possibly mixing stored and inherited values. A stored value of such attribute whenever present, overrides some otherwise 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 DBMS, e.g., My-QL, along the lines we defined. Whatever DBMS is chosen, the result should be a win-win deal. Finally, most of major DBMSs are now interoperable multidatabase systems, [LA86]. SIRs with multibase IEs seem therefore attractive as well.

References


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


