Relations with Stored and Inherited Attributes

Witold Litwin\(^1\)

(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 1\(^{st}\) Normal Form (1NF) thus. A Stored Relation, \(\text{SR}\) 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 \(\text{IR}\), 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, \(\text{SIR}\). 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 \textit{SIR-model} the data model resulting from our proposal. We refer to Codd’s model as to \textit{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 the primary key of R. We thus always have card (R) = card R[S].

Also, any key of R[S] is also a key of R. V’s scheme may result from one or several IEs. 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’. The reference serves the operational specification of 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’.
To respect the condition of always having $\text{card } (R) = \text{card } R[R[S]]$, 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 $R'$. As wished, 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 fulfills 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 must in addition refer to $R$ itself, as already discussed. Actually this reference may be sometimes implicit for convenience, as we’ll show. Also, since Create Table for SIRs defines IAs, the Create View statement becomes useless in SIR-model, except for the backward compatibility. We discuss SIR specific clauses that appear for the other usual SQL DDL statements later on.

The following example illustrates all the discussed points. We use the biblical Codd’s Supplier-Part relational DB, originally illustrating his proposal. We refer below to its probably most known version, 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 S-P as to S-P1. We call the restated one 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 2 shows the S-P2 scheme. It is also optimal, as it will appear. The referential integrity is again the default. Figure 3 shows example extensions of $S$-$P$2 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 $I_{SP}$.

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 $R' \times V$ from which $t$ resulted. E.g. augmented with $(S1, P1, 200)$ for $v \in V'$; $v = (Smith, 20...)$ that 1st line in $SP$ in Figure 3 inherits. The final $SP$ in Figure 3, i.e., $R$ above, is $SP := (S#, P#, QTY)$ left join $(S#, P#, QTY, SNAME...PCITY) \text{ AS } X$ on $SP.S# = X.S#$ and $SP.P# = X.P#$ and $SP.QTY = [SP.S#, SP.P#, SP.QTY, ...]$. 

- 3 -
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.

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

Figure 2 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>
<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 3 S-P2 content. IAs are in Italics.

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 join clauses in I_SP, i.e., 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 3 both values being here SAs, the product has only one tuple (S1, P1, 300, S1, Smith, 20, London, P1, Nut, Red, 12, London). The join clauses select
this one only. The projection removes the duplication of \( S_1, P_1 \) forming a tuple of \( V' \). The final join produces the \( SP \) tuple \((S_1, P_1, 300...London)\). The first three values are the stored ones, all the others are inherited. This is the 1st \( SP \) tuple in Figure 3. 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 3, 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 \((S7, P1, 200, \text{null as } SNAME...\text{null as } PNAME...\text{null as } P.CITY)\). 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:

\[ I_{SP1} (\text{Select } SNAME, \text{ STATUS, S.CITY, } \text{ PNAME, COLOR, WEIGHT, P.CITY From } (SP \text{ Left Join } S \text{ On } SP.S# = S.S#) \text{ Left Join } P \text{ On } SP.P# = P.P#) \]

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, but also preserving all the \( SP \) stored tuples, if any, without matching \( S.S# \). The follow up left join preserves now all the tuples of \( X \), but expands each of them either with the tuple of \( P \) with the matching \( P.P# \) 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:

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

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 \( I_{SP1} \) 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 \( \text{STATUS} \) 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 \( \text{STATUS} \) 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 \( \text{STATUS} \).

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

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, \( \text{STATUS} \) 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:

\[ \text{Create View Status } (S#, \text{ Int}(\text{SUM}(\text{QTY})/100) \text{ AS STATUS FROM } SP \text{ GROUP BY } S# ) \]

However, the immediate practical advantage of \( S \) as the \( SIR \) is that the simplest query \( \text{Select * from } S \) would continue to deliver \( \text{STATUS} \). With the \( \text{Status} \) view, the join clause on \( S \) and \( \text{Status} \) would be necessary. Alternatively, \( \text{Status} \) should be more complex, inheriting not only \( S# \) and \( \text{QTY} \), 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 \( \text{SIR} \) \( SIRs \) 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.

The other usual SQL DDL statements are, we recall, Alter Table, Drop Table and Create Index. With respect to the first statement for an SIR, 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 then that the Alter clause applies also to the IEs. That alteration consists simply of a new expression. Next, the Drop Table simply drops as usual the definition and the eventual 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. Finally, Create Index statement for an SIR is the same, except that it naturally concerns the stored or materialized attributes only.

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 X, P Where X.WEIGHT = P.WEIGHT), (Select WEIGHT_KG / 1000 As WEIGHT_T From P X, 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 From P X, P Where X.WEIGHT_KG = P.WEIGHT_KG);

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: … \text{WEIGHT KG} \ \text{WEIGHT / 2.1, WEIGHT T AS WEIGHT KG / 1000}….

2. We alter \( S \) by replacing \text{STATUS} with the inherited one.

Alter Table \( S \) Drop \text{STATUS} Add (Select \text{Int}(\text{SUM(QTY)/100}) \text{ 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 \text{SIRs}.

3. We wish \( SP \) to implicitly inherit also eventual alterations of \( S \) or \( P \), adding or dropping attributes there. \text{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 \text{A} designates the attribute(s) that ‘\(*’ does not generate in its Select list. We alter the \( SP \) scheme in Figure 2 as follows.

Alter Table \( SP \) Drop \text{I-SP} Add \text{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 \text{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 \text{SIRs}. Actually, it would be the optimal scheme for our DB, as it will appear in Section 3.@

2.2 Data Manipulation

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

In short, no extension to any DML statement of an SQL dialect is required to accommodate \text{SIRs}. An extension to ‘\(*’ semantics may nevertheless be practical as we show soon. For a modification of an \text{SIR}, i.e., the SQL Insert, Update or Delete statement, it continue to act for an \text{SA} and for an IA as it would act on stored relation or a view with such an attribute. The effect on a \text{SIR} may be then a combined one. More precisely, an Insert creates as usual any \text{SA} value. It might insert a value of an IA as well, provided the update propagates to the source \text{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 \text{SAs}. It also deletes from the \text{SIR} the values of all the selected \text{IAs}, although conceptually only, of course. As for \text{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 \text{SAs} and of all \text{IAs} in Figure 3. The Insert statement of the MS Access SQL dialect, Insert \( SP \) (select ‘S4’ as S#, P4 as P#, 100 as QTY); would add the tuple with these (stored) values and, therefore, with all those inherited through \text{I-SP} or \text{I-S} and \text{I-P} etc. 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 \text{CITY} value propagates to \text{S}. To authorize it requires some thinking. The side-effect that the city would change also for any other supply by \text{S1} may indeed surprise. Next, an update of \text{STATUS} as \text{SA} in \text{S} may succeed and as an IA in \text{SP} as well, provided it propagates upward to \text{S}. But if it is the IA \text{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 3. Conceptually, it would also delete all their inherited ones.
2.3 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 2 and Figure 3 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 WEIGTH_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 same 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 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
The declaration 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, if 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#, PH, 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, PH, 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
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 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:

- 12 -
In the absence of MVDs, no decomposition introduces the discussed logical navigation.
(b) Otherwise, a decomposition removing an MVD may still spare the logical navigation to some queries addressing the projections, but not to all.
(c) 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 AC 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 illustrates 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# ->> EMAIL | (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’ = ∅. 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 left gets decomposed to:

- P (P#, PNAME, COLOR, WEIGHT, CITY) and SP (P#, S#, QTY, (Select*/S# From S, SP Where S.S# = SP.S#), (Select*/P# From P, SP 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, (select SNAME, SCITY As CITY, STATUS from SP, SE Where SE.S# = SP.S#)),
- SP (P#, S#, QTY, (Select*/S.S# *From S, SP Where S.S# = SP.S#), (Select*/P.P# From P, SP 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 -> S#, leading to:

- SE (S#, EMAIL), SP (S#, SNAME... (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), SE (S#, EMAIL) P (P#, PNAME, COLOR, WEIGHT)
- SP’ (P#, EMAIL, QTY, (Select*/S.EMAIL, P.P# From S, P Where S.EMAIL = SP.EMAIL 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:

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

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:

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

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 *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 for creating an 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 implement a layer managing then SIRs, say SIR-layer, using the services of the DBMS, Figure 4. The DDL and DML statements for SIRs, i.e., at SIR-layer, should extend to SIRs such statements of the DBMS dialect. The SIR-layer should parse accordingly the former into latter. It should then pass the result to DBMS for the execution, reformattting perhaps the results.

The SIR-layer may represent every SIR as is, by creating the dialect’s stored relation sharing the name and rest of the scheme. Likewise, every view SIR may be passed to the DBMS provided only the reformattting of its Create Table statement to Create View. Other DDL statements and DML ones (queries) about stored-only or view SIRs, also pass as they are.

Creation of an SIR, say R(S, V), is in contrast clearly more involved. R is indeed neither a stored relation nor a view hence no actual DBMS can get the SIR-layer’s Create Table parsed as above. SIR-layer must represent R for the dialect as S and view(s) somehow representing V. The easiest seems to have at least the full view of R, i.e. defined as Select * From R, if R was a stored relation or a view for the DBMS. This is of course impossible for an SIR under the DBMS. One must reconstruct this view then from B and from the view(s) implementing V. This is (fortunately) always possible through the following recursive processing. We define it using pseudo SQL notation.

Let \( I_1, ..., I_n \) be the IEs defining R, to be evaluated in order. Let K be the key of B. For each \( I_j \), let \( I_j^c \) be the view formed from \( I_j \) by adding K to all the attributes in Select clause of \( I_j \) with values being these of K in each tuple selected by \( I_j \) within the Cartesian defined by its From clause. Let it be \( R_0 = S \) and let \( R_1,...,R_n \) be the views produced by successive evaluations of \( I_1, I_2, ..., I_n \), starting with S. Then \( R_n \) is the full view of R, resulting from the following recursive formula. For \( j = 1, ..., n \), one should evaluate:

\[ R_j = \text{select } R_j-1.* , I_j .* \text{ From } R_{j-1} \text{ Left Join } I_j^c \text{ On S.K = } I_j^c.K \]

Indeed this formula results from the characteristic properties of any SIR detailed in Section 2. The left join formula here transforms the original one there for operational use. We using K instead of entire \( R_{j-1} \) denoted as R’ there. K should usually have by far less attributes than any \( R_j \). It creates thus less join clauses and making the overall calculation faster.

In this way, to process a Create Table R statement, the DBMS may first create a stored relation, say R_S, representing R. Then it may store each \( I_j^c \) as a view, say \( I_j^c_F \), produced each from IE \( I_j \) with the renaming of R to R_S in \( I_j \) at least. Some \( R_j \) may need alternatively the renaming to \( R_{j+k} ; k > 0 \), when it
refers to an IA created by \( R_j \). For instance, in Example 2, \( I_2 \) is \( \text{WEIGHT}_T \), hence \( R_2 \) would refer to \( R_1 \) where \( I_1 \) named \( \text{WEIGHT}_K \) creates the IA \( \text{WEIGHT}_K \). Each \( R_j \) may be a temporary view, say named \( R_j \_T \), except for \( R_n \) renamed simply \( R \). These views would be produced by DBMS while it evaluates a query. Alternatively, the DBMS could keep them persistent as all the others. Yet alternatively, SIR layer could dynamically create only a single view defining \( R \), defined then by a single imbricated left join expression, combining all views \( R_i \). This strategy seems less general however. Several imbrications could exceed the operational possibilities of a DBMS. Whatever is the strategy for the views, he SIR layer passes afterwards any query to its SIR \( R \) to DBMS. This one processes the query as is, but towards the view \( R \). It sends the result back to the SIR-layer.

Example 9. 1. Consider \( R = \text{SP} \) from Example 1.3, with IEs \( I_1 \_S \) and \( I_1 \_P \) thus. We have \( R_0 = \text{SP} \_S = (\#S, \#P, \text{QTY}) \) and \( K = (\#S, \#P) \). Let it be also that \( I_1 = I_1 \_S (\text{SNAME}, \text{STATUS}, \text{S.CITY}) \) and \( I_2 = I_1 \_P (\text{PNAME}, \text{COLOR}, \text{WEIGHT}, \text{P.CITY}) \), although it could be the other way around. Now, the SIR layer generates \( I_1 \_F \) as:

Create View \( I_1 \_S \_F \) As select \#S, \#P, \text{SNAME}, \text{STATUS}, \text{S.CITY} From \text{SP} \_S, \text{S} Where \text{SP} \_S.S# = \text{S}.S# ;.

Also, it declares \( I_2 \_F \) as:

Create View \( I_1 \_P \_F \) As select \#P, \text{PNAME}, \text{COLOR}, \text{WEIGHT}, \text{P.CITY} From \text{SP} \_S, \text{P} Where \text{SP} \_S.P# = \text{P}.P# ;.

When a query comes in or before, if the temporary view definitions should stay persistent, the DBMS represents \( SP_1 \), then \( SP_2 \) successively as:

Create View \( SP_1 \_T \) As (select \#S, \#P, \text{QTY}, \text{SNAME}, \text{STATUS}, \text{S.CITY} From \text{SP} \_S Left Join \text{I} \_S \_F On \text{SP} \_S.S# = \text{I} \_S \_F.S# And \text{I} \_S \_F On \text{SP} \_S.P# = \text{I} \_S \_F.S#) ;

Create View \( SP_2 \) As (select \#S, \#P, \text{SNAME}, \text{STATUS}, \text{S.CITY} From \text{SP} \_S Left Join \text{I} \_P \_F On \text{SP} \_S.S# = \text{I} \_P \_F.S# And \text{I} \_P \_F On \text{SP} \_S.P# = \text{I} \_P \_F.P#) ;

The view \( SP \) is the full one we spoke about initially. The SIR-layer passes any query to SIR SP to DBMS as the same query but to view SP.

Figure 4 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 \( \text{EMAIL} \) from Example 8, having only its S-P1 attributes, but with \( \text{STATUS} \) inherited from SP in Example 1.4 and \( \text{SUPPLIES} \) from Example 1.6. Similarly, P has its usual SAs, but also IAs \( \text{WEIGHT} \_K \_G \) and \( \text{WEIGHT} \_T \), calculated through the two IEs in Example 2. The upper part shows the three IAs. Recall from the discussion of the Alter statement that they are all SIRs. In other words, unlike for any current DB, S-P1 in particular, S-P3 has no stored (only) relations. The dimensions of each rectangle reflect the size of the content for the user, i.e., the number of tuples and the number of attribute values per tuple in each relation in Figure 3, plus the two IAs for P. The lower part shows the stored relations and views possibly implementing S-P over some current DBMS. These are generated by the SIR-layer following the rules we just stated. The dimensions of each stored relation rectangle again reflect the number of tuples and their size, in number of (stored) attributes. The length is thus the same as for the SIR, but not the width. The dotted rectangles represent the temporary views. These could be alternatively permanent as well, as discussed.

As the figure shows, for the three SIR schemes, the SIR-layer would generate fifteen 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 would 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 create and compose all twelve manually. The virtual attributes could spare views for P and for P only. The currently available implicit join capabilities would spare nothing. As said, the SIR-layer could alternatively generate even only a single, hence perhaps by far more
complex, view scheme per SIR. It would lead to the minimum of two schemes per each of our SIRs to represent it in an actual DBMS. But, this strategy could end up a bad idea, 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@
processing overhead in practice. Better late than never, the existing DBMSs should get expanded 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 a 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 finally observe also that all three constructs, i.e., SRs, IRs and SIRs root in a common 1NF one. One could call it relation with stored or inherited attributes, or stored or inherited relation (SOIR), in short. Such construct looks may look just a formal exercise at present. Perhaps a nice conceptual feature is nevertheless 1:1 correspondence of a SOIR with a formal 1NF relation, instead of the 1:3 in Figure 1. Our SIR-model design rules, as well as any existing ones, besides obviously apply to SOIRs. There is however may be more to explore there.

On the practical side, the future work should start with the implementation of SIRs over a popular DBMS, e.g., My-SQL, 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.