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). SIRs can make queries less procedural than to SRs only, without impacting the normal form. Queries may become partly or fully free of logical navigation or of selected value expressions. Specific views may provide the same capabilities. Nevertheless, we extend SQL so that declaring IAs for a SIR is always less procedural than creating any such view. Altering a SIR is also always less procedural than altering the view. Finally, our extensions provide for backward compatibility with virtual (computed...) attributes (columns), provided by some DBSs. The latter already avoids selected value expressions to queries, while being also always less procedural to define and maintain than any equivalent view. We motivate our proposals through the biblical Supplier-Part DB. We show how to implement SIRs with negligible operational overhead. In conclusion, we postulate SIRs 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 the stored attribute (SAs) of R defined as usual in an SR. The inherited attributes (IAs) are in contrast defined basically as usual in a view. For every SIR R, a single Create Table R defines both the SAs and the IAs.

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 logical navigation, necessary within every equivalent query to the SRs only. We recall that such navigation in an SQL query occurs whenever it refers to several relations with, usually, equijoins among those. Likewise, a query to a SIR may avoid selected value expressions. Altogether, SQL queries to a DB with SIRs should end up usually less procedural (simpler, more usable...) than their equivalents to a DB with normalized 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 that one can name view R, defining mathematically the same SQL relation and for every attribute in SIR R, with, at least, the same proper name in view R. “Mathematically the same”, means, we recall, the abstraction of the implementation considerations of the relations. In our case, the fact that a value stored in SIR R is only calculated in view R becomes out of the considerations. We recall also that in an SQL relation the attributes are in some order, unlike in a mathematical relation. For every SQL query to SIR R, view R provides then for the same outcome. We call every such view R equivalent to SIR R. In

¹ Université Paris-Dauphine, PSL Research University, CNRS, LAMSADE, 75016 Paris, France. witold.litwin@dauphine.fr
² LAMSADE Research E-Report. Latest update August 10, 2018
fact, such views are already for decades the notorious intuitive “escape route” for clients unhappy with the procedurality of most queries to normalized SRs only or with poor conceptual schemes of some, i.e., without enough attributes. Universal views, providing all the attributes and, possibly, all the values of the DB in one relation, were particularly studied, [MUV84].

We propose extensions to Create Table to accommodate SIRs. Likewise, we propose extensions to Alter Table. We show that for every SIR R, our clauses for the IAs in Create Table can be less procedural than Create View R of any equivalent 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 for every equivalent view R. It will appear also that View R may be more procedural to alter than the SIR R scheme, while no view R may ever be less procedural to alter than SIR R. We show finally how to implement SIRs on a popular DBS with negligible storage and processing overhead. As the need for faithful conceptual modelling and non-procedural queries is universal, we postulate SIRs 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 concerning VAs. Our clauses for SQL aim precisely at the same gain. But they generalize it to every SIR. Consequently, first, for every SIR with solely IAs that would fit as VAs, our Create Table provides for the same gain as the Create Table supporting VAs at present. This is done through the backward compatibility, abstraction made of minor syntactical differences between current SQL dialects. Besides, we gain also for every relation with some or all IAs that could not fit as VAs at present. We can indeed make it into a SIR, while it can only be an equivalent view of that one at present.

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

Section 3 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 4 discusses the related work. Section 5 concludes that SIRs should be standard for relational DBs and proposes 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 said, every SIR has furthermore some SAs and some IAs that may intermix. Every SIR has also a name and scheme defining all its SAs and IAs. The scheme defines every SA as for an SR. It defines every IA through the already mentioned IE that is some relational or value expression, as for a view. Values in IA sub-tuples are basically immaterial, as usual for views as well. For some SIR tuples, the IE may produce null IA sub-tuples.
For obvious practical reasons, as we already hinted to, we consider every SIR as an SQL relation, i.e., where the attribute order matters. Figure 2 illustrates the possible structure of a SIR. 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 other defaults possible, e.g., R_only. The green rectangles and the white ones labelled Null, represent the IAs, valued or nulls. SAs and IAs intermix at the figure.

For every SIR R, we consider a specific equivalent view, noted view R as well. Every attribute in view R that is an IA in SIR R has the same full source name as in SIR R, i.e., has the same proper name and inherits from the same relation. Next, the IAs in view R that are the SAs in SIR R inherit all from the same (hypothetical) SR named R_B and equal to the base of SIR R. We suppose R_B thus to be the SQL projection of SIR R on all its SAs only. Finally we suppose that the IE and view R schemes are the same, except that IE contains in its Select list only all the IAs of view R that are also IAs for SIR R. We call view R conceptually equal to SIR R, CE-view R in short.

It follows that to qualify as CE-view R of some SIR R, every view R has first to inherit as attribute A every attribute A of some SR, named R_B by default. For every tuple R_B.t, there has to be furthermore exactly one tuple R.t’ with all the values in t as sub-tuple. View R should not have other tuples. Notice that for every R_B, if R_B.A1,...,Ak form the primary key, then R.A1,...,Ak is then also a key for CE-view R.

Given all this, we define every SIR R through an extended Create Table R, basically as follows. Basically, means here that later we define some additional rules. We declare each SA as we would for Create Table R_B. We declare each IA as we would for the Select list of CE-view R. The list of these declarations in Create Table R should follow the desired SQL order, i.e., the one that Select * From R should display. Notice that the IA declarations in the Select clause of CE-view R follow the same order. After the declarations of all the SAs and IAs, one provides the From etc. clauses of CE-view R.
Finally, follows every eventual clause of \( R_B \), beyond all the attribute declarations. Those clauses are thus for the primary key, for the referential integrity, for the indexing...

We consider furthermore the notorious SQL naming rules applying to SAs and IAs in SIRs as well. The additional rule is that for every SIR \( R \), one may qualify every SA \( A \) not only as \( R.A \), but also as \( R.B.A \). The latter is the default. The obvious rationale is our reuse for SIR \( R \) of CE-view \( R \) scheme. The clauses From etc. referring to \( R_B \) within view \( R \) remain all valid for SIR scheme, despite the absence of any stand-alone SR named \( R_B \). A less obvious rationale is that referring to \( R_B \) rather than to \( R \) for some other SIR \( R' \), may also avoid the so-called circular referencing. We address this issue later.

For every SIR \( R \), we call \textit{inheritance expression} (IE) all the clauses in Create Table \( R \) defining the IAs. Furthermore, we qualify of \textit{explicit} an IE defined as above, and denote it \( E \) or \( E_R \) for SIR \( R \). The additional rules we hinted to, lead to \textit{implicit} IEs we discuss soon. Observe that every \( E_R \) defines the SQL projection of CE-view \( R \) on all and only IA that are also IAs in SIR \( R \). Every \( E_R \) consists thus (i) of a sub-list of Select clause of Create View \( R \), where every such IA and such IA only is named, and (ii) of all the clauses From, Where... in Create View \( R \). While contiguous in \( E_R \), the elements of the sub-list may be separated by SA schemes in Create Table \( R \) as at Figure 2, we recall. Nevertheless, as a character string, every \( E_R \) is therefore a sub-string of Create View \( R \), although perhaps a distributed one.

Recall furthermore that for every SIR \( R \), the definition of SAs in Create Table \( R \) is the same as in Create Table \( R_B \). Also, for any view scheme, most often, the Select expression defining any view scheme, enumerates every IA in Select list or, if some form all the attributes of some relation \( X \) and are inherited with the same names and values, then Select may contain instead the notorious less procedural generic SQL construct \( X.* \). Every \( E_R \) is then only a substring of Create View \( R \). Declaring an \( E_R \) instead of such a CE-view \( R \), is consequently always less procedural. Using SIRs would bring thus this advantage to every DB considering views qualifying as such CE-views. The rare and only exception is the CE-view with entire Select list reduced to \('*'\) only. According to SQL rules, an IE defined as up to now, may then need at least several \( X.* \) constructs instead. It may consequently be more procedural than the CE-view. In next section, we propose an additional rule for IEs, allowing for the IEs less procedural than the CE-views also in every such case.

Ex. 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 every 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 different SIRs. We will call these variants 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 3 shows the sample data type for each attribute in these relations. Actually, the figure shows S-P2 DB that is our basic SIR DB. Nevertheless, \( S \) and \( P \) relations are the same. S-P1.SP types are these of the SAs with the same names in SIR SP shown there. All the SA definitions at the figure skip the practical details other than the data type, e.g., the length. We underline the primary key, as usual.

Figure 4 shows the original sample data values for S-P1. For S-P1.SP, these are among all those of SIR SP there, 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. As discussed, it does for every SQL query with \('*\)' e.g., for Select \(*\) From \( SP \). The S-P1 scheme is the optimal conceptual one for the relational DB for the discussed application. The criterion is the minimal number of SRs, free of storage and update (normalization) anomalies.
The well-known drawback of S-P1 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. Such a view avoids the logical navigation to more query than any other view of SP with fewer attributes or values. For any view SP, SR SP has to be first renamed, say to SP_B. Every relation in an SQL DB must have indeed a different name. Then, likely the least procedural view SP declaration in SQL is:

(1) Create View SP As (Select SP_B.*, SNAME, STATUS, S.CITY, 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#);

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

![Figure 3 S-P1 and S-P2 schemes.](image)

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 outer joins, each of these tuples expands the latter values with those in a single tuple in S and a single tuple in P, whenever those values exist and with nulls otherwise. The joins select indeed at most a single tuple in S since S# is the key of S. Likewise, they select at most a single tuple of P. (1) requires the outer joins since there are no referential integrity requirements for SP in S-P1. Hence SP_B could have a tuple where S# value or P# value does not exist in S or P. Inner joins would not preserve such an SP_B tuple in view SP, missing the intention of view SP. In contrast inner joins would suffice if the referential integrity clauses were added to SP in S-P1, hence to SP_B.

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.

To have a DB, say S-P2, with S, P and SIR SP instead of S-P1 with S, P and SP renamed to SP_B, and view SP defined by (1), one should figure out first whether the view qualifies as CE-view SP. This is the case. First, SP inherits bijectively every tuple of SP_B as exactly one sub-tuple and has no other tuples. Thus, in particular, \((SP.B.S#, SP.B.P#)\) is the primary key of SP, as \((SP.B.S#, SP.B.P#)\) is for SP_B. The reason for these properties of view SP is that S.S# and P.P# are also the keys. Accordingly, for the first tuple of SP_B at Figure 4 for instance, i.e., with SAs S# = S1 and P# = P1, the join clauses match only one source tuple in S and only one in P. Only a single tuple in view SP results from that is the
first one at the figure. Similarly for SAs S# = S1 and P# = P2 etc. View SP qualifying thus as CE-view SP, the following Create Table SP would create SIR SP for our S-P2:

(2) Create Table SP (S# Char, P# Char, Qty Int, SNAME, STATUS, S.CITY, 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#), Primary Key (S#, P#));

Figure 4, shows the content of SIR SP that would result for the sample data in S-P1. Every SA is in plain text and every IA in Italics. The SAs schemes in S-P2.SP are supposed these of S-P1.SP, hence of SP_B. These SAs and their tuples form the base SP_B of S-P2.SP. The (underlined) key of S-P2.SP is also that of S-P1.SP. Its definition in (2) follows entire $E_S$, as required for every Create Table R for SIR R. $E_S$ is the string: ‘SNAME...P.P#' that happens to be a contiguous one. It is the same substring in (1) hence in CE-view SP, as well as the SQL projection there on the enumerated IAs. These are also all and only IAs in (2). As only a substring, it is strictly less procedural than (1). More precisely, one saves the string ‘Create View SP As (Select SP_B.*,’. This makes Create View SP scheme about 25% more procedural than $E_S$. The remaining part of (2) is exactly as procedural as Create Table SP_B. It is simply the same indeed, except for the name SP_B of course.

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

<table>
<thead>
<tr>
<th>Table SP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>S#</strong></td>
</tr>
<tr>
<td>S1</td>
</tr>
<tr>
<td>S1</td>
</tr>
<tr>
<td>S1</td>
</tr>
<tr>
<td>S1</td>
</tr>
<tr>
<td>S1</td>
</tr>
<tr>
<td>S1</td>
</tr>
<tr>
<td>S2</td>
</tr>
<tr>
<td>S2</td>
</tr>
<tr>
<td>S3</td>
</tr>
<tr>
<td>S4</td>
</tr>
<tr>
<td>S4</td>
</tr>
<tr>
<td>S4</td>
</tr>
</tbody>
</table>

Figure 4 The S-P2 content. IA (proper) names and values are in Italics.

In both (1) and (2), the already recalled SQL ordering makes all the SAs preceding all the IAs. It is our subjective choice. The rationale is that keeping the IAs inheriting from SP_B together, minimizes, in SQL, the procedurality of view SP, through SP_B.*. Note nevertheless that many consider *' less safe for Create View than the list of attributes it represents. This would make the procedurality gain provided by SIR SB even greater. Same would happen if an IA dispersed the SAs within Create Table SP and in in CE-view SP thus. The list of IAs contiguous in $E_S$ would then consist of the same IAs, but non-contiguous in Create Table SP. The same From clause of (2) would follow both lists. Finally, for S-P2, the query Select * From SP; would output the attribute order at Figure 3 and the tuples of Figure 4.

Observe also 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, it refers to the SP base S_B, hence to a part of SP itself. We qualify below of recursive every join in some SIR R referring similarly to a part of R. Actually, a
recursive join may be a $\theta$-join, as we will show. Recursive joins are basically not permitted for an SQL view, we recall. The example suggests them in contrast typical for an E-R.

The graphic at Figure 3 illustrates similarities and differences between our example schemes. From left to right we have first S-P1 with the “biblical” SRs only. Next, there is our S-P2 with SIR SP replacing the original SP. Finally, there is S-P1 with SP renamed by default and the CE-view SP, defined by (1). The colors symbolize SAs and IAs as in Figure 2. The grey rectangles are thus the same for all the DBs. The green one of S-P1 with view SP is as large as SIR SP. It is larger than the green one of SIR SP by its left part. That one is redundant with SP_B, since illustrating the (stored) attributes of SP_B redefined as IAs. This redundancy must cost some procedurality. This one is the core of that adversely affecting our CE-view SP compared to SIR SP. Same redundancy leading to perhaps even greater procedurality would clearly affect every CE-view SP with the attribute order different from that in (1).

2.2 Implicit IEs

For every SIR R, one can define the IE as some E_R. Nevertheless, there are three cases when one needs an IE even less procedural than any E_R could be. We now define such IEs. We call them implicit and denote as I or I_R. We suppose that DBS supporting SIRs internally pre-process every I_R to some E_R that we denote as $E_R$. DBS processes then every $E_R$ as every $E_R$. We propose the rules for this pre-processing.

We already hinted to the first case. It is the possibility of the IE of SIR R less procedural than CE-view R, even when the Select expression of latter is in the form Select * From R1…R2…R3…, with some of these relations being R_B. We do not see any practical case of more than one R_B, so suppose R_B = R1, for instance. We have then:

$$E_R = R2.*, R3.*…\text{ From } R_B…R2….$$  

The procedurality of the list of the attributes in $E_R$ grows linearly with the number of relations listed. Clearly, for any CE-view R, for some number, we guess above eight in practice or for even R2 only, but with long enough proper name, $E_R$ must become more procedural than the view. We define therefore the following rule to solve the case:

Rule 1. I_R has the form: ‘# From R1…R2…R3… with R_B = R_i for some i. If R_i is not the first or the last one in From clause, $E_R$ is:

$$E_R = R1.*, R2.*…R_i-1.*, R_i+1.*…\text{ From } R1…R2…R_i…;$$

We skip the otherwise obvious forms of $E_R$. We suppose furthermore that ‘#' should be in Create Table after all the SAs. The order of the attributes in SIR R would nevertheless be that in From clause. This, as usual for every query or view in form of Select * From R1…R2….

Ex. 2. Suppose for S-P1 that only selected clients should be able to match the supplies of any supplier or part. All the others may still access every relation, nevertheless. The DBA may therefore use a secret function Enc to encrypt SP.S# and SP.P# of every S.S# and P.P#. The DBA may furthermore provide the selected clients with the universal view SP as follows, after renaming S and P as before:

(3) Create View SP As (Select * From (SP_B Left Join S On SP_B.S# = Enc(S.S#)) Left Join P On SP_B.P# = Enc(P.P#));

View SP defined so is clearly also CE-view SP for SIR SP with base SP_B. Given Rule 1, DBA may define I_SP then simply as:

(4) I_SP = # From (SP_B Left Join S On SP_B.S# = Enc(S.S#)) Left Join P On SP_B.P# = Enc(P.P#));

Clause From is the same for (3) and (4), hence I_SP remains less procedural than View SP. Actually, the length is visibly reduced by about 25%. When Create Table SP is declared, DBS pre-process (4) to:

(5) $E_{SP} = S.*, P.*$ From (SP_B Left Join S On SP_B.S# = Enc(S.S#)) Left Join P On SP_B.P# = Enc(P.P#));
Actually, (5) remains visibly less procedural than (3) as well. However, visibly as well, it would no more be so if S and P had longer names, e.g., SUPPLIERS and PARTS or the universal view of supplies had to inherit from more relations for any reasons. Hence, without Rule 1, the goal of an IE always less procedural than the CE-view would not be attainable.@

Our 2nd case concerns the query (result) equal view, QE-view in short, as we call it. Under restrictive conditions on the DB, QE-view R may be the same relation as CE-view R and SIR R thus, except for different source name(s) for some unique proper SA name(s). No SQL query to CE-view R, hence to SIR R, needs then to invoke the full source name of such attributes. Also, for every popular DBS, every query to CE-view R labels every such attribute with the proper name only. Any query result may then result from the same query either to CE-view R or to QE-view R. In the same time, QE-view R may be substantially less procedural to define than the least procedural CE-view R. The least procedural $E_R$ could then turn more procedural than QE-view R as well, contradicting our claim of providing for any SIR R, an IE less procedural than any view R providing for the same query results. A DBA could then also evidently prefer $R_B$ and QE-view R to SIR R.

The following rule provides for our goal.

Rule 2. The Select list of $I_R$ contains only as elements A1,A2… or R1.#, R2.#… . Also, every relation Ri has the attribute Ki as a key, supposed mono-attribute for simplicity. SIR R also has Ki as an attribute. Then, DBS produces $E_R^I$ as follows.
1. For every Xi.#, $E_R^I$ lists all the attributes of Xi except for Ki, in the order of Xi.*.
2. $E_R^I$ lists every A1, A2… in the order of in $I_R$, including every attribute replacing every Xi.# in $I_R$.

Ex. 3. Consider the following variant of S-P2.SP with the attributes ordered as follows:

(6) SP (S#, SNAME, STATUS…, P#, PNAME…, QTY).

Suppose the referential integrity between SP, S and P. The following Create View SP becomes possible, after the default renaming of SP:

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

This view SP is not CE-view SP. The full source names of attributes S# and P# are indeed S.S# and P.P#, instead of SP_B.S# and SP_B.P# in CE-view SP. Nevertheless, with the referential integrity enforced and only then, no query to view SP needs the source names for S# or P# for the same result as when addressing CE-view SP. View SP is QE-view SP for our S-P1 thus.

Create View SP for CE-view SP could have the same From clause as in (7), but would need to enumerate all the attributes of S, P and SP_B in obvious order. $E_{SP}$ would do so for all the IAs only, consequently. It results that (7) is about twice less procedural than Create View SP for CE-view SP. It is also less procedural than $E_{SP}$, as one may easily see. If $E_{SP}$ is the only choice for SIR SP, then DBA may legitimately prefer QE-view SP.

Rule 2 authorizes nevertheless the following simple expression for the IE, being then $I_{SP}$:

(8) $I_{SP} = S.#, P.# From S, P, SP_B Where SP_B.S# = S.S# And SP_B.P# = P.P# ;$

$I_{SP}$ is now less procedural than (7), by about 20%. The difference evidently increases with the number of SAs in SP. No more reasons for the DBA to prefer QE-view SP anymore. The resulting Create Table SP for our SIR SP with $I_{SP}$ would finally be:

(9) Create Table SP (S# Char, S.#, P# Char, P.#, Qty Int From (SP_B Left Join S On SP_B.S# = S.S#) Left Join P On SP_B.P# = P.P#), Primary Key (S#, P#));

We recall that (9) replaces Create Table SP_B with SP_B (S#, P#, QTY) and Create View SP as (7), for our QE-view SP.@
One may finally, easily verify from the example that lower procedurality of \( I_{SP} \) generalizes to any \( I_R \) conform to Rule 2, regardless of the number of constructs \( R_i.\) in the Select list and of relation and attribute names.

Our third and last case is that of SIR \( R \) where (i) every IA could be an already mentioned VA, also named *computed or generated column*, we recall and, basically, when the DBS supports the VAs. One easily sees that every \( E_R \) must then be more procedural than the VA-declarations in Create Table \( R \) for relation \( R \) with VAs. This was actually the rationale for the concept decades ago, although stated without explicit reference to any \( E_R \) of course. If for a discussed SIR \( R, E_R \) was the sole possibility for the IE, the DBA should again choose Create Table \( R \) with VAs explicitly declared. Fortunately, the following rule provides the backward compatibility for every discussed SIR \( R \). It provides it in fact for every DBS supposed supporting SIRs, whether it supports VAs specifically or not.

**Rule 3.** Let \( A \) be a proper IA name and let \( V \) designate a value expression allowed for a VA by the SQL dialect of the DBS supporting SIRs. Suppose also that this DBS itself uses as the kernel some popular DBS supporting VAs at present. Suppose finally that Create Table \( R \) contains elements in the form:

\[
(10) \quad A_1\text{ As } V_1,\ldots,A_k\text{ As } V_k, \quad k = 1,2,\ldots
\]

Then, DBS considers every such element as an IA that is a VA. Their presence in IE makes it \( I_R \), DBS pre-processes every such \( I_R \) to \( E_R \) simply constituted from every IA in Create Table \( R \) other than a VA.@

The rationale for Rule 3 is that the declaration of an IA in form (10) is basically backward compatible with that of a VA at present. “Basically” means here that \( I_R \) should in fact conform to the actual syntax for VAs of the kernel. Minor syntactical variations may occur then. E.g., for MySql SQL, (10) should include parentheses around every VE. For SQL Server in contrast, (10) should apply as is. As we detail in Section 3, we suppose the DBS supporting SIRs with the kernel supporting VAs, to processes every IA conform to Rule 3, as a VA effectively. In contrast, entire Rule 3 does not apply, e.g., to MsAccess as the kernel, since that one does not support VAs. Finally, even if Rule 3 applies, an IA can be declared as VE As \( A \), as usual for a Select list of a view. It is mandatory if it cannot be a VA for the kernel DBS.

Ex. 4. Suppose that in S-P2.P.WEIGHT, the weight of every part is in pounds, while the clients should also know the weight in KG as WEIGHT_KG, placed as the successor of WEIGHT in P. First, the following \( E_P \), placed in Create Table P immediately after the declaration of SA WEIGHT, would do:

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

Notice that in Create Table P with (11), the declaration of the SA CITY would precede the From P clause. \( E_P \) would be again less procedural than any CE-view P or QE-view P. However, suppose now the SQL Server dialect and DBS for S-P2. The following VA declared at the same required place in Create Table P of the SQL Server at present would do as well:

\[
(12) \quad \text{WEIGHT_KG As Round } (\text{WEIGHT } * 0.454,3); \quad E_P
\]

\( E_P \) is then clearly more procedural than WEIGHT_KG defined so. The DBA would be better off declaring P simply this way. Rule 3 however allows the DBA to declare WEIGHT_KG for SIR P through exactly the same expression. The DBA may declare SIR P then through the same Create Table P as for the SQL Server without support for SIRs. There is no more procedurality penalty for SIR P.@

It should be clear from the last three examples that the above rules generalize to every SIR \( R \), where the only choice at present is some view \( R \) that amounts to CE-view \( R \) or QE-view \( R \), or is some VAs. Creating SIR \( R \) should be strictly less procedural than every view \( R \). To create any relation \( R \) with VAs amounts, through Rule 3, simply to create SIR \( R \) where every IA could be a VA at present. Current relations with VAs amount thus simply to specific SIRs, without of course being considered so till now.
2.3 DDL Statements for SIR-model

We already discussed Create Table for SIRs extensively. We now discuss the other SQL DDL statement for SIRs. We suppose every such statement backward compatible with some kernel. E.g., for MySQL SQL, we suppose Create View for SIRs being simply the MySQL Create View, except that among source relations could be a SIR. Similarly for SQL Server etc.

The other SQL DDL statements we consider for SIRs are all the popular ones, i.e., Alter Table, Drop Table, Alter View, Drop View and Create Index. For Alter Table R for some SR or SIR R, we suppose for the former the semantics of Alter Table R of the kernel SQL. E.g., for MySQL kernel thus, Add may create an SA or a VA or may be followed with optional First and After keywords specifying how the added SA mixes with the existing SA and VAs. Also, Alter Table may alter several attributes, unlike for SQL standard. On the other hand, for any kernels, Alter Table R for R that is SR or SR with IAs declared as VAs, may expand R with an IE. This is done only through the clause specific to Alter Table for SIRs, we named IE as well, and refer to as IE-clause. The IE-clause may finally define the IE replacing an existing one, like the Select expression in Alter View X replaces the existing view X scheme.

The IE-clause may be in one of the following forms, differing only by the Select list of IAs and of SA. If A1,...,An are the IAs, the list (A1,...,An) means that all these IAs follow all the SAs and, perhaps, VAs, of the SIR. The latter remain in the order of Create Table, perhaps altered by subsequent Alter Table. In turn, the list (A1,...,An,*) means that all the IAs precede all these SAs. Finally, for every SIR R resulting from Alter Table R, IE-clause may be stated as Create Table R would do to define SIR R, except that if an SA or VA is referred to in Select list of IE-clause, it is by name only. If this list refers to any SA or VA, it has to refer to every SA and VA of SIR R. These attributes should be listed in the current SQL order they would be in Create Table R, i.e., the original one before the alteration or in the altered order. In other words, IE-clause may be like could be the expression following Select keyword in Create View R for CE-view R of SIR R resulting from Alter Table R. Finally, in every of its forms, the list in IE-clause may designate IAs in every way an IA could do.

Next, for every SIR R, we allow Alter Table R to drop the IE through simple Drop_IE verb. This obviously alters SIR R into SR R with VAs eventually. Then, if Alter Table drops, adds or renames any SAs or VAs, new IE clause is optional. Like it could be for Alter View R for CE-view R, resulting from the same Alter Table R_B. Next, for any SIR R, we prohibit to drop all SAs, as usual for every alteration of an SR R, besides. In other words, we prohibit for every SIR R, any alterations into a view instead. If such need occurs, one should use Drop Table R followed by Create View R. Likewise, if a view R should evolve to SIR R, we presume Drop View R followed by Create Table R. This procedure is obviously the simplest to put into practice.

For Drop Table R, we simply consider that one can apply to every SIR R as well. As usual, the manipulation should not violate the referential integrity. It may also trigger a cascade to other SRs or SIRs or the refusal of the statement. Next, we suppose Alter View and Drop View the ones of the kernel, if any for Alter View. Finally, we suppose Create Index to apply to SAs and IAs as the kernel one applies to SAs, VAs and views.

Ex. 5 DBA adds to S-P2.P the IA WEIGHT_KG from Erreur ! Source du renvoi introuvable.. 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.

1. The SQL dialect for SIRs is backward compatible with MySQL.

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

Both IA schemes are so since the IAs could be VAs for MySQL. As the result, Alter modifies SR P into SIR P that, on MySQL, could be relation P with SAs of S-P1.P and two VAs.
2. The SQL dialect for SIRs is backward compatible with DBS without VAs, e.g., MsAccess.

(14) Alter Table P IE (P#, PNAME, COLOR, WEIGHT, WEIGHT_KG / 1000 As WEIGHT_T, Round (WEIGHT * 0.454) As WEIGHT_KG, CITY From P_B);

3. The DBA from (2) above decides to drop WEIGHT_T. The following statements would do for SIR P:

(15) Alter Table P IE (P#, PNAME, COLOR, WEIGHT, Round (WEIGHT * 0.454) As WEIGHT_KG, CITY From P_B);

For view P, if kernel dialect provides Alter View, then the DBA could use:

(16) Alter View P As (Select P#, PNAME, COLOR, WEIGHT, Round (WEIGHT * 0.454) As WEIGHT_KG, CITY From P_B);

Otherwise, e.g., as for MsAccess, DBA would need Drop View P followed (atomically) by Create View P. In both cases, the alteration is visibly less procedural for SIR P.

4. DBA of S-P2 has created SP initially as S-P1.SP SR. Then, s/he decided to alter SP to SIR SP at Figure 3. Thus all the IAs should follow the base SP_B. Regardless of the kernel dialect, the following statement should do:

(17) Alter Table SP IE (S.#, P.# From (SP_B Left Join S On SP_B.S# = S.S#) Left Join P On SP_B.P# = P.P#);@
2.4 Data Manipulation

2.5 Manipulating SIRs

As for DDL, we presume for every DBS supporting SIRs that DML statements for SIRs are backward compatible with the kernel SQL dialect. Then, for every SIR R, every query to SIR R valid also for CE-view R provides the same result as it should provide for CE-view R, given our definition of Ce-view. This defines the outcome of every Select query to SIR R. An update query to view R is valid however only if the view is updatable for the query. In practice, this depends on the kernel. The constraint could impair even a query updating SAs only. For every SIR R, we allow therefore every such update to have the semantics of a query to an SR. More precisely, instead of being valid for CE-view R, the update should be valid for R_B at least.

Ex. 7. The simplest select query: 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 MsAccess dialect as the kernel one for SIR SP, the update query Insert SP (select ‘S4’ as [S#], ‘P4’ as [P#], 100 as QTY); would add the tuple to SP with these values and with all the IA values. For CE-view SP, this query would propagate the update of these attributes to SP_B. Likewise, the query Update SP set QTY = 250 where S# = ‘S1’ and P# = ‘P1’; would update one value in SIR SP. Same update to CE-view SP would propagate to SP_B as well.

Then, the statement: Update SP set QTY = 250, CITY = ‘Paris’ where S# = ‘S1’ and P# = ‘P1’; would do similarly for SIR SP, propagating however also the change to CITY to S_B.CITY for CE-view SP. The perhaps surprising side-effect should be for SIR SP the SP.CITY change for every other supply by S1, since it would be so for CE-view SP. The Insert SP (select ‘S4’ as [S#], ‘P4’ as [P#], 100 as QTY, S.CITY as ‘Rome’); would change to Rome the CITY value in every SIR SP tuple with S# = S4, since it would be so in CE-view SP. Likewise, every update to WEIGHT_KG in SIR SP should fail. Finally, every Delete...From SP... would fail for SIR SP. It would fail for CE-view SP indeed for SQL of MsAccess as the kernel dialect, because of the joins (it would succeed however in QBE of MS Access, perhaps surprisingly). A Delete statement would succeed for SIR SP with this dialect only if formulated as a Delete...From SP_B....

In contrast to MsAccess, SQL Server as kernel would be more restrictive. A view there is updatable indeed only if it inherits from a single SR, unlike for S-P2.SP therefore. Under SQL Server as the kernel DBS thus, the client would need to formulate every above update query as to S, P or SP_B. MySQL is less restrictive. Like MsAccess it accepts some update queries to views over multiple tables. Hence, it would accept some updates to SIR SP, e.g. those in our examples.

3. Implementing SIR DBs

3.1 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 5. 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 every R being an SR, SIR-layer pushes the statement as is down to the kernel. In turn, the processing is clearly more involved for every SIR R. First SIRs obviously need dedicated meta-tables for the IEs. The schemes of these are easy enough to skip the matter. Then, the simplest design seems to represent every SIR R in the kernel in general by its base R_B and CE-view R. SIR-layer simply forwards then every query as is to the kernel. That one executes the query, using
thus view R or R_B. Only for every SIR R defined through Rule 3 specifically, on the DBS supporting VAs therefore, the simplest design appears rather to simply forward to the kernel Create Table R for SIR R, to create a single SR R with VAs.

Accordingly, we qualify of basic (processing) scheme, (BPS), the SIR-layer processing of SIRs. For Create Table R for SIR R in the general case, BPS always starts with the conversion of I_R, if there is any into E'_R. Next, BPS passes the Create Table R_B statement to the kernel DBS, using for that all and only (i) SAs of Create Table R and (ii) IAs declared VAs if any. Then BPS creates the CE-view as follows. Let A1,...,Am list the name of every SA and VA (if there is any) in R and let it contain every IA as it is in Select list of E'_R. Suppose all the attributes in their order in Create Table R, perhaps amended by subsequent Alter Table R. SIR layer meta-tables should maintain this order, as do meta-tables of any SQL DBS. Then, BPS simply issues to the kernel the following statement, with From and Where clauses of E'_R:

(V1) Create View R As (Select A1,...,Am From...Where...)

Ex. 8. (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 Create statement as is to the kernel that creates both relations as usual for SRs. SIR-layer in contrast determines that Create Table SP defines E_{SP} or I_{SP} we discussed. Hence, it calls BPS. BPS eventually pre-processes I_{SP} to E'_{SP}. It both case, it issues the following statements to the kernel DBS. The Create View SP is that in (1) actually.

Create Table SP_B... ; /* With all and only stored attributes of SP at Figure 3.

Create View SP As (Select SP_B.*, SNAME, STATUS, S.CITY, 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#);

(2) Suppose now the kernel dialect backward compatible with MySQL, hence supporting VAs. Suppose also that DBA creates SIR P with IAs WEIGHT_KG and WEIGHT_T, upfront defined as in (12) and (13). BPS passes then Create Table P the DBA submits to SIR-layer as is to the kernel DBS. That one executes the statement. The result is SR P with VAs.

If the kernel dialect does not support VAs, Create Table P for SIR P should define both IAs as for a view, i.e., through (11) for WEIGHT_KG and similarly for WEIGHT_T. BPS generates then the following statements for the kernel:

Create Table P_B...

(18) Create View P As Select P#, PNAME, COLOR, WEIGHT, WEIGHT_KG/1000 As WEIGHT_T, WEIGHT_KG As Round (WEIGHT * 0.454), CITY From P_B;

Notice incidentally that one could modify BPS so to apply this way of proceeding to the kernel enabled with VAs as well. Perhaps, even with the result as efficient as when, as above, BPS effectively creates VAs.@

---

Figure 5  S-P3 DB. Above: SIRs. Below: CE-views and SRs within the kernel DBS.
Figure 5 illustrates BPS outcome for Ex. 8. We refer to the DB with SIR P of the example as to S-P3. We suppose no VAs, hence the creation of view P as in (18). SIR-layer shows SIRs as rectangles. The sizes reflect the number of tuples and tuple width appearing to the client. The lower part displays SRs and CE-views within the kernel DBS similarly.

3.2 SIR-Layer DDL & DML Processing

The above BPS discussion showed the processing of Create Table R for every SIR R. Likewise, Alter Table R and Drop Table R statements require from BPS more processing than calling their kernel counterparts only. Thus, for every Alter Table R submitted to SIR-layer, BPS has at least to find out in the meta-tables whether R is an SR, perhaps with VAs or a SIR R. For the former, if Alter Table R only alters an SA or a VA, BPS passes the statement to the kernel. E.g., it would be so for Alter Table P adding WEIGHT_KG and WEIGHT_T as VAs to SR P. If in contrast, Alter Table R adds an IE, BPS issues the renaming of R to R_B and the creation of the CE-view R. E.g., it would be so for Alter Table P adding WEIGHT_KG and WEIGHT_T as an IE.

Next, for every SIR R, for every Alter Table R altering only SAs or VAs, BPS first generates the Alter Table R_B statement and Alter View R IE, at least formally. Then, if the kernel DBS supports Alter View, BPS sends down both Alter Table R_B and Alter View R, where the latter alters CE-view R into the new one. For both statements, BPS forms the atomic transaction. If the kernel DBS does not support Alter View, BPS issues Drop View R and Create View R within the transaction instead. Finally, for Drop Table R, BPS either simply forwards the statement to the kernel or, again issues the atomic transaction with Drop Table R_B, followed by Drop View R.

With respect to the SIR-layer processing of DML statements, once BPS created the CE-views, SIR-layer simply sends every SIR-layer query to the kernel as is.

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, [L16] simulates BPS for our example SIRs on MS Access as the kernel. For each SIR, a stored MS Access table is its base. The MS Access stored queries simulate the CE-views. The client may appreciate advantages of SIRs, through queries to CE-views. One may also update these views, e.g., to experiment with every manipulation of SP or P we have discussed. As easy bonus, one may simulate the QBE interface to SIRs, the generation of forms, graphics, etc. In sum, one may play with every nice capability of MS Access that made it the most popular DBS by number of licenses, , by far even, almost as if these capabilities were designed for SIRs as well.

3.3 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 by 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.
4. Related Work

We have shown that 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 could be free of selected VEs. If S-P1 had to provide for the same queries, one would need to add the CE-views S, P and SP we have discussed. But then, every IE in S-P2 was less-procedural than Create View of its CE-view and even QE-view when it applied. The views would be also more procedural to maintain, e.g., as we have shown for each view SP.

The same reasons motivated VAs, already decades ago. In fact, every SR R with VAs may be seen as a specific SIR R. First, IE defines only IAs that could be VAs at some DBS. Next, IE is an lk such that (i) for every IA A, instead of being specified in Select list as: VE As A, one writes A as: A As VE and (iii) there is an implicit From R_B clause. E.g., see Ex. 5.1 again. We did not define above however any rules for such an IE. The current processing of a SIR with VAs only by BPS is thus simply, we recall, to recognize the case and create SIR R only as SR R with VAs, provided the kernel DBS making it possible. This provides the backward compatibility we claimed.

Observe nevertheless also that if one does implement the above rule on some DBS already supporting SIRs as above defined, then the DBS will provide for VAs at SIR-layer, even if the kernel DBS does not. This capability could be of interest to some clients hence we leave the goal for future work. Besides, the current capabilities of every popular DBS with VAs are not all that the research has proposed. Especially, unlike today, at least some forms of VAs could be updatable, [LV86]. Implementing those capabilities would thus naturally profit to SIRs more generally as well.

As we mentioned, our example SIR S-P2.SP is a new type of a universal relation that one may call thus a **universal** SIR. As we also hinted to, the idea, known for decades, was that of a single relation per DB. No query would need then the logical navigation. Through often passionate, although now rather extinct interest in the topic there were various proposals for universal relations, [M4], [V11]. None apparently made to the industry. The only practical outcomes seem optional universal views with all the attributes, but not all the values. The **dangling tuples**, e.g., suppliers in S supplying nothing for the time being, make the latter initial goal usually impossible. CE-view SP is such a universal view. If a universal view R qualifies as CE-view R, the universal SIR R should be thus always less procedural to define and maintain. One may expect a DBA or client naturally more often applying the latter, getting more sometimes simpler queries as well. We leave for future research the rules for the relational design of a DB with SIRs, i.e., so that the DB is possibly best normalized and provided with a universal SIR. The basis seems to be a generalization to SIRs of Heath’s and of Fagin’s decomposition theorems, [H71], [F77], as well as of some proposals for the lossless decomposition through outer joins, [JS90], [DD91].

As one could realize furthermore, if a DBS gets provides with SIR-level as described, SIRs could always remain nevertheless only an optional add-on to any DB designed with SRs and views only. In our example, one could always still stay with S-P1. Switching to SIRs should be safe in this sense. No loss of any current and future capabilities of a relational DB could result from. In particular, every current application could continue to run as well. Finally, it is notorious that the “biblical” S-P1 DB was the mold for most of practical ones. One may thus expect the benefits of SIRs extending to most of practical DBS as well.

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 surrogated of the inherited object that is the one with the primary key equal to. This model characterizes also most of popular DBSs. We should mention however that some, so-called, object-relational DBSs proposed different models in in 90ties 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.
5. Conclusion

Stored and inherited relations, (SIRs), appear useful for every popular SQL DBS. Like dedicated views, SIRs may provide for queries free of logical navigation or of selected value expressions. Such a SIR may be then always less procedural to define or alter than the view. A SIR may also seamlessly integrate virtual attributes (VAs) whenever the DBS provides for those. Finally, the implementation of SIRs on popular DBSs appears easy and with negligible operational overhead.

Future work should start with such an implementation. MySQL seems the best current basis. Besides, our current SQL clauses for creation or altering SIRs are only those necessary for the always lesser procedurality of the inheritance expressions (IEs) with respect to any equivalent views. Additional clauses may sometimes decrease that procedurality even more. We have already mentioned the extension to BPS possibly providing for SIRs with VAs even when the kernel DBS does not support VAs. Hence, this extension provides for the procedurality lower than of any explicit IE necessary in every such case at present. One could also attempt to enlarge the class of implicit IEs without From clause, limited to IAs declared as VAs only at present. A VA supports presently indeed, only scalar functions and arithmetical operators and IAs must source in the base of the SIR only. Next, BPS could create CE-view perhaps more efficient, [GL1], [H1], [V87]. The relational design rules for SIRs we have mentioned appear also a promising goal. Finally, most of major DBSs are now interoperable, [LA86]. Multidatabase SIRs, i.e., with IEs inheriting from several DBs, appear attractive as well.

Acknowledgements. We are grateful to Prof. Emeritus Peter Scheuermann for helpful comments.

References

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