Stored and Inherited Relations for SQL Databases

Witold Litwin

(March 2016)

Abstract. A stored and inherited relation (SIR) is a stored relation (SR) with additional inherited attributes, (IAs). SIRs can make queries less procedural than SRs only, without impacting the normal forms. 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. Likewise, altering a SIR is also always less procedural. Finally, our extensions provide backward compatibility with virtual (dynamic, computed...) attributes (columns), available at some popular DBSs. The latter already avoid selected value expressions to queries, while being also always less procedural to define or alter than the equivalent view. 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 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 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 a view, 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 has to refer to attributes in several relations with, usually, equijoin clauses among those relations then. Next, one can define IAs within a SIR through value expressions, letting for SQL queries to the SIR free of these 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 per query.

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, [D4]. 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

1 Université Paris-Dauphine, PSL Research University, CNRS, LAMSADE, 75016 Paris, France. witold.litwin@dauphine.fr
2 LAMSADE Research E-Report. Latest update October 1, 2018
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, [MUV84]. 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 for every SIR R, our clauses defining the IAs in Create Table R 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 with IAs some SR, say R_B, may thus provide simpler queries to R_B at lower procedural data definition cost than every equivalent view R. It will appear also that altering SIR R is always less procedural than to alter or create a view R. The gain is especially substantial when the latter operation follows altering of every SA the view inherits from that alternatively becomes an SA of SIR R. We show finally how to implement SIRs on popular DBSs, with negligible storage and processing overhead. The wish of non-procedural queries being universal, we postulate SIRs defined our way standard on every SQL 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 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. Next, we gain also for every an equivalent view with every value expression defining an IA that cannot become however a VA, since DBS does not support those or the expression is too complex for any VA at present, e.g., contains an aggregate function. Finally, we gain for SIRs with not only with IAs defined through value expressions, but also, perhaps, with 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. For every SIR R, we qualify the stored sub-relation, formed by all the SAs, 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. A fundamental property of every SIR R is that the primary key of R_B is furthermore also a key of R.

Next, for every SIR, the already mentioned IE defines every IA. Values in IA sub-tuples are basically immaterial, as usual for views as well. IE may also produce null IA sub-tuples for some SIR tuples.
As we already mentioned as well, we consider furthermore below every SIR as an SQL relation, i.e., where the attribute order matters. Here, "SQL" means the backward compatibility with some popular SQL dialect, e.g., MySQL dialect in practice. Figure 2 displays a possible structure of a SIR. Each grey rectangle represents a stored sub-tuple. The green rectangles and the white ones labelled Null, represent the IAs, valuated or nulls. SAs and IAs intermix at the figure.

We define every SIR R operationally through the auxiliary concept of a specific SQL view, named view R and called conceptually equal to SIR R, CE-view R, in short. To declare CE-view R, we first suppose existence of some SR R_B, equal to the base of SIR R, including the proper attribute names. Then, CE-view R (i) is mathematically the same SQL relation as SIR R, (ii) inherits every IA that is also an IA in SIR R under the same full source name and (iii) inherits every IA that is an SA in SIR R from R_B. We recall, that condition (i) means that the only difference between SIR R and view R as an SQL relation, (hence where the order of attributes matters), is that every value stored in SIR R is an inherited one in view R. Notice finally that if R_B.A1,...,Ak form the primary key, then R.A1,...,Ak is a key for CE-view R as well, analogously to SIR R thus.

Given all this, we define every SIR R through an extended Create Table R basically in three steps. Basically, means here some refinements later. (1) Start the SIR R scheme within Create Table with the scheme in Create View R for CE-view R, i.e., as the SQL expression after Select keyword in Create View R. (2) Materialize every IA A of view R that should be SA A in SIR R as A is within Create Table R_B. (3) Finally, append every table option that eventually is in Create Table R_B as well, i.e., all those for the primary key, referential integrity, indexing...

We naturally suppose for every SIR scheme defined as outlined that every notorious SQL naming rule applies. 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. One rationale is our reuse of CE-view R scheme. All the clauses From... within view R referring to R_B, remain valid despite the absence of stand-alone SR R_B in the DB with the SIR. A less obvious rationale is that referring to R_B rather than to R from some other SIR R', when R inherits an IA from R', hence refers to R', may avoid the circular referencing between R and R'.
that we prohibit. The latter is well-known for views, being prohibited as well.

The IE in SIR R is in consequence operationally defined through the CE-view scheme restricted to all the IAs that are also IAs in SIR R. Usually, it is the Select sub-list with all these attributes only, followed by all the From... clauses. Besides, without being named so, CE-views are in fact already applied for decades to avoid the logical navigation or selected value expressions to queries, as we spoke about. Using SIRs instead would provide the benefit of always lower procedurality of the IE than of Create View, as we hinted to and will discuss in depth soon.

In fact, we qualify of explicit, every IE defined as a 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 CE-view R on all and only IA that are also IAs in SIR R. These IAs are contiguous in \( E_R \), but may be separated by SAs in Create Table R as at Figure 2, we recall. As a character string, every \( E_R \) is a sub-string of Create View R, although perhaps a distributed one.

Ex. 1. Recall the ‘biblical’ Supplier-Part DB, often named S-P in short, modelling some suppliers, parts and supplies. 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. 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 every practical DB. The variant we pick up below seems the most known, [D4]. 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 original sample data type for every attribute. Actually, the figure shows S-P2 DB that is our basic SIR DB. Nevertheless, S and P are the same SRs. S-P1.SP data types are these of the SAs with the same names in SIR SP shown there. All the SA definitions at the figure skip some practical details of the data type, e.g., the data 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 one for the discussed application. The notorious relational design criterion it conforms to, is the minimal number of SRs free of storage and update (normalization) anomalies, [D12].

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., about every actual client searching for a supply would select the supplier or part name(s). Every such query has 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 query more procedural than it should be, [MUV84]. The notorious “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:

\[
\text{(1) Create View SP As (Select SP_B.*, SNAME, STATUS, S.CITY, PNAME, COLOR, WEIGHT, P.CITY From (SP_B Left Join S On SP_B.S# = S.S#) Left Join P On SP_B.P# = P.P#));}
\]

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 CE-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 CE-view SP, we can define SIR SP according to steps (1) to (3) above through the following Create Table SP:

### S-P2 Scheme

<table>
<thead>
<tr>
<th>Table S</th>
<th>Table P</th>
<th>Table SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>S# Char,</td>
<td>P# Char,</td>
<td>S# Char,</td>
</tr>
<tr>
<td>SNAME Char,</td>
<td>PNAME Char,</td>
<td>P# Char,</td>
</tr>
<tr>
<td>STATUS Int,</td>
<td>COLOR Char,</td>
<td>QTY Int,</td>
</tr>
<tr>
<td>CITY Char;</td>
<td>WEIGHT Char,</td>
<td>SNAME, STATUS, S.CITY, PNAME, COLOR, WEIGHT, P.CITY</td>
</tr>
<tr>
<td></td>
<td>CITY Char;</td>
<td>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.
```

(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 in 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 CE-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 $E_{SP}$, as required for every Create Table R for SIR R. $E_{SP}$ 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 $E_{SP}$. 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.

In both statements (1) and (2) above, 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. The latter choice would make the procedurality gain provided by SIR SP_B 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 $E_{SP}$ 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.

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

Table P

| \( P# \) | \( PNAME \) | \( COLOR \) | \( WEIGHT \) | \( CITY \) |
| P1  | Nut  | Red  | 12  | London |
| P2  | Bolt | Green | 17  | Paris |
| P3  | Screw | Blue | 17  | Oslo |
| P4  | Screw | Red  | 14  | London |
| P5  | Cam  | Blue | 12  | Paris |
| P6  | Cog  | Red  | 19  | London |

Table SP

| \( S# \) | \( P# \) | \( QTY \) | \( SNAME \) | \( STATUS \) | \( S.CITY \) | \( PNAME \) | \( COLOR \) | \( WEIGHT \) | \( P.CITY \) |
| S1  | P1  | 300  | Smith  | 20  | London   | Nut | Red  | 12  | London |
| S1  | P2  | 200  | Smith  | 20  | London   | Bolt | Green | 17  | Paris |
| S1  | P3  | 400  | Smith  | 20  | London   | Screw | Blue | 17  | Oslo |
| S1  | P4  | 200  | Smith  | 20  | London   | Screw | Red  | 14  | London |
| S1  | P5  | 100  | Smith  | 20  | London   | Cam  | Blue | 12  | Paris |
| S1  | P6  | 100  | Smith  | 20  | London   | Cog  | Red  | 19  | London |
| S2  | P1  | 300  | Jones  | 10  | Paris    | Nut  | Red  | 12  | London |
| S2  | P2  | 400  | Jones  | 10  | Paris    | Bolt | Green | 17  | Paris |
| S3  | P2  | 200  | Blake  | 30  | Paris    | Bolt | Green | 17  | Paris |
| S4  | P2  | 200  | Clark  | 20  | London   | Bolt | Green | 17  | Paris |
| S4  | P4  | 300  | Clark  | 20  | London   | Screw | Red  | 14  | London |
| S4  | P5  | 400  | Clark  | 20  | London   | Cam  | Blue | 12  | Paris |

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

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 every join in some SIR R referring similarly to a part of R, of recursive. Actually, a recursive join may be a \( \theta \)- join, as one may easily find out. 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 conceptual similarities and differences between our example DBs. 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 \( I_R \) or \( I_R^c \). We suppose that DBS supporting SIRs internally pre-processes every \( I_R \). The result may be some \( E_I \) that we denote as \( E_R^c \). DBS processes then every \( E_R^c \) as every \( E_R \). Alternatively, the DBS considers \( I_R \) as an IE directly destined for dedicated processing. A rule for each case defines the form and pre-processing of every \( I_R \).

For the first case, 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 \(X\) and are inherited with the same names and values, then Select may contain the notorious less procedural generic SQL construct \(X.*\) instead. In both cases, every \(E_R\) is a proper substring of Create View \(R\), although perhaps distributed within the latter. Declaring an \(E_R\) instead of such a CE-view \(R\), is consequently always less procedural. Using SIRs brings thus this advantage to every DB using such CE-views at present. The rare and only exception is the CE-view with entire Select list reduced to ‘*’. According to widely known SQL rules, an IE as defined up to now, may then need several \(X.*\) constructs instead. It may consequently become more procedural than the CE-view.

More in depth, consider therefore CE-view \(R\) with the Select expression Select * From \(R_1…R_2…R_3…\), with some of these relations being \(R_B\). We do not see any practical case of more than one \(R_B\), so suppose \(R_B = R_1\) for instance. We have then \(E_R = R_2.*, R_3.*…\). The procedurality of Select list in \(E_R\) grows linearly with the number of relations listed. For any CE-view \(R\), for some number, likely above eight in practice or for even \(R_2\) only, but with long enough proper name, \(E_R\) must become more procedural than Create View \(R\).

Consider then the following rule:

**Rule 1.** \(I_R\) has the form: ‘# From \(R_1…R_2…R_3…\)’ with \(R_B = R_i\) for some \(i\). If \(R_i\) is not the first or the last one in From clause, then \(E_R\) is:

\[
E_R = R_1.*, R_2.*, R_3.*…\text{From } R_1…R_2…R_i… ;
\]

\(I_R\) is clearly less procedural than Create View As (Select *…) considered. Same occurs clearly for \(R_B\) being the first or last in \(I_R\). Likewise, we skip the discussion of the obvious generalization of \(I_R\) to multiple \(R_i\). These are useless in practice, to our experience, although an SQL dialect may allow for.

We suppose furthermore for Rule 1 that ‘#’ should be in Create Table after all the SAs. The order of the attributes in SIR \(R\) would nevertheless be that implied by the relations in From clause. This, as usual for every query or view in the form of Select * From \(R_1…R_2…\).

**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# within every supply. The DBA may furthermore provide the selected clients with the universal view SP as follows, after renaming SR SP to SP_B, as already discussed:

(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} = # \text{From } (SP_B \text{ Left Join S On } SP_B.S# = Enc (S.S#)) \text{ 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.* \text{From } (SP_B \text{ Left Join S On } SP_B.S# = Enc (S.S#)) \text{ Left Join P On } SP_B.P# = Enc (P.P#));

Actually, (5) remains visibly less procedural than (3) anyway. However, visibly as well, it would not 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. Hence, without Rule 1, the goal of an IE being always less procedural than the CE-view would not be attained.@

Our 2\textsuperscript{nd} case concerns the query (result) equal view, QE-view in short, as we call it. Under some restrictive conditions on the DB, QE-view \(R\) may happen to be the same SQL relation as CE-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 CE-view \(R\) or SIR \(R\), or to QE-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 attribute is labelled with the proper name
only in the resulting relation at every popular DBS. In other words, for every SIR R, the result is the same regardless of CE-view R, SIR R or QE-view R. Every possible result of a query to CE-view R or to SIR R may then also result from the same query to QE-view R.

In the same time, it will appear that Create View for QE-view R may be substantially less procedural than the least procedural one of CE-view R. It may become then even less procedural than the IE of SIR R. The rationale is that QE-view R scheme may take advantage of ‘*’. This may occur when (i) for some relations Ri; i = 1,..,k; the IE and CE-view R inherits every attribute of each Ri; i = 1,..,k; in the SQL order in Ri, except for some attribute Ai for every Ri, Ai being perhaps composed, (ii) for each Ai, SIR R has an SA R_B.Ai, and (iii) R_B.Ai immediately precedes in Create Table R every IA inherited from Ri. For every Ai, QE-view R may then provide Ri.Ai instead of R_B.Ai. For every Ri, the least procedural $E_R$ has then to enumerate all the IAs from Ri, while Ri.* may suffice for QE-view R. Every $E_R$ could then turn more procedural than QE-view R, contradicting our claim of providing for every SIR R, an IE less procedural than any equivalent view R. A DBA could then also evidently prefer R_B and QE-view R to SIR R.

The following rule provides nevertheless for an $I_R$ sufficient for every SIR R in such a case, as it will appear.

Rule 2. The Select list of $I_R$ contains only 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 $E_R'$ as follows.

1. For every Ri.#, $E_R'$ lists all the attributes of Ri except for Ki, in the order of Ri.*.
2. $E_R'$ lists every A1, A2… in the order of $I_R$, including every attribute replacing every Ri.# in $I_R$. @

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

(6) S-P2.SP (S#, SNAME, STATUS…, P#, PNAME…, QTY).

Suppose the referential integrity between SP, S and P. Consider the following Create View SP in S-P1, 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 from (7) is thus QE-view SP of SIR SP from (6).

The least procedural Create View SP for CE-view SP of SP as in (6) should have the same From clause as in (7), but would need to enumerate all the IAs from S, P and from SP_B in obvious order. $E_{SP}$ would do so for all its IAs and the IAs only, consequently. Create View SP for QE-view SP from (7) would be then about twice less procedural than Create View SP for CE-view SP. It would be then also less procedural than $E_{SP}$, as one may easily verify. If $E_{SP}$ was the only choice for SIR SP, the DBA could legitimately prefer QE-view SP.

Rule 2 authorizes nevertheless for the following $I_{SP}$:

(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
One may easily verify from the example that lower procedurality of $I_{SP}$ generalizes to any multi-attribute $K_i$. 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 other case of QE-view $R$ less procedural than CE-view $R$, hence, perhaps, than $E_R$ as well.

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 (ii) 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. The following rule provides then the backward compatibility for every discussed SIR $R$.

Rule 3. Consider the SQL dialect of the (kernel) DBS supposed expanded to SIRs, already providing for VAs, e.g., as for MySQL SQL as the kernel. It obviously means that DBS supports internally VAs as well. Let A be a proper IA name and let V designate a value expression (VE) allowed for a VA. Suppose finally that Create Table $R$ contains besides SAs, only the elements in the following form, supposed basically conform to that of a VA scheme for the kernel:

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

Then, every such element is supposed an IA that is a VA. The DBS processes the whole Create Table $R$ accordingly as the kernel’s one for relation $R$ with SAs and VAs.

The IAs of (10) constitute the specific $I_{R}$ we spoke about. The rationale for Rule 3 is that the definition of all the IAs in SIR $R$ to create is then backward compatible with that of the relation with the VAs at every popular kernel at present. “Basically conform” for (10) means that the actual syntax of $I_{R}$ should be conform to that of the dialect, while minor syntactical variations may occur among those. E.g., for MySQL SQL, (10) should include parentheses around every V. 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 DBS as the kernel, since that one does not support VAs.

Finally, notice that Rule 3 does not apply if even one of the IAs could not be a VA. Then, the IE has to be explicit or in another implicit form.

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. The most practical way to satisfy the client is to declare WEIGHT_KG as an IA of SIR P, self-inheriting from P.WEIGHT.

1. Regardless of the SQL kernel dialect, the following $E_P$ in Create Table $P$ immediately after the declaration of SA WEIGHT could then do:

\[
(11) \quad E_P = \text{Round} \ (\text{WEIGHT} \ast 0.454,\ 3) \ \text{As} \ \text{WEIGHT}_K \text{G From } P;
\]

Notice that in Create Table $P$ the declaration of SA CITY would separate WEIGHT_KG and From $P$ clause of $E_P$. As claimed in the Introduction, again $E_P$ would be less procedural than any CE-view $P$ or QE-view $P$ with WEIGHT_KG.

2. Suppose now S-P2 specifically on SQL Server. WEIGHT_KG could be there a VA. Rule 3 allows the DBA to declare this IA backward compatibly as:

\[
(12) \quad \text{WEIGHT}_K \text{G As} \ \text{Round} \ (\text{WEIGHT} \ast 0.454,\ 3) ;
\]

(12) is visibly even less procedural than (11). It is also obviously not more procedural than if it was a VA at present.
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 either some view R that amounts to CE-view R or QE-view R, or it is some VAs. Creating SIR R should be strictly less procedural than every view R. Through Rule 3 finally, every current relation with VAs, amounts simply to a specific self-inheriting SIR, 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 continue supposing every such statement backward compatible with some (kernel) dialect. 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 every kernel, 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, one may state the IE-clause as in Create Table R defining SIR R, except that if an SA or VA is referred to in Select list of IE-clause, it is then 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. 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. These procedures are obviously the simplest to put into practice.

For Drop Table R, we simply consider it applying 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 Ex. 4. 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 the SQL 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. Likewise, MsAccess DBA would need an atomic transaction with Drop View SP and Create View SP instead of (14).

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#);@

Observe that altering SR P to SIR P as in (14) is (slightly) less procedural than Create View P for any equivalent view P, CE-view P, in particular (why?). Likewise, the alteration (15) is visibly less procedural than (16) and even less if Drop View P and Create View P should be used instead. Similarly, (14) is visibly less procedural than Drop View P and Create View P that MsAccess would need in its place. Likewise, altering SR SP to SIR SP as in (17), is visibly less procedural than Create View SP for any equivalent view SP or CE-view SP. In fact, the actual view creation is 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. We thus need two statements. To avoid any run-time error for a client, both should form an atomic transaction. The following example details the point for SP. The case of P is similar. An atomic transaction with its add-on procedurality is likewise finally needed, we recall, for Drop View followed by Create View above discussed.

Ex. 6 Consider again S-P1.SP becoming either SIR S-P2.SP or CE-view SP. For the former, the single Alter SP statement (17) suffices. To create the CE-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). To make the whole procedure atomic, SQL Begin Transaction and Commit brackets are necessary. Likewise, SQL Error Code tests for the Commit or Rollback are necessary after each SQL statement within. All this leads to several SQL statements (how many?). The result is altogether clearly several times more procedural than is (17).@

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

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

Our above examples obviously generalize to every SIR. It should be clear thus that to alter any SR R to SIR R, should be always substantially less procedural than altering every SR R to R_B and creating CE-
view R or QE-view R. Obviously, choosing another renaming and creating a view equivalent to CE-view R or QE-view R accordingly, does not change this conclusion. Likewise, for every SIR R, altering R should be always less procedural than altering its CE-view R or QE-view R.

2.4 Data Manipulation for 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 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 by the query, [D4]. In practice, this depends on the kernel DBS. The constraint could impair even a query updating SAs only. For every SIR R, we allow therefore every update to SAs only to be valid iff it would be valid for R_B as a stand-alone SR. It may not be valid however for CE-view R.

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 SIRs
3.1 Basic Processing Scheme
As said already, the most practical way towards a SIR DB seems to reuse as the kernel a popular 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 \( EI_R \). 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 \( A_1, \ldots, A_m \) 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, Where etc. clauses of \( E_R \):

(V1) Create View R As (Select \( A_1, \ldots, A_m \) From...Where...)

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

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 \( E_{SP} \) we discussed. Hence, it calls BPS. If SIR-layer found any of \( I_{SP} \) we discussed, BPS would eventually pre-process it to \( EI_{SP} \). For \( E_{SP} \), it would issues the following two statements to the kernel DBS. The Create View SP is that in (1) actually. We systematically omit below the statements making the presented ones into an atomic transaction.

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#);

We leave as exercise the variants for each discussed \( I_{SP} \).

(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.

(3) If 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 (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; @

Figure 5 illustrates BPS outcome for Ex. 8.3. We refer to the DB as to S-P3. 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 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 IS 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 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 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, [H71], [F77], as well as of some proposals for the lossless decomposition through outer joins, [JS90], [DD91].

As one could realize further, 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. 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. 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 surrogate 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. The open-source Postgres DBS is the most prominent survivor of this trend, [SM96], [P]. Those models of inheritance should not be confused with that of IEs. E.g., Postgres has a dedicated INHERITS clause in its Create Table, creating a sub-relation (sub-table) from the entire inherited relation etc.

5. Conclusion

Stored and inherited relations, (SIRs), appear useful for every popular SQL DBS. Like dedicated views, SIRs may provide 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 an inheritance expressions (IE) 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 the latter. Hence, this extension would lower the procedurality with respect any explicit IE, necessary in every such case for BPS defined as 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.