Stored and Inherited Relations for SQL Databases

Witold Litwin¹

(March 2016)²

Abstract. A stored and inherited relation (SIR) is a stored relation (SR) with additional inherited attributes (IAs), calculated like in a view. SIRs can make common queries less procedural than to the SRs only, without impacting the normal forms. A query may become partly or fully free of logical navigation or of selected value expressions. A dedicated view can provide for the same queries. However, we extend SQL to SIRs so that adding IAs to Create Table can be always less procedural than Create View for any such view. View maintenance may be also more procedural. Finally, our extensions are backward compatible with and generalize the popular tools also avoiding some value expressions to queries through additional attributes in Create Table, being similarly less procedural than Create View. We motivate our proposals through the biblical Supplier-Part DB. We extend the Normal Forms, the Heath’s and Fagin’s theorems to SIRs. Extended Heath’s theorem produces lossless and logical navigation free decomposition. Extended Fagin’s theorem decomposition is lossless as well, but logical navigation over projections remains occasionally necessary. We show how to implement SIRs with negligible operational overhead. We postulate SIRs and our extensions as standard on every SQL DBS 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 only values calculated on-the-fly 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. No one followed however, to the best of our knowledge.

Below, we refine our proposal, especially for SQL DBs. We call our construct Stored and Inherited Relation, (SIR), Figure 1. For every SIR R, we suppose every stored attribute (SA) of R defined as usual. For every tuple of R, the formula we call inheritance expression (IE) defines furthermore every IA value. Accordingly, it completes every stored sub-tuple of R, towards the entire R-tuple. Sometimes, IE may find no value to inherit. In every such case, IE preserves the sub-tuple and completes it with a null instead.

We refer to Codd’s relational model and DB, i.e., with two constructs only, as to Stored Relation or View (relational) model, (SRV-model) and SRV DB. We believe the reader familiar with the SRV-model and SQL in particular. We recall nevertheless that for every SRV DB, its conceptual scheme consists exclusively of some SRs. These are possibly as few as possible without normalization anomalies. Relational views, inheriting from the conceptual scheme or other views present then this scheme differently to different clients. We qualify of SIR-model and of SIR DB the relational model and a DB supporting SIRs. The conceptual scheme of a SIR DB may consist of SRs or SIRs. A view may inherit also from SIRs.

We show that a SIR may have the conceptual scheme more faithful to the reality than the SR it expands. The rationale is that the IAs may model conceptual properties inconvenient as SAs. The latter could adversely impact the normal form (NF) of the SR or lead to impractically frequent updates. By addressing SAs and IAs in the same SIR, an SQL query may totally or partly avoid the

¹Université Paris-Dauphine, PSL Research University, CNRS, LAMSAD, 75016 Paris, France.
witold.litwin@dauphine.fr
²LAMSAD Research E-Report. Latest update June 18, 2018
logical navigation, necessary within every equivalent query to the SRs only. We recall that such navigation in an SQL query occurs when the query has to refer to several relations with, usually, equijoins among those. Likewise, a query to a SIR may avoid selected value expressions. Altogether, SQL queries should end up usually less procedural (simpler, more usable...) than their equivalents to SRs only, by the basic measure of the number of characters to type-in.

On the other hand, it is easy to see that for every SIR R, there is always at least one view R in SRV-model, defining mathematically the same SQL relation, i.e., where the attribute order matters, not only the value set. We recall that this order is the ascending one of column IDs in a DBS meta-table such as Sys.columns in SQL Server. That order is initiated by the top-down or left-to-right order of the attributes in Create Table or in Select clause of Create View statement. View R may also have, for every attribute, the same proper name as in SIR. View R is then equivalent in this sense to SIR R, differing only in implementation. For every stored value in SIR R, view R provides indeed the same value, but calculated through inheritance from some relation. View R provides for the same simpler queries as SIR R. Actually, since decades, such views are notorious “escape route” for clients unhappy with the procedurality of most queries to the normalized SRs only. Universal views, providing all the attributes and, possibly, all the values of the DB in one relation, were particularly studied, [MUV84].

We propose clauses declaring an IE in Create Table. Likewise, we propose extensions to Alter Table altering an IE. We show that our clauses are so that, for every SIR R, our declaration of the IE within Create Table R can be less procedural than any equivalent Create View R. Every SA in our Create Table R remains also declared as usual, we recall. SIR R expanding some SR R_B may thus provide for simpler queries to R_B at lower procedural data definition cost than every equivalent view R. It will appear also that View R may be more procedural to alter than the IE, while no view can ever be less procedural to alter. We show finally how to implement SIRs on a popular DBS with negligible storage and processing overhead. The need for faithful conceptual modelling and non-procedural queries is universal. We postulate SIRs therefore as new standard capability of every relational DBS.

We do it especially since some popular DBSs provide in fact already for limited SIRs for decades. Unknowingly of course, as it will appear. These are SRs possibly carrying also so-called virtual attributes (VAs) or computed, generated... columns. We recall that one defines every VA as a named value expression in Create Table through a dedicated clause. Queries avoid the expression through simple reference to VA name. The advantage of the whole capability is that whatever is the number of VAs in Create Table, their declarations are altogether always less procedural than any Create View otherwise needed. The advantage extends to all the other SQL DDL statements concerned by VAs. Our clauses for SQL aim precisely at the same gain. But we generalized it to every SIR. The IE can consequently support value expressions not possible as VAs at present, e.g., with aggregate functions. For an IA that could be a VA, the IE may also avoid the logical navigation, or reduce it at least. Our clauses are in particular designed for backward compatibility with current ones for VAs, abstraction made of minor syntactical differences between current SQL dialects.

Next section discusses the basic concepts of SIR-model. It also introduces our clauses for an IE in Create Table. We then motivate our proposal by applying SIR-model to the “holy” for SRV-model Supplier-Parts DB. We finally complete the introduction of our SQL extensions; discuss the utility of SIR-model and the related work.

Section 3 extends the basic relational design rules to SIRs. We generalize the NFs other than 1NF. We restate Heath’s and Fagin’s theorems, [H71], [F77]. The restated theorems decompose SRs or SIRs. They create lossless or presumed to be so decompositions into two projections at each step, as do their originals. For an SR, our both projections have the original SAs, but at least one is a SIR. For the restated Heath’s theorem, the latter projection has all the attributes of the former as IAs. A query to SAs and IAs there is free of the logical navigation over the original projections.

The property extends to the decomposition of SIRs as well. The end result is that, if the conceptual scheme of a SIR DB results from successive restated Heath’s decompositions only, it is logical
navigation free. One projection must be the universal SIR, with all the attributes in the DB and all the data values. As for a universal view over the projections at present, this freedom, however, lasts only until a dangling tuple appears in a projection. The decomposition ceases to be lossless then. A query selecting a dangling tuple and a tuple elsewhere may require the logical navigation. Even so, SIRs may reduce the latter with respect to the original one, as we will show.

In contrast, for the restated Fagin’s decomposition, the need of occasional navigation over the projections always remains. Altogether, it will appear that SIR DB produced by the restated theorems should always have the same SRs as these produced by the original theorems. But, some SRs would become parts of SIRs. The new design should typically bring then 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

As Figure 1 illustrates, every SIR is a 1NF relation, i.e., a finite subset of a Cartesian product of atomic attributes (columns) over some domains, subject to every algebraic or predicative operation and aggregate or scalar function applying to 1NF relations. As usual for SRV-model, we also basically consider an IA value immaterial. Likewise, every SIR has a name and scheme defining all its SAs and IAs. Every SA scheme is usual as well, e.g., as defined by an SQL dialect. The already mentioned IE defines all the IAs of a SIR. Every IA inherits from SRs, views or SIRs. The IE selects IAs and calculates tuples of IA values through some relational or value expression like in a view. The expression must be so that every IA is functionally dependent on the key of the SR constituted from all and only SAs of the SIR.

Figure 2 illustrates the resulting possible structure of a SIR as an SQL relation, i.e., where the attribute order matters. Each grey rectangle represents a stored sub-tuple. For every SIR R, all these sub-tuples form a stored sub-relation we qualify of base of R. The base has its proper default name. We use R_B below, but presume actual DBAs choosing perhaps different defaults, e.g., R_simply. The green and white labelled Null rectangles represent the sub-tuples forming altogether the SQL view constituted from all and only IAs. We recall that unlike a relational algebra view, an SQL one may contain duplicates and null tuples. We name this view R_V. For every stored (base) tuple, R_V defines one and only one inherited tuple. Together with R_B-tuple this sub-tuple forms R-tuple. Each green rectangle represents a valued R_V sub-tuple. Some of these sub-tuples can be duplicates. Alternatively, an R_V sub-tuple may have no inherited values, i.e., can be a null sub-tuple. Each line of Null rectangles at the figure symbolizes a null sub-tuple.
Furthermore, for every SIR R, we have every SA of R with proper name A belonging to both R and R_B. As usual for SQL naming conventions, we consider that A can have then as full source name R.A or R_B. For reasons we discuss soon, the choice of R_B.A should be usually always preferable. Next, we qualify of conceptually equal to R, CE-view R in short, the already discussed view R, where instead of SIR R, there is some SR named B sourcing every IA with proper name A in B and in view R, and such that A is also an SA A in SIR R. CE-view R defines thus relation R formally equal to SIR R with, in addition, the same full source name B.A in R for every IA A in CE-view R that is an SA in SIR R. Every SIR R has therefore the same conceptual properties as its CE-view R. It also provides for the same queries. These properties actually motivated our name choice for such a view.

The former property does not characterize every view R providing the latter. More precisely for every SIR R there is CE-view R. Under some restrictive conditions on the DB, for some SIR R may nevertheless also view R that is formally equal to CE-view R, except that every IA A inherited “as is” by CE-view R from B, is inherited also as A by view R with the same values, but from another relation, say X. View R provides then for the same simpler queries, but is conceptually different of SIR R in the sense that some attribute with full source name B.A in CE-view, has X.A as such name in view R. We call every such view R query equivalent to SIR R or QE-view R in short. The utility of QE-view R may be that it provides for Create View R substantially less procedural than for CE-view R. We provide an example of a QE-view illustrating this point later.

Finally, observe that for every view X to qualify as CE-view X of some SIR X, view X has to respect all the properties of CE-view above discussed. Thus, there should be some SR that one can (re)name as Y, from which view X inherits every tuple t as a sub-tuple t’ with the same values and the same proper name of each IA as that of its source SA. Next, the primary key of Y has to be a key of view X. Therefore, every other IA A of view X should be functionally depended on that key, being perhaps null in the tuple with no value of A inherited. One can use view X then as the basis for creating SIR X scheme. For that one, on the one hand, one has to (re)materialize every IA of t’ to SA of t. These SAs form then the base Y, i.e., X_B by default, of SIR X. Next, one has to (re)consider references to the original SR Y in view X scheme, as those addressing the base Y instead. On the other hand, one has to retain the part of view X scheme defining IAs remaining not materialized as the basic scheme for the IE. If view X is an SQL view, i.e., is created with Create View X, the order of the attributes of SIR X IAS is finally that of all the IAs in the Select clause of view X.

SIR X replaces in this way the couple: SR Y and CE-view X. As it will appear, if CE-view X is an SQL one, then the IE is always less procedural than Create View X. Hence SIR X is also always less procedural to define than the couple.

2.2 Creating SIRs.

We manipulate every SIR DB scheme as usual through some DDL (Data Definition Language). As we hinted to already, operationally, we consider an SQL-like DDL. We suppose the statements of this DDL extending some existing SQL dialect with clauses for IEs. We call that dialect the kernel for the resulting SIR-model dialect. We presume the kernel to be the dialect of some popular DBS.

Accordingly, we define every SIR through extended Create Table R statement. For every SR R, the extended Create Table R is simply the present one of the kernel. For every SIR R in turn, one defines every SA as one would do in the kernel’s Create Table R_B statement. For the IE, according to what we just outlined, one basically defines it as one would define all and only IAs of SIR R within CE-view R using the kernel’s Create View R statement. In the nutshell, the latter consists thus of (i) every element in Select list in Create View R defining an IA of SIR R. As already illustrated at the figure, these elements may be dispersed among SAs of SIR R in Create Table R. Observe that it is also so for already mentioned VAs in present Create Table R on a DBS with that capability.

Next, we suppose the definitions of all the IA and SA in Create Table R (ii) followed by every subsequent clause in Create View R, i.e. From clause, perhaps Where etc. Finally, any relevant
clauses of kernel’s Create Table R_B applying to several SAs at once or to entire table, e.g., a multi-attribute primary key or index definition, or table partitioning specifications, may follow the former clauses.

Furthermore, like SAs of some SR R defined presently with Create Table R statement, we presume SAs and IAs in every SIR R, to initially be ordered according to the usual SQL attribute order in Create Table R. This order can get altered through Alter Table extended to SIRs. We extend to SIRs of course the other SQL DDL statements as well as we show soon. Whether the initial order is altered or not, we consider that DBS maintains the current one somehow, depending on the implementation. Accordingly, as usual, for every SIR R, the query Select * From R or the element R.* in a query should provide all the attributes of R in that order. We say “should”, as some popular DBSs apparently do not guarantee it already for an SR, e.g., SQL Server.

Furthermore, we consider that notorious SQL naming rules apply to SAs and IAs in SIRs. The additional rule, we already hinted to, is that for every SIR R, every SA A of R can be qualified not only as R.A, but also as R_B.A. The latter name is even the default full source name. The rationale is the use of CE-view to define the IE. The former may indeed use R_B as prefix. The rule may further avoid the so-called circular referencing among SIRs. We address that issue later.

Finally, for every SIR R, we call the IE defined as the above explicit. We denote an explicit IE as E and may index it with the name of “its” SIR, e.g., E_R. Some IEs may use a shorter representation that we call implicit. We denote every latter IE as I or I_R. We address the implicit IEs later.

Example 1. Consider the ‘biblical’ Supplier-Part DB, modelling some suppliers, parts and supplies under SRV-model. A supply contains some quantity of a part shipped by some supplier. A supplier may supply nothing for the time being. Likewise, a part may be not supplied. This DB motivated the original proposal of the relational model, [C69], [C70]. Variants settled the relational (conceptual schema) design rules of SRV-model, based on NFs as known. Through those rules, Supplier-Part DB 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. We refer to it as S-P1. We restate S-P1 into variants with SIRs. We will call these S-P2, S-P3...

S-P1 has three notorious relations: S (S#, SNAME, STATUS, CITY), P (P#, PNAME, COLOR, WEIGHT, CITY), SP (S#, P#, QTY). Figure 4 shows sample data types for these relations. S-P1.SP types are these of SAs SP.S#, SP.P# and of SP.QTY of SIR SP there. The definitions of SAs are self-explaining. We underline the primary key attributes, as usual. Actually, the figure shows our basic SIR DB that we call S-P2. SP in S-P2 is no more an SR, but a SIR. The generic ‘#’ denotes its IE. We explain why soon.

Figure 4 shows the original sample values among all these of SIR SP there. The values of S-P1.SP are accordingly to the attribute names. For the relational algebra, considered by the original S-P1 proposal, the order of attributes in a relation, hence the one at the figures, does not matter. It does however for every SQL query with ‘*’, e.g., for Select * From SP. The S-P1 scheme is optimal for the relational DB. The criterion is the minimal number of relations free of storage and update anomalies.

The well-known drawback is that most of practical Select queries to SP also need values from S or P. E.g., a client searching for a supply rarely does not select the supplier or part name(s). These queries have to logically navigate over SP and S or P through inter-relational joins SP.S# = S.S# or SP.P# = P.P#. It is notorious that clients usually dislike the logical navigation, at least as making the query more procedural than most clients feel it should be, [MUV84]. The well-known practical “escape route” is to add to S-P1 a view, named view SP, providing the image of SP with every tuple preserved bijectively and expanded with every matching value of every attribute of S and of P or with nulls otherwise. For any view SP, SR SP has to be first renamed, say to SP_B by default. The rationale is that every relation in a DB must have a different name. Then, likely the least procedural general view SP declaration in SQL is:
(1) Create View SP As (Select SP_B.*, SNAME, S.CITY, STATUS, PNAME, COLOR, WEIGHT, P.CITY From (SP_B Left Join S On SP_B.S# = S.S#) Left Join P On SP_B.P# = P.P#);

Indeed, for each SP-B tuple, view SP has one and only one tuple with the same values of the attributes with the same full source names. Through the joins, each of these tuples expands these values with those in a single tuple in S and a single tuple in P. The joins select a single tuple in S since S# is the key of S. Likewise; they select a single tuple of P. There are no referential integrity requirements on S-P1. Hence SP_B could have a tuple where S# value or P# value does not match any values in S or P. Then, null values complete such an SP_B tuple in view SP. It is indeed the left outer join that defines SP in (1).

Unlike for the original SR SP, the SQL formulation of a typical query to SP, such as name of the supplier, quantity supplied and name of the part for every supply with supplier Id ‘S1’, does not need the logical navigation anymore. The query becomes notably less procedural, as one may easily verify.

Under SIR-model, first one may figure out whether view SP can qualify as CE-view SP. This is indeed the case. First, SP inherits bijectively every tuple of SP_B as a sub-tuple. Then, (SP.S#, SP.P#) is a key of SP, like (SP_B.S#, SP_B.P#) is for SP_B. The reason is that S# and P# are also the keys of S and of P, respectively. Then for instance, for the first tuple of SP_B at Figure 4, i.e., with SAs S# = S1 and P# = P1, the join clauses match only one source tuple in S and P.P# = P1 matches only one in P. Only a single tuple in view SP results from that is the first at the figure. Similarly for SAs S# = S1 and P# = P2 etc. Therefore, the following single Create Table SP can do, instead of the couple:

(2) Create Table SP (S# Char, P# Char, Qty Int, SNAME, S.CITY, STATUS, PNAME, COLOR, WEIGHT, P.CITY From (SP_B Left Join S On SP_B.S# = S.S#) Left Join P On SP_B.P# = P.P#);

At Figure 4, the content of SIR SP as defined by (2) shows the SAs in plain text and the IAs in Italics. SA schemes are the same as in S_P1.SP, hence as they would be in Create Table SP_B for (1). These SAs and their tuples form the base SP_B. The (underlined) key of SIR SP is also the same according to our general assumptions on SIRs, although we omitted its definition in (2). The IE within Create Table SP, hence $E_{SP}$, is the string: $E_{SP}$ = ‘SNAME…P.P#’. It is only a part of Create View SP, hence strictly less procedural than the latter. One may easily observe that this property generalizes to every SIR R with respect to CE-view R defining the IE. For SIR SP especially, one saves the string ‘Create View SP As (Select SP_B.*,’ View SP scheme is thus about 25% more procedural than $E_{SP}$. Actually, Figure 3 shows an even much less procedural equivalent expression for $E_{SP}$ that we justify soon.

In both (1) and (2), the already recalled SQL ordering makes all the SAs preceding all the IAs. It is a subjective choice, minimizing, in SQL, the procedurality of view SP, through the use of SP_B.*. The query Select * From SP would output the SAs and the IAs in the order at Figure 3 and Figure 4. Note that many consider ‘*’ less safe for Create View than the list of attributes it represents. This would make the difference of procedurality with SIR SB even greater. In fact, SAs and IAs in SIR SP could intermix arbitrarily, implying the same order of the IAs in CE-view SP. The list of IAs in $E_{SP}$ would then
consist of non-contiguous strings, followed by the same From clause as in (2). Observe finally that in (1), any prefix SP_B, in joins there, refers to SR SP that is one of the source relations of view SP. In (2) in contrast, as already stated for a SIR in general, it refers to a part of SP itself. We qualify below every latter join of recursive. Actually, a recursive join may also be a $\theta$-join, as we will show. Recursive joins are basically not permitted for an SQL view, we recall. $E_{SP}$ example suggests them in contrast typical for an $E_R$.

Finally, observe that, in principle, view SP defines only an image of the conceptual scheme (CS) of SP1, consisting of the three SRs. Presenting an image of the CS targeting specific clients is the role of any view, already in ANSI-SPARC Reference DB architecture, we recall. SIR SP in contrast is intended to replace SP-1.SP in the DB scheme. The graphic at Figure 3 illustrates the difference for our example DBs.

The graphic shows how the original S-P1 conceptual scheme relates to S-P2 conceptual scheme with SIR SP defined in the upper part of the figure and how both relate to S-P1 scheme with CE-view SP as an external scheme. The colors symbolize SAs and IAs as in Figure 2. One may observe that view SP appears redundant in part with respect to the SR SP. It is indeed, as, by definition, it redefines all the (stored) attributes of SP_B as IAs. The redefinition must always cost some procedurality, necessarily adversely affecting CE-view SP compared to SIR SP. The example illustrated this point in detail, as did also other properties of SIRs we hinted to.

We recall furthermore that SP was criticized as too poor or, in other words, unfaithful model of a supply almost immediately after S-P1 proposal. E.g., in reality, at least supplier and part names characterize every supply, not only the related IDs and the quantity supplied. Adding the former however as SRs to S-P1.SP, would introduce the anomalies. This would be against CS design rules of SRV-model. The issue led, in particular, to the popular ER model as a tool for a more faithful CS definition, we recall, [C76].

<table>
<thead>
<tr>
<th>S-P2 Scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Table S</strong></td>
</tr>
<tr>
<td>S# Char, SNAME Char, STATUS Char, CITY Char;</td>
</tr>
<tr>
<td>/* Every attribute from S and P, except for */ S# and P#.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S-P1 DB</th>
<th>S-P2 DB</th>
<th>S-P1 DB with CE-View SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>SR</td>
<td>S</td>
<td>SR</td>
</tr>
<tr>
<td>SR</td>
<td>SP</td>
<td>SIR SP</td>
</tr>
<tr>
<td>SR</td>
<td>P</td>
<td>SR</td>
</tr>
</tbody>
</table>

Figure 3  S-P1 DB and S-P2 DB schemes.

In contrast to view SP, SIR SP is intended to be an element of the CS, replacing S-P1.SP. As a conceptual model, it states that every supply inherits every property of its supplier and of the part supplied. The obvious practical rationale is the avoidance of the logical navigation to possibly the largest number of queries to SP. In the same time, all the attributes of SIR SP that are not these of SR SP do not create any anomalies to SP-B. A storage anomaly can indeed only concern an SA. An update anomaly does not concern any IA in SP, under the usual view update propagation rules for CE-view SP. E.g., if one modifies SNAME to Dupont for the tuple (S1, P1), the new value propagates to
S.SNAME. It then virtually appears back in SP for every supply by supplier S1. SIR SP constitutes in this way the most faithful or the richest, in other terms, conceptual model of a supply, possible for a 1NF relation in this case. ER model seems less useful or perhaps even useless then for a SIR DB than it could be for the SRV DB with the SRs the same as the bases of the SIRs.

2.3 Implicit IEs

It is sometimes possible in Create Table R to declare an implicit IE $I_R$ instead of $E_R$. We denote then the replaced $E_R$ as $E'_R$. $I_R$ is less procedural than $E'_R$ and any equivalent $E_R$ (defining the same SIR R). $I_R$ may also provide the backward compatibility of an IA resulting from specific VE with the industrial notation for the aforementioned virtual attributes (VAs), [LV86]. With slight syntactical differences, some, but not all, popular DBSs provide for VAs, also named computed or generated columns, we recall. E.g., MySql does it, with parentheses around every VE. SQL Server provides for VAs as well, but without parentheses around VEs. In contrast, MS Access does not provides for VAs at all. If for some $E_R$, every IA could be a VA, then every such $E_R$ would be more procedural than the current notation of these VAs. In contrast, for every such $E_R$, there is a backward compatible $I_R$ as we define it below. The compatibility means here that $I_R$ can be identical to the present notation, with or without parentheses, at wish.

<table>
<thead>
<tr>
<th>S-P2 Content</th>
</tr>
</thead>
</table>

**Table S**

<table>
<thead>
<tr>
<th>S#</th>
<th>SNAME</th>
<th>STATUS</th>
<th>CITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>Smith</td>
<td>20</td>
<td>London</td>
</tr>
<tr>
<td>S2</td>
<td>Jones</td>
<td>10</td>
<td>Paris</td>
</tr>
<tr>
<td>S3</td>
<td>Blake</td>
<td>30</td>
<td>Paris</td>
</tr>
<tr>
<td>S4</td>
<td>Clark</td>
<td>20</td>
<td>London</td>
</tr>
<tr>
<td>S5</td>
<td>Adams</td>
<td>30</td>
<td>Athens</td>
</tr>
</tbody>
</table>

**Table P**

<table>
<thead>
<tr>
<th>P#</th>
<th>PNAME</th>
<th>COLOR</th>
<th>WEIGHT</th>
<th>CITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>Nut</td>
<td>Red</td>
<td>12</td>
<td>London</td>
</tr>
<tr>
<td>P2</td>
<td>Bolt</td>
<td>Green</td>
<td>17</td>
<td>Paris</td>
</tr>
<tr>
<td>P3</td>
<td>Screw</td>
<td>Blue</td>
<td>17</td>
<td>Oslo</td>
</tr>
<tr>
<td>P4</td>
<td>Screw</td>
<td>Red</td>
<td>14</td>
<td>London</td>
</tr>
<tr>
<td>P5</td>
<td>Cam</td>
<td>Blue</td>
<td>12</td>
<td>Paris</td>
</tr>
<tr>
<td>P6</td>
<td>Cog</td>
<td>Red</td>
<td>19</td>
<td>London</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>S.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 reduced procedurality of an $I_R$ usually results from implicit From clause, together with every optional usual follow up clause. The specific generic character we denote as ‘#’ and define below, allowed within or even instead of A1,...,An. An list of IAs in $E_R$ may also reduce the procedurality, whether From clause is present or not. Likewise, an IA A defined through a VE with a subquery or with Group By clause in $E_R$ may have a shorten definition in $I_R$. That one may be as simple as: ‘VE As A’ or ‘A As VE’. All these capabilities apply only iff $R_B$ and all the other source relations of $E_R$ fulfill specific conditions. The DBS rewrites then $I_R$ to specific $E_R$ that we denote as $E'_R$. It does it as the preprocessing of Create Table R or Alter Table R processing we specify later for every SIR R, supposed defined through some $E_R$ for this processing. The rewrite rules appear open-ended. We define now only some for $I_R$ capabilities we just announced. They appear sufficient for most applications. We
leave the study of eventual additional rules for the future. If some rules do not apply to \( R \), then the IE must be an \( E_R \).

Rule 1. Every IA in \( R \) inherits only \textit{locally} that is from \( R_B \) only, through a VE without an aggregate function or a subquery. Then, the implicit From clause in \( I_R \), i.e., the one of \( E_R \), is simply: From \( R \). No other clauses may follow that one in \( E_R \). The list \( A_1 \ldots A_n \) is the same for \( I_R \) and \( E_R \), unless modified through Rule 5 below. The list contains thus, in order, all the IAs and SAs defined by Created Table \( R \) and every Alter Table \( R \) prior to \( E_R \) calculation.

Rule 1 is quite obvious for SQL fans. The following example illustrates it.

Ex. 2. Suppose that \( P \) should also provide for every part its weight in KG after the original weight presumed in pounds. The following declaration could do, given Rule 1:

Create Table \( P \) (\( P# \) Char, \( PNAME \) Char, \( COLOR \) Char, \( WEIGHT \) Real, \( \text{Round} \) (\( WEIGHT \) * \( 0.454,3 \)) AS \( WEIGHT_{\text{KG}} \), \( CITY \) Char);

\( S-P1.P \) became thus \( SIR P \) with an implicit IE:

\[ I_P = \text{Round} \ (\text{WEIGHT} * 0.454,3) \text{ AS WEIGHT}_{\text{KG}} \]

Rule 1 would generate \( E_P \) as follows:

\[ E_P = \text{Round} \ (\text{WEIGHT} * 0.454,3) \text{ AS WEIGHT}_{\text{KG}} \text{ From } P; \]

In Create Table \( P \), From clause would not actually follow \( WEIGHT_{\text{KG}} \), but \( CITY \) Char term. Notice finally that \( E_P \) would also be less procedural than Create View \( P \) for \( WEIGHT_{\text{KG}} \).

The rule that follows avoids the logical navigation to \( I_R \).

Rule 2. Only some IAs, at most, inherit according to Rule 1. Every other IA \( A \) inherits from IAs in source relations other than \( R_B \), aliased \( R_B \) included. Let \( X_1,\ldots,X_m \) be these relations. Suppose also that every \( A \) either inherits its value as is from some source attribute \( X \) or \( A \) results from VE as in rule 1, but involving one \( X \) or more and, perhaps, some SAs in \( R_B \). Also, for every \( I = 1\ldots m \), let \( X_i.K_i \) be the primary key of \( X_i \), perhaps composed as: \( X_1.1, X_1.2 \ldots \). Suppose furthermore that for every \( X_i.K_i \), there is some \( R_B.K_i \) as well. The latter attributes are not necessarily disjoint. Then, IE of \( R \) may have the implicit From clause:

\[ \text{(3) From } R_B \text{ Left Join } X_1 \text{ On } R_B.K_1\_1 = X_1.K_1\_1 \text{ And } R_B.K_1\_2 = X_1.K_1\_2 \ldots \text{ And Left Join } X_2 \text{ On } R_B.K_2\_1 = X_2.K_2\_1 \text{ And } \ldots \text{ Left Join } X_m \text{ On } R_B.K_m\_1 = X_m.K_m\_1 \ldots \; ; \]

The list \( A_1 \ldots A_n \) in \( I_R \) is as for Rule 1 or shorted through Rule 6 below.

Ex. 3. Relations of \( S-P1 \) fulfill conditions of Rule 2. That is why From clause of \( E_{SP} \) in (2) conforms to (3). It thus can be made implicit, providing for \( I_{SP} \)

\[ I_{SP} = \text{SNAME, S.CITY, STATUS, PNAME, COLOR, WEIGHT, P.CITY} \]

\( E_{SP} \) in (2) becomes \( E'_{SP} \) for \( I_{SP} \). The latter is here less than half long, hence notably less procedural. The savings are even greater with respect to Create View \( SP \) of CE-view \( SP \).@\]

\( E_{SP} \) contains recursive left equijoins, as (3) generally. A DBA manually defining equivalent \( E_{SP} \) or \( E_R \) generally, could have different preferences, e.g. towards right joins only or even towards some or all inner equijoins, under some restrictive conditions. Left joins also commute. The DBA could specify in the join on \( P \) in \( E_{SP} \) as first, unlike in \( E'_{SP} \) thus etc. Every such \( E_{SP} \) would remain nevertheless about as procedural as \( E'_{SP} \) and less procedural than “its” Create Views \( SP \). The kernel dialect may require furthermore parentheses in (2), imbricated so to impose the left to the right order on joins, e.g., MS Access. For only a single implicit inner equijoin and mono-attribute primary key, MS Access has in fact a completion rule for QBE queries similar to Rule 2. MS Access calls these joins \textit{automatic}. SRV-model does not consider implicit joins, although there were proposals for adding those, [LA86], [LSW91].
The following Rules 3 and 4 avoid the pain of nested subqueries in $E_R$ or in From clause at least.

Rule 3. Only some but not all IAs of $R$ inherit accordingly to rule 2 or even none. Suppose the former case for the time being. Let $A'1, A'2...$ denote these IAs in their order in Create Table. Let then $X'1, X'2...$ denotes the respective source relations. Suppose every other IA, say, $Ai1, Ai2..., numbered again in the order of these attributes in Create Table ; $i1,i2... \in (1,2...n)$ ;. Next, suppose every $Ai$ inheriting respectively through some VE $V1, V2...$, where each $V$ contains an aggregate function not within a subquery, if $VE$ was followed by Group By $VE$. Furthermore, suppose every $Aik ; k = 1...n$ ; inheriting from some $Xik_B$, where every $Xik_B$ has key $Kk$ with the property common to every $X$ in Rule 2. To simplify, suppose that every $K1, K2...$ is atomic, as it should be usually. Suppose finally that arbitrarily for the time being only every $A$ precedes every $A'$ in Create Table. Then, the implicit From clause, as well as the explicit From clause for $E'_R$ thus, results from the following pseudo SQL expression:

(4) From (SELECT $R_B.*$, (Select $V1$ From $Xi1_B$ Where $Xi1_B.K1 = K1$) As $Ai1$ From $Xi1_B$, (Select $V2$ From $Xi2_B$ Where $Xi2_B.K2 = K2$) AS $Ai2$ From $Xi2_B...$), $A'1, A'2...$ FROM $R_B$ Left Join $X'1$ On $R_B.K1_{1} = X'1.K1_{1} And...$);

The final ‘And...’ continues for $X'$ as it continued for $X$ in Rule 2. The whole Left join clause reduces to From $R_B$ only if none of the IAs conforms to Rule 2. Expression (4) complicates in the way easy to see for composed $K1$ or $K2...$. Also, if IAs $A$ and $A'$ intermix, then (4) changes adequately. Next, the $A1,...,An$ list defining the $I_S$ from every $A$ and $A'$ is as for Rule 2. Finally, observe that From clause preserves every tuple of $R_B$ as it should.

Ex. 4. Consider STATUS stored attribute in S‐P2. Imagine that, behind the scene, at one point in time, DBA decides to calculate 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 will have status 1 etc. The status of the supplier not supplying any parts for the time being should be null. Continuing having such STATUS as an SA is clearly not the most practical choice. In S‐P1, likely the most practical approach would be to either calculate STATUS value in every query needing it or (i) to drop STATUS from S and rename it as, say $S_0$, (ii) to Create the view S with $S_0.*$ and STATUS calculated as above. This view would be in fact the CE‐view S of SIR S with STATUS as an IA. For this purpose, since S‐P2 is a SIR DB, the DBA may alter S, using the Alter statement for SIRs we present later. The DBA may then (a) drop STATUS, (b) rename S as $S_B$ since it plans to create SIR S and (c) recreate STATUS as the $I_S$:

$I_S = \text{Round} ( \text{SUM} (\text{QTY}) / 100) \text{ As STATUS}$

STATUS applies an aggregate function. According to Rule 3, $E'_S$ is:

$E'_S = \text{STATUS FROM (SELECT $S_B.*$, \text{Select round ( \text{SUM} (\text{QTY}) / 100) From $SP_B$ WHERE $S_B.S# = [S#]$) AS STATUS FROM $S_B$)}$;

$E'_S$ is more than three times longer than $I_S$ here. The difference in procedurality is thus even greater than for Ex. 3. $I_S$ is consequently also several times less procedural than Create View for CE‐view S, the latter being as always more procedural than $E'_S$ as well. Observe also that $I_S$ almost minimizes the procedurality of any STATUS scheme. It contains only the proper name and the VE. Besides, since the VE employs an aggregate function and refers to attributes beyond S, STATUS cannot be a VA at present. Hence, having S‐P1 on a DBS with VAs would not help to avoid either declaring STATUS in every query needing it or creating and maintaining CE‐view S.

The result for SIR S with calculated STATUS would be values for suppliers S1...S4. STATUS would be null in contrast for S5. Indeed, this supplier does not supply anything at present.

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

(Q3) Select * From S ;
In contrast, if $S$ was a base equal SR, i.e., $S$ without STATUS, the least procedural equivalent query, also serving eventually as the scheme of the CE-view of our SIR $S$ and (partly) as $E_S$ alternative to $E'_S$, 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 than (Q3) by far. We recall also that the non-procedurality of queries like Q3 charmed in its time the DB community to SEQUEL, former name for SQL, we recall. In contrast, one knows well that clients usually dislike Select clause sub-queries, (as well as From clause sub-queries as in $E'_S$). Many have hard time to even figure out the semantics of these. Some could rather equivalently apply the popular Group By instead. The result is then even more procedural.

Cherry on the cake, Q3.1 queries may not work on a popular DBS, e.g., MS Access, if one adds Order By STATUS clause. Creation of CE-view $S$ followed by Q3 with Order By becomes mandatory. At the procedural cost of Create View respectively higher than of each IE above.

Observe also that $E'_S$ refers to SP_B since QTY is an SA. This is the default choice, but we allow every $E_S$ to refer to R instead, a priori, as well, i.e., to SP here. Not for above $E'_S$ however. That choice would create the so-called circular referencing among SIR $S$ and SIR SP. The latter would occur then also among the CE-view $S$ and CE-view SP. In general, circular referencing occurs when view R1 attempts to inherit from view R2, while view R2 inherits from view R1. This would be the case of view $S$ and of view SP here as one can easily figure out. The cycle can be transitive. Every popular DBS we are aware of forbids the circular referencing. We do not permit for circular referencing between SIRs either. Every R_B inherits from nothing. Hence, no referencing to it can reveal circular.

Actually, it may happen for SIR $R$ that an IA is sourced in other IAs in R, e.g., for an IA defined through a value expression (VE). We prescribe any circular referencing among such IAs as well. Again, today, for VEs in views, any circular referencing is similarly prohibited.

Rule 4 below applies to some IEs where Rule 3 cannot. It may avoid actually an even more consequent nesting.

Rule 4. Everything stated in rule 3 holds, except that some Aj1, Aj2… among A1, A2… of $i_R$ inherit each from some Ai1, Ai2… resulting from Rule 3, through a subquery $S_{j1}, S_{j2}$…, each perhaps with an aggregate function and each defining respectively Aj1, Aj2…. Also, let S' be $S$ where name R_1 replaces R in every reference to R. Again, suppose arbitrarily every Aj following every other A in Create Table. Then, the implicit From clause results from the pseudo SQL expression:

(5) FROM (Select $R_1.*$, $S'_{j1}, S'_{j2}$… From (Select $R_B.*, R_B.A'1, R_B.A'2$… (Select V1 From Xi1_B Where Xi1_B.K1 = K1) As Ai1 From Xi1_B, (Select V2 From Xi2_B Where Xi2_B.K2 = K2) AS Ai2 From Xi2_B…), A'1, A'2… FROM $R_B$ Left Join $X'1$ On $R_B.K1_1 = X'1.K1_1$ And…) As $R_1$);

Here, the 1st subquery defines the relation with all the attributes of the relation produced by its own nested subquery, and with every Aj1, Aj2…. The nested subquery defines the relation termed $R_1$ with all the attributes resulting from Rule 3. Also, as for Rule (5), if some Aj intermix with other IAs differently from the above, expression (4) changes adequately. Next, A1,…,An list of $i_R$ is again as for Rule 2. Finally, observe that as for (4), the nesting preserves every tuple of $R_B$, as it should.

Observe that in Rule 4, the level of nesting is two, as in the example that follows. We have not seen a practical need for more levels. Observe also for expression (4), that every subquery realizes, for each Ai, the calculation that could alternatively result from nested From clause with Group By Xi1_B.A… without Having or Order By in $E_R$.

Ex. 5. This example illustrates Rule 4 with an IA of $S$ defined through a subquery in $i_S$. Besides, it shows that a recursive join may be not an equijoin. Consider indeed that DBA declares STATUS as above and an IA named RANK as follows. For each supplier $s$, if STATUS ($s$) is not null, then RANK ($s$) is one plus the number of Suppliers with STATUS higher than that of $s$. Otherwise, RANK ($s$) is null. In
short, RANK inherits from the just defined STATUS. Finally, RANK should follow STATUS in S. The clause of $I_S$ declared in Create Table S right after the above clause for STATUS, fulfils then the requirements:

$I_S$ consists now of both clauses: that defining STATUS and the above one for RANK. IIF is the scalar function, e.g., of MS Access SQL dialect. The clause contains a subquery, but it still defines RANK through a VE, perhaps surprisingly to some. The join within the subquery, is recursive and not an equijoin. The VE is locally-inheriting, i.e., inherits values from S only. Also, subquery refers to S and not to S_B since Status is an IA. Through Rule 4, we now have:

$$E_S = \text{SELECT } S_1.* \text{ FROM } (\text{SELECT } S_B.* \text{ FROM } \text{SP_B WHERE } S_B.S# = S.S# \text{ AS STATUS FROM } S_B) \text{ AS } S_1;$$

$I_S$ is now even less procedural than $E_S$ than for STATUS only. Like STATUS, RANK also cannot be a VA at present at any popular DBS we are aware of. Finally, to get the feeling of utility of SIRs in this case, consider the simple query of obvious interest:

(Q4) Select * From S;

As it was for STATUS, because of the already outlined limitation of SQL, to the best of our knowledge, at least some popular DBSs do not allow for a single equivalent query to the base equivalent relation, i.e., to S (S#, SNAME, CITY). One solution is again to rename S, e.g., to S_B, then to create at least the CE-view S with STATUS only and to define RANK in the query. However, CE-view S integrating STATUS only would not suffice if one attempts to add Order By RANK clause to RANK defined in the query. CE-view of S with STATUS and RANK followed by Q4 with Order By RANK is necessary. Again, the Create View S has to be more procedural than $I_S$.

Rule 5. Every IA A of R inherited through a VE V, has to be declared in $E_R$ as: $V$ As A. An $I_R$ can alternatively declare every such A as: A As V. When $E_R$ is computed then according to every other relevant rule, the latter notation is reversed to the former one.

Ex. 6. $I_P$ from Ex. 2 can be alternatively defined as:

$I_P = \text{WEIGHT}_\text{KG} \text{ AS Round(WEIGHT} * 0.454,3)$

Same for $I_S$ for STATUS and that for STATUS followed by RANK.

Rule 5 allows for, as we call it, secondary form for the declaration in Create Table R or Alter Table R of any IAs under consideration. Unlike the basic form in every $E_R$ rule 5 refers to, the latter is also the one of a VA, abstraction made of dialect-dependent syntactical details, e.g., of the parentheses we spoke about. The convenience of Rule 5 is thus that Create Table R for SIR R with IAs that could be VAs only at some DBS, is then backward compatible with Create Table R there, e.g., at MySQL. In particular thus, as we aforementioned, every table R with VAs at every DBS with that capability is in fact a specific SIR R, limited to the local inheritance only, without being recognized so. Create Table R for such SIR with $I_R$ declared through Rule 1 with Rule 5, but also the one with $I_R$ declared through Rule 1 without Rule 5, are each equally procedural as Create Table R with the VAs instead at present. All are also less procedural than CE-view R would be. Decades ago already, the rationale for tables with VAs was thus visibly the same with respect to the latter feature as ours for SIRs more generally.

Rule 6. Suppose that for every relation X as in Rule 2 above, $E_R$ selects nominally every attribute of X except for X.K. Then, for every such X, $I_R$ may replace this enumeration with the single element X.#. Also, in every A1…An list in $I_R$, # alone in this list stands for X1.#, X2.#,...Xm.#, each as in Rule 2 and as follows. First, that list can be all such relations within explicit From clause of $I_R$. Otherwise, it results from all the SAs in R_B. For every A there, A is Ki as for Rule 2, iff there is exactly one relation X in the
DB with key X.Ki. Every such X enters then From clause in $E_2^R$. If for some A, there are more relations with A as key, then # alone is inapplicable as element of $I_R$.

Ex. 7. $E_{sp}$ in (2) fulfils the conditions for Rule 6. Hence, one may shorten it several times to $I_{sp}$ that is simply:

$$I_{sp} = S.\#, P.\#$$

In fact, Rule 7 allows for shortening $I_{sp}$ even to ‘#’ alone, standing then for all the attributes of S and of P. These are indeed the only relations in the DB fulfilling conditions of Rule 2 given (2). Figure 3 and Figure 4 illustrate the overall result for SP. If there is also a relation, say S_P with past suppliers, in the DB, with S# as key as well, then $I_{sp}$ with ‘#’ alone needs S or S_P in From clause. If From clause of $I_{sp}$ is nevertheless implicit, despite S_P and S, then the list of the attributes of $I_{sp}$ cannot contain ‘#’ only, S.# or S_P.# need to be there instead.

Furthermore, through Ex. 7, S-P2 shows that for every qualifying SIR R, Rule 6 provides for $I_R$ visibly less procedural than every (equivalent) $E_R$. The need for SIRs like SP appears very common in fact, since S-P1 molded most of practical relational DBs.

In particular, for every SIR R with QE-view R substantially less procedural than CE-view R, we hinted to before, an $I_R$ Rule 6 may provide for, may be less procedural than Create View for QE-view R as well. $E_R$ in contrast could reveal more procedural than QE-view R creation, although CE-view R creation would still remain even more procedural, as always. Rule 6 completes in this sense the practical interest of SIRs. It is also necessary for our initial claim, we recall, that for every SIR R, there is always an IE with lesser procedurality than that of every Create View providing for the same queries. Example below illustrates all this for SP-2. One will easily see that the result generalizes to every SIR R with the discussed QE-view R.

Ex. 8. Consider the following variant of S-P2.SP we name SP’. Create Table SP’ orders the attributes of SP’ differently from SP, namely we have SP’ (S#, SNAME, STATUS…, P#, PNAME…, QTY). Suppose also that the referential integrity between SP’ S and P is enforced. The following view SP’ becomes possible:

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

View SP’ is not CE-view of SP’. The full source names of attributes S# and P# are indeed S.S# and P.P#. They need to be SP_B.S# and SP_B.P# in CE-view, thus. Nevertheless, under the referential integrity enforced, every query addressing view SP’ provides the same data and attribute names as when it addresses CE-view SP’. These names are unique proper names or full source names we recall. View SP’ is QE-view SP’ for our DB thus.

CE-view SP’ would need to enumerate all the attributes in Create Table SP’, in the same order. $E_{sp}$ would need to consequently as well. QE-view SP’ does not, profiting from ‘*’. This makes the latter about twice less procedural than CE-view SP’. QE-view SP’ is also clearly the least procedural QE-view for any variant of SIR SP differing only by the attribute order. One may also easily see that QE-view SP’ is also less procedural than any revised $E_{sp}$ by about 1/3. However, Rule 6 applies also to SP’. Observe, besides that one could not define QE-view SP’ with ‘*’ otherwise. Rule 6 provides consequently for $I_{sp} = S.\#, P.\#$. This one is then less procedural than QE-view SP’, ten times less even, more precisely. Such non-procedurality is out of reach for every QE-view SP, obviously.

Observe as we claimed, that the same holds for every SIR R with any number of source relations X1,X2… each with the attributes in the attribute list ordered as above and with the referential integrity with R enforced. Likewise, there is thus QE-view R with X1.*… There is however similarly also $I_R$ with X1.#… Hence, $I_R$ remains less procedural than Create View for QE-view R. Any other QE-view or CE-view would have Create Table even more procedural, as it would need to enumerate some attributes from X1 or X2 etc. That is why we could claim our DDL proposals sufficient for every SIR R, to have an IE less procedural than any Create View R for the same simpler queries.
Rule 7. An IA does not fit the assumptions of rule 1 or of rule 2, or ...of rule 4. Then, there is no \( I^*_R \) for \( R \).

Ex. 9. Suppose that, for any reasons, \( SP.S# \) was named \( SP.S_{Id} \) instead. None of above rules would apply to such \( SP \). No \( I^*_SP \) would exist accordingly.

We recall however that our rules are intended open-ended. Future work may perhaps restrict Rule 7, bringing some \( I^*_SP \) for the Ex. 9 case, restricting Rule 7 consequently.

### 2.4 DDL Statements for SIR-model

We already discussed the extension of Create Table to SIRs. Likewise, we now discuss the extensions to the other SQL DDL statement. We suppose every result backward compatible with some kernel. E.g., we suppose the Create Table able to create also every SR that the kernel would. Likewise, we consider that Create View for SIRs is the kernel’s Create View, except that a source relation can be a a SIR.

We suppose generally the same for all the other usual SQL DDL statements, i.e., Alter Table, Drop Table, Drop View and Create Index. More specifically, for Alter Table, we suppose that each of its kernel dialect clauses for an SA, at least thus: Add Alter and Drop, apply to an IA in general, unless kernel constraints restrict a capability to an SA only. E.g., Indexing on IAs may be disallowed, if the kernel does not provide for view indexing or the VE is non-deterministic. The latter constraint applies already to SRs with VAs that are, we recall, limited SIRs.

Suppose thus, e.g., MySQL as the kernel. Then, an alteration of an SR in a SIR DB or of a SIR, may add an IA, after any specified SA or IA or before the first attribute, or after every existing attribute. The latter is the default and the only choice besides if SQL Server is the kernel. An alteration of a SIR may also modify or drop an existing IA. Likewise, it can rename the SIR or an IA. In the latter case, the MySQL clause: Rename Column old_column_name To new_column_name, changes the IA source name. We allow then also for the additional option: Rename Column column_name As column_alias. This one may also modify an existing alias or may drop it, if column_alias is column_name. For an IA defined through VE, we suppose that the altering towards a new VE may use the basic or the secondary notation. The latter is the case of VAs today. An alteration of/towards a SIR may also apply ‘#’.

An alteration of an IA as above stated may contradict the existing IE. For every such case, regardless of the kernel, we provide for Alter Table the clause specific to SIRs: Refresh SIR_expression. SIR_expression designates the part of Create View that follows the Select verb for the CE-view of the SIR that should result from the alteration. The expression specifies in this way the new IE and the order of the IAs among the existing SAs. The latter should remain unaffected.

Next, as usual for an SR, we prohibit every Alter Table to drop all SAs. For every SIR \( R \), we thus prohibit transforming it into view \( R \). If the need for dropping all SAs occurs, the extended Drop Table simply drops as usual the scheme, the IE included. It also drops the stored tuples. The manipulation cannot of course violate the referential integrity if any, as usual. It may thus, also as usual, trigger a cascade to other SRs or SIRs or the refusal of the statement. For a SIR \( R \) intended to be replaced by some view \( R \), preserving some or all IAs of SIR \( R \), a Create View should end up the job then. Besides, we do not suppose any extensions to kernel’s Drop View and Alter View, if the kernel supports the latter. In turn, if a view \( R \) should evolve to SIR \( R \), we presume basically Drop View \( R \), followed by Create Table \( R \). This procedure is obviously the simplest to put into practice. Finally, we consider that Create Index statement for a SIR reuses the syntax of the kernel, although, as said already, indexing of IAs may reveal more restricted than of SAs in practice.

Ex. 10. 1. DBA adds to S-P1.P the IA WEIGHT_KG from Ex. 2. S/he also adds WEIGHT_T converting WEIGHT_KG further to tons. For application dependent reasons, WEIGHT_T should precede in the scheme WEIGHT_KG. MySQL is the kernel DBS.
Alter Table P Add After WEIGHT WEIGHT_T As WEIGHT_KG / 1000, WEIGHT_KG As Round (WEIGHT * 0.454);

Both IA schemes are in secondary notation and could be VAs. As the result, Alter modifies SR P into SIR P with \( I_P \) constituted from both IA schemes.

2. DBA alters furthermore SA STATUS in S-P1.S to IA STATUS we discussed.

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

Unlike for (1) above, Alter Table of MySQL does not provide such capability, since STATUS cannot be a VA there. The intended result is that SR S becomes SIR S with \( I_S \) from Ex. 4. The stored values of STATUS get dropped. In practice, a warning should precede the actual processing.

Ex. 11. Suppose for S-P2.SP, Figure 3, the statement Alter Table SP Drop Status. This statement would invalidate the existing IE, defined simply as \( I_{SP} = '#' \). Hence, one should use instead:

Alter Table SP Refresh (S#, P#, QTY, SNAME, S.CITY, PNAME, COLOR, WEIGHT, CITY);

The statement defines new \( I_{SP} \) say \( I_{SP}' = 'SNAME, S.CITY, PNAME, COLOR, WEIGHT, CITY' \). It does not alter the SAs.@

Below, we refer to S-P2 DB with P and S altered as discussed above and with RANK, as S-P3 DB. Notice that S-P3 scheme has SIRs only.

We finally illustrate with a few examples our point that SIR R maintenance due to an alteration may be also less or even much less, procedural than for the SRV DB with CE-view R or QE-view R instead.

Ex. 12. Consider indeed first S-P1.SP and the unanticipated need for evolution towards either S-P2.SP or towards CE-view SP, or QE-view SP alternatively. To expand SR SP to our SIR SP, single Alter SP Add # would do. To alternatively create the CE-view SP or even QE-view SP’, one has to first rename SP into SP_B. This costs one statement. Then, one has to formulate the selected Create View. It is easy to see that each choice would be by far more procedural than \( I_{SP} \). It would also more procedural than every equivalent \( E_{SP} \), as we discussed. To avoid any run-time error, the whole procedure must be an atomic transaction. SQL Begin Transaction and Commit brackets are thus necessary. Likewise, are necessary the SQL Error Code tests for the Commit or Rollback after each statement. All this leads to several SQL statements (how many?) and quite a headache, comparatively.

Ex. 13. Similar trouble appears for every SA name change or addition or deletion. E.g., work out the example of shortening QTY to Q, for either S-P2.SP and CE-view SP or for SP’ and QE-view SP’. In practice today, such alterations often create run-time errors. The rationale is that DBA often manages only the conceptual scheme. Views and applications are private to clients. The clients are often not aware in real-time of alterations the DBA performs. In turn, the DBA is often not aware of every view. A run-time error of an application, often costly, may appear weeks or months after the alteration. The debugging is then usually anything but easy.

Ex. 14. Finally, suppose that SP had the referential integrity enforced and that DBA decides to drop it. Suppose also that S-P1 DBA previously created CE-view SP with the joins in the WHERE clause we have shown and that S-P2 DBA used simply \( I_{SP} \). Once there is no more referential integrity, S-P2.SP_B or S-P1.SP may get, e.g., the tuple (S10, P10, 50) with S10 and P10 not in S and P, respectively. To enforce the new requirement, S-P2 DBA has to drop the Foreign Key clauses for SP in Alter Table SP, according to the kernel’s SQL dialect. The S-P1 DBA needs to do the same for SP. In addition however, s/he needs to alter CE-view SP to one with outer joins preserving SP. For QE-view SP, one has to alter it even more extensively, modifying the list of the attributes to that of CE-view SP. Otherwise, for both views, the output of a query could be silently errored. Some queries would silently miss the above tuple, e.g., 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 for S-P1, unlike for S-
P2, even if one adequately alters the CE-view or QE-view, each procedure requires again an atomic transaction with several statements (how many this time?).@ The examples obviously generalize to many if not most of actual DBs, illustrating the attractiveness of this facet of SIRs as well.

2.5 Data Manipulation

2.6 Manipulating SIRs

Every SIR is a 1NF relation, by definition. Every relational algebra operator operates on 1NF relation as defined by the mathematical model, Figure 1. Whether an attribute is a SA or an IA is immaterial to the operator. Every operator applies thus not only to an SR or a view, but to a SIR as well. One may project, select or join thus the SIRs. Likewise, applies to SIRs every SQL Select statement. Operationally, every statement acts on any SIR R as it would act upon CE-view R. This includes every query with VEs like those we have discussed, as well as every query with the special clauses like: Top k, Group By, Order By, Drill Down... Same applies to every update query, i.e., SQL Insert, Update or Delete statements. These statements also act on every SIR R as upon CE-view R, especially with respect to the update feasibility and propagation. In short thus, regardless of the kernel dialect, no current DML statement requires extensions to apply to the SIRs.

Ex. 15. 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, in the same attribute order, but not necessarily in the same tuple order, we recall. Supposing MS Access as kernel for DBS supporting S-P2, would make 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’; should also succeed. The change to CITY would propagate to S. The possibly surprising side-effect would be SP.CITY change for every other supply by S1. The Insert SP (select ‘S4’ as [S#], ‘P4’ as [P#], 100 as QTY, S.CITY as ‘Rome’); would affect S4.CITY everywhere in SP similarly. Next, every update of S-P2.SP.STATUS would succeed, since S.STATUS is an SA. But, every update to S-P3.S.STATUS would fail, STATUS being an IA defined through a VE. Next, for S-P3.P, Delete * From P where [P#] = ‘P1’ would succeed, since VEs there are arithmetic only in CE-view P, propagating to P.B. But, even the simplest delete from S-P3.S, i.e., Delete * From S, would fail, since STATUS in CE-view S is defined through a subquery. Likewise, Delete * from S_P2.SP would fail, despite SP not having any VE in the scheme, but having joins, unlike S-P2.P. Surprisingly however, it would succeed under QBE in MS Access directly, hence perhaps also for SIR S-P2.SP with another kernel.

2.7 Usability of SIRs & Related Work

Our motivating example illustrated how SIRs may make a relational DB less-procedural, hence more usable by usual meaning of this qualifier. First, with respect to queries to S-P1, the equivalent ones to S-P2 and S-P3 were free of logical navigation or with reduced one, or were freed of selected VEs. If S-P1 should provide for the same queries, one would need to create the CE-views S, P and SP we have discussed. But then, every IE in S-P2 and S-P3 was less-procedural than Create View of its CE-view. The views were also more procedural to maintain. Same reasons motivated VAs, already decades ago. The examples illustrated how implicit IEs seamlessly, i.e., with the same usability, integrate that capability and even generalize it to VEs impossible to be VAs at present.

SIRs are also as in footsteps of the universal relation idea we already hinted to, also known since decades. That relation should have as attributes all these in the DB as well as all the values so that no query to the DB should need the logical navigation. Through often passionate, although now rather extinct interest in the topic there were various proposals for universal relations, [M4]. None apparently made to the industry, in the sense that it is up to DBA only or to the client to create a universal SR or, rather, a universal view. As we signaled S-P2.SP was actually in the footsteps of this
effort, being however, as a SIR, also a new type of a universal relation. The gain is again lesser creation cost, hence higher usability, than that of the (universal) CE-view or QE-view. S-P2.SP made logical navigation free not only for our example Q1 query, but generally for every query to S-P2 etc., not selecting a dangling tuple in S or P. For a latter query, the logical navigation may get reduced at least, as our example showed. We will also show soon that this usability generalizes further, to every DB without a multivalued dependency (MVD) in fact, [F77]. In presence of an MVD, selected queries not aimed at dangling tuples may nevertheless need the logical navigation. This one can however get reduced as well.

Besides, as we also mentioned, SIR-model is backward compatible with the SRV-model. The latter is indeed a strict sub-model of the former. The trivial condition to stay within the current model with the SIR-model is simply to refrain of IEs. In our example, it one can still stay with S-P1 instead of profiting from S-P2 etc. Switching to the SIR-model is safe in this sense. No loss of any current capabilities of S-P1, when expanded to S-P2 etc., may result from. Every application of it on a popular DBS should continue to run thus, if this DBS gets capabilities to manage SIRs as well.

It is notorious that the “biblical” S-P1 DB was the mold for most of practical ones. We may thus reasonably expect all the nice behavior discussed extending to most of practical DBs as well.

From the theoretical standpoint, we already recalled through our examples that insufficiency of the conceptual modelling by normalized SRs only were known almost since the inception of the relational model. There is no more such trouble with our example SIRs. These are intended upfront for the conceptual schemes and absent from the SRV-model. No more need to twist the purpose of views to enrich the actual conceptual scheme. Besides, no more need for the data dictionary identifying these views among usually many more. Likewise, there is no more need for the ER modelling, with its own troubles. Finally, one observed perhaps for all SIRs in S-P2 etc., that their bases were normalized as required by the SRV-model, i.e., were as in S-P1. Actually, we transpose these normalization principles to SIR-model formally soon.

Finally, one could observe from the example that the inheritance model for IEs is the original one of the relational model. That is, the foreign key value is the surrogate of the inherited object that is the one with the primary key equal to. This model characterizes also most of popular DBSSs. We should mention however that some, so-called, object-relational DBSs proposed different models in in 90ties. The open-source Postgres DBS is the most prominent survivor of this trend, [SM96], [P]. Those models of inheritance should not be confused with that of IEs. E.g., Postgres has a dedicated INHERITS clause in its Create Table, creating a sub-relation (sub-table) from the entire inherited relation etc.

3. SIR-model Relational Schema Design

At the peak of its glory, four decades ago, research on SRV-model issued countless proposals for somehow optimal (conceptual) relational scheme design. 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.

The relational scheme design goal was the removal of the anomalies. We preserve the goal for SIRs. Through IEs, we aim in addition at avoiding the logical navigation and specific VEs to queries. 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 could contradict S, where SNAME is anyhow already. Using SP instead, avoids the trouble.

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

Ex. 16. 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. It focus primarily on the logical navigation, avoiding selected VEs is a secondary goal. Ideally, we optimistically start with the attempt of the universal stored relation, say U, for the entire DB. U has the potential of avoiding the logical navigation to every query, 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. The notorious cost with respect to U is the logical navigation over the projections for most queries.

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 a SIR, say R again, is not only lossless, but also that 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 queries in (c) should be uncommon.

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 usually wise to choose A with fewest
attributes. Likewise, A should be the primary key of AB. B does not depend then on any proper
subset of A so 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 discussed principles in mind, we restate the theorem for an SR and a SIR as
follows. Let ABCD with ABC being as for the original theorem and D denoting IAs or
D = . In the
former case, ABCD is a SIR. In the latter case, it denotes the SR ABC. We decompose ABCD into
AB (A, B) and ACD (A, C, D, B). Here, B denotes B inherited through l_{ACD} = B, generating thus the
implicit recursive equijoin clause: On AB.A = ACD.A for E_{ACD}.

As the original decomposition, the restated one is thus also into two schemes and clearly lossless.
But, while AC was an SR, ACD can be a SIR and ACDB is a SIR with base AC. This decomposition is
possible only for the SIR-model. It produces AC as the stored projection of ABCD, as the original
theorem does for ABC. But, it also preserves all the original attributes together, although as A, B, C,
D. It avoids thus, as promised, the logical navigation to any queries selecting the original attributes.

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. For the reasons already abundantly discussed, the
creation of this view should be always more procedural than l_{ABC}, i.e., l_{ABCD} for D = . Observe finally
that query Q (B,D) to ABCD, making sense for the restated theorem only thus, would also remain free
of the logical navigation after the decomposition.
The decomposition thought the Heath Theorem is usually 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 the 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.

![Image of restated (a) and original (b) Heath’s decompositions, as well as, (c) restated Fagin’s decomposition.](image)

In practice, we start the decomposition, by Heath’s or Fagin’s theorem, from the optimistic assumption of a universal relation, usually noted U, for the DB. The decomposition of U until the final best normalized schema usually requires several steps. E.g., from U for S-P1, till its final 3-relation scheme, there are obviously two successive Heath’s decompositions. Observe that if all the decompositions are the Heath’s ones only, then the restated theorem always produces one projection that is a universal SIR. Every query to the DB can address that one and is then logical navigation free. Again, the universal view, formally equal to U, would lead to the same result, but at a higher definition cost, as we have seen for S-P2.

As we mentioned in the introduction and seen in the motivating example, 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 the motivating example. Queries of obvious practical interest may require the logical navigation again. Even so, ABCD may still reduce that one with respect to AC only. Actually, queries Q3 and Q4 already pointed out this property, although we did not refer there to Heath’s theorem.

Next, Fagin’s theorem also states that in presence of MVD A \[ \rightarrow\] B | C in the presumably stored relation ABC (A, B, C), its decomposition into AB (A, B) and AC (A, C) is lossless. Now, suppose B’ being a (perhaps empty) subset of B such that A \[ \rightarrow\] 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 ABCD a stored relation or a SIR as above stated. The restated decomposition creates ABDC’ (A, B, D, C’) and ACDB’ (A, C, D, B’) where \[ I_{ABDC'} = C' \] and \[ I_{ACDB'} = B' \]. As Figure 5.c illustrates, C’ and B’ avoid the logical navigation for any query to BD and C’ or to B’ and CD in the projections, unlike for the original decomposition. Only a query to B/B’ and C/C’ still needs it. As notorious, the result of such a query should be however typically awkward, as B and C are supposed mutually independent. Such query are therefore unlikely.

Unlike for FDs only and the exclusive use of Heath’s decompositions then, for an MVD in contrast, our decomposition does not avoid completely the logical navigation that the original Fagin’s 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.

Ex. 17. 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 as the only SR. In short notation we have:

U (EMAIL, S#, SNAME, STATUS, CITY, P#, PNAME, COLOR, WEIGHT, PCITY, QTY).

Notice the necessarily different names for the supplier and part cities, unlike in S and P of S-P1 or S-P2. U is potentially the optimal SR 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. Regrettfully, 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 may create two projections, intended as SRs, say SE and SP as follows, with C’ = (SNAME, STATUS, CITY) and B’ = ∅:

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

SE is now a SIR, with IAs in Italics and $I_{SE} = \text{SNAME, STATUS, CITY}$. We thus have also:

$E_{SE} = I_{SE}$ From SE Left Join SP On SE.S# = SP.S#,

with $I_{SE}$ denoting in short its actual extension. The base SE_B (S#, EMAIL) would be the projection for the original Fagin’s decomposition. SP is the same for both decomposition. SE is in the restated BCNF. It would not be if any of its IAs, e.g., SNAME, was an SR. The IAs of SE spare the logical navigation to every queries to EMAIL and to any of its IAs, unless one inserts a dangling tuple into SE. Otherwise, these queries would necessarily navigate over SE_B and SP. In contrast queries selecting emails and an attribute in SP that was not inherited in SE would still need to navigate through the projections. We come back to these queries later on. We will show that practical queries 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 may result from $I_{SP} = S.#$, with the implicit join $S.S# = SP.S#$. The projection SP is again an SIR, with $I_{SP}$ defining $B$. All the tentative SAs of the decomposed SP remain thus preserved in the projection SP, as the IAs sourced in S. Notice that this does not change anything for SE scheme. $S$ is as for the original decomposition. It is in BCNF, hence can be definitively made an SR. 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:

$$P \ (P#, \ PNAME, \ COLOR, \ WEIGHT, \ PCITY) \text{ and } SP \ (P#, \ S#, \ QTY, \ SNAME, \ STATUS, \ SCITY, \ PNAME, \ COLOR, \ WEIGHT, \ PCITY).$$

Here, the IAs of SP result now from revised $I_{SP} = S.#, P.#$, with the implicit join $SP.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, with PCITY supposed (arbitrarily) renamed to simpler CITY, becoming PCITY in SP only. We underlined the primary key stored attributes.

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

$$P \ (P#, \ PNAME, \ COLOR, \ WEIGHT, \ CITY),$$

$$SE \ (S#, \ EMAIL, \ SNAME, \ STATUS, \ CITY) \quad /* \text{ IAs from } S$$

$$SP \ (P#, \ S#, \ QTY, \ SNAME, \ STATUS, \ SCITY, \ PNAME, \ COLOR, \ WEIGHT, \ P.CITY \text{ as PCITY}) \quad /* \text{ IAs from } S$$

Notice that SP scheme is that of S-P2 from Example 1. Because of IEs, as for S-P2, most practical queries to S-P4 are 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 -> S#, leading to:

$$SE \ (S#, \ EMAIL), \ SP’ \ (EMAIL, \ SNAME...P#...S#),$$

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

$$S’ \ (EMAIL, \ SNAME, \ STATUS, \ CITY, \ S#),$$

$$SE \ (S#, \ EMAIL), \ P \ (P#, \ PNAME, \ COLOR, \ WEIGHT, \ PCITY)$$

$$SP’ \ (P#, \ EMAIL, \ QTY, \ S#, \ SNAME, \ STATUS, \ CITY, \ PNAME, \ COLOR, \ WEIGHT, \ PCITY).$$

Here, $I_{SP} = S’.#, \ P.#$ with the implicit recursive joins $SP’.EMAIL = S’.EMAIL$ and $SP’.P# = P.P#$ define 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-P4-S that we calculate STATUS as in Example 1. The only change to $S$ would be:

$$S \ (S#, \ SNAME, \ STATUS, \ CITY),$$
where \( \text{STATUS} \) results from the \( I_S = \text{STATUS INT}(\text{SUM(QTY)}/100) \), with the implicit \text{FROM SP_B} and the implicit join \( \text{S.S#} = \text{SP_B.S#} @ \).

We recall finally that the order of attributes in a mathematical relation does not matter, while it does in SQL. As formulated, our decomposition orders every relation with all the \text{SAs} first. One may need to reorder the results thus adequately.

4. Implementing SIR DBs

4.1 Basic Processing Scheme

As said already, the most practical way towards the SIR-model enabled DBS, seems to transparently manage a 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 a SIR DB 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 \( R \) statement received, SIR-layer should determine the type of the relation to create. For \( R \) being an SR, SIR-layer pushes the statement as is down to the kernel. In turn, the processing of \( R \) with IEs, being a SIR thus by SIR-level is clearly more involved. First SIRs obviously need dedicated meta-tables for the IEs. The schemes of these are easy enough to skip details. Then, the simplest design seems to represent every SIR \( R \) in the kernel by its base \( R_B \) and its CE-view \( R \). SIR-layer simply forwards afterwards every query as is to the kernel for execution using view \( R \). SIR-layer avoids the complex burden related to.

Accordingly, we qualify of basic (processing) scheme, (BPS), the SIR-layer algorithm for creating CE-views within the kernel we propose now. BPS always starts with the conversion of \( I_R \), if there is one (and only one then, we recall), into \( E^I_R \). Next, BPS passes the Create Table \( R_B \) statement to the kernel DBS, using for that all and only \text{SAs} of Create Table \( R \). Then BPS creates the CE-view as follows. Let \( A_1,\ldots,A_m \) contain every \text{SA} in \( R \) and every \text{IA} in \( E^I_R \), all the attributes being in the order resulting from Create Table \( R \) and all the subsequent Alter Table \( R \). SIR layer meta-tables should maintain this order. Then, BPS simply issues to the kernel the following statement, with From and Where clauses of \( E^I_R \):

\[
(V1) \text{Create View } R \text{ As } (A_1,\ldots,A_m \text{ From...Where...})
\]

Ex. 18. (1) We submit to SIR-layer S-P2 scheme at Figure 3. SIR-layer finds no IEs in Create Table \( S \) and Create Table \( P \). It passes each statement as is to the kernel that creates both relations as usual for SRs. SIR-layer in contrast determines that Create Table \( SP \) defines \( I_{SP} \) that we discussed, hence applies BPS. BPS creates \( E^I_{SP} \) and issues the following statements to the kernel DBS. The actual Create View below contains in fact the extension of \( E^I_{SP} \), defined in IE1 formula in the motivating example.

Create Table \( SP_B \); /* From all and only stored attributes of \( SP \) at Figure 3.
Create View \( SP \) As (Select \( SP_B.*3, E^I_{SP} \));

(2) Suppose now that DBA creates S-P3. To implement S-P3.S, BPS generates Create Table \( S_B \) as just above, then generates \( E^I_S \) for STATUS and RANK as in the motivating example and finally, send the following statement, with the extension of \( E^I_S \), to the kernel:

Create View \( S \) As (Select \( S#, SNAME, E^I_S, \text{CITY} \));

For relation \( P \), BPS generates \( E^I_P \) and send the following statement to the kernel:

Create Table \( P_B \);

Create View \( P \) As Select \( P#, PNAME, \text{COLOR}, \text{WEIGHT}, \text{WEIGHT\_KG}/1000 \) As \( \text{WEIGHT\_T}, \text{WEIGHT\_KG} \) As Round (\( \text{WEIGHT} \ast 0.454 \)), \text{CITY} \text{ From } P_B;
Finally, for SP, BPS generates the same statement as for S-P2.SP, extended however with all the additional IAs in S and in SP.@

Figure 6 illustrates Example 6.2. The SIR-layer shows the SIRs as rectangles. SIRs constitute the Conceptual Scheme (CS) of S-P3, in ANSI-SPARC DB Reference Architecture. The sizes are intended to reflect the number of tuples and the number of attribute values per tuple as seen by the client, i.e., supposedly as in Figure 4, augmented with the IAs proper to S-P3. The lower part shows under the same convention the SRs and the CE-views. These are the Internal Schema (IS) of S-P3, together with various usual underlying physical data structures. We leave as exercise, the BPS definition of the kernel statements for the IS construct(s) for views at the SIR-layer, i.e., for External Schemes (ESs) of SIR DB, in ANSI-SPARC terms.

One may obviously tailor BPS as we defined it above to specifics of the kernel. E.g., one can take advantage of VAs when available, saving the CE-view when all IAs are VAs. We leave the analysis of such enhancements for the future.

4.2 SIR-Layer DDL & DML Statements Processing

We suppose BPS designed also for all the other DDL statements for SIRs. As for Create Table statement for a SIR, Alter Table and Drop Table also require from BPS more processing than calling their kernel counterparts only. As the result, Alter Table R adding an IE to SR R, as our example Alter adding WEIGHT_KG and WEIGHT_T to SR P did, triggers renaming of R to R_B and creation of the resulting CE-view R. For Alter concerning an SA of SIR R, BPS issues the Alter Table R_B statement. Finally, for alteration of any IAs of R, through Refresh clause in particular, BPS first produces the new $E_R$. That one can be based on the existing one, being however altered as specified in Alter Table R, i.e., with additional attributes or without some or with some renamed. Alternatively, it produces entirely new $E_R$ from SIR-expression in Refresh clause. Next, BPS produces the Create View R for CE-view R resulted from the alteration, as it would do from $E_R$ for Create Table R. Then, BPS issues the atomic transaction with Drop View R to the kernel, followed with Create View R for the new CE-view. For this purpose BPS obviously explores the meta-tables at SIR-layer. We skip easy, but tedious details. Finally, for Drop Table R for SIR R, BPS issues, again as the atomic transaction, the sequence starting with Drop Table R_B, followed by Drop View for every partial view of R, if any and for the CE-View. Notice that future work may tailor the BPS as just outlined, e.g., to the availability of VAs.

With respect to the SIR-layer processing of 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 however 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 a view update query at present, if 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.

Observe that, in our implementation, the kernel DBS always processes a query to a SIR through its CE-view. It is known for decades that the processing of the query through a view may be faster than of the (equivalent) query addressing the source SRs directly, [H1]. The rationale is a possible partial materialization of views or of join clauses through indices, e.g., [V87], [GL1], [LZ7]. In our case this may concern the joins in CE-views, reflecting the recursive and usually outer ones that should be common to IEs, as one could see. Provided the kernel DB has thus some view materialization capabilities, queries to SIRs could be accordingly both less procedural and substantially faster than their equivalents to SRs only.

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, MS Access and MySQL are less restrictive: they accept update queries to views over multiple tables. 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, MS Access 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 MS Access, 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 Ex. 18 with a manual simulation on MS Access as the kernel. For each simulated SIR, a stored MS Access table is its base. The MS Access stored queries simulate the CE-views the BPS would create. The client may appreciate advantages of SIRs, through queries to CE-views. One may also alter and update any views, e.g., to experiment with every SIR definition, manipulation and processing aspect we discussed. As easy bonus, one may experiment the QBE interface for SIRs, generate forms, graphics, etc. In sum, one may play with all nice capabilities of MS Access 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 own query evaluation overhead is negligible as well. Within the kernel, the processing of every query to a SIR costs the same as the processing of the same query to the CE-view. Hence, there is no incidence on the query evaluation overhead of SIR-layer. Altogether, perhaps surprisingly, the enticing capabilities of SIRs appear thus practically almost without overhead cost.

5. Conclusion

Stored and inherited relation, (SIR), appears a useful construct for a relational DB. Through the IAs, a SIR may be conceptually richer than an SR with the same SAs, while IAs do not introduce any anomalies. SIRs alleviate in this way the notorious 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 be free of logical navigation or with reduced one. They can be free of selected value expressions as well. The cost for a SIR for these advantages with respect to the SR with the same SAs, hence without the latter, is the IE. With respect to the CE-view or QE-view, alternatively providing the same advantages, the IE should be always less procedural than the Create View. Especially, if the IE is implicit. Likewise, the view maintenance is more procedural than that of the SIR and may be source of big trouble. Recall that it is lesser procedurality of relational assertions that decisively attracted the users of equivalent navigational queries to CODASYL DBs.

Our extensions to Create Table for SIRs seamlessly integrate VAs, i.e., can be made backward compatible with the current clauses for those. Every relation with VAs is indeed a SIR with specific IAs, defined in Create Table through an implicit IE, we recall. The clients apparently found VAs useful, since they remain in use for decades now. Our extensions to SQL for SIRs only expand this capability, to a larger class of view-saving value expressions and to IAs saving the logical navigation. User should find them thus useful accordingly as well.

SIR-layer appears a higher level interface to SQL DBs. Its implementation over a popular DBS looks easy and with negligible operational overhead. The future work should start with that implementation. Depending on the kernel’s capabilities, it may be wise to include enhancements to BPS we have mentioned. But even without these, the result should be a win-win deal. Better sooner than later the existing DBSs should provide 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 restate those proposals for SIRs as well, especially the proposals for the lossless decomposition using outer joins, [JS90].

Finally, most of major DBSs are now interoperable, [LA86]. Multidatabase IEs seem thus attractive as well.

Acknowledgements

We thank Peter Scheuermann for helpful comments.

References

[C70] Codd, E., F. A Relational Model of Data for Large Shared Data Banks. CACM, 13,6,1970.
[LZ7] Larson, P., Zhou J. Efficient Maintenance of Materialized Outer-Join Views. oin view need not be more expensive
than maintaining an inner-join view. ICDE 2007.


symposium/talks/mendelzon.pdf.

[MUV84] Maier, D, Ullman, J. D., Vardi, M., Y. On the foundations of the universal relation model. ACM-TODS, 9, 2, 283-
308, 1984.


1998.
