Stored and Inherited Relations

Witold Litwin

(March 2016)

Abstract. The universally applied Codd’s relational (data) model has two constructs: a stored relation, with stored attributes only and a view that has only the inherited ones. In 1992, we proposed an additional construct, mixing both types of attributes. Examples showed it attractive, but, no further work followed. We now come back to our proposal. We show that normalized stored relations expanded with inherited attributes may be more faithful to the reality. ER-modelling becomes useless. SQL queries become simpler since less procedural, freed partly or totally of logical navigation, or freed of selected value expressions. We propose clauses defining inherited attributes in Create Table. We add clauses for inherited attributes also to Alter Table. We illustrate our proposal with motivating examples, rooted in the “biblical” Supplier-Part database. Adding inherited attributes to stored ones as we propose appears in practice always less procedural than declaring views necessary for the same more faithful schemes or the same simpler queries at least, in every DB with stored relations and views only. The views may be also more procedural to maintain. We show also how our construct subsumes two popular partial helpers with less procedural SQL queries without views. We generalize the Normal Forms, the Heath’s and Fagin’s theorems to our construct. The restated Heath’s theorem decomposes into logical navigation free projections unlike its original. The restated Fagin’s theorem does not have this nice property. It may still spare however the logical navigation to likely most of practical queries. We show how to implement our construct on popular DBSs with negligible operational overhead. We conclude that the DBSs should accommodate our proposal better sooner than later and we discuss further research.

1. Introduction

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

No one followed the lead however, to the best of our knowledge. Below, we come back to our proposal in depth. We call our construct Stored and Inherited Relation, (SIR), Figure 1. We define every stored attribute (SA) of a SIR as usual for an SA scheme. One or more formulae that we call inheritance expressions (IEs) define for every SIR, its inherited attributes (IAs). We refer to Codd’s relational model, i.e., with the two constructs only, as to Stored Relation or View (relational) model, (SRV-model). Likewise we qualify of SRV DB every database, (DB), conform only to this model. In turn, we qualify of SIR-model and of SIR DB the relational model and DB supporting SIRs. We believe the reader familiar with the SRV-model and SQL in particular. We recall nevertheless that the conceptual model of any SRV DB consists of some SRs. Relational views, inheriting from these SRs or other views may present this scheme somehow differently to selected clients. The conceptual scheme of a SIR DB may contain also SIRs. A view in a SIR DB may inherit from SIRs as well.

We will show that IAs may model properties inconvenient to be SAs. It then appears as first benefit of the SIR DB with SIRs expanding some or all of the SRs of the conceptual scheme of some SRV DB,
that the SIR DB conceptual scheme may be more faithful to the reality. The practical advantage that will appear is that SQL queries to the SIR DB scheme may become less procedural, i.e., simpler, by the simplest linear measure which is the number of characters to enter. The rationale is that by addressing IAs, queries may become totally or partly free of the logical navigation otherwise necessary, or of selected, perhaps complex relational or value expressions, necessary for equivalent queries to the SRV DB as well.

On the other hand, it will appear that for every SIR, an SRV DB with the same SRs always provides for the view that may have the same name as the SIR, the same full source attribute names, the same attribute order, and, finally, always the same tuples as the SIR. In the nutshell, such a view defines the same SQL relation, but with IAs only. We call it conceptually equivalent to the SIR or its CE-view in short. It follows that for every SIR, its CE-view allows for the same, less procedural, queries. Declaring a CE-view, without being named so, is in fact the basic current “escape route” for the Database Administrator (DBA) of an SRV DB, to provide an acceptable view of a selected normalized SR to clients not happy with its resulting conceptual scheme and the induced complexity of queries to the DB involving the SR. We recall that in the SRV-model, the conceptual DB scheme is constituted exclusively from the normalized SRs.

Under some conditions on the SRV DB, there may be also, for a SIR, a view differing from CE-view only by inheritance of some attributes from relations other than in the CE-view, while providing for the same less procedural queries. We qualify it of query equivalent view of the SIR, QE-view in short. The interest of a QE-view may be to have Create View less procedural than the CE-view.

Nevertheless, as it will appear, the advantage of a SIR should be the IEs altogether less procedural than either Create View in practice. Also, either view scheme may be more procedural to maintain when the DB scheme evolves. Finally, implementing a SIR on popular DBs will appear costing a negligible storage and processing overhead. We will conclude that altogether, the popular DBSs should accommodate SIRs better sooner than later. Especially, since, as we will show, some already provide for decades a restricted type of SIRs we call self-inheriting, without realizing that it is so.

Next section details the SIR-model. We discuss the basic concepts and main SQL extensions for SIRs. We then illustrate these and the advantages of SIRs through a motivating example. We reuse the “biblical” Supplier-Parts DB. We extend with IEs at first one SA then, finally, all. We then complete the discussion of the SQL extensions for SIRs and of the utility of our proposal.

Section 3 extends the present basic relational (conceptual) schema design rules to SIRs. We first generalize the NFs other than 1NF. Next, we restate the Heath’s and Fagin’s theorems, [H71]. The restated theorems create the lossless decomposition of an SR, or presumed to be so, into two projections with the SAs of the original theorems. However, one or both of the new projections can be a SIR. If the projections result from the restated Heath’s theorem, then one of projection gets also all the attributes of the other as the IAs. A query to any SAs and IAs in this projection saves the logical navigation, originally necessary, since the equivalent query would have to address both projections.

Accordingly, if the conceptual scheme of a SIR DB results from the restated Heath’s decompositions only, it becomes logical navigation free in this sense. As we will show, it must contain indeed between the projections a universal one that is a SIR with all the attributes of the DB. This freedom, however, lasts only until, as usual, dangling tuples appear in a projection. As known, the decomposition joins cease to be lossless then. A query selecting a dangling tuple in a projection and a tuple in other projections may again require the logical navigation. Even so, as it will appear, SIRs may often reduce the latter, with respect to the original one.

In contrast, we also show that for the restated Fagin’s decomposition, the need of occasional navigation over the projections always remains. Altogether, it will appear that a SIR DB scheme produced by the restated theorems should have the same SRs as these produced by the original one. But, since some of these are now parts of SIRs, the new design should typically bring the discussed advantages. A motivating example in the section backs up our claims.
Section 4 discusses the implementation of SIR-model over an existing DBS. We specify an algorithm mapping SIRs into SRs and views there. We show how to process queries to SIRs. We also show that the storage and processing overhead of a SIR implemented as proposed is negligible. Section 5 concludes that SIRs improve the usability of the relational DBs and overviews the future work.

2. SIR-Model

2.1 Overview

Figure 1 shows the SIR-model versus the SRV-model. As already discussed, the SR construct is the same for both models. In the SRV-model, the SRs of a DB constitute its conceptual scheme. In SIR-model, it may be the case of SRs and/or of SIRs. An SRV-model view furthermore, i.e., an IR, is also a view for the SIR-model. The inverse is not true, as also already mentioned. In both models, views adapt the conceptual scheme to sub-communities of DB users, as the ANSI-SPARC architecture stipulates for any DB. Next, as for both constructs in SRV-model, an SIR is a finite subset of a Cartesian product of atomic attributes (columns) over some domains, subject to any algebraic or predicative operations, and aggregate or scalar functions, applying to 1NF relations. However, as we already stressed, unlike for an SR or a view, every SIR mixes SAs and IAs. As usual, we basically consider an IA value immaterial. As usual as well for an SR or a view, every SIR has a name and the scheme defining all its SAs and IAs. Every SA scheme is a usual one, e.g., as in a popular SQL dialect. Every IA in contrast results from a specific formula that we call inheritance expression (IE).

An IE defines one or several IAs. These may be inherited from SRs, views or SIRs. Each SIR scheme contains one or several IEs. Each IE there defines a disjoint subset of IAs. Next, every IE selects IAs and calculates tuples of IA values through some relational or value expression. The necessary requirement is that, for every SIR tuple \( t \), every IE contributes with at most one sub-tuple of IAs values to become a sub-tuple of \( t \). Figure 2 illustrates these principles. The SIR at the figure has multiple IEs. Each tuple of, e.g., IE1, appends to some SIR tuple, containing the sub-tuple with values of all the SAs and the sub-tuples with values from all the others IEs. We named all SAs collectively as B at the figure for reasons we explain soon. Each green rectangle represents an inherited sub-tuple.

We define IEs through SQL-like statements that we detail below. Basically, they generalize these defining at present the Select clause sub-queries. As known, every such subquery defines for each tuple \( t \) produced by the query, a single (attribute) value or a null entering \( t \). This result steams from the half outer join semantics of any such sub-queries, preserving every tuple produced by the query without the subquery. An IE Select statement basically generalizes this semantics to a sub-tuple of perhaps several values or nulls entering every tuple \( t \) of the SIR with it.

For instance, the expression \( A \) From \( F \) Where \( F.B = R.B \) with mono-attribute \( A \), could serve a Select clause subquery of a query to some relation \( R \), named in the From clause of the query outside the subquery. It could also be an IE, as it will appear, within the scheme of some SIR \( R \). In contrast, the expression \( A1, A2 \) From \( F \) Where \( F.B = R.B \) could only be an IE defining two IAs. As for the subquery,
the first IE could generate a null value for A, as if its inner join was the half outer one, preserving every tuple of R/A. Likewise, the inner join serving 2nd IE, would act as a half outer one, preserving every tuple of R/(A1,A2), i.e., creating perhaps one or two nulls.

More generally, our basic form for an IE is: <SQL Select list of IAs> [From...>] [Where...] ','|',,'|';'. The list of IAs conforms to the usual syntax for SQL Select clause, but with some restrictions we will address and without Select keyword itself. It may contain thus IAs proper names or the prefixed IA names, both perhaps with aliases. Alternatively, unlike an SQL query, it may contain only a single IA defined through a value expression. The rationale comes from the implementation algorithm for SIRs we propose in Section 4. For the same reason, the IA name is then the default name of the IE. The IEs without value expressions do not have default names. It may be practical however to name it informally, e.g. in the SQL comment.

The From clause contains the names of all the source relations for the IAs, again according to the usual SQL syntax. The SIR itself may be among, being eventually aliased. The clause may be implicit, i.e., made explicit transparently to evaluate the IE. Implicit From means that the IAs proper names perhaps with some prefixed attribute names determine the set of all the source relation(s) for the explicit From clause.

For an explicit From, Where clause is optional, as usual for SQL. An explicit Where depends on the Select list. For the one with IAs proper names or the prefixed IA names only, both perhaps with aliases, the Where clause may contain only the explicit equijoins that we call recursive and discuss below. There is the conjunction of such clauses, one per every source relation. For the implicit From, the explicit join clauses are the explicit natural equijoins. We will discuss later the explicit Where for IEs with value expressions. If From is implicit the Where clause is as well. Its explicit content depends on Select list in the same way as the Where clause content for the implicit From made explicit.

If From and Where clauses are implicit, then ‘,’ terminates the IE if an SA or an IE with the explicit clauses, or an IE with a value expression follows, the IE being in basic or so-called secondary form we define below. If From is explicit, ‘,’ terminates it unconditionally. In both cases, the terminating IE contains as the IAs all these preceding ‘,’ either since the previous definition of an SA or since the previous IE or since Create Table keyword if no SA or IE precedes the IE. Next, ‘,’ may alternatively terminate an IE, being the usual end of every SQL statement. The IE contains the same IAs then as for the termination with ‘,’. Finally, ‘,’ always forces the termination of an IE with the implicit From, the IAs in the IE being again determined as above stated. This may serve if one wishes to separate two IEs without value expressions and with the implicit clauses only. Next subsection shows an example, but also that there does not seem to be any good reasons for doing it. Notice finally that the termination rules enforce also our rule of a single IA per IE if the IA results from a value expression.

For every SIR R, and every IE I of R, even if From and Where clauses are implicit, it is always the usual SQL syntax and semantics that determines every tuple t' of I and determines for every t' every tuple t of R / I that t' expands and guarantees for every t that t' is the only tuple in I to do it for t. Both From and Where clauses are therefore considered explicit for any such evaluations. We speak about the usual SQL syntax only, since, as known, there is no formal definition of SQL. We have the SQL standard and dialects. As known as well, in practice, the syntax of the Select clause sub-queries that, as we have indicated, the IEs basically generalize is also dialect-dependent. Examples below mainly follow the SQL Server dialect.

The basic form of an IE with an IA resulting from a value expression is the usual one for SQL, albeit without Select verb, namely: <value expression> As <IA name> [From...] [Where...]. For any such IA, we provide also the secondary form: <IA name> AS (]<value expression>|)] ‘,’|’,,’|’.. An IE with a value expression may make every value of its IA inheriting from SAs or other IAs only from the SIR it contributes to. We speak then about a self-inheriting IE. If the value expression in addition does not contain an aggregate function, then the secondary form is also the common one, i.e., ignoring dialect-dependent details, for a virtual (computed, calculated, generated...) attribute (VA), [LV86].
Some but not all popular DBSs provide VAs, e.g., MySQL, (with parentheses required) or SQL Server (no parentheses), but not MS Access. The convenience of the secondary notation is mainly that an IA that is a VA may be defined as at present. However, as it will appear, we let IEs in this form to provide also for value expressions impossible for VAs at present, e.g., with aggregate functions. As for a VA today, some may indeed prefer also for such an IA the notation where IA name is first and not following the value expression. Besides this convenience difference, the secondary form and the basic one with implicit From and Where clauses are equally procedural. For an IA definition through a value expression with these clauses being explicit, we consider however the basic form mandatory, for obvious implementation convenience.

For an IE defining an IA through a value expression, an explicit IE may be any SQL expression respecting all the constraints on an IE. The rules for the IE with implicit From and Where are in contrast more restrictive. First, suppose a self-inheriting IE. Then, its explicit form has only From R clause and no Where clause. Alternatively, Select may consist of a value expression with or without an aggregate over attribute(s) of another relation(s), perhaps imbricated into an arithmetic expression or a scalar function. Then, the explicit From and Where clauses are supposed as for an IE without any value expression. In every case, Select clause in secondary form is supposed reformatted into the basic one for its explicit representation.

Now, more in depth with respect to the attribute naming rules for a SIR scheme, every SA and IA should have a unique attribute name, as usual in a relation. Every SA has unique proper name that is its attribute name in the SIR. A unique proper name of an IA is also usually its default attribute name. The default may be inconvenient however. The IE’s Select list allows then for an alias. Moreover, an IA proper name in a SIR may create a name conflict, with some SA or IA name elsewhere in the SIR. It then may be aliased with a non-confictual name. Alternatively, if the IE inherits the IA from its source relation as is, then its full source name may be the default IA attribute name in the SIR. In other words, it is the proper source name prefixed by the name of the source relation. Some popular DBSs, e.g., Ms Access, do similarly for views. If it is a value expression that defines an IA, its proper name should be unique in the SIR as well.

Next, as known and appearing in our examples of the IEs above, for every query \( Q \) with a Select clause subquery \( q \), producing for each tuple \( t \) selected by \( Q \) at most a single value \( v \), \( q \) may contain one or more specific join clauses. Each join may be a \( \theta \)-join, although it is an equijoin usually. It matches a value of some attribute calculated from relation(s) referred to in From clause of \( q \), with value of some attribute named in clauses of \( Q \) outside \( q \). Accordingly, a basic form of an IE \( I \) in a SIR \( R \) may contain a similar join that we call recursive. Indeed, \( I \) serve to define \( R \), while \( R \) values outside \( I \) serve through the join(s) to define \( I \) tuples. Notice that as for any Select clause subquery, the recursive join within \( I \) may be an inner one, but still may generate null values, as if it was the half outer join, preserving every tuple of \( R/I \).

As already mentioned, we allow for a variant of the basic form of IE with all the recursive equijoin clauses being implicit, i.e., added transparently when one evaluates the IE for a tuple, as we discuss later on. This is the case of the implicit From and Where clauses in an IE we have discussed. The implicit recursive join means for every source relation \( S \) the natural one in the (explicit) Where clause between \( R' \) and \( S \). Usually, it should be between the primary and foreign keys. We recall however that the recursive join is then in fact evaluated as a half outer one, preserving every tuple of \( R' \). It should preserve thus also every eventual tuple of \( R' \) where the foreign key value is unknown in its source relation. If an IE implicitly inherits from several source relations, then we presume all the explicit recursive (natural) joins `AND'ed in the Where clause, as already mentioned. For a single implicit equijoin clause and mono-attribute primary key only, MsAccess has in fact a similar rule for queries, although that join means only an inner join. We come back to that rule more in depth later. SRV-model does not consider implicit joins, although there were proposals to add these, [LA86], [LSW91].
Now, let us write an SIR R as \( R(B, V) \) with B containing all the SAs, and V all the IAs. V consists thus of the columns labelled with IEs at Figure 2. For every R, we require B to be a stored relation for SRV-model, i.e., without duplicates. We call B a stored relation or base of R. We consider for reasons that will appear that each base has its own relation name available for queries that is \( R_B \) by default. Like any relations, every SIR has one or more keys. We require all keys of every base among these. We recall that for any relation, all attributes are functionally dependent on every key. For any key value of any R, we have thus for each SA and each IA, at most one (atomic) value, respectively stored or inherited. The rationale is, again, the requirement of 1NF for any SIR. Finally, we always have \( \text{card}(R) = \text{card}(R_B) \). All this is also the rationale for the requirement we made above that for every R tuple, for every IE, its recursive join matches at most one IE tuple.

An IE may inherit from a stored relation, a view or an SIR. It may happen that an SIR or a view R1 inherits from an SIR or a view R2 and vice versa. By extension of the current terminology for views, we say that R1 and R2 are then in circular reference. We forbid circular references in an SIR DB, like they are prohibited at present at any existing major DBS we are aware of. The DBSs detect them at the run-time, issue some SQL Error Code and abort the query. Our rationale for extending this policy to SIR DBs is that the basic processing scheme (BPS) of SIR DBs we detail in Section 4, supposes the implementation over an existing major DBS. That one takes care of the actual query evaluation for the SIR DB. For any circular reference in the SIR DB, BPS would then create a circular reference in the DBS. The query to the affected SIRs or views in the SIR DB would consequently not get processed.

The straightforward way to avoid a circular reference in an SIR DB is to rewrite one of the IEs so it refers to the stored relations only. These can be bases of SIRs that the IE refers to. Also, they can be the origin of IAs in those SIRs. The latter applies to IAS views as well, including in present DBs. We illustrate the idea with examples soon.

![Figure 2. An SIR with multiple IEs. SAs are grey, IAs are green. IAs follow here all the SAs, but it is not always so. IEs bring here respectively three, four and as many tuples as there are stored ones. V unions all the IAs.](image-url)

We suppose SIR DB schemes defined as usual through some DDL (Data Definition Language). Operationally, we consider from now some SQL-like DDL. We suppose the statements of this DDL extend to SIR-model some existing SRV-model conform SQL dialect. We call the latter the kernel for the resulting SIR-model dialect. We presume the kernel to be the dialect of some major DBS. E.g., Create Table for an SIR R may extend the SQL Server or MySQL Create Table R to the capability to defining IEs. In any such extension, the left to right order of attributes in R is the top-down and left-to-right in the IEs order of the attribute definitions in R. We recall that, while the attributes order has no importance for a formal definition of the relations, i.e., any relations that differ only by this order are equal, it differentiates any otherwise formally equal SQL relations, because of different results of the Select * From... query.
SIRs being 1NF relations no DML (Data Manipulation Language) kernel statement needs syntax extensions. For update queries in particular we suppose that the kernel SQL Update, Insert or Delete statements apply to SIRs as well. More precisely, for every SIR, say R, we consider that an update query to R is valid only if it does not attempt to modify any IA defined by a value expression or a Select clause subquery, directly or transitively. In other words, the query may only modify an SA or an IA that R inherits as equal to some SA, directly or transitively. The expected result is as if the modification concerned the SA directly. The rationale is that at present, no popular DBS allows to modify an IA defined by a value expression or a subquery, even if it is theoretically possible, [LV86]. Thus, an insert into a SIR R, creates the tuple(s) in R with every SA instantiated as in the query. The primary key of R_B must be among these. The insert may instantiate an IA. The result is the instantiation of the source SA. That one is then inherited back by the IA, and by every IA for which SA is the source. If an insert does not instantiate an IA, then this one gets the value defined by the IE, or becomes null as it was discussed. Likewise, an update to existing tuples proceeds as usual for any SA and an update to an IA, provided valid, propagates to the source SA. Finally, a delete removes from R the tuples specified in the Where clause of the Delete statement, regardless whether this one concerns SAs or IAs. Finally, as usual, any of these operations gets blocked if it violates the referential integrity or usual check constraints etc.

As usual for a Select clause, that of an IE may enumerate the IA to create by names. Likewise, for non-procedurality of the clause, i.e., the number of characters defining it, basically, the generic character ‘*’, perhaps prefixed with a relation name, may replace the enumeration of all the attributes of a source relation or even of all the source relations. For IEs, it appears however also useful to have a similar generic character, enumerating all the attributes except for all these already in some recursive equijoin clause, being sources of foreign keys in the SIR in particular. We denote this character as ‘#’ and will justify it in our motivating example.

As we hinted to, every SIR has the CE-view, constituting the conceptually equivalent relation with IAs only. More formally, first, for every SA of the SIR, there is a unique IA in CE-view having the same full name and always the same values as in the SIR. Next, for every IA of the SIR, there is a unique IA in the CE-view, with the same full source name and, again, always the same values. Besides, finally, the CE-view has no other attributes. It follows, in particular, that for every SIR, its CE-view allows for the same less procedural queries. As we hinted as well, under some conditions on the SRV DB, a SIR may also have a QE-view. That one may be less procedural to define than the CE-view, while providing still for the same simpler queries. Basically, the lower procedurality may result from SQL ‘*’ possible for the QE-view Create View, unlike for the CE-view. A condition for the existence of QE-view may be, e.g., the referential integrity between some source SRs of the CE-view. As said however, IEs should remain altogether usually less procedural than either view. For QE-view however, this can be possible only because of ‘#’ in IEs. Besides, it will appear that a maintenance operation eventually required for both views may be more procedural for QE-view, with more potential for a serious trouble, while actually not necessary for a SIR at all.

Lower procedurality of a DDL or DML statement makes it generally preferred to any more procedural equivalents. For their goal, SIRs should thus be more practical than views. We’ll show later that they should also be more practical than two partial helpers with simpler queries, also replacing more procedural views, already in use by popular DBS for decades. We now illustrate all the discussed points with a motivating example. We complete afterwards the discussion of the DDL for SIRs and of their utility.

2.2 Motivating Example

We reuse the biblical Codd’s Supplier-Part relational DB. Its variants motivated the original proposals, [C69], [C70]. These settled the relational (conceptual schema) design rules of SRV-model, based on NFs as known. These, as well the Codd’s DB itself, molded about any practical DB created since. The
variant we picked up is probably the most known, [D4]. It is often named S-P in short, although we refer to it as S-P1. We restate S-P1 into variants with SIRs. We call these S-P2, SP3...

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, a supplier may supply nothing for the time being. Likewise, a part may be not supplied. S-P1 conceptual schema of the enterprise consists of three stored relations: S for suppliers, P for parts and SP for supplies. As known, this scheme is optimal for the relational design rules of SRV-model.

1. Figure 3 shows the conceptual scheme of a SIR-model DB that we call S-P2. Like the scheme of S-P1, it consists also of three relations. The scheme is the optimal one under SIR-model for S-P2, as it will appear, for the enterprise that the S-P1 models best under SRV-model. S-P2 scheme was in fact also our motivating example in [LKR92], with however a different Create Table SP syntax for IEs. The current one appears more practical. Figure 4 shows example extensions of S-P2 relations, i.e., a possible content of each. We use self-explaining statements to define every relation. We underline the primary key attributes, as usual.

<table>
<thead>
<tr>
<th>S-P2 Scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table S</td>
</tr>
<tr>
<td>S# Char,</td>
</tr>
<tr>
<td>SNAME Char,</td>
</tr>
<tr>
<td>STATUS Char,</td>
</tr>
<tr>
<td>CITY Char;</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Figure 3 The S-P2 scheme.

<table>
<thead>
<tr>
<th>S-P2 Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table S</td>
</tr>
<tr>
<td>S#</td>
</tr>
<tr>
<td>S1</td>
</tr>
<tr>
<td>S2</td>
</tr>
<tr>
<td>S3</td>
</tr>
<tr>
<td>S4</td>
</tr>
<tr>
<td>S5</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>S1</td>
</tr>
<tr>
<td>S2</td>
</tr>
<tr>
<td>S3</td>
</tr>
<tr>
<td>S4</td>
</tr>
<tr>
<td>S5</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Table SP

<table>
<thead>
<tr>
<th>S#</th>
<th>P#</th>
<th>QTY</th>
<th>SNAME</th>
<th>STATUS</th>
<th>CITY</th>
<th>PNAME</th>
<th>COLOR</th>
<th>WEIGHT</th>
<th>PCITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>P1</td>
<td>300</td>
<td>Smith</td>
<td>20</td>
<td>London</td>
<td>Nut</td>
<td>Red</td>
<td>12</td>
<td>London</td>
</tr>
<tr>
<td>S1</td>
<td>P2</td>
<td>200</td>
<td>Smith</td>
<td>20</td>
<td>London</td>
<td>Bolt</td>
<td>Green</td>
<td>17</td>
<td>Paris</td>
</tr>
<tr>
<td>S1</td>
<td>P3</td>
<td>400</td>
<td>Smith</td>
<td>20</td>
<td>London</td>
<td>Screw</td>
<td>Blue</td>
<td>17</td>
<td>Oslo</td>
</tr>
<tr>
<td>S1</td>
<td>P4</td>
<td>200</td>
<td>Smith</td>
<td>20</td>
<td>London</td>
<td>Screw</td>
<td>Red</td>
<td>14</td>
<td>London</td>
</tr>
<tr>
<td>S1</td>
<td>P5</td>
<td>100</td>
<td>Smith</td>
<td>20</td>
<td>London</td>
<td>Cam</td>
<td>Blue</td>
<td>12</td>
<td>Paris</td>
</tr>
<tr>
<td>S1</td>
<td>P6</td>
<td>100</td>
<td>Smith</td>
<td>20</td>
<td>London</td>
<td>Cog</td>
<td>Red</td>
<td>19</td>
<td>London</td>
</tr>
<tr>
<td>S2</td>
<td>P1</td>
<td>300</td>
<td>Jones</td>
<td>10</td>
<td>Paris</td>
<td>Nut</td>
<td>Red</td>
<td>12</td>
<td>London</td>
</tr>
<tr>
<td>S2</td>
<td>P2</td>
<td>400</td>
<td>Jones</td>
<td>10</td>
<td>Paris</td>
<td>Bolt</td>
<td>Green</td>
<td>17</td>
<td>Paris</td>
</tr>
<tr>
<td>S3</td>
<td>P2</td>
<td>200</td>
<td>Blake</td>
<td>30</td>
<td>Paris</td>
<td>Bolt</td>
<td>Green</td>
<td>17</td>
<td>Paris</td>
</tr>
<tr>
<td>S4</td>
<td>P2</td>
<td>200</td>
<td>Clark</td>
<td>20</td>
<td>London</td>
<td>Bolt</td>
<td>Green</td>
<td>17</td>
<td>Paris</td>
</tr>
<tr>
<td>S4</td>
<td>P4</td>
<td>300</td>
<td>Clark</td>
<td>20</td>
<td>London</td>
<td>Screw</td>
<td>Red</td>
<td>14</td>
<td>London</td>
</tr>
<tr>
<td>S4</td>
<td>P5</td>
<td>400</td>
<td>Clark</td>
<td>20</td>
<td>London</td>
<td>Cam</td>
<td>Blue</td>
<td>12</td>
<td>Paris</td>
</tr>
</tbody>
</table>

Figure 4 The S-P2 content. IA (proper) names and values are in Italics.
The S, P relations have SAs, but none has any IAs. Both are the same as in S-P1, i.e., have the same scheme and tuples. In contrast S-P1.SP is now expanded into a SIR. It keeps the original SAs with their schemes i.e., (S#, P#, Qty) and values. These SAs form its base SP_B. SP_B keeps also its original primary key, (S#, P#). This one becomes also the primary one for the entire S-P2.SP, i.e., with its IAs. Two IEs define the latter. All the IA of SP form V in our generic notation. The left to right order of SP attributes is the top-down and left-to-right in the IEs order of the definitions in SP, i.e., (S#, P#, QTY, SNAME...SCITY, PNAME...PCITY). The IA names and values are in Italics at the figures. The IA names displayed at Figure 4 are proper names only, as through Select * From SP. The choice of IAs means that for the DBA, every property of a supplier and of a part is also as that of any supply. We examine the rationale for it later. Having thus, e.g., SP.SNAME, models for the DBA the conceptual property of any supply stated in natural language as: name of the supplier of the supply and so on for every other IA in SP.

As the comments show at Figure 3, we refer to the two IEs in SP-2.SP scheme as I_S and I_P. I_S definition starts after ‘,’ terminating QTY scheme, accordingly to the general principles of IEs. I_S inherits thus IAs resulting from the generic ‘#’. These are all and only attributes of S, except for S#. Their default names are SNAME, CITY, STATUS. CITY does not need prefix, since there is no name conflict with CITY inherited from P through I_P. Each I_S tuple inherits values of the enumerated source attributes in one tuple of S. Likewise, I_P inherits all and only attributes of P, except for P#. I_P refers to P.CITY, because of the name conflict otherwise with CITY in S, in presence of the implicit From clause. The default name of CITY inherited from P in SP could be P.CITY. The prefixing avoids indeed the name conflict in SP with CITY IA brought by I_S. Nevertheless, I_P renames P.CITY into an alias, unique in SP, avoiding the prefixing, perhaps annoying for some. I_S has the implicit From S clause, unlike for the usual SQL syntax. The prefixing of ‘#’ with S trivially suffices to make it explicit whenever needed. If the clause was explicit, From S thus, I_S ‘#’ does not need its prefix. I_P From clause is also implicit. IAs proper names and P.CITY determine its From P explicit form. Each IE has also an implicit Where clause, thus with an implicit recursive natural join in it and no other clauses. The implicit Where clause for I_S can be thus formulated as Where SP.S# = S#, with the latter S# referring implicitly to S.S#, because of the presumed From S clause, as usual then for SQL. As required for at least one IE in an SIR, I_S matches thus an SA in SP. S.S# is here also the attribute involved in the recursive join, we spoke about while introducing ‘#’. Respectively, the same occurs for I_P through its implicit recursive (natural) join clause SP.P# = P#. Finally, observe that we could rewrite both IEs into a single one, say I.SP, namely as:

(I.SP) S.#, PNAME, COLOR, WEIGHT, P.CITY As PCITY.

Actually, I.SP has even lower procedurality than I_S and I_P as at Figure 3, gaining however only the very modest ‘,’ after S.#. That is why we said previously that there does not seem to be any practical reasons for using ‘,’. With explicit From and Where clauses I.SP would become:

(I.SP’) S.#, PNAME, COLOR, WEIGHT, P.CITY PCITY From S, P Where SP.S# = S# AND SP.P# = P#.

For the first tuple of SP, with SAs S# = S1 and P# = P1, the recursive join clause in I_S matches I_S tuple (Smith, 20, London). The reason is that S.S# = S1 for this tuple. SP.S# = S1 does not match any other S.S# value. Hence, the clause makes I_S producing only one tuple for this SP tuple, as required. The obvious reason is that S# is the primary key of S. The same occurs clearly for any other tuple with SP.S# = S1 and, besides, similarly for every SP tuple. For the first SP tuple, the selected I_S tuple becomes the sub-tuple of IAs due to I_S. Likewise, I_P creates, for the first SP tuple, the sub-tuple (Nut, Red, 12, London). Together with the B sub-tuple (S1, P1, 300) and the I_S sub-tuple just discussed, all three form the (entire) SP tuple. Similar calculation determines every other SP tuple at Figure 4. Notice that the IAs defined by I_S as well as these of I_P and, alternatively, these of I.SP are all functionally dependent on the key (S#, P#), as required for IAs of any SIR.

2. We first illustrate our claim about the lower procedurality of the IEs than of Create View for the CE-view. We recall that in a relational DB every relation must have a unique name. Suppose
therefore that the DBA of S-P1 wished to create a view SP upfront, hence the SR SP there was created as SP_B instead. The least procedural CE-view, i.e., the shortest, seems then:

Create View SP As (Select SP.*,SNAME,STATUS,CITY,PNAME,COLOR,WEIGHT, CITY As PCITY From (SP Left Join S On SP.S# = S.S#) Left Join P On SP.P# = P.P#);

This is visibly by far more procedural than both I_S and I_P considered together. If the referential integrity between S, P and SP is enforced, then CE-view creation can be simpler:

(V_SP1) Create View SP As (Select SP.*,SNAME,STATUS,CITY,PNAME,COLOR,WEIGHT,CITY As PCITY From SP,S,P Where SP.S# = S.S# And SP.P# = P.P#);

It is still visibly no competition to the IEs.

Next, providing the referential integrity between S, P and SP, the only choices for a QE-view of SP here are S#.S# instead of SP.S# or P.P# instead of SP.P#. Each choice provides for the same queries as SIR SP. None reduces nevertheless the procedurality of CE-view. Consider however the following variant of S-P2.SP. Suppose first that one does not rename P.CITY. As for some popular DBS, e.g., of MsAccess, its name in SP will stay then P.CITY. Suppose next that SP scheme should be SP (S#.S#, SNAME, STATUS...,P#, PNAME..., QTY) and that the referential integrity is again enforced. Then, the following QE-view becomes possible:

(V_SP2) Create View SP (Select S.*, P.*, QTY From S, P, SP Where S# = S#.S# And P# = P.P#);

Use of '*' makes QE-view about twice less procedural than CE-view SP. It is also clearly the least procedural form of either view for any variant of SIR SP with respect to the attribute order. With respect to the IEs, notice that I_P may now be also simplified to P.#. Not only the CE-view, but even the much simpler V_SP2 QE-view remains still thus of no match to the IEs.

We now show similar result for the secondary notation. Consider that WEIGHT in P expresses the weight of a part in pounds, while in the conceptual model every part should also have its weight in KGs with precision of 1G, this property being modelled as the IA named WEIGHT_KG, following WEIGHT in P. To take care of, one could expand S-P1.P, renamed P_B, with the following IE using the secondary notation:

(W1) WEIGHT_KG AS (Round (WEIGHT * 0.454,3)).

The notation would be the same for VA WEIGHT_KG in the DBS providing these, e.g., SQL Server or MySQL, but not MsAccess. Relations with VAs there are thus a specific SIRs without being explicitly considered as such. An alternative IE definition, through the basic notation could be:

(W2) Round(X.WEIGHT * 0.454,3) As WEIGHT_KG

Here again both From and Where clauses are implicit. Since W2 does not contains an aggregate function and it expands P, its implicit clause is simply From P. P is also an example of a self-inheriting SIR where, in addition, the IA inherits from and only from the tuple of P it expands. Furthermore, observe that W2 with its explicit clauses are also parts of Select clause subquery and of the query to S-P1.P selecting WEIGHT_KG calculated for each tuple in P. Namely, the latter could be:

(W3) (Select Round(X.WEIGHT * 0.454,3) From P X Where P# = P.P#) As WEIGHT_KG From P;

One could expand (W3) into the CE-view P of SIR P with WEIGHT_KG, after renaming S-P1.P to, say, P_B. However, in practice one would rather define WEIGHT_KG in CE-view P through the less procedural value expression that would be simply (W2), namely as:

Create View P As (Select P#, PNAME, COLOR, WEIGHT, Round (X.WEIGHT * 0.454,3) As WEIGHT_KG, CITY From P_B);

The view scheme is however still about four times more procedural than (W1) or (W2). The rationale is obviously the enumeration of the SAs of P in the view scheme. The ratio would be the smallest,
about two, if WEIGHT_KG was added after all the SAs, as one could use ‘*’ then. This would be however an unusual choice to separate WEIGHT and WEIGHT_KG by CITY. Notice that WEIGHT_KG being also a VA through (W1) illustrates why the popular DBs propose the VAs already.

We could alternatively wish WEIGHT_KG not to appear in P, but only in SP, after the IA WEIGHT. Instead of I_P, the following three IEs would do. The 2nd one is W1 again, but could be W2 as well:

P#, PNAME, COLOR, WEIGHT, Round (X.WEIGHT * 0.454,3) As WEIGHT_KG, CITY

The implicit clauses for the two IEs around W1 are these of I_P. We could similarly modify I_SP. The implicit clause for W1 is however now Notice also that SP.WEIGHT_KG cannot be a VA in present DBSs. Notice finally that P.WEIGHT_KG is not updatable in present DBSs, as any VA besides. Decades old research showed that one could change this state-of-the-art for VAs rather easily nevertheless, [LV86]. In the meantime, WEIGHT_KG would not be updatable under our assumptions anywhere in S-P2 as well.

3. We now illustrate our statements about possibly greater procedurality of a CE-view or QE-view maintenance operation than for the SIRs. Essentially, either view may need an alteration consequent to that of a source SR, while this SR may be the base of the SIR, not inherited anywhere. As perhaps the simplest case, suppose that the DBA of S-P1 did not anticipate the need for the CE-view S-P2.SP. If SP may get IEs, a single Alter SP statement will do (see the DDL for SIRs discussion below). If one needs to create a CE-view in contrast, the DBA has to first rename SP into, say, SP_B again, using one Alter statement. Then, the DBA has to define the CE-view SP or a QE-view. Both Alter and Create View statements must constitute an atomic transaction, to avoid any run-time errors for application programs. SQL Begin Transaction and Commit brackets are thus necessary as well, together with the SQL Error Code tests for the eventual Rollback after each statement. All this leads to several (how many?) statements and quite a headache for the DBA, comparatively.

Similar reasoning with even greater difference to procedurality applies to an SA name change. Suppose indeed that the DBA decides to shorten QTY to simply Q, when either S-P2.SP or its CE-view SP already exists. Also, a CE-view scheme with ‘*’ or that of QE-view, may require a maintenance procedure if a new attribute gets appended to a source relation. Taking this attribute automatically to the account through ‘*’ may indeed create a run-time error for a form or an application program querying the view. Such an error seems actually quite frequent. Often DBA manages indeed only the DB conceptual scheme. Views and applications are private to clients. The clients appear often not aware of alterations the DBA performs. In turn, the DBA is in general not aware of all the views and their apps. That is why, besides, ‘*’ is not advised for embedded SQL.

As perhaps an even more instructive example, suppose that DBA drops the default referential integrity for above variants of S-P2. Hence S-P2.SP_B and S-P1.SP may contain, e.g., the tuple (S10, P10, 50) with S10 and P10 not in S and P, respectively. The S-P2 DBA has nothing to do. The S-P1 CE-view with WHERE clause has to be altered to the one with the explicit outer joins. Likewise, one has to alter V_SP2 QE-view scheme to the CE-view. Otherwise, for each of these schemes, the output of a query could be silently errored, with the above tuple missing, e.g., for the simplest query Select * From SP. Silent errors are of even greater concern than the run-time ones. However, for our purpose what matters mainly here is that, while S-P2 DBA has nothing to do, we recall, for the S-P1 DBA or a client, even if one adequately alters the CE-view or QE-view, each procedure requires again an atomic transaction with several statements.

4. We now illustrate our claim that SIR-model may provide for more faithful conceptual modelling then the SRV-model. We recall that Codd’s original relational design rules, [C69], [C70], taught today to undergrads, propose the conceptual schema of a relational DB to be the (i) the smallest collection of 1NF stored relations that satisfies the largest community of the DB clients, called commonwealth by Codd, as the reality model. This provided however, (ii) that every relation in this scheme is free of the, so-called, storage and update anomalies, initially termed by Codd in fact strong redundancies that could result from stored redundant values or an update, insert or delete. We use n-ary relations
with as many attributes as possible, to fulfil the requirement (i), instead of, e.g., using the binary ones only. We recall that the latter were the basis of once popular semantic data model, [A74]. Condition (ii) boils down in practice today to BCNF for every relation with functional dependencies (FDs) only and to 4NF for those that present multivalued dependencies (MVDs). Relations in 4NF that would not be in 5NF are almost unheard of. For a community with specific needs, not met by the common scheme, views should produce the DB view. We recall that, for the relational model, this was the way to respect the ANSI-SPARC reference architecture, recommended in its time and till today, as mandatory for every data collection pretending to be a DB.

It’s truism to say that Codd’s principles became universal for the relational DBs. Codd used S-P1 to illustrate those. S and P relations did not lead to major controversies. This was immediately not the case of SP. Many found that the minimal three attribute model of SP is not faithful to any practical conceptual one. An actual supply always presents at least some other properties of the supplier and of the part supplied. These are the names at least or even all the properties of each ultimately. Adding any of the attributes modeling these properties to S-P1.SP would create the anomalies however. Hence, according to Codd, as known for decades, they should not be there, [D4]. Especially, since a query or a view can present to the client any of the “missing” attributes as IAs using the foreign keys of the minimal model whenever the need occurs. Many in the conceptual modelling community remained however unconvinced by this postulate, as widely known. The popular ER model resulted from, postulating instead that a tuple of SP in fact does not model an actual supply, but rather a relationship between two entities that were a supplier and a part [C76]. This one introduced however its own problems. E.g., to consider an actual box with parts as a relationship only, not as a (perhaps even heavy) entity by itself, may not convince some. Also, whether a marriage is an entity or a relationship never got a clear answer.

In S-P2 we expand S-P1.SP into a SIR with the base SP_B = S-P1.SP, and with all the others attributes of S and of P. Hence SP models now, in the greatest contrast to Codd’s scheme, not the minimum, but the maximum of the conceptual properties of S and of P that could characterize SP. Thus, SP scheme is now not only more faithful to the reality, but, even, cannot be more faithful with respect to S and P properties the actual supply could present. All these are the IAs however hence avoid any storage anomalies. Likewise, no updates to SP_B may create an update anomaly. Next, according to our general principles for SIRs, an update to any IAs of S-P2.SP applies to the sources in S or P that are always SAs. A view update to the CE-view of S-P2.SP would do the same, indeed. As known and one may easily verify, e.g., on MsAccess, this prevents any update anomaly for the view, hence for S-P2.SP. Likewise, S-P2.SP is free from the insert and the delete anomalies. Hence, S-P2.SP can serve as a conceptual scheme using the same best normalized stored relations as S-P1. In contrast, the latter with S-P1.SP expanded with the SAs formally equal to IAs of S-P2.SP, would not be the one required for S-P1 under the SRV-model.

5. Now we show that a SIR DB may, consequently, avoid or reduce the logical navigation in many queries to its conceptual scheme, with respect to the equivalent queries addressing the conceptual scheme of the base-equal SRV DB. The term designates the SRV DB with the same stored (base) relations as the stored ones or the bases of SIRs in a SIR-DB. Thus, S-P1 and S-P2 are base-equal. Consider now the client needing P#, PNAME and QTY of every supply by Smith. For S-P2, the SQL query could be:

(Q1) Select P#, PNAME, QTY From SP Where SNAME = ‘Smith’;

An equivalent query to S-P1 could be:

(Q2) SELECT PName, SP.p#, SP.s# FROM S INNER JOIN (P RIGHT JOIN SP ON P.P# = SP.P#) ON S.S# = SP.S# WHERE S.SName ="smith";

Q1 has no logical navigation, as it selects all its data from a single relation. Q2 does it, as it must address all three S-P1 relations. The result is visibly by far (more than two times by our crude measure) more procedural than Q1. This stresses the practical importance of avoiding the logical
navigation, the need identified for a little while already anyhow [MUV84]. If the referential integrity is mandatory for S-P1, the query equivalent to Q1 could be simpler, but still by about twice as procedural as Q1:

(Q3) Select SP.P#, PNAME, QTY From S, SP, P Where SNAME = ‘Smith’ And S.S# = SP.S# And SP.P# = P.P# ;

The SQL standard would besides rather recommend writing Q3 as even more procedural:

(Q3.1) Select SP.P#, PNAME, QTY From S Inner Join (SP Inner Joins P On SP.P# = P.P#) On S.S# = SP.S# Where SNAME = “Smith” ;

Actually, if one creates Q3 using QBE, popular DBSs, e.g., MsAccess, would translate it to SQL only as Q2.1.

Actually, Q1 avoids the logical navigation since it may select all its tuples in SP only. A query to S-P2 may however select a dangling tuple in S or P, i.e., a tuple that is not a projection of a tuple in SP. In practice, it would be a supplier not supplying any parts at present or, similarly, a non-supplied part. The logical navigation may be impossible to avoid then. We stated that even then, a SIR may make the logical navigation simpler, since shorter than for the equivalent S-P1 query. Even worse, a popular DBS, e.g., MsAccess³, may be unable to execute the latter. Consider indeed that the referential integrity exists at least between P and SP. Suppose also that that S-P2 client wishes the data in Q3 below for every supplier in S, even if there is no related supply for the time being, e.g., for S4 in Figure 4. Recall that every latter supplier would be modelled as a dangling tuple. Any SQL query to S-P2 expressing this wish requires the logical navigation between S and SP. The least procedural is the half outer join:

(Q3) SELECT S.S#, S.SName, SP.p#, SP.PName FROM S LEFT JOIN SP ON S.S# = SP.S#;

For S-P1, the equivalent query would need the logical navigation also between P and SP. The following query could do:

(Q4) SELECT S.S#, S.SName, SP.p#, SP.qty, P.PName FROM S LEFT JOIN (P inner JOIN SP ON P.P# = SP.P#) ON SP.S# = S.S#;

Q4 is more procedural than Q3, enough to require a typical client to think for a couple of minutes at least how to formulate it. Unlike it is for Q3. In particular, since the left and the inner join do not commute in Q4, illustrating the rationale for the current SQL standard we spoke about. More generally, because of this and other annoying properties, [DD91], the navigation through outer joins gained the well-earned reputation of being even more awkward than through the inner ones only. Perhaps, that is why, e.g., the MsAccess outer join processing remains bugged since its earliest version. As one result, if S-P2 was implemented over MsAccess as we discuss in Section 4, Q3 would work. Q4 in contrast, cherry on the cake, would have its execution refused. A message would explain that it contains a non-supported join expression. That is clearly a bug, as Q4 conforms to SQL standard. As one may easily experiment, the bug would be independent of our implementation.

Query Q1 was a specific one. As final point here, observe however that S-P2.SP contains all the attributes of S-P2, although a few are renamed. We call it a universal SIR, as a new type of the universal relation [MUV84]. S-P2 avoids therefore the logical navigation through S-P1 not only for Q2 and its variants, but for every query addressing several relations in S-P1, provided its S-P2 equivalent may select all the tuples in S-P2.SP. Stated differently, the S-P1 query should select in S or P only the tuples that are projections of some S-P2.SP tuples. Still in other terms, the S-P1 query should not select dangling tuples in S or P. Most queries to S-P stated in the literature as representative of

³ Perhaps surprisingly, MsAccess is the most popular relational DBS by number of licensees, allegedly in hundreds of millions.
practical ones are of this kind, e.g., in [DD91] & its later edition. We recall that these tuples model indeed suppliers and part in an unusual situation.

6. Next, the following formulation of Q2 illustrates the similarity of the Select-clause sub-queries and of the IEs in basic form. The following query using a Select clause subquery for PNAME is indeed also equivalent to Q2:

(Q2.3) Select P#, (Select PNAME From P Where SP.P# = P.P#) As PNAME, QTY From S, SP Where SNAME = "Smith" And S.S# = SP.S# ;

The subquery here is indeed largely the same as it would be for an IE, say I_PNAME, inheriting PNAME only for our SIR SP, for any reasons, with Select keyword and explicit recursive join added to. The From and Where clauses of I_PNAME would be then the same as for I_P expanded similarly. Notice finally that Q2.3 is equivalent to Q2 for the default referential integrity, but remains the only correct between both also without. The reason is the de facto outer join semantics of the inner join in the subquery. E.g., if S-P1.SP contained, say, the tuple (S1, P7, 200), without P# = P7 in P yet, then only Q2.2 would correctly select the tuple (P7, null, 200).

7. We now illustrate that the IEs can make the conceptual scheme more faithful also through the value expressions impossible for VAs presently. As VAs nevertheless, these may shield the client from declaring them in queries to the base equal stored relations. Even more, such IEs may again make executable queries that would not be otherwise. Alternatively, as VAs again, they may save to the DBA the effort of creating the auxiliary CE-views hiding the value expressions. These views would be again largely more procedural than the IEs declaring those.

As first case, consider STATUS stored attribute in S-P1. Imagine that, behind the scene, one calculates it, e.g., as the total quantity delivered by a supplier divided by hundred and rounded to its integer part. Thus, the supplier of 100 - 199 parts has status 1 etc. The status of the supplier not supplying any parts for the time being is null. Having such STATUS as an SA could be clearly impractical. If S-P1 was a SIR DB, the DBA could simply alter S, using the Alter statement for SIRs we present later, dropping STATUS as an SA and recreating it as the following IE with implicit From and Where: Int ( SUM(QTY) / 100) As STATUS. Alternatively, the same IE in the secondary form would be simply: STATUS As Int ( SUM(QTY) / 100). In both forms, QTY is not prefixed, since only SP has this attribute. With all the clauses being explicit, STATUS would consequently be, in basic form only:

Int ( SUM(QTY) / 100) As STATUS From SP Where S# = S.S#.

Both the secondary and the basic forms with the implicit From and Where, minimize the procedurality of STATUS to the least possible. Namely, one provides only the IA name and the formula defining it. Besides, no VA can define STATUS at present, as the value expression employs an aggregate function and refers to attributes beyond S. The semantics of the basic definition with the Select keyword added and explicit From and Where clauses, is about that of the Select clause subquery calculating STATUS within a query to S-P1, e.g., within Q.3.1 just below. The only difference is that the clause: As Status is not in the subquery, but immediately follows it. The result for the IE, as well as for the subquery would be some STATUS values for Suppliers S1...S4 and a null for S5, as supplying nothing at present.

The client of SIR S may get S, through the simplest SQL query, namely:

(Q3) Select * From S ;

In contrast, if S without STATUS remained a stored only relation, the least procedural equivalent query would be:

(Q3.1) Select S#, SNAME, CITY, (Select Int ( SUM(QTY) / 100) From SP Where S.S# = S#) As STATUS From S ;
Q3.1 is clearly more procedural by far. We recall also that the non-procedurality of queries like Q3 charmed in its time the DB community to SEQUEL. Furthermore, clients usually dislike Select clause sub-queries. Many have hard time to even figure out the semantics of these. Cherry on the cake, a query with such a subquery may not work on a popular DBS. E.g., on MsAccess one cannot expand Q3.1 neither with Order By STATUS clause nor with Order By Int(SUM(QTY) / 100), i.e., the value expression in the STATUS scheme. More precisely, MsAccess refuses to execute the former, while it accepts the latter, but with errored result. S-P2.S user may trivially add the Order By clause.

We leave the definition of the CE-view S with the calculate STATUS as an exercise, e.g., on the basis of Q3.1, showing in particular that definition even more procedural.

As the second example, here is another IA, also impossible as VA currently as well. Besides, the recursive join defining it below is not an equijoin. Consider indeed that after declaring the above IA STATUS in S-P2.S we wish to append to S, as immediately following STATUS, an IA named RANK. For each supplier s, RANK(s) value is either one plus the number of Suppliers with STATUS higher than that of s or RANK(s) is null if STATUS(s) is null. The following IE in secondary notation, declared in S scheme next after STATUS IE, fulfils our requirements:

\[
\text{RANK} \equiv \text{IIF}(\text{status is not null}, (\text{select count(*) +1 from S X where status > s.status}), \text{null})
\]

Here IIF is the scalar function, e.g., of MsAccess SQL dialect. The IE is again a value expression, perhaps surprisingly to some. Its form is the secondary one. There is one recursive join within the subquery in IIF, illustrating our point that such join may not be an equijoin. The implicit From and Where clauses for RANK are: From S. Besides, RANK cannot be a VA at present at any popular DBS we are aware of. Finally, consider the simple query of obvious interest:

**Select * From S Order By RANK;**

As for STATUS alone, to the best of our knowledge no current popular DBS allows for a single equivalent query. Even with complex sub-queries and some logical navigation. The most practical solution is to create the CE-view of S with, again, the price of higher procedurality. Notice that this one has to be higher than for CE-view of S without RANK. The calculus of RANK requires indeed that of STATUS. As the result, we are not aware of any DBS, where a single Select expression calculates STATUS and RANK. Hence, the CE-view of S with RANK has to refer to some partial view defined first calculating STATUS. So, now we need even two Create View statements at least. We revisit this constraint in Section 4.

8. We now show an example of a circular reference among IEs we spoke about and of a way out from such a prohibited situation. Suppose that the STATUS computation should apply to S in SP-2. In (3) we supposed S-P1 DB with only S expanded to a SIR. In S-P2 in contrast, SP is a SIR and happens to inherit STATUS from S through I_S. As the above IE STATUS inherits it from SP, a circular reference among IEs would result. A way out is to observe that, fortunately, S.STATUS inherits from SAs of SP only. By definition these are not IAs thus and are all in the stored relation SP_B, the base of SP. As we have discussed in general, one may create therefore STATUS through the IE in Create Table S, with the implicit From SP_B:

\[
\text{STATUS} \equiv \text{Int } (\text{SUM(QTY) / 100});
\]

9. We will finally illustrate a way to define an IE whose Select expression potentially inherits several tuples per tuple of the SIR yet without the IE. We recall that IE must be so that this finally does not occur. Consider that DBA of S-P1 finds that a conceptual property of every part, to become the attribute following all the others already in P scheme at Figure 3, should be the list of all the suppliers of the part. The list called SUPPLIERS should contain for each part, the values of S#, SNAME, QTY, for every supplier of the part. The values should be in descending order on quantity supplied and in the ascending one on SNAME. For every part not supplied for the time being, SUPPLIERS should be null. If P is a SIR, altering the original P with the following IE will make the DBA happy. The LIST aggregate
function casts for each supplier all the selected tuples into a single Char type value (a string thus), [L3]:

SUPPLIERS LIST (SP.S#, SNAME, QTY) Where SP.S# = S.S# Order By Qty Desc, SNAME

The explicit join is there since it is not a recursive one. The From SP, S clause is in contrast implicit. There is also the recursive implicit join ‘AND’ed to the explicit one, namely P.P# = P#, the latter P# sufficing to designate SP.P# of course. Both implicit clauses complete SUPPLIERS when it is actually created by Create Table or Alter Table, as we detail in Section 4. The result, e.g., for P1 would be P tuple:

(P1, Nut...London, (S2, Jones, 300 ; S1, Smith, 300))

Without LIST, SUPPLIERS would not be an IE. The expression could indeed inherit more than one tuple for some P tuples, e.g., for that identified by P1. Also, like for STATUS before, observe that if P was in S-P2, then P altered as above would create a circular reference. The way out is the reference to SP_B instead of SP, as above for STATUS. Observe that the resulting IE would be fine for the altered S-P1 as well. Next observe that if P had to be a stored table only, making SUPPLIERS an SA would usually be impractical. The DBA would need to define a rather complex Trigger on SP. If the DBA is nevertheless in good mood and still wants to spare the additional procedurality to queries, creating a view should be more practical. However, this creation would be again more procedural than simply putting SUPPLIERS into Create Table P. Observe, finally again, that the simplest SQL query Select * From P is now equivalent to that with two joins between S, P and SP at present, i.e., in S-P1 still under SRV-model. Actually, without SUPPLIERS, the equivalent query would be more procedural even with S-P2. Nevertheless, with respect to S-P1, it would reduce the logical navigation by half still, sparing the join between S and SP. @

2.3 DDL Statements for SIR-model

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. Also, as mentioned for S-P1, we suppose the Foreign Key clauses possible for SIRs as well. Furthermore, Create Table for SIR-model may in particular define a stored (only) table. More generally, every DDL statement for SRV-model enters the SIR-model DDL by definition. As one wisely said, who can more, can less, Figure 1.

For views, in particular, we suppose the reuse as is of the kernel’s Create View DDL statement. The rationale is the absence of the recursive join(s) in a view. Reuse of Create View seems then easier to implement than some view scheme only specific syntax in Create Table for SIR-model. We come back to this issue in Section 4.

The other usual SQL DDL statements are, we recall, Alter Table, Drop Table, Drop View and Create Index. With respect to the generalization to SIRs of Alter Table, we suppose that it keeps its kernel dialect clauses Add, Alter or Drop with all their capabilities, to operate on any SAs. We add the extension specific to SIRs, namely that Alter applies also to the IEs. The alteration may add an IE, after or before any specified SA or IA or after all the attributes, by default. It may also replace or drop an existing IE. This may transform an SIR into a stored relation or vice versa. However, we prohibit Alter to drop any attribute serving a recursive join. In particular, Alter cannot transform a SIR in a view.

If such a need occurs, the extended 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 to cascade to other SIRs or be refused if a violation could result. Furthermore, as for any kernel we are aware of, if a view scheme should get altered, one should use Drop View followed by Create View statement. But, for SIR-model specifically, a view should also have potential to evolve into an SIR, i.e., get some SAs. We suppose then the Drop View followed by Create Table. Finally, we suppose that Create Index statement for an SIR applies naturally to its base only, reusing the syntax of the kernel one.
Example 2.

1. Suppose that we add to S-P2 the already discussed IA WEIGHT_KG and the IA WEIGHT_T converting it further to tons. For application dependent reasons, WEIGHT_T should precede in the scheme WEIGHT_KG.

   Alter Table P Add After WEIGHT WEIGHT_T WEIGHT_KG / 1000,
   WEIGHT_KG AS Round (WEIGHT * 0.454,3)) ;

   These IEs follow the secondary notation and would logically follow WEIGHT attribute in P, hence precede CITY. Main SQL dialects allow for both IAs as VAs and would also use the same syntax for their Alter statements. However, only the syntax of WEIGHT_KG IA is that of a VA today. The one of WEIGHT_T is little simplified.

2. We alter S in S-P2 by replacing SA STATUS with the IA we discussed.

   Alter Table S Alter STATUS As STATUS (Select Int (SUM(QTY)/100));

   Notice the implicit From SP_B and the implicit join in STATUS. Also notice that with P and S altered as above, all relations in S-P2 become SIRs. Below, we refer to the DB with P and I_P altered with IAs WEIGHT_T, WEIGHT_KG and S with STATUS IA, as to S-P3 DB.

3. Suppose now that the DBA of S-P3 decided that a conceptual property of every supply, of obvious interest to every client, is the total weight of the parts delivered, expressed according to each of the three units of measure above. Also, the DBA decided that these values should precede any IA already in SP. The following Alter Table statement may respond to the DBA wishes:

   (A1) Alter Table SP Add Before SNAME SPWEIGHT As WEIGHT*qty, SPWEIGHT_KG As WEIGHT_KG*qty, SPWEIGHT_T As WEIGHT_T*qty ;

   (A1) contains three IEs, since it defines each IA through the secondary notation. If these IAs were VAs, the syntax would be the same or almost, e.g. we would have SPWEIGHT As (WEIGHT*qty) for Alter Table of SQL Server. Alternatively, one could define all the IAs in a single IE, using the basic notation. The procedurality of both statements would be obviously the same. The only difference besides would be in implementation as we already stated. The basic form would be in contrast the only choice if one rather wished to refer to the originals in P, (for no good reason), using the following (single) IE:

   (A2) Alter Table SP Add Before SNAME WEIGHT*qty As SPWEIGHT, WEIGHT_KG*qty As SPWEIGHT_KG, WEIGHT_T*qty As SPWEIGHT_T From P ;

   In SRV-model, one choice for SP would be three more SAs, with their storage cost and update procedures within a trigger on SP, fired whenever a supplied quantity changes. The only other choice would be the CE-view SP. On a popular DBS providing VAs, hence a limited form of SIRs already, we recall, one more possibility would be one SA, e.g. SPWEIGHT and two VAs. We encourage the reader to try out all these choices, to find out how much more procedural they are than each above statements.@

2.4 Data Manipulation

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

In short, SIRs do not require extensions to any current DML statements. For a modification of an SIR, i.e., the SQL Insert, Update or Delete statement, each statement should act as we already discussed in Section 2.1.
Example 3. The simplest for SP SQL Select statement Select * From SP would show all the SP values, of all SAs and of all IAs in Figure 4. Supposing MS Access SQL as the kernel dialect, would make the statement Insert SP (select 'S4' as S#, P4 as P#, 100 as QTY); adding the tuple with these stored values and with all the virtual IA values. The statement Update SP set QTY = 250 where S# = 'S1' and P# = 'P1'; should normally succeed, updating one stored value in SP. The statement: Update SP set QTY = 250, CITY = ‘Paris’ where S# = ‘S1’ and P# = ‘P1’; also succeeds. The change to CITY may propagate to S indeed, as the CE-view of SP would let it to propagate, e.g., under MsAccess. The side-effect that might surprise would be the city change for every other supply by S1. Next, an update of SP.STATUS succeeds provided S.STATUS is an SA. But if S.STATUS is the above defined IA, any update to it in SP or even S must fail. Finally, the statement Delete SP Where S# = ‘S1’; would erase as usual physically from the DB all the values of the stored attributes in the selected tuples in Figure 4. Formally, it would also erase all the inherited values.

2.5 Utility of SIRs & Related Work

Usability of Relational DBs

The motivating examples illustrated how SIRs may make a relational DB less-procedural. In the nutshell, IEs in the SIRs in S-P2 or S-P3 were always less-procedural, than the CE-view or QE-view creation statements that S-P1 would need for the same more faithful to reality relations or, at least, for the same simpler queries. IEs concurred thus to better usability of S-P2 and of S-P3 than that of S-P1 with a CE-view or a QE-view of S-P2 or of S-P3.

Next, all the IEs in our example were components of the conceptual models of S-P2 or S-P3. They should serve thus the “commonwealth” of our DBs clients. This was actually the intention of the concept of an IE in general. One consequence was that the ER-model became useless for these DBs. In contrast, any relational views are in principle and, as we discussed, often in practice, only for some but not all users of a DB. The view concept was indeed intended so already in the ANSI-SPARC DB architecture that the SRV-model claimed conformity to. Serving more users may appear a better usability of S-P2 and of S-P3 than that of SP-1 with the CE-view or the QE-view as well. Besides, using a view as element of the conceptual scheme, is twisting the concept so that, in particular, we recall from the above, it contradicts the very own original design principles of the SRV-model itself.

Furthermore, every SIR in our example contained some SR of the S-P1 conceptual scheme. This is general of course, in the sense that for every conceptual scheme with SIRs, each SIR contains an SR of some conceptual scheme in SRV-model, presumably best normalized. The latter actually becomes the base-equal equivalent of the former. Any views are additional relations in contrast. A DB with more relations for the same purpose is in general considered less usable. Let us recall that the popular in its time universal relation research postulated to ultimately have only one relation in the DB, [M4]. Also, the alterations of SIRs we have discussed that would alter SR schemes in S-P1 required quite procedural changes to these separate relations, the CE-view or QE-view specifically. An eventual view maintenance trouble had potential for costly practical implications. In contrast, these alterations did not bother the SIRs in S-P2. This is another facet of better usability of S-P2 or S-P3 than that of S-P1 with one of the discussed views.

All our examples showed benefits from SIRs for the “biblical” DB, i.e., S-P1 only. However, as known, this one molded most of countless practical relational DBs. SIRs should benefit the usability of these DBs accordingly.

SIRs and Virtual Attributes

From SIR-model perspective, VAs may be seen as a partial helper with less procedural queries without views, present in the popular DBSs we spoke and some others about for decades. The rationale for this capability was that it is less procedural to define the VAs than to create the CE-view of the expanded relation. Our examples have shown that IEs generalize the VAs. Every VA scheme one can define at present is an IE in secondary notation, e.g., WEIGHT_KG. The inverse is not true, as
we have shown as well, e.g. for STATUS or RANK. Relations with VAs are SIRs without being explicitly named so. More precisely, they are self-inheriting SIRs only. Our proposal applies thus to VAs as is. Alternatively, one may see the SIR concept as formalization and generalization of a relation with VAs so that one also partly or totally avoids the logical navigation without additional views. The concept of an IE as we defined it can be seen accordingly as generalizing towards that double goal the de facto IE concept applying solely to VA schemes. In the same time, the “generalized” IEs respect in practice the rationale for the VAs, namely lower procedurality than that of any alternate views. Actually, we say “in practice”, since, one may construct a SIR, where CE-view or QE-view is less procedural than even the least procedural IEs for the SIR. One way towards is to design a DB with a SIR where for every IE in it, (i) From clause is explicit as less procedural than all the prefixes of proper names in Select list, (ii) all the recursive join clauses are explicit, (iii) the view can have the join clauses as the conjunction in the Where clause, i.e., the referential integrity with all the source relations is enforced. Then, for each IE in the SIR, the view may save one ‘ From ‘ and one ‘ Where ‘ keywords, at the expense of ‘ AND ‘ keyword in Where clause and one ‘,’ in the From clause, separating the source relation(s) of the IE from the list already there. Altogether, the view saves seven characters per IE. Enough of IEs may then compensate for the additional procedurality of the view, consisting of the list of SAs of the SIR and of ‘Create View As (Select ‘ string. Rapid calculation that we skip here, expanding S‐P2.SP as an example with more IEs and accordingly to constraints (i), (ii) and (iii), shows that about ten IEs would be the minimum for the searched goal. We are not aware of any application that would meet all these constrains.

**SIRs and Implicit joins**

There is another partial helper on some popular DBSs for decades, aiming at the same goal as VAs, but for the logical navigation. With that tool, the client avoids specifying selected inter-relational joins. The DBS adds these by default. They are consequently called implicit or automatic joins, [LA86], [LSV91]. E.g., MsAccess and SQL Server, provide the tool, although, as for VAs, not to the extent of the research proposals. For the purpose, in both DBSs, the query must be first designed in QBE, through the interactive graphical interface. The joins may be pre-declared in graphical QBE-like database sub-scheme. One may display in this one any stored or inherited relation and pre-declare inner or half outer joins. The interface allows also for the referential integrity definition. This declaration generates the implicit inner join between the primary and the foreign key attribute.

The DBA may activate besides an option called auto joins. Then, if attribute A is the primary key of one of the query (source) relations, say R and some other relation(s) of the query contain non-key attribute(s) A, then, for each of those relations, the DBS automatically creates an inner join on A with R. If several relations of the query have A as the primary key, then the DBS generates the auto joins only for the first one to be chosen for the QBE query. These design choices are somehow murky and bugged for decades, at least in MsAccess. They may lead to QBE queries that do not translate to SQL or do not execute while they should. Nevertheless, the tool appears globally useful, being maintained by Microsoft in each new release since twenty years.

It should appear from our motivating examples that IEs also generate implicit joins. For each IE in S-P2, for instance, its recursive join is also an implicit half outer join for a query otherwise to S or P and S-P1.SP. It may boil down to implicit inner join, as we discussed. The obvious major advantage of the IEs is the absence of the graphical interface hassle and of the subsequent limitation to interactive queries only. Some advantage is also that the query, e.g., Q1 above, may address a single relation. For the implicit joins interface, an equivalent query must still address all three relations in S-P1. It must remain thus slightly more procedural than Q1.

The existing tools provide however also for implicit joins that could not be within an IE, e.g., between an SA and a view or between two views. These may nevertheless apply also to a SIR and a view or to two views in a SIR DB, perhaps of SIRs besides. The precondition is that one generalizes the current
graphical interface to SIRs. Actually this should be easy, as the interface may display any CE-view. Hence to display a conceptual scheme with SIRs at present, assuming a SIR DB implemented at one of these DBS as we discuss in Section 4, it suffices to display all the SRs, if there is any in the DB, and all the CE-views of the SIRs, [L6]. In this way, again the SIR-model appears again the “umbrella” for both VAs and the implicit joins tools.

**Backward Compatibility of SIR-model**

As we also mentioned, the SRV-model is a strict sub-model of the SIR-model. Thus every SRV-DB is also a SIR-DB, but not vice versa of course. The trivial condition to stay within the current relational model for a SIR DB, is simply to refrain of SIRs. Switching to the SIR-model is safe thus. No loss of any current capabilities of a relational DB may result from. In this sense, expanding with IEs any SRs in a SIR DB being also an SRV DB is always backward compatible. Consequently, any application that runs on the latter will continue to run as is on the former.

Example 4. Figure 3 and Figure 4 illustrate that if for a SIR-model enabled DBS, for any reasons, we wish S-P1 DB only, it suffices to drop I_S and I_P from S-P2 scheme. The price to pay is that any queries become then restricted to these possible under SRV-model only as well, with all the eventual disadvantages we lengthily discussed. Moreover, any queries to S-P1 remain executable on S-P2 as well. S-P2 is thus backward compatible with any queries to and applications running on S-P1.

**Other Models of Inheritance**

Finally, it is the common knowledge that the concept of inheritance with its sub-classes, sub-types, sub-tables... is “the heart” of the object oriented (OO) programming and systems. It got also incorporated in 90ties into the, so-called, object-relational database systems. The popular open-source PostgreSQL DBS is the most prominent survivor of this trend, [S96], [P]. The inheritance in PostgreSQL has a dedicated DDL clause, called INHERITS, in its Create Table for a sub-table. The clause incorporates into the sub-table the entire scheme of the super-table. The inherited scheme is expanded with the schemes of the attributes specific to the sub-table. The tuples in the sub-table, e.g., in (state) Capitals table in the flagship inheritance example in PostgreSQL tutorial, are not supposed to be also in the super-table, i.e., in Cities table in this example. Each tuple of the latter models only a city that is not a state capital. The SQL DML semantics was consequently modified to realize a default UNION of the content of the super and of its sub-table(s), when a Select query addresses the former, e.g., for the selection of some attributes of any city using the SQL clause From Cities. To avoid such a union, the explicit keyword ONLY is necessary in the Select expression. Notice finally that one has to know also upfront whether a city to insert is a capital or not.

This implementation of the inheritance concept, including the modification of the SQL DML semantics, is unique to PostgreSQL to the best of our knowledge. It clearly differs from that for an SIR. First it concerns only schemes and only of all the stored attributes that becomes the schemes of stored attributes in the sub-table. For an SIR, an IE concerns both schemes and tuples. As for views, the source can be SAs, but also IAs, while the result is only IAs. The SQL DML semantics remains also unchanged. For the PostgreSQL example, since no tuple of Capitals is supposed to be in Cities, while Capitals is still supposed to have the same stored attribute schemes as Cities, SIRs would be of no use.

The rationale for it is that SIR-model has a different conceptual model of inheritance. For PostgreSQL example, the Capitals table could have the same intention, but Cities should be a stored relation for all the cities. To avoid any relational anomalies in Capitals, the only shared stored attributes should be, as usual, the primary key of Cities, becoming the foreign key in Capitals. This is actually the basic interpretation of inheritance for the declaration of sub-tables in MsAccess, we discuss soon. It is also advised for modeling inheritance under SQL Server (see any SQL Server Tutorial). In our case, an IE in Capitals scheme that would become then that of an SIR, could inherit consequently from the other attributes of Cities, but as IAs only. It could inherit from all of them, mimicking PostgreSQL example scheme of Capitals, or only from some. The latter capability could be useful if Cities had for instance
also an attribute indicating the distance of a city to its state capital. Finally, an IE could in particular transform some inherited values. Examples are easy to imagine. The PostgreSQL INHERIT clause does not provide any equivalent capabilities.

**Usability and Optimality of a Relational Conceptual Scheme**

Since decades, there is a widespread informal consensus that the SRV-model relational DB conceptual schema consisting of stored relations only is not sufficient in practice. One reason is the implied procedurality of the queries. The “corrective” procedure of adding views for a more usable scheme appears too procedural as well. This is perhaps the reason why, from various universal (view) relation proposals and now extinct passionate interest in the topic, [M4], none made to the industry. The only partial attempts of more usable conceptual schemes that made it are the VAs and the implicit joins. SIR-model subsumes both tools as we discussed. It provides in this way for the most usable, in the discussed sense, conceptual schemes at present.

At the peak of its glory, three-four decades ago, research on SRV-model issued countless proposals for somehow optimal (conceptual) relational schemes. In practice, remain basically the NFs, the Heath’s Theorem and the Fagin’s one. SIR-model (conceptual) scheme design rules obviously should preserve that effort for the (stored) bases of SIRs. The new need is the integration of IEs with. The SRV-model basic design rules need to be restated accordingly. Such a proposal is the subject of what follows.

3. **SIR-model Relational Schema Design**

The relational scheme design goal was the removal of the anomalies. We preserve the goal for SIRs, but, through IEs, aim also on the queries free of the logical navigation and value expressions. We continue with S-P2 as the motivating example. We first restate the NFs for SIRs. Next, we restate accordingly the Heath’s and Fagin’s theorems. The restated theorems generate the same lossless decompositions, but with SIRs as projections, instead of the present ones. The benefit from the restated Heath’s theorem is the total absence of the logical navigation, otherwise necessarily generated by the (original) Heath’s theorem. The restated Fagin’s theorem avoids in contrast the original logical navigation only partially. Both results take care of the IAs defined through the value expressions.

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 would contradict S, where SNAME is anyhow already. Using SP instead, avoids the trouble.

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

Example 5. 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) 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 Sir DB 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 presents a (non-trivial) multivalued dependency (MVD). The latter, by far more frequent, characterizes schemes with the functional dependencies (FDs) only. The least number of relations means the grouping of all attributes functionally dependent on the same one(s) into possibly one relation, with the latter as the primary key. Possibly means here the respect of a myriad of other less or more fuzzy criteria, e.g., not “too many” null values for some attributes.

Designing a scheme is furthermore usually a many-steps process. Ideally, we optimistically 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, practical chances for U in 4NF are zilch. We usually perform then a decomposition of U into projections, i.e. we suppose that the DB consists of these projections as SRs instead. The decomposition must be lossless, producing the projections whose equijoin equals the decomposed relation. Any projection may end up proven in 4NF or proven in BCNF and free of any MVDs. It is then in 4NF thus as well. Or, a projection may not end up so. We decompose any such projections again. We continue, until every projection is anomaly-free, in which case we create it as an SR.

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

We now generalize these principles to the SIR-model, i.e., U and the projections may be SIRs. Such schemes were out of scope of the original methodology, of course. In other words, even U may contain IAs, e.g. the aggregate ones we showed. For FDs and MVDs used for the decompositions, we nevertheless originally assimilate all these IAs to SAs. We apply to the projections the restated NFs. Then, in contrast, we consider any IA again as is. For the Heath’s and Fagin’s theorems rested for SIRs, our goal is that the decomposition of an SIR, say R again, is not only lossless, but also at least one of the projections inherits some, possibly all, attributes of R. The result aimed on is that the lossless decomposition possibly does not cost us the logical navigation through the projections, unlike for the original theorems. We leave for the future eventual restatement of multitude of other rules aiming on best schemes, [D12], [F11], [V11], Rissanen’s work included.

The major gain that will appear below is that, for the same stored relations and the same size optimal schemes for a DB altogether, the DB using SIRs effectively spares the discussed logical navigation. More precisely, the optimal SIR scheme will be always as follows:

(a) The SRs whose schemes are projections resulting from original Heath’s and Fagin theorems become bases of SIRs resulting from the restated theorems or remain the same SRs.
(b) In the absence of MVDs, no restated decomposition creates the logical navigation through the projections.
(c) Otherwise, a restated decomposition removing an MVD still spares or at least reduces the logical navigation for some queries addressing the projections, but not for all such queries.

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

Indeed, first, the Heath’s theorem states, we recall, that for any stored relation ABC (A, B, C) and an FD A -> B, the decomposition AB (A, B) and AC (A, C) is lossless. That is: ABC (A, B, C) = AB (A, B) Join AC (A, C). In practice, as known well, we may have several choices for A, B and C. As every decompositions doubles A, for stored relation ABC, it is wise to choose A with fewest attributes, at least for this reason. Likewise, A should be the primary key of AB. B does not depend on any proper subset of A then and AB is in 2NF at least. Also, for reasons previously invoked, we should hunt for the largest B. We may end up nevertheless with AB not in 3NF at least hence B may get decomposed in turn, etc. With all the principles stated in mind, we restate the theorem for ABC being a stored relation or an SIR, as the decomposition into AB (A, B) and ACB (A, C, B). Here, B denotes B inherited through the IE: B from AB, with the implicit recursive join A = ACB.A. The position of B within ACB does not matter of course for the decomposition.

![Diagram](image)

Figure 5. Restated (a) and original (b) Heath’s decompositions, as well as, (c) restated Fagin’s decomposition.

As the original decomposition, the restated one is also into two schemes and clearly lossless. But, while AC was a stored relation, ACB is an SIR with base AC. This decomposition is possible only for the SIR-model. Unlike the original one, while it conserves AC as a projection of ABC, it also preserves the original attributes A, B, C together. It avoids thus, as promised, the logical navigation to queries selecting B and C.

Figure 5 illustrates both decompositions. At Figure 5.a, query Q (B,C) selecting B and C, say for each tuple that was in ABC, is logical navigation free for the restated decomposition. It may indeed access only one projection. For the original decomposition however, Figure 5.b, the equivalent query Q’ cannot be. It has to join the projections. Query Q remains possible, but only provided the expense of creating the CE-view ABC from the projections.

The decomposition thought the Heath Theorem is usual called lossless. The term appears partly misleading, in the light of the restated decomposition. Each original projection loses indeed some attributes of the decomposed relation. Only the recovery through a join is lossless. It is the reason for the logical navigation in every query to attributes in different projections. Also, it is the reason for the join view providing the recovery, necessary in SRV-model to avoid the navigation. Only the restated decomposition should qualify as lossless, since one of the projections keeps all the original attributes and values.

In addition, the view scheme is always more procedural than the IE in that projection. Also, if there are several decomposition steps, as usual, e.g., there are obviously two for S-P1 and one creates the join view at every step, every new step not only creates new view, but also need the altering of some previous view(s). Such decomposition process is sufficiently procedural to be useless in practice. Rather, one generates all the projections first, then the universal (join) view. Creating that view may
still require partial views, e.g., for STATUS and RANK. Even without these, IEs remain always less procedural in practice, as we discussed.

As we mentioned in the introduction, finally, the freedom from the logical navigation provided by the restated decomposition(s) holds nevertheless only until, as often, dangling tuples enter a stored projection. The join is no more lossless. It will appear, but is easy to observe already, that the content of S and of SP at Figure 4 illustrates the case, because of the dangling S4 tuple in S, already mentioned in Example 1.2. Queries of obvious practical interest may require the logical navigation then again. Even so, SIRs should typically reduce that one with respect to the current needs. Actually, queries Q3 and Q4 in that example already pointed out these facts, although we did not refer there to Heath’s theorem.

Next, the Fagin’s theorem also states that in presence of MVD \( A \rightarrow\rightarrow B \mid C \) in the presumably stored relation ABC (A, B, C), its decomposition into AB (A, B) and AC (A, C) is lossless. Now, suppose \( B' \) being a (perhaps empty) subset of B such that \( A \rightarrow B' \) and let \( C' \) be a (perhaps empty) subset of C, where \( A \rightarrow C' \). Actually, we may about always expect either \( B' \) or \( C' \) non-empty, but not both, as in the example that follows. We restate the theorem as follows. Suppose ABC a stored relation or an SIR. The restated decomposition creates \( ABC' \) (A, B, C') and \( ACB' \) (A, C, B') where (i) the IE: \( C' \) from \( ACB' \), with the implicit recursive join \( ACB'.A = A \) defines \( C' \) and (ii) the IE: select \( B' \) from \( A = ABC' \), with the implicit recursive join \( A = ABC'.A \) defines \( B' \). As Figure 5c illustrates, \( C' \) and \( B' \) avoid the logical navigation for any query to B and C' or to B' and C in the projections, unlike for the original decomposition. Only a query to B/B' and C/C' still needs it.

For an MVD thus, our decomposition does not avoid completely the logical navigation that the original decomposition may create. If we wanted to, as long as there is no dangling tuples, a full auxiliary view scheme of ABC, as at Figure 5.b, would be the only option for both decompositions. But, without this price, our decomposition do limits the navigation to fewer queries. Furthermore, as it will appear the remaining queries should usually have the logical navigation through the final optimal scheme of the DB partly limited, i.e., with fewer joins. Notice that the restated theorem again conserves each original stored projection as is or as the base of one of the SIR projections.

On these foundations, the generic schema generation algorithm for SIRs is quite analogous to that for the stored relations only. More precisely, U remains the starting point, except that it may have IAs upfront. From there, we perform the same, wisely chosen, successive decompositions eliminating MVDs and “annoying”, i.e., anomaly creating, FDs. However, at each step, we now use a restated theorem instead. If we face both dependencies, the restated Fagin’s theorem works first. We naturally end up with the same stored relations, hence the same size scheme, but also with less logical navigation, as claim (b) states. If there are no MVDs, we remove the discussed logical navigation entirely, as claim (a) states. Finally, the rationale for claim (c) is that in a real-life DB, MVDs are rare with respect to annoying FDs. Also, \( B' \) or \( C' \) usually have several attributes, unlike B/B' or C/C'. Even for a decomposed MVD, most queries to the projections should be normally logical navigation free as well.

The following example illustrates all the debated points.

Example 6. 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-P1. Each supplier has now one or more contact email addresses. Each address may serve for any inquiry about the supplies or the supplier itself. Each address is the value of new stored attribute EMAIL. Every address is for one and only one supplier. We redesign the S-P scheme under SIR-model accordingly. We call the result S-P4.

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

\[ U \text{(EMAIL, } S\#, \text{ SNAME, STATUS, CITY, PH, PNAME, COLOR, WEIGHT, PCITY, QTY)}. \]
Notice the necessarily different names for the supplier and part cities, unlike in S and P of S-P1 or S-P2. U is potentially the optimal stored relation for S-P4, unless proven otherwise. What’s easy, since EMAIL already introduces the MVD: S# -> EMAIL | (SNAME, CITY, STATUS, P#...QTY). U is not in 4NF thus. Regretfully, U cannot be the optimal S-P1 scheme. We have to decompose it. We have MVDs and obviously FDs. We start with the restated Fagin’s theorem. The decomposition creates two projections, say SE and SP, as follows:

SE (S#, EMAIL, SNAME, STATUS, CITY), SP (S#, SNAME, SCITY, STATUS, P#...QTY).

SE is now an SIR, where the IE: SNAME…CITY From SP, with the implicit recursive join S# = SP.S# in Where clause defines all the IAs, in Italics. Its base (S#, EMAIL) would be the stored relation for the original Fagin’s decomposition. SP is the same for both decomposition. We have now thus C’ = (SNAME, STATUS, CITY) 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 spare the logical navigation to any queries to EMAIL and to any of its IAs. Otherwise, these queries would necessarily navigate over SE and SP. In contrast queries selecting emails and an attribute in SP that was not inherited in SE would still need to navigate, i.e. would require a clause expressing SE join SP clause. We come back to these queries later on, showing that practical ones should require lesser navigation anyway, backing up our earlier claim.

SE has no more MVDs, hence it is also in 4NF. SP has no more MVDs neither. But, is not in (restated) BCNF (hence neither in 4NF). The restated Heath’s theorem applies. For all the already discussed reasons, we choose the following decomposition, with S# as A, in particular since it is a single attribute key:

S (S#, SNAME, STATUS, CITY), SP (S#, P#, PNAME…PCITY, QTY, SNAME, STATUS, SCITY).

Here, the IAs in SP result from the IE, say I_S: # From S, with the implicit join Where S# = SP.S#. The projection SP is again an SIR, with I_S defining B. The related supposedly stored attributes of the decomposed SP, remain thus preserved in the projection SP, as IAs from S. Notice that this does not change anything for SE scheme. S remains the stored relation, as for the original decomposition. It is in BCNF. SP however still isn’t in restated BCNF. Its projection on the stored attributes isn’t in BCNF in SRV-model indeed, given the FD : P# -> PNAME, COLOR, WEIGHT, PCITY. We thus apply the restated Heath’s theorem again. One gets SP decomposed to:

P (P#, PNAME, COLOR, WEIGHT, PCITY) and SP (P#, S#, QTY, SNAME, STATUS, SCITY, PNAME, COLOR, WEIGHT, PCITY).

Here, the IAs of SP result from again from I_S and from, say, I_P: # From P, with the implicit join Where P# = P.P#. Now S-P4 has every relation in BCNF, hence in 4NF, as there are no more MVDs. The optimal scheme is as follows. We underlined the primary key stored attributes.

S (S#, SNAME, STATUS, CITY),

P (P#, PNAME, COLOR, WEIGHT, PCITY),

SE (S#, EMAIL, SNAME, STATUS, SCITY) /* I_S defines IAs

SP (P#, S#, QTY, SNAME, STATUS, SCITY, PNAME, COLOR, WEIGHT, PCITY) /* I_S and I_P define IAs.

Notice that SP scheme is almost that of S-P2 from Example 1. The only difference is PCITY instead of CITY in P. That is why the S-P2 scheme is the optimal one as well. Because of IEs, most practical queries is now clearly logical navigation free. However the already signaled queries to SE and SP are not. Some of these queries, e.g., select every P# supplied by supplier with given EMAIL, seem of marginal interest only. Clients in practice need also names. Then, the restated decomposition still reduces the logical navigation by two joins otherwise necessary, i.e., SE with S to get SNAME and SP with P to get PNAME. We may thus reasonably expect at least some logical navigation spared for
practical queries to SE and SP together and for most of such queries to projections of a decomposed MVD in general.

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

\[
\text{SE (S#, EMAIL), SP'} (\text{EMAIL}, \text{SNAME}, \text{P#} \ldots \text{S#}),
\]

where the IE, say ISE: S# From SE, with implicit join Where EMAIL = SP’.EMAIL defines S#. SE is again in BCNF. But now, SP’ is also free from any MVD, hence we do not need Fagin’s decomposition for it neither. However, SP’ isn’t (yet) in restated BCNF. Through successive restated Heath’s theorem decompositions, the final scheme for S-P would be:

\[
\begin{align*}
S' & (\text{EMAIL, SNAME, STATUS, CITY, S#}), \\
\text{SE} & (\text{S#, EMAIL}), \\
\text{SP'} & (\text{P#, EMAIL, QTY, S#, SNAME, STATUS, CITY, PNAME, COLOR, WEIGHT, PCITY}).
\end{align*}
\]

Here, an IE, say I_S’: # From S’ with implicit join Where EMAIL = S’.EMAIL and I_P: # From P, with the implicit join Where P# = P.P# defines the IAs in SP’. Now, if a supplier had \(m\) email addresses on the average, S’ and SP’ would have each \(m\) time more stored values on the average than, respectively, S and SP. We have more stored values than before, i.e., a sub-optimal result, as predicted.

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

\[
\text{S (S#, SNAME, STATUS, CITY),}
\]

where STATUS results from the IE: STATUS INT(SUM(QTY)/100), with the implicit FROM SP_B and the implicit join Where S.S# = S#.@

Notice finally, that the order of attributes in a relation mathematically does not matter. In SQL it does however. If so, one may move around the IEs produced by the enhanced decomposition. One may even cut an IE into \(x > 1\) IEs, so to accommodate SAs among IAs of the original IE.

4. Implementing SIR DBs

4.1 Basic Processing Scheme

As said already, the most practical way towards the SIR-model enabled DBS, is to transparently manage an SIR DB by an existing (kernel) SQL DBS. One way is to create the SIR-layer managing the SIR DB through calls to the kernel services, Figure 6. For the kernel, SIR-layer appears as any clients. SIR-layer processes every DDL or DML statement for an SIR through the internal generation of these for the kernel. It’s obviously useful to have the SQL syntax at the SIR-layer as compatible as possible with the kernel SQL dialect. Below, we presume the total immersion of the kernel syntax in the enhanced one.

In particular, for the Create Table statement received, SIR-layer should determine the type of the relation to create. Without any IEs, it is a stored relation. SIR-layer should push down then to the kernel the statement as is. In turn, the processing of Create Table of a SIR by SIR-level should be clearly more involved. First SIRs obviously need dedicated kernel meta-tables for the IEs. The schemes of these are easy enough to skip details. Let us deal first with the creation of a SIR, say R (B, V) again. The simplest design seems to represent R in the kernel by the already discussed CE-view R. SIR-layer simply forwards afterwards every incoming query to the kernel basically as is. The kernel takes care of the query evaluation. SIR-layer avoids this complex burden.

It is then up to SIR-layer to implement every CE-view R when it gets the CREATE TABLE R statement with IEs. Below, we propose that SIR-layer performs the task through the requests for the creation of (i) the stored relation B and of (ii) the sequence of view(s) creating progressively V by adding for each
next view the IAs within a selected IE to all those already in B or in the latest previous view. The view R is the last one. As it will appear, this one may however only perform some attribute reordering of the latest previous view to get these in their order in SIR R. We recall that the order of the attributes in the CE-view R should be then that of the top-down and left to right attribute definitions in CREATE TABLE R. We now define this algorithm. For sake of comprehension, we remain reasonably informal.

We qualify the overall result of basic (processing) scheme, (BPS), for SIRs.

As we have seen, an IE may define (a) one or more IAs through a Select list, provided none is defined through a value expression, e.g., _S IE. Alternatively, it may define (b) an IA through a value expression in a self-inheriting IE e.g., for WEIGHT_KG and RANK IEs. Next, (c) an IA may result from a value expression involving attributes sourced in other relation(s) and not being arguments of aggregate function(s). This could be the case of the IE: SPWEIGHT As WEIGHT*qty that we have discussed, if WEIGHT was only in P. Finally, (d), it may define an IA through a value expression with aggregate over an attribute in a different relation, e.g., as for STATUS. The term with the aggregate may be in particular involved in an arithmetical expression with some (non-aggregated) attributes of the SIR, perhaps bracketed by scalar functions. SIR-layer starts, at least conceptually, with the rewriting of every IE into its canonical form. We say conceptually, since the only intention in this form is to have a convenient representation of an IE for the BPS processing rules that follow. First, every IE in its canonical form has all its eventual implicit clauses made explicit. Making such clauses explicit for a Create Table or Alter Table, involves for BPS tedious, but rather easy details we skip here. Likewise, BPS expands eventually ‘#’ into the explicit list of the IAs. Then, the canonical form depends on the IE. Let I’ be an IE and I the same one but in the canonical form. Let us start with an I’ of type (a), hence basically written as: A [From F’] [Where W’]. Here A denotes all the elements of the Select clause, F’ is the content of explicit From clause and W’ is the one of Where clause.

As said W’ is then the conjunction of the recursive joins between the SIR, say again R, and all the source relations, say, S1...Sn. Namely, one has: R.J1 = S1.J1 AND R.Jn = Sn.Jn. Each J may be a composed attribute, all its element attributes being then ‘AND’ed as well. For I in the canonical form, BPS replaces this conjunction and the whole explicit Where clause thus, with imbricated half-joins preserving R in From clause, i.e.:


If a J is composed, then the On clause is followed by the conjunction of all its component attributes. Also, in practice, additional n parentheses brackets may be required by the dialect of SQL of the core DBS. E.g., for I’ = I_SP, BPS outputs I as follows:

I = SNAME...CITY As PCITY From (SP Left Join S On SP.S# = S.S#) Left Join P On SP.P# = P.P# ;

For I’ in secondary notation BPS converts it into the basic form. Next, if I’ is self-inheriting, i.e., case (b) above, BPS makes I = I’. For case (c), if BPS finds or produces the explicit Where, then it proceeds as for I’ of type (a). If it does not, the IE must be errored, as it cannot guarantee a single value per tuple of all other SAs and IAs. For type (d) finally, BPS forms I by reformatting the term with the aggregate in I’ into the format of the SQL subquery in Select clause of a query to the SIR. E.g. for STATUS we have:

I = (Select Int ( SUM (QTY) / 100) From SP_B Where S_B.S# = S#) AS STATUS From S_B.

Let us denote R the CE-view of SIR R. Once all the IEs of SIR R in canonical form, BPS creates the CE-view R through the following sequence of partial views.

Let I1...In be the IEs in R scheme, each in its canonical form, numbered in the order of their evaluation. BPS chooses the order letting each IE to find all the attributes it refers to in the CE-view resulting from the previous IEs. For instance, it chooses the view creating WEIGHT_KG as preceding that creating WEIGHT_T. Idem for STATUS and RANK. Let it be R0 = B, operationally named R_B in the kernel. Next, let R1...Rn be the views produced each respectively by the evaluations of I1/I2...In, as follows. We denote Ij (X), an Ij having formally X as SIR name, although X is a view name below. For
SIR $R$, $I_j(X)$ means in practice thus the replacement everywhere in $I_j$ of name $R$ with $X$. BPS creates the series of the following views

(V1) Create View $R_i$ As Select $R_{i-1}.*$, $I_j(R_{i-1})$; /* $i = 1...n$

View $R_n$ attributes may be in the order of these in SIR $R$ or not. Recall that while relations $R_n$ and $R$ remain then nevertheless equal for the relational algebra, they are not for SQL. For the latter case, instead view $R_n$, BPS creates thus the view, say $R_n'$, with all and only attributes of $R_n$ but ordered as $R$. CE-view $R$ is accordingly either view $R_n$ or view $R_n'$, operationally named by BPS view $R$.

View $R$ is the CE-view $R$, since, indeed, for $n = 1$, view $R_1$ is either the join of $B.*$ and of $I_1.*$ on some matching attributes, preserving every tuple in $B$ through the recursive join clauses in From of $I_1$ or is $B.*$ with additional value expression based attribute value for each $B$ tuple, or is $B.*$ tuple followed by a null $I_1$ tuple. Supposing $I_1$ well-formed, i.e., respecting IE’s principle, the result is a single tuple of $R_1$ for each tuple of $B$. Likewise, for $n = 2$, $R_2$ joins every tuple of $R_1$ that serves as $R'$ in turn, with the matching tuple of IAs of $I_2$ or extends each $R'$ tuple with the value expression based attribute value, or with the null $I_2$ tuple. Again, provided $I_2$ well-formed, the result is a single tuple of $R_2$ for each tuple of $R_1$. Etc.

Summing up, to process a Create Table $R$ statement, the SIR-layer starts thus with the rewrite of all the IEs needing it into the canonical form. Then, it requests from the kernel to create the stored relation $R_B$. Next, it requests to create the view $R_1$, named, say, $R_1$, then $R_2$ as $R_2$ etc. until the view $R_n$. SIR-layer verifies finally whether this one would be a CE-view. If so, it renames it $R$. Otherwise it replaces $R_n$ with the reordered view $R$.

Example 7. (1) We submit to SIR-layer S-P2 scheme at Figure 3 to create. SIR-layer applies BPS and finds no IEs in Create Table $S$ and Create Table $P$. It passes each statement as is to the kernel. It finds IEs in Create Table $SP$. It proceeds according to rules (i) to (V6) above. It does not find in $I_S$ any attribute created by $I_P$ and vice versa. It therefore chooses $I_1 = I_S$ and $I_2 = I_P$. Consequently, according to BPS, the SIR-layer issues the statements:

Create Table $S$... /* Usual creation of a stored table with the attributes at Figure 3.

Create Table $P$... /* Idem

Create View $SP_1$ As select $SP_B.*, SNAME, STATUS, CITY As SCITY From $SP_B$ Left Join $S$ On $SP_B.S# = S.S#$; /* $I_S$ is in the canonical form

Create View $SP$ As select $SP_1.*, PNAME, COLOR, WEIGHT, CITY As PCITY From $SP_1$ Left Join $P$ On $SP_1.P# = P.P#$; /* $I_P$ is in the canonical form, $SP_1$ is in the canonical form, $SP_2$ is $SP$ directly, hence there is no reordering.

(2) Suppose that the user wishes to create the DB S-P3, Example 2. For relation $S$ that became an SIR, we recall, BPS generates the following statements for the kernel:

Create Table $S_B$... /* With $S#$, SNAME, CITY

Create View $S_1$ As Select $S_B.*, (Select Int ( SUM (QTY) / 100) From $SP_B$ Where $S_B.S# = S#$) AS STATUS FROM $S_B$; /* STATUS IEs, see Example 1.3, became a subquery in Select clause.

Create View $S$ As SELECT $S_1.S#$, $S_1.SName$, $S_1.STATUS$, $S_1.City$ FROM $S_1$;

Then, for relation $P$, BPS generates $I_1 = WEIGHT\_KG$ and $I_2 = WEIGHT\_T$, since the latter refers to the former. Consequently, BPS issues the following four statements for $P$ to the kernel:

Create Table $P_B$...

Create View $P_1$ As Select $P_B.*, WEIGHT/2.1$ As WEIGHT\_KG From $P_B$; /* $I_1$ is the canonical WEIGHT\_KG.
Create View P_2 As Select P_1.*, WEIGHT_KG/1000 As WEIGHT_T From P_1; /* I_2 is the canonical WEIGHT_T.

Create View P As Select P#, PNAME, COLOR, WEIGHT, WEIGHT_T, WEIGHT_KG, CITY From P_1;
/* CE-view of P after attribute reordering in P_2.

Finally, for SP, BPS generates same statements for the kernel as for S-P2, except that I_P, hence also SP, include now also SPWEIGHT_T and SPWEIGHT_KG, we recall.@

Figure 6 illustrates Example 7.2. The figure recalls that S-P3 has only SIRs. The SIR-layer shows these as rectangles. Each size reflects the number of tuples and the number of attribute values per tuple as seen by the client. This perception corresponds to Figure 4, augmented with the IAs proper to S-P3. The lower part shows under the same convention the stored relations and views possibly implementing S-P3 over some current DBS. We represent the views created by BPS in the bottom up order. A rectangle name reflects a view name and the IE used for its creating. The CE-view P restoring the original order is shown as P(*).

View representations at the figure illustrate the IEs creating them. Each rectangle length is the same as for the SIR-layer rectangle. But it is not so for the width, for the rectangles representing the stored relations especially.

As the figure shows, for the three SIR schemes, the BPS would generate nine relational schemes in DBS. Three would be the stored relation schemes. Hence, they would define the relational conceptual scheme of S-P3 in the SRV-model. The views would help the queries to S-P3 to be less procedural, avoiding some logical navigation, value expressions and perhaps additional mandatory auxiliary views, as we discussed. Without SIRs, the user or DBA wishing simpler queries would need to basically create all these six views 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 more complex but single view per SIR. It would lead to the minimum of two schemes per SIR in an actual DBS, or even only one if SIR-layer generates the virtual attributes. But, the single-view strategy could end up a bad idea for multiple IEs, as pointed out.

Finally, to appreciate the non-procedurality gain our sample SIRs may bring, perhaps formulate to S-P1, assumed in SRV-model, the query equivalent to the following simple one to S-P3: Select SNAME, SUPPLIERS, PNAME From SP Where STATUS > 2 And SPWEIGHT_T > 1 ORDER BY STATUS@
4.2 SIR-Layer DDL & DML Statements Processing

With respect to the other DDL statements for SIRs, the Alter Table and Drop Table also require more processing than their kernel counterparts. For Alter, the SIR-Layer may request from the kernel to create/drop sub-queries when a SIR gets/loses IAs. We skip the easy, but tedious general discussion. As an easy example, consider how adding RANK to S alters the above views of S. In contrast, the SIR-layer sends every CREATE VIEW statement as is to the kernel. That one processes the statement as usual, given that, for every SIR named in the statement, it (hopefully) finds its CE-view.

With respect to DML statements, once BPS created the CE-views, SIR-layer sends every SIR-layer query as is to the kernel. For every SR or view the query names, the kernel proceeds as usual. For every SIR in the query, the kernel processes its CE-view instead. For a SIR-layer update query to a SIR in particular, the kernel proceeds accordingly with the CE-view update. The kernel may then be unable to satisfy for some queries the intended semantics of the SIR update queries from Section 2.1. As widely known, view updates in popular DBSs are indeed subject to numerous limitations. Cherry on the cake, every DBS has somehow different limitations. We take therefore the simplest practical stand for the SIR-layer, i.e., that an update query to SIR R is valid iff the kernel processes it. As for view update queries at present, if a SIR update query turns invalid, the client may attempt to rephrase it. Ultimately, one may always address the rephrased update(s) directly to the SRs or to the bases of SIRs involved in the initial query. We leave for the future the ambitious goal of enforcing the full semantics of SIR-layer update queries, in spite of the limitations of a specific kernel. E.g., for SQL Server a view is updatable only if it inherits from a single SR, unlike S-P2.SP thus. If SQL Server is the kernel DBS thus, the client has to rephrase every update query to this SIR to update queries to S, P or SP_B. In contrast MsAccess and MySQL are less restrictive: they accept update queries to views over multiple tables as well. However, details differ. E.g., some UPDATE SP queries to S-P2 could be valid for both kernels. Even more uniformly, none of these DBSs would let for any DELETE From S-P2.SP..., unfortunately. Strangely, MsAccess would however process some of these deletes if they come through its graphical interface. Finally, an INSERT SP... could be valid for MySQL but would be always invalid for MsAccess, etc.

SIR-layer should be implemented in some host language, obviously calling the Embedded SQL interface of the kernel. This is a future work. In the meantime, [L6] backs up Example 7 with a manual simulation on MsAccess as the kernel. For each simulated SIR, a stored MsAccess table is its base. The MsAccess stored queries simulate all the views the BPS would create. The result lets an MsAccess user to actually appreciate advantages of SIRs, through queries to the CE-views. One may also alter and update any views, e.g., to experiment with every SIR-layer processing facet above discussed. As an easy bonus, one may experiment the QBE interface, generate forms, graphics, etc. In sum, one may play with all nice capabilities of MsAccess that made it so popular, almost as if they were designed for SIRs as well.

4.3 Operational Overhead of SIR-layer

The kernel storage for a SIR is in practice the one for its base. CE-view storage is negligible provided the view is not materialized, as we suppose. The storage for the kernel meta-tables for the IEs should be obviously negligible with respect to the typical one for the DB data. Next, as shown, the optimal scheme with SIRs has the same size and the same stored attributes as the optimal one for SRV-model. Hence, the storage for the values of these SIRs within the kernel is the same. The optimal DB with SIRs should cost thus negligibly more in storage than the optimal DB with the stored relations only for the same application.

For DDL statements, the processing cost of each, including of CREATE Table using BPS, is clearly negligible. For DML, since the SIR-layer passes every query as is to the kernel, its query evaluation overhead is negligible as well. Altogether, perhaps surprisingly, the enticing capabilities of SIRs appear practically without overhead.
4.4 Enhancing BPS

Examples show that BPS should usually suffice. As usual for a DBS component, one can nevertheless enhance it. Below, we discuss a few possibilities. Some aim on kernels with specific capabilities only. Whether any enhancements actually prove practical enough to get implemented, is anyone’s guess at present. We leave the exhaustive investigation of any BPS enhancements for the future work.

One goal for BPS enhancements may be fewer views \( R_i \). The rationale is potential speed-up of queries through fewer view mappings. An ultimate rule is to make enhanced BPS imbricating the SIR base and all the successive views as relations within a single From clause of view \( V \). A potential drawback is that a DBS does not support several such imbrications in a single statement, although it supports the processing of the same views separately stated. E.g., we are not aware of any DBS that would support such a single view with our STATUS and RANK. Likely, since the latter refers to the former, while the former is being defined within a different subquery imbricated within the same FROM clause. As for (I-SP) above, one may simplify sometime such imbrications further to more usual half-joins on relation names only. A partial easy rule is also to avoid generating the CE-view \( R_n \), if view \( R_{n-1} \) is already an equivalent one. Instead, enhanced BPS can generate the CE-view directly from \( R_{n-2} \). It suffices to take into the account the attribute order in the SIR as well. In Example 7.2, SIR-layer would then generate view \( P \) directly from \( P_1 \), gaining one mapping.

Another rule aiming at the same purpose is to collapse selected IEs defined in the secondary notation only into a single view. In Example 7.2 this could spare \( P_2 \). With both rules, view SP would even replace \( P_1 \). The kernel would end up with two view mappings less than for BPS. The kernel supporting VAs could even replace CE-view \( P \) that BPS generates for S-P3 at Figure 6 with \( P \) with \( S \)As and two VAs. Notice, however that for Example 7.1 in turn, the discussed rules would complicate BPS, but would gain nothing. The complication would even naturally little slow down the SIR-layer.

Yet another issue is that of the circular references. Examples show enhancements to BPS avoiding these in the DBS, despite having some in the SIR DB scheme. They point to rewriting rules of the affected SIRs or views during the actual scheme generation. The goal here is the DBA comfort, not the usual query execution speed-up. E.g., one could let in this way the DBA of S-P3, to declare STATUS referring simply to SP as in S-P2, Example 1.3, despite the induced circular reference between \( S \) and SP. The enhanced BPS could automatically rewrite it into the IE referring to SP_B, as discussed in Example 1.4.

5. Conclusion

Stored and inherited relation, (SIR), appears a useful construct for a relational DB. Through its inherited attributes (IAs), a SIR may be conceptually more accurate than a stored (only) relation (SR) with the same stored attributes, while IAs cannot introduce any anomalies. SIRs alleviate in this way the well-known limitation of SRs, the dark side of the normalization. The popular ER model, proposed precisely because of this limitation, appears useless.

The operational gain of a SIR DB with respect to the SRV DB with the same SRs only is typically less-procedural SQL queries. These may have no or reduced logical navigation and be free of selected value expressions. The procedural cost for the SIR DB scheme with respect to the SRV DB consists of the IEs. The SRV DB scheme may also provide for the same simpler queries at the cost of the auxiliary views. The SQL statements creating these however should be always more procedural, than these for the equivalent IEs. Likewise the maintenance of the former may be more procedural and the source for a potential trouble.

With respect to the existing helpers with less procedural queries without the views provided by popular DBSs, a SIR DB may seamlessly integrate VAs. Every VA scheme is indeed an IE. Likewise, IEs may generate some implicit joins, making that tool useless for those. The capabilities of the tool beyond these of the IEs appear easily adaptable to SIRs as well. The clients apparently found useful
these helpers. Otherwise they would not be proposed for decades. SIRs should therefore appear useful to the clients as well.

The SIR-layer appears as a higher level interface to SQL DBs. Implementing it over a popular DBS looks easy and without operational overhead in practice. The future work should start with such an implementation. Depending on the kernel’s actual capabilities, it may be wise to include some of the enhancements to BPS we have discussed. But even without these, the result should be the win-win deal. Better sooner than later the existing DBSs should provide thus for SIRs.

On the theoretical side, the design rules for SIRs based on restated NFs and Heath’s and Fagin’s theorems appear about as easy as the current ones. However, the decompositions based on these two theorems exclusively, are only the tip of the iceberg of known proposals, [D12]. Future work could adapt those proposals to SIRs as well, especially the proposals for the lossless decomposition using outer joins, [JS90].

One may also observe further that all the three constructs, i.e., SRs, IRs and SIRs root in a common 1NF construct. One could call it relation with stored or inherited attributes, or stored or inherited relation, say (SoIR) in short. As the name hints, a SoIR may be a stored one, or a view or an SIR. The construct may look esoteric. Observe however that our Create Table for an SIR is in fact that of 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 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, outlined already in [LKR92], is of attributes possibly mixing stored and inherited values. A stored value of such an attribute whenever present, overrides the eventually inherited value. This may be obviously practical. For instance if color of a part in P is green, while an S-P2 user getting a supply involving the part in SP rather sees it as light green, for any reasons, then the update overriding the color in the related SP tuple could bring the global happiness. Future work could explore that issue as well.

Finally, most of major DBSs are now interoperable, [LA86]. SIRs with multidatabase IEs seem attractive as well.

References


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


[F11] The rise, fall, and rise of dependency theory: Part 2, the rise from the ashes. Presentation. Sigmod/Pods 2011


symposium/talks/mendelzon.pdf.


[P] Postgres SQL. https://www.postgresql.org/
