SQL for Stored and Inherited Relations

Witold Litwin
Université Paris-Dauphine PSL
Paris, France
Witold.Litwin@dauphine.fr

Keywords: SQL Databases, Information Systems, Non-procedural Data Definition and Manipulation, Logical Navigation

Abstract: A stored and inherited relation (SIR) is a stored relation (SR) extended with inherited attributes (IAs), calculated as for a view. IAs can make queries to a SIR free of the logical navigation or of selected value expressions without affecting the normal form of the SR. A specific view of the SR can offer the same advantages. The so-called virtual (dynamic, computed…) attributes (columns), possibly extending SRs at some DBSs, can do as well for supported value expressions, being then less procedural to define than every alternate view. We propose extensions to SQL providing for the IAs always less procedurally defined than any alternate view. Likewise, every altering of a SIR leading otherwise to a view altering would be less procedural. Finally, to define IAs instead of virtual attributes would be at most as procedural. We motivate our proposals through the biblical Supplier-Part DB. We show how to implement SIRs with negligible operational overhead. We postulate SIRs standard on every popular SQL DBS.

1. INTRODUCTION

Universally applied Codd’s (relational) model for a Database (Management) System (DBS), [1] & [2] has two constructs: a stored relation, often called base table, 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. Clients or applications provided the stored tuples. The SR definition (scheme) does not allow calculating any of these. A view, also called Inherited Relation (IR), has only the inherited attributes. These get basically only values calculated on-the-fly from SRs or from other views, through a statement of some data definition language (DDL). The statement is basically an SQL Select query, stored within the view scheme. In 1992, we proposed an additional construct, [11]. It was also a 1NF relation, but mixing the stored and the inherited attributes. Examples showed the construct attractive. No further work followed however, to the best of our knowledge.

Below, we refine our proposal specifically 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 an SR. We define the inherited attributes (IAs) basically as usual for views, through some relational or value expression we refer to as to Inheritance Expression (IE). For every SIR R, a single Create Table R defines both the SAs and the IE. The IAs of a SIR may model properties inconvenient as SAs. First, supposing the SR formed from the SAs within the SIR normalized, the latter choice could also adversely impact this normalization. Next, it could imply impractically frequent updates. By addressing SAs and IAs in the same SIR, an SQL query may furthermore totally or partly avoid the logical navigation, otherwise necessary for every equivalent query to the scheme with normalized SRs only. We recall that such navigation occurs whenever a query refers to several relations with, usually, equijoins among. Next, one can define IAs within a SIR through value expressions, letting for SQL queries to the SIR free of these expressions. Altogether, an SQL query to a DB with SIRs should end up usually less procedural (simpler, more usable…) than the equivalent to a DB with normalized SRs only, by the basic measure of fewer characters per query. We recall that clients usually prefer DB languages with less procedural statements. Likewise, we recall that lesser procedurality was the driving force for DBS evolution. In particular, this was the notorious rationale for the evolution from Codasyl to the relational model, given the assertional data definition and manipulation within the latter.

On the other hand, one may observe 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 SA in SIR R with unambiguous proper name, having an IA bearing, at least, the same proper name. We recall that mathematically the same” means the abstraction of the implementation. In our case, whether a value is stored in SIR R or calculated in view R becomes irrelevant. We recall also that in every SQL relation, the attributes are in some order, unlike in a mathematical relation, [3]. View R provides then the same outcome at least for every SQL query to SIR R where the unambiguous proper names above are not prefixed. Actually, one knows such prefixing useless in queries, i.e., the outcome is independent of. We
call every such view R equivalent to SIR R. In fact, the equivalent views are already for decades notorious "escape route" for clients unhappy with the logical navigation or value expressions within the usual queries to normalized SRs only. An equivalent view may in particular be a universal one, providing all the attributes and, possibly, all the values of the DB in one relation, [17]. These views were particularly studied.

We propose extensions to Create Table to accommodate SIRs. Likewise, we propose extensions to Alter Table. The extensions consist of SQL clauses specifically for IAs. We show that, consequently, for every SIR R, IE in Create Table R can be less procedural than Create View R of any equivalent view R. SIR R expanding with IAs some SR, say R_B, provides then for simpler queries to R_B at lower procedural data definition cost. Likewise, it will appear that for every SIR R, altering R can be always less procedural than altering view R. The former is especially advantageous, when it concerns an SA. A highly procedural atomic transaction formed from altering of R_B and of view R is mandatory otherwise. We show finally how to implement SIRs on popular DBSs, with negligible storage and processing overhead. Non-procedural queries being a universal wish, we postulate SIRs defined our way standard on every popular DBS.

We do it especially since some popular DBSs provide unknowingly already for limited SIRs for decades. These are SRs possibly carrying also so-called virtual attributes (VAs) or computed, generated... columns. We recall that one declares a VA as a named value expression in Create Table. Queries avoid the expression by simply referencing the name. The advantage of the whole capability is that for any number of VAs in Create Table, their declarations are altogether always less procedural than any Create View of an equivalent 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 the declarations generalize the gain to every SIR. More specifically, first, for every SIR with solely IAs that could be VAs, our Create Table provides at least for the same gain as with VAs at present. Next, we gain also over every equivalent view with every value expression defining an IA that cannot become a VA, since DBS does not support VAs or the expression is too complex for a VA at present, e.g., contains an aggregate function. Finally, we gain also for every SIR with, in addition or instead, IAs avoiding the logical navigation, as already discussed.

Next section defines SIRs for SQL DBs. We refer to the relational model with SIRs as to SIR model and to SRV model otherwise (SR or View model). We illustrate our proposals through the application to the notorious Supplier-Parts DB. Section 3 discusses the implementation of SIRs over a popular DBS. This seems the most practical approach. We specify an algorithm mapping SIRs into SRs and views there. We analyze the storage and processing overhead of a SIR implemented as proposed. We show it negligible. Section 4 discusses the related work. Section 5 concludes that SIRs should be a standard capability of SQL DBs and proposes future work.

2. SIR MODEL

2.1 Overview

As Figure 1 illustrates, every SIR is a 1NF relation (table), 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. We suppose also for every SIR R that the part formed by all the SAs is by itself a 1NF relation that 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. An easy to see property of every SIR R is that the primary key of R_B is also a key of R. For obvious practical reasons we consider that the former is in fact the primary key of R as well.

As stated already, we suppose that SIRs are SQL relations in practice and so that every notorious SQL naming rule applies to SIRs as well. We consider also a specific rule, namely that for every SIR R, one may qualify every SA A not only as R.A, but also as R_B.A. The latter qualification is the default. The rationale for this rule will appear soon.

Next, for every SIR, the already mentioned IE defines every IA. Values in IA sub-tuples are basically immaterial, as usual for views. IE may also produce null IA sub-tuples for some SIR tuples. As we already mentioned as well, as usual for every relation in practice, we consider below every SIR as an SQL relation. The attribute order matters thus, unlike theoretically for a relation. Here, "SQL" means more precisely the backward compatibility with some popular SQL dialect, e.g., MySQL dialect, referred to as the kernel (dialect). More precisely, we intend every SQL dialect providing for SIRs in the way we define in what follows, to preserve every capability of the kernel. Below, we also refer to every SQL dialect, DB or DBS providing for SIRs as SIR SQL, SIR DB and SIR DBS respectively.

Figure 2 displays a possible structure of a SIR. Each grey rectangle represents a stored sub-tuple. The green rectangles represent the valuated IAs. The white ones
labelled Null represent IAs with nulls. SAs and IAs intermix at the figure.

We define every SIR R operationally through the auxiliary concept of a specific SQL view. Given some SR R, we call that view conceptually expanded view (of) R and denote as C-view R or view R simply. Technically, to declare C-view R, one first renames SR R. The renaming is necessary since no view and an SR may share a name in an SQL DB. We suppose R_B as default new name. C-view R inherits then, on the one hand, every SA A of SR R as R_B.A. It contains furthermore some other IAs, sourced in some SRs or views. Let these be R1…Rk. For every i = 1…, Ri is different of R_B or is an alias of that one, i.e., is declared ‘R_B As Ri’. The characteristic property of every C-view R is finally that, for every tuple t’ of SR R, there is exactly one tuple of view R with t’ as sub-tuple and that view R has no other tuples.

The rationale for C-views is that every C-view R presents every tuple of SR R extended with some or all attributes and values that, for DBA, conceptually characterize it as well. Some of these may result from value expressions over values in SR R. However, none of all these attributes is in SR R. Each one would create indeed some notorious normalization anomaly there. They are therefore within other relations. These are R1…Rk above. For every Ri, it is typically the key that represents all the attributes of Ri in SR R. It is there a foreign key, one calls therefore also sometimes surrogate of Ri. For every Ri, the attributes that DBA selects for inheritance in C-view, expand the surrogate of Ri in C-view R. Every surrogate itself inherits in contrast every of its values in C-view R from SR R.

Every query to SR R and any of those attributes requires the logical navigation through inter-relational joins. By addressing attributes in the same relation, an equivalent query to C-view R may avoid that one. The result, avoiding also selected value expressions, defining some of the C-view R attributes, is notoriously less procedural. We will illustrate it with examples soon. Actually, lesser procedurality notoriously characterizes in this way most of practical queries. Without being named so, C-views avoid in this way the logical navigation and selected value expressions in queries since decades.

We intend SIR R as a single construct merging C-view R and SR R. The intended result is an SQL relation formally equal to C-view R and with the same full source name of every attribute, assuming R_B the source name for every SA in SIR R, as we just did. The only intended difference between SIR R and C-view R is that as a DB relation, Figure 1, SIR R has R_B as the base, i.e., every attribute R_B.A of view R is materialized back in SIR R into SA A of SR R.

Accordingly, we define SIR R through Create Table R of SR R, expanded with the scheme of every IA inherited in view R from R1…Rk. The order of SAs and of IAs in SIR R should be of the IAs in C-view R. The whole statement should appear as constructed through the following steps. Write down, after the declaration: ‘Create Table R As (’, the C-view R scheme, i.e., the entire SQL expression that would follow Select keyword in Create View R. If the scheme includes ‘*’ or R_B.* term, expand the term so that the result contains every attribute of R_B, referred to by its proper name. Also, if relevant, refine the remaining part of C-view to its implicit form we discuss soon. Next, expand every R_B.A to its declaration intended for Create Table R for SR R. Finally, append every clause of that Create Table R eventually remaining. Such clauses declare a multi-attribute primary key, indexing, partitioning...

An alternate view of the Create Table R for SIR R could be as if one constructed it as follows. Start with Create Table R for SR R. Add to the list of attributes every IA intended for C-view R where it would inherit from R1..Rk. The resulting order of attributes in Create Table should be as in view R. Again, if relevant, refine the remaining part of C-view to its implicit form. Finally, insert From… clauses intended for view R after the last attribute of the list.

![Figure 1: SRV-model versus SIR model.](image-url)
view $R$, followed by the From… clauses of the view. Given that for every SIR $R$, the SAs schemes have the same procedurality as for SR $R$, IE of SIR $R$ has strictly lower procedurality than Create View $R$ for C-view $R$, as we hinted to and will illustrate with examples. SIR $R$ becomes consequently more advantageous than SR $R$ and C-view $R$ for the avoidance of the logical navigation or of selected value expressions.

In what follows, we qualify of explicit, every IE with the above sub-list. We denote it as $E$ or $E_R$ for SIR $R$. The refinements we hinted to, define implicit IEs. These are defined differently and introduced in next subsection. Observe that every $E_R$ defines the SQL projection of C-view $R$ on all and only IA that are also IAs in SIR $R$. Observe finally, that while these IAs are always contiguous in $E_R$, they may be separated by SAs in Create Table $R$, as at Figure 2, we recall.

Ex. 1. Recall the ‘biblical’ Supplier-Part DB, often named S-P in short, modelling some suppliers, parts and supplies. Every 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. S-P 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 these rules, S-P molded about the most known, [3]. We refer to it as S-P1. We restate S-P1 into variants with different SIRs. We call S-P2 the variant that follows.

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 original sample data type for every attribute. Actually, the figure shows S-P2 DB. S-P1.S and P are the same SRs as in S-P2. For S-P1.SP, data types are these of S-P2.SP at the figure. The latter is however SIR SP that we present in detail soon. All the SA definitions at the figure skip some practical details, e.g., the data length. We underline the primary key, as usual.

Figure 2: SIR structure as an SQL relation.

Figure 4 shows the original sample data values for S-P1. For S-P1.SP, these are among all those of SIR SP there, according to the attribute names. For the relational algebra, considered by the original S-P1 proposal, the order of attributes in a relation, hence the left-to-right one at the figures does not matter. As known, it does for SQL queries with ‘*’, e.g., Select * From SP. The S-P1 scheme is the optimal one for the discussed application. The notorious relational design criterion it fulfills is the minimal number of SRs free of storage and update (normalization) anomalies, [5].

The well-known drawback of S-P1 is that practical Select queries to SP usually need values from S or P as well. E.g., about every actual client searching for a supply needs the supplier or part name(s). These are evidently conceptual attributes of every supply. However they are not in SP, since the notorious normalization anomaly would make SP losing its BCNF form. Every related query has then 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 hate the logical navigation, feeling it making the queries more procedural than they should be, [17]. The well-know “escape route” for S-P1 is adding the (universal) 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 queries than any other view of SP with fewer attributes or values. To create view SP, one has to rename first SR SP, to, say, SP_B, since every relation in an SQL DB must have 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.$P#$);

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. 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 C-view SP. This is the case. First, view SP inherits bijectively every tuple of SP_B as exactly one sub-tuple and has no other tuples. In particular, (SP_B.$S#$, SP_B.$P#$) is the primary key of SP_B and (SP.$S#$, SP.$P#$) is the one of view SP. The rationale for all these properties is that S.$S#$ and P.$P#$ are also the keys for S and P, respectively. 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 C-view SP, we can define SIR SP as above discussed through the following Create Table SP:

(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 3 shows S-P2 scheme. Figure 4 shows the content of SIR SP that would result for the sample data of S-P1. Every SA is in plain text and every IA in Italics. We suppose the SAs schemes in S-P2.SP these of S-P1.SP, hence of SP_B for C-view SP. These SAs and their tuples form also 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 Create Table SP in (2) above follows entire E SP thus. The list of IAs contiguous in dispersed the SAs within Create Table SP and in in C-view by SIR SB even greater. Same would happen if an IA Create View than the list of attributes it represents. The rationale is that many consider '*' less safe for in SQL, the procedurality of view SP, through SP_B.*. One can define some E_R for every SIR R. For some SIRs, the IE can also be a specific expression that we call implicit and denote as I_R. Every I_R is intended to be less procedural than an E_R could ever be. As it will appear, in three cases, it is an I_R and not any E_R that reaches our already mentioned goal, i.e., of always providing an IE less procedural than any equivalent view.

2.2 Implicit IEs

As said above, the IE 'SNAME…P.P#' for SIR SP is an explicit one that we denoted thus E_SP. One can define some E_R for every SIR R. For some SIRs, the IE can also be a specific expression that we call implicit and denote as I_R or I_R. Every I_R is intended to be less procedural than an E_R could ever be. As it will appear, in three cases, it is an I_R and not any E_R that reaches our already mentioned goal, i.e., of always providing an IE less procedural than any equivalent view.

The reduced procedurality of an I_R may result from new generic character we denote as '#'. In two cases, I_R with '#' may be the only option for IE less procedural than every equivalent view. In first case, view R requiring I_R is a specific C-view R. Otherwise it is a view that we call query equivalent to SIR R, Q-view in short. It is not C-view R, but an equivalent one that still provides in practice for the same non-procedural queries as C-view R, hence SIR R. In practice means here that the query does not (uselessly) prefix unambiguous proper attribute names, as discussed in the Introduction. When Q-view R is a possibility, its advantage may be Create View R even less procedural than an E_R could be, hence less procedural than Create View for C-view R as well. Nevertheless, every Q-view R remains more procedural than possible for I_R, as it will appear.

Finally, I_R may provide backward compatibility with the virtual attributes (VAs), when every IA of SIR R and of C-view R thus, could be a VA and the kernel SQL supports the VAs. Every E_R would be in this case more procedural than SR R with the VAs, although it would be still less procedural than Create View R for C-view R. Actually, as we already said, but in somehow different terms, it is the lesser procedurality of an
SR with VAs than of C-view R with IAs same as the VAs, was the rationale for the VAs and their popularity for decades already.

We suppose that DBS supporting SIRs pre-processes (rewrites) every $I_R$ with '#'. The result is Create Table R with some $E_R$ that we denote as $E'_R$. DBS processes then every Create Table R with $E'_R$ as any Create Table R with some $E_R$. There is basically no pre-processing for an $I_R$ defining VAs. A rule for each case defines the syntax of $I_R$ and the pre-processing or absence of it. We now focus on these rules.

For the first one, recall that for every SIR R, the definition of SAs in Create Table R is the same as in Create Table R_B. Also, most often, for any view scheme, the Select expression either enumerates every IA in Select list or, if some IAs form all the attributes of some relation R and are inherited with the same names and values, then Select may contain the notorious less procedural generic SQL construct X.* instead. As already stated, in both cases, first, every $E_R$ is a proper substring of Create View R, although perhaps distributed within the latter. Consequently, declaring an $E_R$ instead of C-view R, is always less procedural. Using SIRs brings thus this advantage to every DB using C-view at present. The rare and only less procedural generic SQL construct X.* instead.

Indeed, suppose C-view R with the Select expression Select * From R1...R2...R3..., with one of these relations being necessarily non-aliased R_B, e.g., R_B = R1. The least procedural form of $E_R$ is then $E_R = R_2.*, R_3.*...$ From R_B...R2...;: The procedurality of Select list in $E_R$ grows linearly with the number of relations listed. For any C-view R, for some number, likely above eight in practice or for fewer, with long enough proper names, $E_R$ must become more procedural than the C-view, by far even.

Consider now the following rule:

Rule 1. Create Table R contains, after the last SA scheme, $I_R$ in the form: '# From R1...R2...R3...';, with some $I_R = R_B$ or $R_i = R$; $i \geq 1$. Let also $R'1$, $R'2$... be, successively, all the relations among the former that are not a non-aliased R_B. Then, $E'_R$ is:

$E'_R = R'_1.*, R'_2.*, ...$ From R1...R2...;

The terms $R'_1.*, R'_2.*, ...$ that precede R_B in From clause, insert into Create Table R before the first SA scheme. All the others replace #. @

Anyone even only basically familiar with SQL, realizes that if Create View R As (Select * From R1...R2...) defines C-view R, then if the terms in $E'_R$ and R_B.* in their order within From clause were in Select clause of view R, they would constitute simply a more procedural equivalent of '*' in the same time, $E'_R$ is the least procedural $E_R$ in this case. Every other $E_R$ requires explicit enumeration of some IAs. As said, even $E'_R$ can reveal nevertheless necessarily more procedural than the discussed Create View R with '*' only in Select list. In contrast, $I_R$ permitted through Rule 1 is always less procedural than the latter. It is indeed always free of 'Create View R As (Select' and of 'R_B.*, ' substrings, while equal to the remaining one(s).

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, encrypting SP $S$# and SP.P# of every supply. The DBA may furthermore provide the selected clients with the following universal view SP, after renaming SR SP to SP_B, as already discussed. The right join replaces the left one in (1) for the sake of the example.

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

View SP defined so may clearly be C-view SP for SIR SP with base SP_B. Given Rule 1, DBA may define $I_{SP}$ simply as:

(4) $I_{SP} = # From (S Right Join SP_B 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 one declares Create Table SP, DBS applies Rule 1 and pre-processes it using (4) to:

Create Table SP (S.*, $S$# Char, P# Char, Qty Int, SNAME, STATUS, S_CITY, PNAME, COLOR, WEIGHT, P_CITY, P.* From (S Right Join SP_B On SP_B.S#= Enc (S.S#)) Left Join P On SP_B.P#= Enc (P.P#));

The right join replaces the left one in (1) for the sake of the example.

(5) $E'_{SP} = S.*, P.*$ From (S Right Join SP_B On SP_B.S#= Enc (S.S#)) Left Join P On SP_B.P#= Enc (P.P#));

In Create Table SP, S.* term of $E'_{SP}$ precedes all the SAs, since S precedes SP_B in the right join within From clause. P.* replaces #. The list S.*, SP_B.*, P.* would be simply a more procedural expression of '*' in (3) that we spoke about in general terms.

As in general for every $E_R$ and C-view R, $E'_{SP}$ in (5) is also less procedural than Create View SP of C-view SP defining it as $E_{SP}$, i.e., Create View SP As (Select S*, S.P.*, P* From (S Right Join SP_B...);. In fact, one may easily see that (5) remains also less procedural than (3). $I_{SP}$ as in (4) is not thus really necessary here for our goal. However, visibly, it could not be so if relations S and P had instead longer names, e.g., SUPPLIERS and
PARTS_IN_STOCK. This would prove our point that without Rule 1, we could not attain our goal of an IE being always less procedural than the C-view it may replace.

As hinted to, our 2nd case concerns the Q-view. Under some restrictive conditions on the DB, Q-view R may happen to be the same SQL relation as C-view R and SIR R, except for different source name(s) for some unique proper SA name(s) in SIR R. If so, whether an SQL query to C-view R or SIR R, or to Q-view R selects every such an attribute by its full source name in the view or SIR R, or by its proper name only, the result at every popular DBS, will have every such attribute labelled with its proper name only. Every possible result of a query to C-view R or to SIR R may then also result either from the same query to Q-view R or the same query, except that every discussed attribute bears its proper name.

In the same time, Create View for Q-view R may be substantially less procedural than the least procedural for any C-view R. It may become then even less procedural than substantially less procedural than the least procedural for any attribute labelled with its proper name only. Every possible result from the same query to Q-view R or the same query, except that every discussed attribute bears its proper name.

The following rule provides fortunately, in conditions (i) to (iv), for $I_R$ sufficiently non-procedural, as it will appear.

Rule 2. The attribute list in $I_R$ contains terms $R_1.$, $R_2.$,..., where, for every relation name $R_i$, $R_i \neq R$ and $R_i \neq R_B$. Also, every $R_i$ has some attributes $A_{1i}, A_{2i},...$, enumerated here in their SQL order such that for each, there is an attribute of $R_B$ with the same proper name and the SQL order of the latter is that of the former. For each $R_i$, the latter attributes are also consecutive in Create Table $R$ and every term $R_i.$ is consecutive to those. Then and only then, DBS produces $E'_R$ within Create Table $R$ as follows.

1. For every $R_i.$, $E'_R$ lists, in the order in $R_i$, all the attributes of $R_i$ except for every $A_{1i}; A_{2i}$.
2. Create Table $R$ contains all the attributes in $E'_R$ in order that preserves the attribute order for each $R_i$, except that for every $R_i.A_{1i}.., R_B.A_{1i}..$ replaces the former.

The most typical case for an $R_i$ should be a single $R_i$ with all the other attributes of $R_i$ following that one, e.g., as in S-P1.S. Composed $R_i$ preceding all the other attributes, e.g., as in S-P1.SP, should be next typical. The following example illustrates Rule 2.

Ex. 3. Suppose, just for the sake of the example, that $S$ is atypical, namely is $S$ (SNAME, $S.#$, STATUS, CITY). Suppose also the referential integrity between $SP$, $S$ and $P$. DBA of S-P2 can then declare the following SIR, instead of (2), with the advantage of visibly less procedural Create Table:

(6) Create Table SP (S.#, S.#, P.#, Char, Qty Int From $S$, $P$, SP_B Where SP_B.S# = $S$.S# And SP_B.P# = $P$.P#; Primary Key ($S.#$, $P.#$));

Clause Where of (6) is visibly less procedural than the one with outer joints in (2). It is however obviously possible only if the referential integrity is enforced as specified. The IE for scheme (6) contains ‘#’, hence is $I_S$:

(7) $I_S = S.#, P.#$ From $S$, $P$, SP_B Where SP_B.S# = $S$.S# And SP_B.P# = $P$.P# ;

Rule 2 implies the preprocessing to $E'_S$:

(8) $E'_S = \text{SNAME}, \text{STATUS}, \text{S.CITY}, \text{PNAME}, \text{COLOR}, \text{WEIGHT}, \text{P.CITY}, \text{QTY}$ From $S$, $P$, SP_B Where SP_B.S# = S.S# And SP_B.P# = $P$.P#;

Finally, Rule 2 implies:

(9) Create Table SP (SNAME, S.#, STATUS, S.CITY, P# Char, P.#, PNAME, PNAME, COLOR, WEIGHT, P.CITY, Qty Int From $S$, $P$, SP_B Where SP_B.S# = S.S# And SP_B.P# = P.P#, Primary Key ($S.#$, P#));

If DBA considered C-view SP instead, the least procedural one would be:

(10) Create View SP As (SNAME, SP_B.S#, STATUS, S.CITY, SP_B.P#, PNAME, COLOR, WEIGHT, P.CITY, QTY From $S$, $P$, SP_B Where SP_B.S# = S.S# And SP_B.P# = P.P#);

(10) is clearly more procedural than (8), hence would not be a practical alternative. However, DBA can be interested only in the freedom from the logical navigation for practical queries, i.e., those where no unique proper attribute name is (uselessly) prefixed with source name. E.g., for (10) a query Select $S$# From SP Where SNAME = 'Smith' would be a practical one, while Select SP_B.S# From SP... would not. The following view would make then more sense for the DBA than (10):

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

Indeed, view SP defined by (11) would, first, be visibly less procedural than (10), taking advantage of ‘*.’ It is however not some C-view SP. The full source names of
attributes $S_\#$ and $P_\#$ are indeed $S.S_\#$ and $P.P_\#$, unlike in (10) and unlike it should be in any $C$-view $SP$. The DBA can nevertheless realize that (11) respects conditions (i) to (iv) we stated for every $Q$-view. We have in particular $R_1 = S$, $A_{11} = S_\#$, $R_2 = P$ and $A_{12} = P_\#$. Then, with the referential integrity enforced and only then, view $SP$ (11) respects (iii), since $S_\#$ and $P_\#$ are the keys of respectively $S$ and of $P$. Finally, the respect of (iv) is obvious.

It follows that view $SP$ (11) is $Q$-view $SP$. As desired, every query $Q$ to this view provides then effectively the same outcome whether $Q$ uses the full source names for $S_\#$ or $P_\#$ in (11) or not. If not, for every $Q$, the outcome would be the same as if $Q$ addressed $C$-view $SP$ (10). For the latter, this outcome would be also the same as for $S_\#$ and $P_\#$ prefixed with $SP_B$. Altogether thus, the DBA can effectively substitute $Q$-view $SP$ (11) to $C$-view $SP$ (10) and, in fact, to any $C$-view $SP$ equivalent to (10), as one can easily realize.

Create View $SP$ (11) appears furthermore visibly less procedural than $E_{SP}'(8)$. Thus, if $E_{SP}$ was the only choice for our $DBA$, choosing $Q$-view $SP$ would be more practical than choosing $SIR$ $SP$ defined by (9). This conclusion extends to any $E_{SP}$, as visibly none can take advantage of ‘*’. With respect to $I_{SP}$ (9) however Create View $SP$ (11) remains visibly more procedural, by about 20%. It would be so also visibly for any equivalent variant of (11). Hence, choosing $SIR$ $SP$ would still always remain more practical.@

One may easily verify from the example that lower procedurality of $I_{SP}$ generalizes to any multi-attribute $A$, common to some $R_1$ and $R_B$. It also 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. Observe finally, that there is no $Q$-view $R$ less procedural than $C$-view $R$, hence, perhaps, than $E_R$ as well, other than when Rule 2 suffices.

Our last case is that of the kernel SQL providing for the already mentioned SRs with VAs, e.g., MySQL or SQL Server SQL. We recall that one declares at present a VA $A$ generally with the term that we call VA-term. The generic form of a VA-clause is: $A$ As $V$, e.g., for SQL Server. Here, $V$ designates an SQL value expression (VE) permitted for the kernel’s statements, the latter necessarily supporting VAs. Then, one should simply preprocess (rewrite) every $V$ As $A$ term into the VA-term of the kernel.

The resulting Create Table $R$ statement is the kernel’s one. The expected result for SIR DBS is creation of SR $R$ with VAs, as it would be for the kernel SQL over a current DBS.

Every IE subject to Rule 3, is some $I_R$, as $A$ does not have any From… clauses, mandatory for every $E_R$. We denote every such $I_R$ as $I'_{R}$. It is easy to see that for every $I'_{R}$, $E'_{R}$ simply is:

$$E'_{R} = I'_{R} \text{ From } R_B;$$

Therefore, the following rule could also serve, instead of Rule 3:

Rule 3’. For every Create Table $R$ with $I'_{R}$, append From $R_B$ clause to last SA or IA.

Observe that $E'_{R}$ is in fact $E_{R}$ of SIR $R$ one could create instead of $SR_{R_B}$ and $C$-view $R$, whenever the latter is formally the same relation as $SR$ $R$ with VAs. Such view is always possible, but, also always, more procedural to create, hence against the practical interest, we recall.

Regardless of the rule applied, every SIR $R$ defined through $I'_{R}$ would be the same as $SA$ $R$ with VAs, although every computed value would be called VA for the kernel SQL and IA for SIR $R$. If the VA-terms that Rule 3 produces are the generic ones, than every $I'_{R}$ is obviously equally procedural as the VAs it defines. In other words, SIR $R$ is then equally procedural to create as is SR $R$ with VAs. Otherwise, e.g., for MySQL server, any $I'_{R}$ is less procedural than VAs it produces. In other words, every SIR $R$ is then even less procedural than SR $R$ with VAs.

Furthermore, $E'_{R}$ is visibly always more procedural than $I'_{R}$. If $E_{R}$ was the only possibility for SIR $R$, then SR $R$ with the generic VAs would remain always less procedural than SIR $R$. It is only $I'_{R}$ that through Rule 3 or Rule 3’ allowed us to claim in the Introduction about the same or lesser procedurality of SIR $R$. Likewise, it allowed us to claim that every SR $R$ with VAs is in fact, unknowingly of course, a specific SIR $R$, where one expresses simply differently every IA since decades.

If the kernel SQL does not provide for VAs, like, e.g., MS Access, or semantic of some $V$ is not allowed by the kernel, then Rule 3 does not apply. We do not destine Rule 3’ at present for operational use hence $E_{R}$ is the only option. One rationale is that whenever Rule 3 applies, it requires evidently less processing. Also, even if Rule 3’ applied, the procedurality gain it could provide would be beyond our present goal. We leave the subject for future work.

Indeed, as the example that follows illustrates, Rule 3’ could serve when the kernel does not provide for VAs or when VAs cannot use some $V$. For instance, no VA can use any aggregates, as far as we know. In all such cases, Rule 3’ would provide for the additional procedurality gain over $E_{R}$, i.e., the gain of ‘From R_B’. However, every $E_{R}$ under consideration already gains over the $C$-view $R$. Our present goal is any gains,
provided there are some. The additional gain through Rule 3′ would be thus a "cherry on the cake".

Ex. 4. Suppose that S-P2.P.WEIGHT provides for the weight of every part in pounds, while the clients should also know the weight in KG. This, as the attribute named WEIGHT_KG, succeeding to WEIGHT in P.

1. Suppose MS Access as the kernel dialect. Rule 3 implies that \( E_P \) is the only option, e.g.,
\[
(12) \quad E_P = \text{Round} (\text{WEIGHT} \cdot 0.454, 3) \quad \text{AS WEIGHT_KG}
\]
From P;

The Select list of \( E_P \), i.e., WEIGHT_KG scheme, should be in Create Table P immediately after SA WEIGHT scheme. From P clause in (12) should follow SA CITY. As claimed in the Introduction, it is easy to see that \( E_P \) is less procedural than would be any Create View for C-view P.

2. Suppose now S-P2 on SQL Server. WEIGHT_KG could be a VA. Rule 3 allows declaring WEIGHT_KG as:
\[
\text{Round (WEIGHT } \cdot 0.454, 3\text{) As WEIGHT_KG}.
\]
This declaration constitutes \( I_P^P \). It is clearly even less procedural than (12). If SQL Server SQL is the kernel, Rule 3 would preprocess it to VA-term:
\[
(13) \quad \text{WEIGHT_KG As Round (WEIGHT } \cdot 0.454, 3\text{)}
\]
For SQL Server, Create Table P with (13) would mean then creating SR P with SAs as at Figure 3 and with VA WEIGHT_KG defined by (13). Both clauses are obviously equally procedural. For MySQL, the preprocessing by Rule 3 would transparently add parentheses around V in 13. IA would end up slightly less procedural.

Next, \( E_P \) (12) could be \( E_P' \) here. To define (12) instead of \( I_P^P \) when Rule 3 provides for the latter, always remains an option, nevertheless. It does not make practical sense however, obviously.

3. Suppose now still for S-P2 at SQL Server SQL as the kernel, that P should be created not only with WEIGHT_KG for every part, but, also with the total weight of the supplies of this part. E.g., in order to foresee the requirements on the warehouse with the supplies. That weight should be computed as the last attribute of P, named T_QTY.

VE for T_QTY needs an aggregate function. T_QTY cannot be then a VA for SQL Server. Rule 3 prohibits then now also for WEIGHT_KG to be a VA in the resulting SIR P. Some \( E_P \) is the only option. With WEIGHT_KG defined by (12) thus, one may then create T_QTY in Create Table P after CITY through the following \( E_P \):
\[
E_P = \text{WEIGHT_KG} \ldots \text{T_QTY AS WEIgHT } \cdot (\text{Select Sum (QTY) From SP Where SP.p# = SP.p#}) \quad \text{FROM P_B}.
\]
To declare \( I_P^P \) with WEIGHT_KG and T_QTY for any SQL kernel at present, one would need Rule 3’. FROM P_B would be the procedurality gain. T_WEIGHT scheme would end up as non-procedural as a VA of SQL Server. For Rule 3, \( E_P \) is the only issue.@

Summing up, Rule 3 makes creation of SIR R at most as procedural as that of SR R with the VAs being all and only IAs of SIR R. Whenever \( E_R \) is the only choice, it remains still also always less procedural than Create View R for C-view R. All three examples illustrate finally that for every SIR, there is always a definition of the IE at most as procedural as some SQL capability available at present for the same goal.

### 2.3 DDL Statements for SIR Model

We already discussed Create Table for SIRs extensively. We now focus on the other SQL DDL statement for SIRs. We continue supposing every such statement backward compatible with some (kernel) dialect. E.g., for MySQL SQL as the kernel, we suppose Create View for SIRs, i.e., of SIR SQL, being simply the MySQL Create View, except that among source relations could be a SIR. We suppose similarly for SQL Server as the kernel 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 R 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 IA intended as VA or may be followed by optional First and After keywords specifying how the added SA mixes with the existing SA and VAs. Also, one Alter Table may alter several attributes, unlike for SQL standard. On the other hand, for every kernel, Alter Table R for R that is an SR 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. Every IE-clause defines new IE replacing an existing one, if any. It acts thus similarly to every Select expression in any Alter View at present, replacing the existing view scheme. IE-clause is finally mutually exclusive with the existence of IAs defined as VAs.

The IE-clause defines the IE and, necessarily, the placement of each IA among all the SAs. The latter are defined by Create Table and, perhaps, successive Alter Table statements, including the one with the IE-clause. The IE-clause for SIR R may define all this in the terms of C-view R after the Select keyword. As for \( I_R \), the IE-clause may alternatively contain instead of some or even all such terms the generic character '#', or terms Ri.#, where Ri ≠ R_B and Ri ≠ R. As for an IE, the rationale is to have IE-clause less procedural even when C-view or Q-view definition takes advantages of '*' we have discussed. Every IE-clause with '#', is preprocessed to as \( I_R \) with would be, with however the additional insert(s) by name or as R_B.*, of every SA into the list of the attributes resulting from the preprocessing. Thus, for sole '#', there is the
additional insert of R_B.* at the position determined by R_B or R in From clause. The additional inserts for every term Ri.# are the (unique) names of every SA with the proper name of some Aj in Ri. The position of each insert is determined by that of Aj among the attributes of Ri, as we discussed for Rule 2.

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. Then, if Alter Table drops, adds or renames any SAs, new IE clause is optional. Like would be optional the Alter View R statement for C-view R resulting from Alter Table R_B with the same alterations of SAs. Next, for any SIR R, we prohibit to drop all SAs, as usual for every alteration of every SR R, besides. In particular, we prohibit thus for every SIR R, any alterations into a view instead. If such need occurs, one should use Drop Table R followed by Create Table R. Likewise, if view R should evolve to SIR R, we presume Drop View R followed by Create Table R. These procedures are obviously the simplest to put into practice.

For Drop Table R, we simply consider it applying also to every SIR R. As usual, one should not violate the referential integrity. Likewise, the statement may cascade or may get refused. Notice that dropping SIR R is substantially less procedural than dropping SR R_B with any equivalent view R. Single Drop Table R replaces indeed here an atomic transaction.

Next, we suppose Alter View and Drop View as in the kernel. Observe then that dropping the IE in SIR R is consequently substantially less procedural than dropping C-view R. The former requires indeed only Alter Table R Drop IE statement, The latter requires a multi-statement atomic transaction with Drop View R and Alter Table R_B Rename to R. Interestingly, observe that, in contrast, to drop VAs, may be more procedural than to drop C-view. One has to drop those individually indeed, in every kernel’s Alter Table we are aware of. Still, the drawback apparently did not affect the popularity of VAs. We thus can hope the same for the SIRs. Otherwise, the additional statement Drop IE R could obviously solve the issue.

Finally, we suppose for Create Index for SAs or IAs the syntax of the kernel one for SAs, VAs and views.

Ex. 5. DBA adds to S-P2.P the IA WEIGHT_KG from Ex. 4. S/he also adds WEIGHT_T converting WEIGHT_KG further to tons. For application dependent reasons, WEIGHT_T should precede WEIGHT_KG.

1. MySQL is the SQL kernel dialect for SIRs.

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

Both IA schemes are so since these IAs could be VAs. As the result, Alter modifies SR P into SIR P that, e.g., on MySQL, could be S-P1.P with two VAs added. By not needing parentheses around the value expressions, (14) is (slightly but still) less procedural than the similar altering adding VAs WEIGHT_T and WEIGHT_KG directly under MySQL.

2. The SQL dialect for SIRs does not have VAs, e.g., MS Access.

(15) 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);

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

3. The DBA from (2) above decides to drop WEIGHT_T.

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

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

(18) 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#);

Suppose now that DBA rather prefers to create SIR SP as in Ex. 2. IE-clause would be then as I_{SP} in (4). The preprocessing would rewrite it to:

IE (S.*, S#, P#, Qty, SNAME, STATUS, S.CITY, PNAME, COLOR, WEIGHT, P.CITY, P.* From (S Right Join SP_B On SP_B.S# = Enc(S.S#)) Left Join P On SP_B.P# = Enc(P.P#), Primary Key (S#, P#));

Likewise, for the alteration to SP from Ex. 4, IE-clause would be defined as in (9). The preprocessing would insert SA names, making IE-clause like in (10) without the data type declarations.

Altering SR P to SIR P as in (15) is (slightly, but still) less procedural than Create View P for any equivalent view P, C-view P, in particular (why?). Likewise, the alteration (16) is visibly less procedural than (17). The difference increases if one uses Drop View P followed by Create View P instead of (17), e.g., for MsAccess kernel. Likewise, altering SR SP to SIR SP as in (18), is visibly less procedural than Create View SP for any equivalent view SP or C-view SP. In fact, the actual view creation should be typically even more procedural by far. The reason is that since the view should be named as the existing SR, SQL requires first to rename the SR. This needs
one more statement with its procedurality adding on. Furthermore, to avoid any run-time error for a client, both statements should typically be an atomic transaction. That one requires further SQL statements. The following example details the point for SP. The case of P is similar. An atomic transaction is likewise needed for Drop View followed by Create View above discussed.

Ex. 6 Consider again S-P1.SP becoming either SIR S-P2.SP or C-view SP. For the former, the single Alter SP statement (18) suffices. To create the C-view SP in contrast, one has to first rename SP into SP.B. This costs one Alter SP Rename To SP_P statement. Then, one has to formulate the already mentioned Create View SP as in (1). For the atomicity, SQL Begin Transaction and Commit brackets are necessary. Likewise, SQL Error Code tests for Commit or Rollback should follow every DDL statement. All this leads to several SQL statements (how many?). The result is clearly several times more procedural than (18).@

Similar savings occur for any equivalent view SP. It is also so for SIR SP variant (6) and Q-view SP (7).

Finally, SA name change, SA addition or deletion leads to similar advantages of SIRs. E.g., work out the shortening of SP.B.QTY to Q, (i) for S-P2.SP and C-view SP and (ii) for SP variant (6) and its Q-view SP.

Our examples obviously generalize to every SIR. It should be clear thus that to alter any SR R to SIR R, should be always several times less procedural than renaming every SR R to R.B and creating C-view R or Q-view R. Obviously, another renaming does not change the outcome. Next, for every SIR R, altering an IA A through IE-clause, should be always less procedural than altering A in C-view R or Q-view R. In the same time, it should be also clear, especially from the exercise on QTY, that altering an SA of SIR R should be always several times simpler than altering R.B.A and C-view R or dropping and recreating view R instead. Finally, every altering of SIR R with IAs preprocessed to VAs, by adding, modifying or dropping such IAs, is equally or less procedural that the same operation on SR R with these VAs today.

2.4 Data Manipulation for SIRs

As for DDL, we presume for every DBS supporting SIRs that the syntax of every DML statement (query) for SIRs is backward compatible with the kernel SQL dialect. The only operational difference is that a name in the statement may refer to a SIR or its base. With respect to query semantics then, we consider for every query Q referring to any SIR R that the outcome of Q is that of Q addressing C-view R instead. If Q refers to R.B, the outcome is that of R_B being the stand-alone SR for C-view R. Every update query Q addressing SIR R is accordingly valid (executable) only if C-view R is updatable by Q. In practice, the validity of equivalent Q’s may depend on the kernel DBS, [D4]. The constraint may impair even Q updating IAs of R only. Q may refer then to R.B, being valid iff valid for R.B as the stand-alone SR for C-view R. The following example illustrates and motivates all these proposals.

Ex. 7. The simplest select query: Select * From SP to S-P2 would show all the SP values, of all SAs and of all IAs in Figure 4. The attribute order would be the same, but not necessarily the tuple order, we recall. Suppose now MS Access dialect as the kernel. The update query Q = Insert SP (select ‘S4’ as [S#], ‘P4’ as [P#], 100 as QTY); would add one tuple with these values and, formally, all the IA values that every query to C-view SP selecting * where S# = S4 would show afterwards. If Q addressed C-view SP, the update would propagate to SR SP_B. Next, Q; Update SP Set QTY = 250 where S# = ‘S1’ and P# = ‘P1’; would update one QTY value in SP. Same Q to C-view SP would propagate to SP_B as well.

Then, for S-P2, Q: Update SP set QTY = 250, S.CITY = ‘Paris’ where S# = ‘S1’ and P# = ‘P1’; would accordingly update the tuple. But, since SP.S.CITY is an IA, Q would propagate the new CITY also to every other supply by S1 there, perhaps surprisingly. It would be so indeed for Q and C-view SP. The changed CITY would also propagate to S.B.CITY, besides. Next, Q = Insert SP (select ‘S4’ as [S#], ‘P4’ as [P#], 100 as QTY, S.CITY as ‘Rome’); would change CITY value to Rome in every SP tuple with S# = S4. Again, since it would be so in C-view SP. Likewise, every update to WEIGHT_KG in SIR SP would fail. Finally, every Delete…From SP Where… would fail in S-P2. It would fail indeed for C-view SP under MS Access 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 Where…….

SQL Server as kernel would make updates to S-P2.SP even more restrictive. An SQL Server view is updatable only if it inherits from a single SR, unlike C-view SP thus. Under SQL Server kernel accordingly, S-P2 client would need to reformat every Q above towards S, P or SP_B. MySQL is less restrictive than SQL Server. Like for MS Access, views over multiple tables accept some updates. In particular, MySQL kernel would accept every successful update above.

3. IMPLEMENTING SIRs

3.1 Basic Processing Scheme

As already said, the most practical way towards a SIR DB seems the reuse of a popular SQL DBS as the kernel DBS
with its SQL as the kernel dialect. One way is to create the SIR-layer, managing SIR SQL DDL and DML statements through the calls for the kernel services, Figure 5. For the kernel DBS, SIR-layer appears then as any clients.

In particular, for the Create Table R statement received, SIR-layer determines first the relation to create. If R is an SR, SIR-layer forwards the statement as is. In turn, the processing must be more involved for every SIR R, except for R with VAs. First the former 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 basically represent every such SIR R in the kernel as the stand-alone SR R_B and C-view R. SIR-layer simply forwards then every query Q as is to the kernel. This one processes Q either towards view R or towards R_B only. Only for every SIR R with VAs, hence for the kernel with VAs as well, the simplest design, implied actually by Rule 3, appears that SIR-layer simply forwards Create Table R received. The kernel creates SR R with VAs accordingly.

We qualify of basic (processing) scheme, (BPS), the SIR-layer algorithmic for SIRs implemented as above defined. Thus, for Create Table R for SIR R in every case other than applying Rule 3, BPS always starts with the conversion of I_R, if there is any into E'_R. Next, BPS passes Create Table R_B statement to the kernel DBS, using for that all and only SAs of Create Table R. Then BPS creates the C-view simply as follows. Let A_1,...,A_m be the list of the names of every SA and IA in Select list of E'_R, in the original order. Then, BPS simply issues to the kernel the following statement, with From, Where etc. clauses of E'_R:

Create View R As (Select A_1,...,A_m From...Where...) Ex. 8. (1) We submit to SIR-layer S-P2 scheme at Figure 3. BPS finds no IEs in Create Table S and Create Table P. It passes each statement to the kernel that creates each SR. BPS determines that Create Table SP in contrast defines E_SP we discussed. If BPS found any of I_SP we discussed, it would eventually pre-process it to E'_SP. For E_SP, BPS issues the following two statements to the kernel DBS. We systematically omit below the statements making an atomic transaction from the presented ones, obviously necessary.

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

Create View SP As (...; /* Statement (1).

We leave as exercise the variants for each I_SP already discussed.

(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 (13) and (14). BPS forwards Create Table P from SIR-layer as is to the kernel DBS. The result is SR P with VAs.

(3) Suppose that the kernel dialect does not support VAs. Create Table P for SIR P may only define both IAs as for a view, i.e., through (12) for WEIGHT_KG and similarly for WEIGHT_T. BPS generates two statements for the kernel:

Create Table P_B.../* With attributes of P at Figure 3.

(19) 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; @

Figure 5 illustrates BPS outcome for Ex. 8. We refer to the DB as to S-P3. SIR-layer shows SIRs as rectangles. Each size reflects the number of tuples and tuple width appearing to the client. The lower part displays SRs and C-views within the kernel DBS similarly.

3.2 BPS of other DDL & of DML Statements

Alter Table R and Drop Table R for SIR-layer also require from BPS more processing than calling their kernel counterparts only. For every SIR R, each statement requires in fact the atomic transaction that DBA should formulate to R_B and C-view R instead. We recall from Section 2.3 that the latter is always more procedural than the former, usually several times. In more detail, for every Alter Table R at SIR-layer, BPS has first 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 has the IE-clause, BPS issues the renaming of R to R_B and the creation of the C-view R. E.g., it would be so for Alter Table P adding WEIGHT_KG and WEIGHT_T as an IE.

For every SIR R in contrast without VAs, every Alter Table R altering only SAs makes BPS issuing Alter Table R_B statement. If the kernel DBS supports Alter View, BPS generates also Alter View R, addressing C-view R of course. The latter takes to the account the IE-clause produced by the preprocessing if there is any. We skip easy details of Alter View generation. BPS then sends down the atomic transaction with both statements. If the kernel DBS does not provide Alter View, BPS issues the transaction with Drop View R followed by Create View R instead. If Alter Table R alters IE-clause only, BPS generates similarly only Alter View R or Drop View R followed by Create View R. Finally, if Alter Table R alters an SA and IE-clause, BPS generates the atomic transaction with either
As motivating example, spell out BPS outcome for Alter Table SP for Ex. 6 and its follow up in Section 2.3.

Next, for every Drop Table R, BPS either simply forwards the statement to the kernel or, again issues the atomic transaction with Drop View R followed by Drop Table R_B. Finally, we recall, for every SIR-layer DML statement, BPS simply sends it to the kernel as is.

One should obviously implement BPS in some host language calling the Embedded SQL of the kernel. This implementation is a future work. In the meantime, [13] 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 C-views. The client may appreciate advantages of SIRs, through queries to C-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, almost as if these were designed for SIRs. We recall that, by number of licenses, those capabilities made MsAccess by far the most popular DBS at present.

3.3 Operational Overhead of SIR-layer

The kernel storage for every SIR data is in practice the one for the base data only. C-view storage should be obviously always negligible. The storage for the SIR-layer metatables should be clearly larger. But, it should remain still typically negligible with respect to the data storage. Altogether, the storage for a SIR DB should be only negligibly greater than that required by the DB with the SIR bases as stand-alone SRs only or with C-views or Q-views in addition.

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 SIR R costs the same as the processing of the same query to C-view R or to R_B. Hence, the SIR-layer overhead through BPS has no incidence on the query evaluation in practice. Altogether, perhaps surprisingly, the enticing capabilities of SIRs should come almost without the operational overhead.

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 should be free of logical navigation or with reduced one, or could be free of selected VEs. If S-P1 should provide for the same queries, one would need to add the C-views or Q-view we have discussed. But then, every IE in S-P2 was less-procedural than Create View of its C-view or Q-view. As shown, the views would be also more procedural to maintain.

As already mentioned, same rationale already motivated VAs, decades ago. As discussed also, every SR with VAs is a specific SIR R. SIRs generalize thus the old rationale for VAs to SRs with IAs too complex to be VAs at present, e.g., T_QTY, or to those helping with the logical navigation. The rationale for VAs proved appreciated, since VAs are now popular for decades. We may thus reasonably hope SIRs becoming popular as well.

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, [12]. Implementing those capabilities could perhaps 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 needs then the logical navigation. Through often passionate, although now rather extinct interest in the topic there were various proposals for universal relations, [16], [20]. 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 goal usually impossible.

C-view SP is a universal view. If a universal view R qualifies as C-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 often 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, [9], [6], as well as of some proposals for the lossless decomposition through outer joins, [15], [4].

As one could realize as well, if a DBS gets provides with SIR-level as described, SIRs could always remain an optional add-on to any DB designed with SRs and views only. In our example, one could always still stay with S-P1. Implementing SIRs as proposed should be always 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. Also, 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, observe from the example that the inheritance model for IEs is the original one of the relational model. As we hinted to, 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 DBSs. We recall however that some, so-called, *object-relational* DBSs proposed different models in 90ties. The open-source Postgres DBS is the most prominent survivor of this trend, [19], [18]. 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) as we have defined those, appear useful for every popular SQL DBS. Like a C-view and Q-view, a SIR may provide queries free of logical navigation or of selected value expressions. The SIR may be then always less procedural to define or alter than any equivalent view. A SIR may also seamlessly integrate virtual attributes (VA) when the kernel DBS provides those. Finally, the implementation of SIRs on popular DBSs appears easy and with negligible operational overhead.

Future work should obviously start with the implementation. MySQL seems the best kernel. Besides, our currently proposed SIR SQL clauses for creation or altering SIRs aim only on the always lesser procedurality of an inheritance expressions (IE) with respect to every equivalent view. Additional clauses could decrease the procedurality further. One possibility is lifting the restriction on Rule 3’ we have outlined. The relational design rules for SIRs we have mentioned appear also a promising goal. Next, BPS could perhaps create more efficient C-views, [7], [8], [14], [21]. Finally, most of major DBSs are now interoperable, [10]. Multidatabase SIRs, i.e., where some IEs inherit from several DBs, appear attractive as well.

ACKNOWLEDGMENTS

We are grateful to Prof. Emeritus Peter Scheuermann.

6. REFERENCES

**S-P2 Scheme**

<table>
<thead>
<tr>
<th>Table S</th>
<th>Table P</th>
<th>Table SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>S# Char, P# Char, SNAME Char, STATUS Int, CITY Char;</td>
<td>S# Char, P# Char, SNAME Char, STATUS Int, CITY Char;</td>
<td>S# Char, P# Char, SNAME Char, STATUS Int, CITY Char;</td>
</tr>
<tr>
<td>WEIGHT Char,</td>
<td>WEIGHT Char,</td>
<td>WEIGHT Char,</td>
</tr>
<tr>
<td>CITY Char;</td>
<td>CITY Char;</td>
<td>CITY Char;</td>
</tr>
<tr>
<td>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#));</td>
<td>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#));</td>
<td>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#));</td>
</tr>
</tbody>
</table>

**Figure 3: S-P1 and S-P2 schemes.**

**S-P2 Content**

<table>
<thead>
<tr>
<th>Table S</th>
<th>Table P</th>
<th>Table SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$ Smith 20 London</td>
<td>$P_1$ Nut Red 12 London</td>
<td>$S_1$ P1 300 SNAME Status 20 London S.CITY Nut Red 12 London</td>
</tr>
<tr>
<td>$S_2$ Jones 10 Paris</td>
<td>$P_2$ Bolt Green 17 Paris</td>
<td>$S_2$ P2 300 SNAME Status 10 Paris S.CITY Bolt Green 17 Paris</td>
</tr>
<tr>
<td>$S_3$ Blake 30 Paris</td>
<td>$P_3$ Screw Blue 17 Oslo</td>
<td>$S_3$ P3 400 SNAME Status 30 Paris S.CITY Screw Blue 17 Oslo</td>
</tr>
<tr>
<td>$S_4$ Clark 20 London</td>
<td>$P_4$ Screw Red 17 London</td>
<td>$S_4$ P4 300 SNAME Status 20 London S.CITY Screw Red 17 London</td>
</tr>
<tr>
<td>$S_5$ Adams 30 Athens</td>
<td>$P_5$ Cam Blue 12 Paris</td>
<td>$S_5$ P5 400 SNAME Status 30 Athens S.CITY Cam Blue 12 Paris</td>
</tr>
</tbody>
</table>

**Figure 4: S-P2 content. IA (proper) names and values are in *italics*.**

**Figure 5: S-P3 DB. Above: SIRs. Below: C-views and SRs within the kernel DBS**