SQL for Stored and Inherited Relations

Witold Litwin

▶ **To cite this version:**

Witold Litwin. SQL for Stored and Inherited Relations. 21st International Conference on Enterprise Information Systems (ICEIS 2019), May 2019, Heraklion, Greece. pp.37-48, ⟨10.5220/0007676700370048⟩. ⟨hal-02309464⟩

HAL Id: hal-02309464

https://hal.science/hal-02309464v1

Submitted on 9 Oct 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

HAL Authorization

SQL for Stored and Inherited Relations

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

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), [1] & [2] 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. 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 values basically only calculated on-the-fly from SRs or from other views through a statement of some data definition language (DDL), usually 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 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 when 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, [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 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

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 12, No. xxx
ISSN 2150-8097.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx Latest update 11/16/2018

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 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 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. 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 easy to spot 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 it *conceptually expanded* view (of) R and denote as CE-view R or view R simply. To declare 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. CE-view R inherits then, on the one hand, every SA 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…k, 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 CE-view R is finally that, for every tuple *t'* of SR R, there is exactly one tuple *t* of view R and view R does not have any other tuples.

The current usual rationale for a CE-view R, without being named so, is that it presents every tuple of SR R extended by the attributes and values that in fact conceptually characterize it as well. However, none is in SR R, since each would create notorious normalization anomalies as an SA there. CE-views are useful then for decades for avoiding the logical navigation or selected value expressions to queries. These are otherwise consequent to the discrepancy, as well-known and as we spoke about. We will recall this point with examples soon.

We intend SIR R in this context to be a single construct replacing both: CE-view R and SR R. The intended result is an SQL relation equivalent to CE-view R, with also 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 CE-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 the SA of SR R.

Accordingly, we define SIR R through Create Table R of SR R, expanded with the definitions of every IA inherited in view R from R1…Rk. The resulting order of the SAs and IAs in SIR R should be that of the corresponding IAs in CE-view R. In practice, one way to proceed is to write down, after the declaration: 'Create Table R As (', the view R scheme, i.e., the entire SQL expression that would follow Select keyword in Create View R. If the scheme

included R_B.* term, then expand the term to the proper names referred to. Next, expand every R_B.A to the declaration it would have in Create Table R for SR R. Both steps may constitute a single pass, obviously. Finally, append every clause of the latter Create Table R eventually remaining. Such clauses may declare a multi-attribute primary key, table indexing, partitioning…

An alternate way towards the same Create Table R for SIR R can be to start with Create Table R for SR R. Then, add to the list of attributes every IA intended for view R where it would be inherited from R1..Rk. The resulting order of the attributes should be the one of view R. Finally, one inserts From… clauses intended for view R after the last attribute of the list.

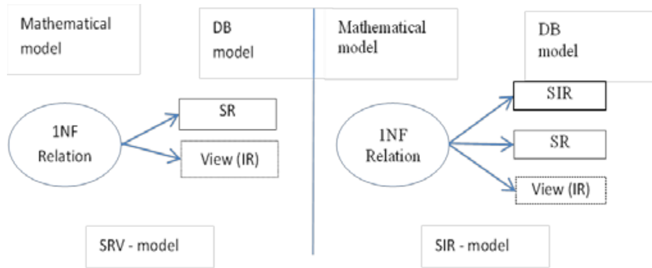

Figure 1: SRV-model versus SIR model.

Observe that the result, no matter which way constructed, conforms to our generic requirement on the primary key of every SIR R. Also, observe that our rule for default source naming of SAs in SIR R, keeps all the clauses From… within view R referring there to R_B, valid for SIR R as well. Instead of stand-alone SR R_B, they simply refer to the base of SIR R, equal to the former as an SQL relation and with respect to the full attribute naming. While this is the primary rationale for our specific to SIRs naming rule, one can figure out also a less obvious one. Namely, referring to R_B rather than to R whenever possible, from some other SIR R', when R already inherits an IA from R', hence refers to R', may avoid the *circular referencing* between R and R'. We prohibit it, as it is for views. Referring to R_B may avoid such referencing since every R_B inherits from nothing by definition.

For every SIR R, we consequently define the IE through CE-view scheme with Select list restricted to all and only IAs being IAs in SIR R as well. If CE-view R enumerates every IA that is an SA in SIR R or declares all as R_B.*, then IE is a strict Select sub-list of the Select list in view R followed by all of the From… clauses. For the same procedurality of the SAs schemes as for SR R, IE of SIR R has then strictly lower procedurality than that Create View R for CE-view R, as we hinted to and will illustrate with examples soon. SIR R becomes consequently more advantageous than SR R and CE-view R for the avoidance of the logical navigation or of selected value expressions.

In fact, 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 CE-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. 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, [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 (<u>S#</u>, SNAME, STATUS, CITY), P (<u>P#</u>, PNAME, COLOR, WEIGHT, CITY), SP (<u>S#</u>, <u>P#</u>, 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 it in detail soon. 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.

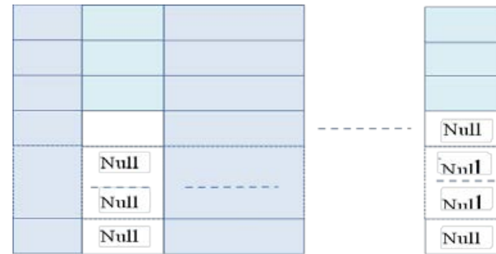

Figure 2: SIR structure as an SQL relation.

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 fulfils 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). 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 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 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 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 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 contiguous one. It is the same substring in (1) hence in CE-view SP, as well as defining 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 reminded 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 SB even greater. Same would happen if an IA dispersed the SAs within Create Table SP and in in CE-view SP thus. The list of IAs contiguous in E_{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 for the tuples of Figure 4.

Observe also that in (1), any prefix SP_B, in joins refers to SR SP that is one of the source relations of view SP. In (2) in contrast, it refers to the SP base SP_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 θ-join, as one may easily find out. Recursive joins are basically not permitted for SQL views, we recall. The example suggests them in contrast typical for IE*s*.

The graphic at Figure 3 schematizes the proposed evolution of the "biblical" SR SP in S-P1 into SIR SP in S-P2. At the left, we have S-P1 scheme. Next, we have S-P1 with SP renamed to the default name of SP_B and the CE-view SP, as defined by (1). This is what DBA could do best at present to avoid the logical navigation within queries to SP. The view contains the sub-view that is a virtual copy of SP_B, with every SA of SP_B becoming an IA. Finally, at the right, SP_B replaced its copy, becoming the base SP_B of our SIR SP. 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 sub-part. That one is fully redundant with SP_B, as just discussed. The redundancy costs view SP the clause S_B.* in (1), with respect to the IE in SIR SP, as defined by (2). This is the core of the higher procedurality of Create View SP with respect to the IE in Create Table SP for SIR SP. In other terms, it is the cause of lower procedurality of Create Table SP as in (2) than of Create Table SP_B followed by Create View SP as in (1).

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 or I_R. Every I_R is less procedural than an E_R could ever be. As it will appear, in three cases, an I_R allows reaching our already mentioned goal, i.e., of always providing an IE less procedural than any equivalent view. There may be no sufficiently simple E in these cases.

The reduced procedurality of I may result from generic character '#' specific to I's. In two cases, I_R with '#' may be the only expression less procedural than every equivalent view. In first case, view R requiring I_R is a specific CE-view R. Otherwise it is a view that we call *query equivalent* to SIR R, QE-view in short. It is not CE-view R, but an equivalent one that still provides in practice for the same non-procedural queries as CE-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 QE-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 CE-view R as well. Nevertheless, every QE-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 CE-view R thus, could be a VA and the kernel DBS support 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 CE-view R. Actually, as we already said, but in somehow different terms, it is the lesser procedurality of an SR with VAs than that of CE-view R with IAs same as all the VAs, was the rationale for the VAs and their

popularity for decades already.

We suppose that DBS supporting SIRs internally pre-processes every I_R. For I_R with '#', the result is Create Table R with some E_R that we denote as E^I_R. DBS processes then every Create Table R with E^I_R as any Create Table R with some E_R. For an I_R defining VAs, the result is the direct processing of Create Table R for SIR R as if it defined an SR with VAs at present. A rule for each case defines the form of I_R and the pre-processing. We now address these rules.

For the first rule, 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. 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 such a CE-view R, is always less procedural. Using SIRs brings thus this advantage to every DB using such CE-views at present. The rare and only exception are the CE-views with entire Select list reduced to '*'. Such Select list is inapplicable to an IE. It would redefine IAs defined as the SAs forming the base of the SIR. According to widely known SQL rules, every E_R may then at best contain one or more X.* terms instead, each inheriting all and only IAs from X. E_R may consequently be more procedural than the CE-view, by far even.

Indeed, suppose CE-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 = R2.*, R3.*… From R_B…R2….;. 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 fewer, but with long enough proper names, E_R must become more procedural than Create View R.

More generally, consider now the following rule:

Rule 1. Create Table R contains I_R in the form of: '# From R_1…R_2… ;'. There, R_i = R_B not followed by AS keyword for some unique $i = 1,2….$ Let R_{i1}, R_{i2}… be all the names among R_1, R_2… with $i1 < i2….$, where $i1 \neq i$ and $i2 \neq i….$ Then, first, E^I_R is:

E^I_R = R_{i1}.*,R_{i2}.*… From R_1…R_2…;

Next, the terms in E^I_R insert into Create Table R, instead of #, according to their order within From clause and with respect to R_B there. The terms R_{i1}.*,R_{i2}.*…R_{i-1} insert before the first SA scheme. All the others replace #.@

In other words, E^I_R has one and only one X.* term for every X in From clause that is not (non-aliased) R_B. Anyone even only basically familiar with SQL, should realize that if Create View As (Select * From R1…R2…) defines CE-view R, then the terms in E^I_R and R_B.* in Select clause, in their order within From clause, constitute simply a more procedural equivalent of '*'. In the same time, E^I_R is the least procedural E_R in this case. Every other E_R requires explicit enumeration of some IAs. As said, even E^I_R can reveal nevertheless necessarily more procedural than the discussed Create View R. In contrast, I_R permitted by Rule 1 must be 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# for every supply. The DBA may furthermore provide the selected clients with the following universal view SP as follows, after renaming SR SP to SP_B, as already discussed. The right join replaced 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 is clearly also CE-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# = S.S#) Left Join P On SP_B.P# = P.P#), Primary Key (S#, P#));

E^I_{SP} is then equal to:

(5) E^I_{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^I_{SP} precedes the SAs, since S precedes SP_B in the right join within From clause. P.* replaces #. The list S.*, SP_B.*, P.* is equivalent to * in (3).

As in general for every E_R and CE-view R, E^I_{SP} in (5) is also less procedural than Create View SP of CE-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 would not be so if S and P had long enough names, e.g., SUPPLIERS and PARTS_IN_STOCK instead of S and P. Enough to prove our point that without Rule 1, we could not attain our goal of an IE being always less procedural than the CE-view it may replace.@

As hinted to, our 2nd case concerns the QE-view. 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, 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 ; IE and CE-view R inherit 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 proceeds 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. The least procedural E_R has to enumerate then for every Ri all the IAs from. In contrast, for every Ri, Ri.* may suffice for QE-view R. Even such E_R could then turn more procedural than QE-view R. This would contradict our already mentioned goal. DBA could again reasonably prefer SR R_B and QE-view R to SIR R.

The following rule provides nevertheless for every such case, for an I_R sufficiently non-procedural, as it will appear.

Rule 2. The attribute list in I_R contains only terms A1,A2… or R1.#, R2.#,... . Also, every relation Ri has some key attribute Ki, suppose mono-attribute for simplicity. SIR R also has Ki as an attribute. Then, DBS produces E^I_R as follows.

1. For every Ri.#, E^I_R lists all the attributes of Ri except for Ki, in the order of Ri.*.

2. E^I_R lists every A1, A2… in the order of I_R, including every attribute replacing every Ri.#.@

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

(6) S-P2.SP (SP_B.S#, SNAME, STATUS, S.CITY, SP_B.P#, PNAME, COLOR, WEIGHT, P.CITY, QTY).

Suppose also the referential integrity between SP, S and P. The following Create View SP for CE-view SP is now clearly among the least procedural ones:

(7) Create View SP As (SP_B.S#, SNAME, 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#);

Consider furthermore the following Create View SP:

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

View SP defined by (8) is not CE-view SP. The full source names of attributes S# and P# are S.S# and P.P#, unlike in (7). 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 without prefixing S# and P#. View SP from (8) is thus QE-view SP of SIR SP as in (6).

CE-view SP of SP as in (6) may have the same From… clauses in Create Table SP as in (8) for QE-view SP. It is the case of (7), in particular. However, unlike in (8), the Select clause for every CE-view SP must enumerate all the IAs from S, P and from SP_B in the order of (6). Every E_{SP} would need to do so as well, for all its IAs and the IAs only, consequently. As the result, every E_{SP} would be more procedural than Create View SP (8), as one can easily verify. If such an E_{SP} was the only choice for SIR SP, the case would contradict our goal. Also, once more, the DBA could legitimately prefer QE-view SP (8) to SIR SP.

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

(9) 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 (8), by about 20%. The difference would evidently increase 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:

(10) Create Table SP (S# Char, SNAME, 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#));

We recall that (10) replaces Create Table SP_B with SP_B (S#, P#, QTY) and Create View SP as (8), for our QE-view SP.@

One may easily verify from the example that lower procedurality of I_{SP} generalizes to any multi-attribute Ki. It also generalizes to any I_R conform to Rule 2, regardless of the number of constructs Ri.* in the Select list and of relation and attribute names. Observe finally, that there is no case of QE-view R less procedural than CE-view R hence, perhaps, than E_R as well, other than those where Rule 2 suffices as discussed till now. In other terms, QE-view R takes advantage of * over CE-view R only for the order of attributes in SIR R obviously analogous to that of our example.

Finally, one may of course apply '#' to reduce non-procedurality even if E_R already reaches its goal. E.g., one may reduce the list of IAs in (2) and at Figure 3, simply to S.#, P.#.

Our last case is that of the kernel SQL providing for the already mentioned SRs with VAs, e.g., MySQL or SQL Server SQL. By our definition of any SIR SQL dialect for SIRs, every such SIR SQL dialect should also provide for VAs, at least through the same statements. We assimilate then, for every such SIR SQL and only then, that every SR R with VAs is SIR R created using a specific I_R. That one defines every IA A as one would define A as a VA for the kernel. Altogether, every such I_R defines E^I_R that would be defined as E_R using CE-view of SR R extended with and only with, every IA defined as VA. More precisely the correspondence is basically as follows. Basically, means that we make abstraction of minor syntactical discrepancies between VA schemes among the dialects. E.g., MySQL requires parentheses around every value expression (VE), unlike SQL Server. A dialect may also support a VE that another does not etc.

Regardless of these discrepancies, we suppose every discussed I_R to contain for every IA the term:

(11) A As VE.

This term appears common to every SQL dialect with VAs at present. Every term (11) generates: VE As A for E^I_R. After the last such term, E^I_R contains 'From R_B' clause. Besides, every clause about SAs in Create Table R with I_R remains the same in Create Table R with E^I_R.

E^I_R instead of I_R remains always an option for every qualifying SIR R. Like, in absence of E^I_R, CE-view remains notoriously at present an option and the only one besides, instead of SR R with VAs. We recall that through Rule 3, the latter constitutes actually the same relation as SIR R. Both options are visibly more procedural. They do not make thus practical sense. We suppose therefore the following rule:

Rule 3. If Create Table R of SIR SQL contains discussed I_R, then it is kernel's Create Table R.

Consequently, as we detail in Section 3, SIR DBS creates SIR R as

the kernel would do for the statement, i.e., possibly as SR R with VAs, unless an error occurs. In other words, unlike for the other forms of I_R we have defined, SIR DBS does not preprocess any such I_R into E^I_R. Consequently, we call simply sometimes below VA every IA defined as in (11) and we refer to SIR R with I_R subject to Rule 3 as to SIR R with VAs. Observe that if SIR R over the kernel with VAs contains even one IA that is not a VA, then some E_R or I_R due to Rule 1 or Rule 2 is the only possibility. Finally, we do not define any VA-like I_R for SIR SQL where the kernel does not provide for VAs, e.g., MS Access. An E_R or I_R due to Rule 1 or Rule 2 is then again the only option.

Observe that actually one could lift the latter restriction, generalizing Rule 3 adequately. An additional gain to procedurality of Create Table for SIR R would result. Through some thinking about, one can foresee this one equal or about equal, or superior to that provided by I_R with respect E^I_R for a kernel with VAs. We leave this whole interesting subject for the future work.

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

1. Suppose MS Access as the kernel dialect. E_P is the only option, e.g.,

(12) E_P = Round (WEIGHT*0.454,3) 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, again E_P would be less procedural than any CE-view P or QE-view P or any other equivalent view with WEIGHT_KG.

2. Suppose now S-P2 on SQL Server. WEIGHT_KG could be a VA. Rule 3 allows declaring WEIGHT_KG as:

(13) WEIGHT_KG As Round (WEIGHT * 0.454,3) ;

For SIR SQL of S-P2, i.e., using here SQL Server SQL as the kernel, this declaration constitutes I_P. Through Rule 3, Create Table P with (13) would mean that of SQL Server creating SR P with SAs as at Figure 3 and with VA WEIGHT_KG defined by (13). Actually, it would be the case. The resulting backward compatibility thus, of the clause possible for declaring WEIGHT_KG for S-P2 with that possible for WEIGHT_KG as VA at SQL Server, makes both clauses obviously equally procedural.

Clause (12) could be E^I_P here as is. Its syntax would be indeed valid for SQL Server, but not e.g., for MySQL as the kernel. It remains an option, but does not make practical sense. (13) is visibly even less procedural.

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. Neither, it can be for even any SQL dialect of our knowledge, besides. 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 = WEIGHT_KG…, T_Weight AS WEIGHT * (Select Sum (QTY) From SP Where SP.p# = SP.p#) FROM P;

To declare I_P with WEIGHT_KG and T_QTY as VAs in Create Table P, one would need to lift, for SIR SQL with SQL Server as the kernel, the restriction we have mentioned. Actually, here the generalization of Rule 3 could be easy. Namely, one could simply add the sub-rule rewriting I_R into E^I_R iff an IA declared VA in SIR SQL for SIR R is not VA for the kernel SQL. 'FROM P' would be the resulting procedurality gain for Create Table P here.@

Observe that the example illustrates in particular our point in the Introduction that every SR R with VAs is in fact a specific SIR R. More precisely, it is SIR R with I_R providing, through Rule 3, for VA scheme for every IA and for implicit From R clause, mandatory in E^I_R. Every I_R is then also less procedural than E^I_R. E.g., as it appears for (12) and (13). If I_R is not a possibility at present, a generalization of Rule 3 we have hinted to could make it possible for some SIRs. Similar gain to E_R being the only possibility for those at present would result from, e.g. as in Example 4 for SIR P with T_QTY.

Summing up, Example 2 and 3 illustrate that, through Rule 1 and 2 we effectively always have SIR R with IE less procedural than Create View R of every equivalent view R could be. Example 4 illustrates that for every Create Table R for SIR R, if an SR R with intended IAs defined as VAs is an option, then through Rule 3, one can have also have every such IA with a VA scheme. The example illustrates in particular thus our point in the Introduction that every SR R with VAs is in fact a specific SIR R. More precisely, it is SIR R with I_R providing, through Rule 3, for the VA scheme for every IA and for implicit From R clause of E^I_R. If the kernel SQL dialect does not provide for VAs, Create Table R for SIR R as above defined can only define E_R. That one defines then every IA scheme as in CE-view and has From R clause of the view.

It follows that whenever Rule 3 makes I_R a possibility, I_R is always less procedural than E^I_R could be. Actually, Create Table R for SIR R where IAs could be all VAs becomes exactly as procedural as Create Table R with these VAs of the kernel. Whenever E_R is the only choice, it remains still also always less procedural than Create View R for CE-view R. As the overall result illustrated by all three examples, there is no SIR R, where the only choice would be an IE necessarily more procedural than some SQL capability available at present for the same purpose.

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, 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 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 or an SR with 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. Every IE-clause defines new IE replacing an existing one. It does so similarly to every Select expression in an Alter View at present, replacing the existing view 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 IR could do.

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

(14) 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., 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) ;

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

(16) 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:

(17) 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 MS Access, DBA would need Drop View P followed (atomically) by Create View P.

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

(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#);@

Observe that altering SR P to SIR P as in (15) is (slightly) less procedural than Create View P for any equivalent view P, CE-view P, in particular (why?). Likewise, the alteration (16) is visibly less procedural than (17) and even less if Drop View P and Create View P should be used instead. Similarly, (15) is visibly less procedural than Drop View P and Create View P that MS Access would need in its place. Likewise, altering SR SP to SIR SP as in (18), 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 (18) 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 (18).@

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, 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 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 renaming 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. Next, for every SIR R, altering an IA A through the IE-clause, should be always less procedural than altering A in CE-view R or QE-view R. In the same time it should be clear also that altering an SA of SIR R is always exactly as procedural as altering R_B. But, whenever view R inherits or should inherit an SA A of R_B explicitly by name, altering R_B.A requires altering view R as well. As the exercise around QTY above should illustrate, altering SIR R instead, should be then always several times less procedural. Finally, if every added or modified or dropped IA could be a VA, altering SIR R is as procedural as adding, modifying or dropping this VA 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 CE-view R instead. If Q refers to R_B, the outcome is that of R_B being the stand-alone SR for CE-view R. Every update query Q addressing SIR R is accordingly valid (executable) only if CE-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 SAs of R only. Q may refer then to R_B, being valid iff valid for R_B as the stand-alone SR for CE-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 CE-view SP selecting * where S# = S4 would show afterwards. If Q addressed CE-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 CE-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 changed CITY also to every other supply by S1 there, as perhaps surprising side-effect. It would be so indeed for Q and CE-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 SIR SP tuple with S# = S4. Again, since it would be so in CE-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 CE-view SP under MS Access 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 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 CE-view SP thus. Under SQL Server kernel accordingly, S-P2 client would need to reformulate 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 to reuse 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. As the obvious primary choice and as till now, we suppose SQL at the SIR-layer backward compatible with the kernel SQL dialect.

In particular, for the Create Table R statement received, SIR-layer determines the type of the relation to create. For R being an SR, SIR-layer forwards the statement as is to the kernel. In turn, the processing must be 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 basically represent every SIR R in the kernel by creating, upon receiving Create Table R, the stand-alone SR R_B equal to the base R_B and CE-view R. SIR-layer simply forwards then every query Q as is to the kernel. This one directs Q towards view R or R_B. Only for every SIR R defined through Rule 3, on the DBS supporting VAs therefore, the simplest design appears rather that SIR-layer simply forwards to the kernel Create Table R. The latter creates the SR R with VAs.

We qualify of *basic (processing) scheme,* (BPS), the SIR-layer algorithmic for SIRs represented 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^{I}_{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 CE-view simply as follows. Let A1,…,Am be the list of the names of every SA and IA in Select list of E^{I}_{R}, in the original order. Then, BPS simply issues to the kernel the following statement, with From, Where etc. clauses of E^{I}_{R}:

Create View R As (Select A1,…,Am 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^{I}_{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.

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 at 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 as for 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. 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 BPS of other DDL & of DML Statements

Like Create Table R for SIR R, Alter Table R and Drop Table R at SIR-layer 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 CE-view R instead. We recall from Section 2.3 that the latter is always more procedural than the former, possibly several times. In more detail thus, 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 an IE-clause, 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.

For every SIR R in contrast, every Alter Table R altering only SAs or VAs makes BPS issuing Alter Table R_B statement. If the kernel DBS supports Alter View, BPS generates also Alter View R, addressing CE-view R of course. It 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 SA or VA and IE-clause, BPS generates either Alter Table R_B followed by Alter View R or it generates Alter Table R_B and Drop View R followed by Create View R.

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, for processing of SIR-layer DML statements, BPS simply sends every query to the kernel as is.

BPS should be implemented in some host language, obviously 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 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, almost as if these were designed also for SIRs. We recall those capabilities made MsAccss the most popular DBS by number of licenses at present, by far even.

3.3 Operational Overhead of SIR-layer

The kernel storage for every SIR data is in practice the one for its base only. Next, CE-view storage should be obviously always negligible. Finally, the storage for the SIR-layer meta-tables should be clearly larger, but still typically negligible with respect to the data storage. Altogether, the storage for a SIR DB should be thus only negligibly greater than that required by SRV DB with the stored relations only or with CE-views or QE-views in addition, 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 SIR R costs the same as the processing of the same query to CE-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 be almost without any operational overhead cost in practice.

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 CE-views or QE-view we have discussed. But then, every IE in S-P2 was less-procedural than Create View of its CE-view or QE-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 IEs too complex to become VAs only at present or helping with the logical navigation. The rationale for VAs proved appreciated, since VAs remain 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 would need 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 initial goal usually impossible.

CE-view SP is 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, [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 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. 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, [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 CE-view and QE-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 (VAs) 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 such an implementation. MySQL seems the best current basis. 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 we have illustrated in Example 4. Next, the relational design rules for SIRs we have mentioned appear a promising goal. Also, BPS could perhaps create more efficient CE-views, [7], [8], [14], [21]. Finally, most of major DBSs are now interoperable, [10]. Multidatabase SIRs, i.e., with IEs inheriting from several DBs, appear attractive as well.

ACKNOWLEDGMENTS

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

6. REFERENCES

[1] Codd, E., F. Derivability, Redundancy and Consistency of Relations Stored in Large Data Banks. IBM Res. Rep. RJ 599 #12343, Aug. 19, 1969.

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

[3] Date, C.J. An Introduction to Database Systems — Eighth Edition since 1975. Pearson Education Inc. ISBN 0-321-18956-6, 2004.

[4] Date, C. J., & Darwen, H. (1989). Watch out for outer join. Date and Darwen Relational Database Writings, 1991.

[5] Date, C. J. Database Design and Relational Theory, Normal Forms and All That Jazz. O'Reilly, 2012, 278.

[6] Fagin, R. Multivalued Dependencies and a New Normal Form for Relational Databases, ACM TODS, 2,3, 1977, 262-278.

[7] Goldstein, J. Larson, P. Optimizing Queries Using Materialized Views: A Practical, Scalable Solution. ACM SIGMOD 2001.

[8] Halevy, A.,Y. Answering queries using views: A survey. The VLDB Journal 10: 270–294, 2001.

[9] Heath, I., J. Unacceptable file operations in a relational data base. ACM SIGFIDET '71 Workshop on Data Description, Access and Control, 19-33.

[10] Litwin, W., Abdellatif, A. Multidatabase Interoperability IEEE COMPUTER, (Dec. 1986).

[11] Litwin, W. Ketabchi, M., Risch, T. Relations with Inherited Attributes. Hewlett Packard Lab. Palo Alto, CA. Tech. Rep. HPL-DTD-92-45, (April. 1992), 30.

[12] Litwin, W. Vigier, Ph. Dynamic attributes in the multidatabase system MRDSM, IEEE-ICDE,1986.

[13] Litwin, W. Supplier-Part Databases with Stored and Inherited Relations Simulated on MS Access. Lamsade Technical E-Note, 2016. pdf.

[14] Larson, P., Zhou J. Efficient Maintenance of Materialized Outer-Join Views. ICDE 2007.

[15] Jajodia, S., Springsteel, F., N. Lossless outer joins with incomplete information. BIT, 30, 1, 34-41, 1990.

[16] Mendelzon, A. Who won the Universal Relation wars? Stanford InfoLab, 2004, http://infolab.stanford.edu/jdu-symposium/talks/mendelzon.pdf .

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

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

[19] Stonebraker, M. Moore, D. Object-Relational DBMSs: The Next Great Wave Morgan Kaufmann, 1996. 2nd Ed. In 1998.

[20] Vardi, M., Y. The rise, fall, and rise of dependency theory: Part 1, the rise and fall. Presentation. Sigmod/Pods 2011.

[21] Valduriez P. Join indices. ACM Trans. Database Syst., 12(2), 218–246, 1987.

S-P2 Scheme

Table S	**Table P**	**Table SP**
S# Char,	P# Char,	S# Char,
SNAME Char,	PNAME Char,	P# Char,
STATUS Int,	COLOR Char,	QTY Int
CITY Char;	WEIGHT Char,	SNAME, STATUS, S.CITY, PNAME, COLOR, WEIGHT, P.CITY
	CITY Char;	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#));

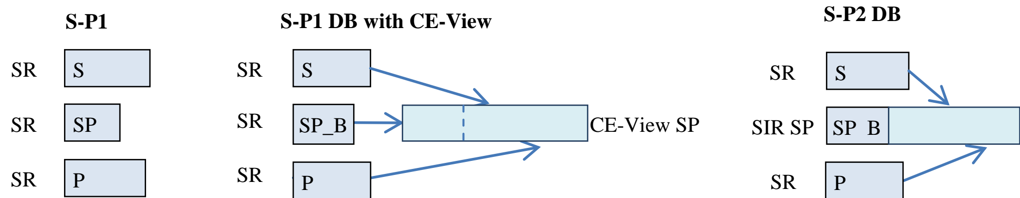

Figure 3: S-P1 and S-P2 schemes.

S-P2 Content

Table S

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

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

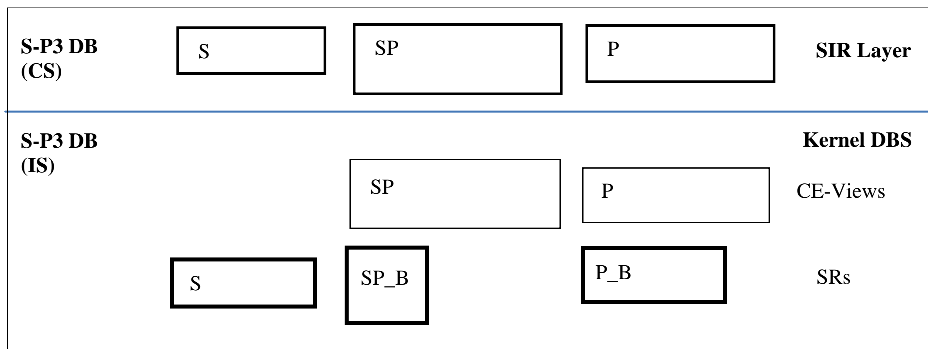

Figure 5: S-P3 DB. Above: SIRs. Below: CE-views and SRs within the kernel DBS.