Stored and Inherited Relations for Logical Navigation Free and Calculated Attribute Free SQL Queries to Base Tables

Witold Litwin
Université Paris Dauphine France
Witold.litwin@dauphine.fr

Abstract — A *stored and inherited relation* (SIR) is a 1NF stored relation enlarged with inherited attributes (IAs). The latter make SIRs as normalized base tables the only known view-savers for logical navigation free (LNF) and calculated attribute free (CAF) SQL queries to such tables, [1]-[5]. Recall that LNF means no equijoins between foreign and referenced keys, while CAF queries avoid possibly complex value expressions, e.g., with aggregate functions and sub-queries. We now show that one may consider typical present base table schemes with foreign keys as defining SIRs termed *natural*. The latter provide for LNF queries, unlike the same scheme base tables at present. Next, we show how one may add CAs to such SIR schemes, getting CAF queries as bonus. Below, we first recall SIRs basics. Then, we discuss the natural SIRs. Next, we analyze the enlarged schemes. Then, we generalize the relational design to SIRs, the concept of NFs especially. Afterwards, we show that providing SIRs over a popular relational DBS, e.g., SQLite3, should be simple. Preexisting applications could remain unaffected, while new ones could profit from LNF and CAF queries. We conclude that major relational DBS should become SIR-enabled "better sooner than later". LNF and CAF query should become the standard, simplifying the life of, likely, millions of SQL clients.

Keywords—Relational model, Foreign Key, Inheritance, Logical Navigation, SQL, Calculated Attributes, Stored and Inherited Relation

I Introduction

The relational model as defined by Codd has two 1NF constructs (abstractions), [6], [7]. A *stored* relation (SR), also called *base relation or table*, consists of stored attributes, (SAs), only. Values of these attributes are not calculable from other attributes in the DB (that is why they have to be stored). An *inherited* relation, more commonly called *view* or *view table*, consists of (relationally) *inherited* attributes, (IAs), only. One calculates every IA from SAs or other IAs, through a stored (relational) query called *view scheme*. Originally, one supposed every IA calculable only. Later, it appeared practical sometimes to maintain a (stored) snapshot of selected IAs, refreshed whenever needed. Such views and IAs were termed *materialized*, [17], [16], [20]. Although stored, a materialized view is not an SR. It is indeed entirely calculable through its (view) scheme.

Recently, we proposed to add the *stored and inherited relation* (SIR) construct to this model, [1]-[4]. The construct roots in [21], part of the popular in nineties trend to harness inheritance in the relational DBs. E.g., see [26] or Postgres, [25], or later proposals, [12]. A SIR, say R, is a 1NF relation with both SAs and IAs, the primary key (PK) being SAs only. We refer by default to the projection of R on its all and only SAs as to R_ and call it *base* of R. We also say that the IAs *enlarge* R_ and refer to the IAs scheme as to *Inheritance Expression* (IE). The crucial advantage of SIRs as base tables over the logically the same base tables, but SRs only, as required by the present model, is that no IA may create a normalization anomaly. Unlike it would often happen if the same attributes were SAs instead. Two important advantages for queries to a DB with SIRs without any normalization anomaly follow, with respect to the equivalent queries to the DB with normalized SRs only, i.e., the queries providing for the same output:

(1) A query Q addressing any SAs or IAs of SIR R can be *Logical Navigation Free* (LNF), while an equivalent query Q', addressing normalized R_ as stand-alone SR named R, would typically require some LN. Recall that the LN concept designates the typical joins between the base tables. These are equijoins on foreign keys, (FKs), [7], and their referenced keys that are PKs with the same proper names as the FKs, usually, [24]. Recall also that the normalized SRs as base tables of an SQL DB suffice for every SQL query to the DB. If Q' is such a select-project-join query, Q consists then, typically, from the select-project part of Q' addressing SIR R only. Q is then in practice always less procedural, i.e., requiring fewer characters, than Q'. In addition, joins are often felt dreadful, the outer ones especially, while the LN often needs the latter, [9], [19]. Not surprisingly, clients typically at least dislike the LN. We designate any SIR free of LN for some queries as SIR *for LNF queries*.

(2) An IA in SIR R can be a *calculated* attribute (CA), i.e., defined through any relational and value expressions or sub-queries, perhaps with scalar or aggregate functions. Any query Q to SIR R with CAs may then be free of defining any of these, selecting every CA in Q by name only. I.e., Q can be a *CAF* query, avoiding the procedurality of the CA specifications within the equivalent Q' to SR R with the same SAs and without any IAs instead, i.e. to R_ renamed to R. SAs with the same names and values as CAs would do in theory as well, but most often would denormalize the base table to 2NF at best (as we recall by examples later on). We designate any SIR with CAs as SIR *for CAF queries*.

A SIR can provide for both LNF and CAF queries. At present, the only practical way to provide for these capabilities of any SIR R is view R that we call *conceptually equivalent* to or the *canonical* view of SIR R, *C-view* R in short. Every C-view R is simply logically, i.e., mathematically, equal to SIR R. I.e., the attribute names and order are the same, as well as every tuple. In particular, one defines all the IAs in SIR R as one would define those in C-view R, the same From clause included. The From clause should be besides such that for every R_ tuple one could insert, given R_ constraints, C-view should contain a tuple with the one of R_ as the sub-tuple. The only difference is then physical, namely every SA A in R_ is IA A in view R, each of these IAs being inherited from the same stand-alone SR R_. That R_ may actually be a pre-existing base table R we referred to in (1) that one had to rename somehow to create view R. Recall that SQL prohibits any same name relations in a DB. SIR R, there is a bijection between SIR R and C-view R.

The "price" for (1) or (2) for SIR R with respect to R_ alone as a base table, may be the procedurality of the IE, i.e., the minimal number of characters or keystrokes to define it. For SQL extended to SIRs, [1], it is, basically, an additional procedurality for Create Table R, [1]. The equivalent price for C-view R in SQL at present is the procedurality of Create View R. The general advantage of every SIR R is that the IE can be less procedural than the Create View R, [1]. The rationale is that the latter has to at least, redefine as an IA every SA of R_. This obviously costs some procedurality. By the same token, to create SIR R is always less procedural than to create R_ and C-view R. In popular terms, every SIR R is a *view-saver* for C-view R. Actually, SIRs are also less procedural to maintain, [1].

All this is our rationale for SIRs. We follow the general trend in DB-science and in entire CS in fact. Recall that this is why the relational model took over the Codasyl one, although the latter was already in use, e.g., in Oracle Codasyl DBS. Likewise, it is why it took finally over every other earlier DB model. The assertional (declarative…) relational algebra queries or, better, the predicative ones, were indeed in general considerably less procedural than any equivalent navigational ones in any of these models. Also, it is the lower procedurality of the higher-level programming languages for general programing that wiped out the use of assemblers for it….

See oldies on the subject, e.g., early editions of [8].

Below, we qualify Create Table extended to SIRs of *SIR Create Table*. One can define *SIR Create Table* for every popular SQL dialect: of MySQL, SQLite, Postgresql… Given such a *kernel SQL* (dialect), any SIR Create Table specifies every SA as in the kernel at present. But, in SIR Create Table, these SAs may interlace with the IAs. Every SIR Create Table R may furthermore include similarly any table constraints and options of the kernel. Recall that these specify the PK, FKs, etc.

Next, with respect to the IAs, a SIR Create Table R, may define an IA *explicitly*. We speak then about an *explicit* IA. For every SQL kernel, one may define explicit IAs as one would do for in C-view R. One may thus define every such IA individually or indirectly, whenever appropriate, through SQL R'.* construct, R'≠ R. As in a Create View, the IE with such explicit IAs always ends up with From clause. In a SIR, however, we recall, table options may follow.

Besides, for any *kernel SQL* providing for, so-called, *virtual* (generated, computed, dynamic…) attributes or columns, (VAs), [22], we supposed that SIR Create Table preserves this capability. As we recall later, base tables with VAs are beyond Codd's relational model. The feature was introduced by the DB industry (Sybase). As we also remind later, any VA is in fact a specific CA, hence an IA. We qualified every IA declared as a VA of explicit as well. We qualified of *explicit* the IE with explicit IAs only, as well as the entire SIR Create Table with.

In [1] also, we qualified of *SIR-enabled* or of *SIR DBS* in short, every DBS (or DBMS, as some prefer) providing for SIRs, in addition to every present SQL kernel capability of a popular DBS. We also considered that every SIR DBS embeds some present DBS, referred to simply as the *kernel* (DBS). The term: kernel SQL above refers in fact to the SQL dialect of the kernel DBS. Every *SIR SQL* dialect provides then at least (i) for Create Table of the kernel DBS, generalized as above outlined and (ii) for Alter Table generalized to alter IEs, as we outline later. All the other SQL DDL and DML statements of the SIR DBS formulate in SIR SQL as for the kernel.

In [2], we conjectured from a motivating example that one could further design SIR DBS so that some present Create Table R schemes, i.e., defining the SAs only sufficed as SIR Create Table R schemes providing for LNF queries. The missing IE part of such SIR schemes, i.e., the IAs and the From clause, could be then determined from the DBS meta-tables, apparently easily. We qualified of *implicit* any missing IAs and the IE with, as well as the Create Table with. In particular, we qualify of *empty* the IE of the motivating example, i.e., whenever an implicit Create Table R defined R_ only, in fact, although named R instead. We supposed finally that every SIR DBS transparently preprocesses every implicit IE to the explicit one for any further processing.

The obvious gain from implicit schemes was reduced procedurality. In particular, the procedurality of any Create Table R with empty IE, was clearly the minimal possible for declaring SIR R with the intended (explicit) IE. For the motivating example, it meant in particular, that a base table R defined as at present could be in fact SIR R providing for LNF queries, instead of requiring the LN at present for the equivalent queries. Accordingly, clients could profit from LNF queries without any additional work for the DBA to define the IE, not to mention the Create View R for C-view R. In the same time, no normalization anomaly could ever follow.

We proposed consequently that, in addition to the capabilities proposed in [1] for every SIR DBS, every SIR DBS presumes also implicit every Create Table submitted. Iff it appears true, SIR DBS preprocesses the Create Table to the explicit one and processes the latter as in [1]. The general principles of such preprocessing remained yet to be stated. They seemed however simple.

We uphold the conjecture in [3]. We have shown that, beyond the motivating example, it holds more generally (a) if one considers FKs as E. Codd originally, apparently, [7], [13], hence not only as SQL at present, (b), if one considers every Create Table R with such FKs as the implicit Create Table R of a so-called *natural* SIR. We also argued that base table schemes usual at present fit (a) and (b). In other words, on a SIR DBS, every such Create Table R defines not "only" an SR, perhaps with VAs, as it would do at present, but the natural SIR R. Yet in other words, unlike today, usual DB schemes could provide for LNF queries, without any additional procedurality for the DBA. We outlined an implementation of SIR DBS supporting implicit Create Table for natural SIRs. The proposal extended accordingly the so-called SIR-layer, [1], intended to provide SIR SQL, while reusing internally some kernel SQL. We conjectured that one could provide with such a SIR-layer every kernel DBS, at the effort of a couple of months of programming only. This would make LNF queries *de facto* standard, making happier SQL clients, likely in millions today. Accordingly, we postulated every relational DBS to become SIR-enabled "better sooner than later".

Below, we continue analyzing DBs with SIRs that we call now simply *SIR DBs*. We first illustrate the above overview of SIRs with motivating examples. We also discuss more in depth our definition of FKs, [7], [13]. Next, we discuss natural SIRs, also more in detail than in [3]. We focus particularly on the rules producing the explicit scheme of natural SIRs from the present schemes of base table with FKs as the implicit ones. We then extend these rules to implicit schemes of SIRs other than the natural ones, with CAs especially. The goal is always the least procedural SQL schemes for SIRs for LNF queries as well as for SIRs for LNF and CAF queries.

Afterwards, we generalize Alter Table for SIRs defined in [1] to the implicit schemes. We show how one may provide then LNF queries to preexisting DBs, while keeping the legacy applications running. We outline the relational design of SIR DBs, extending the meaning of an NF in particular. We show that our proposal is decades overdue. Finally, we detail why to provide for SIRs over any present DBS kernel should be simple. In conclusion, we uphold our postulate of making every DBS SIR-enabled "better sooner than later".

II Natural SIRs

II.A Explicit SIRs By Example

Our framework for motivating examples is the "biblical" S-P DB, Fig. 1. S-P seems the first DB illustrating the relational model, [8]. It is also the mold for about every present DB. Hence, properties of S-P generalize accordingly.

Ex. 1. Suppose that DBA actually declares the base table S-P.SP as follows:

(1) Create Table SP (S# Char 5, P# Char 5, QTY INT Primary Key (S#, P#));

Here, the Primary Key definition is a table constraint. Suppose further that DBA declares the following view, after renaming SP to SP_:

(2) Create View SP AS Select SP_.S#,SP_.P#,QTY, SNAME, STATUS, S.CITY, PNAME,COLOR,WEIGHT,P.CITY From SP_ Left Join S On (SP_.S#=S.S#) Left Join P On (SP_.P#=P.P#);

Recall that no two relations in an SQL DB can share a name. To rename S-P.SP somehow is thus necessary for (2). Next, observe that (i) every SP tuple (S#, P#, QTY) at Fig. 1 is also logically the proper sub-tuple (S#, P#, QTY) of a tuple of view SP defined by (2) with the same (S#, P#), (ii) that view SP can contain only one such tuple since S.S# and P.P# are keys, and that, finally, (iii) (2) does not define any other tuples. These properties make view SP (2) the canonical view of SIR SP, i.e., the C-view SP, declared as follows:

(3) Create Table SP(S# Char 5 ,P# Char 5,QTY INT{SNAME, STATUS, S.CITY, PNAME, COLOR, WEIGHT,P.CITY From SP_ Left Join S On (SP_.S#=S.S#) LEFT JOIN P On (SP_.P#=P.P#)} Primary Key (S#, P#));

Here, the brackets {} delimits the IE. If the IE was disseminated within the Create Table, i.e., with IAs among SAs, then {} should bracket every sub-list of IAs, except for } of the last one, if followed by From clause. To avoid any name conflicts, we also suppose the brackets not allowed within the relation names dealt with by SIR DBS. If this is problematic, one may suppose other brackets, e.g., !! for MsAccess kernel.

Observe that the IE in (3) is the part of (2) defining the SQL projection on every IA in (3) and only on such IAs. But, SP_ referred to in (3) is the base of SIR SP, that is S-P.SP (1), preserved in (3), although implicitly renamed for the referencing in From clause, the same besides as in (2). Below, we refer to DB S-P with SIR SP (3) instead of SP (2) as to DB S-P1.

Fig. 2 shows the scheme of S-P1. Observe that the IE in (3) is there the explicit one for S-P1.SP, since it defines every IA and the From clause as in (2). Fig. 3 shows S-P1 content, given that of S-P at Fig. 1. For convenience, the name and content of every IA at Fig. 3 are *italic*. The SP content at Fig. 3 could actually be also the one of C-view SP (2). Provided however that, with our notation, every SP column name & value in straight font at the figure becomes *italic* as well. One would denote so indeed every column of every view, of view SP in particular.@.

Ex. 2. In S-P there is no referential integrity between SP and S. Hence SP could have tuple *t* with S# not in S, e.g., *t* = (S6, P1, 100). Suppose now that instead of From clause in view SP, one declares: From SP Inner Join S On… . View SP resulting from could not be C-view SP. Indeed, there would not be any tuples in the view with sub-tuple *t*. In contrast, such a view could be C-view SP if the referential integrity between S and SP was declared.@

Ex. 3. Recall that for a DML or a DDL statement *S*, the procedurality, say *p* (*S*), is the minimal number of characters (keystrokes) to express *S*, without convenience spacing especially. In Introduction we recalled that for every SIR R and C-view R, *p* (IE) in Create Table R is always smaller than *p* (Create View R). In our example, *p* (Create View SP) (2) is $p_1 = 156$. For the IE in (3), $p_2 = 112$, with the character count excluding '{', as replacing the usual SQL separator ',' that would be there without the IE, but including '}'. The latter also replaces a mandatory separator that is either ',' or a (single at least) space, depending on the context. So C-view SP is at least $(p_1 - p_2) / p_2 * 100 = 39$ % more procedural than the IE. In other words, the IE saves $(p_1 - p_2) / p_1 * 100 = 28$ % of *p* (C-view SP). All these savings for the DBA work provide for the same service for the client, i.e., the same queries. SIR SP (3) is thus a view-saver for view SP (2). Simply put, on a SIR DBS, adding view SP (3) to S-P, instead of creating SIR SP (3), would be just a waste of time.@

Ex. 4. Consider the need for every PNAME supplied by Smith. The corresponding SQL query to S-P, say Q_1, requires then the LN through the same equijoins between SP and S, and P in From clause as in (3) or equivalent joins. Hence, for the same need expressed as query, say Q_2, to S-P1, the From clause in (3) would do, while the selection on SNAME and projection on PNAME in Q_2 would be the same as in Q_1. Q_2 would be thus an LNF query. Being free of any LN, Q_2 would be then always substantially less procedural than Q_1, regardless of the actual LN in the latter. View SP (2) could provide for Q_2 as well, although with substantially more procedural scheme, as we've seen. The view is the only possibility at present.

The possibility of an equivalent LNF Q_2 to S-P1 instead of Q_1 to S-P with LN like in (3) or equivalent one for Q_1, clearly extends to any select-project part of Q_1. The extension implies only that, sometimes, some IAs in Q_2 may need qualified names. E.g., consider any query retrieving supplier's and part's CITY.@

Ex. 5. Suppose that P.WEIGHT is in pounds, while queries often need it in KGs. Suppose the latter provided by the attribute named, say, WEIGHT_KG, calculated as INT(WEIGHT * 0.454) and preceding WEIGHT. Adding WEIGHT_KG as an SA to P would create the normalization anomaly. Making it a calculated attribute in every query in need of, would increase the procedurality of the latter. The classical solution valid for every present DBS is to rather create the convenient view P. A query with WEIGHT_KG could invoke it by name only, becoming a CAF query for it, accordingly. It is easy to see however that view P could then be also C-view P for SIR P with explicit Create Table P as follows:

(4) Create Table P As (P# Char 4, PNAME Char 20, COLOR Char 10 {INT(WEIGHT * 0.454) As WEIGHT_KG} WEIGHT Int, CITY Char (30) {From P_} Primary Key (P#));

Observe that, again, we have *p* (IE) < *p* (Create View P). Observe also that for popular DBSs providing for VAs we spoke about in Introduction, WEIGHT_KG could enlarge P accordingly. E.g., at MySQL, the enlarged P could be:

(5) Create Table P As (P# Char 4, PNAME Char 20, COLOR Char 10, WEIGHT_KG As INT(WEIGHT * 0.454), WEIGHT Int, CITY Char (30), Primary Key (P#));

The obvious benefit is further reduction of procedurality, with respect to view P as well therefore. On (yet hypothetical) SIR-enabled MySQL, with the *canonical* implementation proposed in [1] and recalled in Section 5, say SIR MySQL, P could be in fact declared through the same statement. Formally, it is SIR P with (5) as the implicit scheme. The IE consists of the definition of WEIGHT_KG in the format of a VA only, i.e., without From P clause. This IE could be preprocessed to the explicit scheme (4) on SIR MySQL. However, for obvious practical reasons, we consider any Create Table with VAs as exclusive to SIR DBSs with the kernel DBS supporting such VAs, e.g., to MySQL for (5) here. Actually, SIR MySQL would forward it to MySQL as is. That one would consequently process it as MySQL would do for WEIGHT_KG VA presently. We recall the related details when we overview the implementation of a SIR DBS in Section VI.

Incidentally, this processing of WEIGHT_KG is the reason why there is no {} brackets around WEIGHT_KG there. The rationale will appear in Section VI. It will also appear that (4), with {} thus, would be the only possibility for SIR P on any SIR DBS implemented canonically over the kernel DBS not supporting any VAs, [1]. E.g., {} would be necessary for SIR MSAccess.

Besides, the vocabulary of the kernels supporting VAs extends the concept of base table to every table with SAs and VAs as well. The reason seems to be the presence of SAs, hence the use of Create Table still, although conveniently extended. This reason motivated us to generalize analogously the previously introduced term of the base of a SIR, [1]. Namely, in what follow, for any SIR R we now call *base* of R, i.e., R_, the projection of R on every SA and every CA, if there is any, with the scheme that could define a VA for the kernel. In practice, it means that one could define such a CA without {} brackets. Thus, e.g., for P as in (4), P_ = (P#,PNAME, COLOR ,CITY, WEIGHT). In contrast for P as in (5), we have: P_ = (P#,PNAME, COLOR ,CITY, WEIGHT_KG, WEIGHT).

The current example illustrates further that every present base table with VAs is in fact a specific SIR. Given the lasting popularity of VAs, since Sybase in the eighteens to our knowledge, we may rationally hope for the future popularity of SIRs, as providing more generally for CAF queries and/or LNF queries.

Ex. 6. Suppose that, in addition to LNF queries, SP clients wish for queries selecting QTY for some supplies that the former is always followed by the values named PERCENTAGE. The latter should provide for every supply selected, the percentage that the QTY there constitutes with respect to the entire supply of the part supplied. Having to specify the value expression calculating PERCENTAGE in every query in need, e.g., by a sub-query that follows, with its substantial procedurality thus, would be anything but practical for the clients. To simply add PERCENTAGE as an SA to SP after QTY, clearly would not make DBA happy. The only practical approach is to make it a CA of SIR SP or of C-view SP providing for LNF queries as well. The queries could invoke PERCENTAGE simply by name, becoming CAF queries for. One could create the required SIR SP, e.g., through the explicit scheme as follows:

(6) Create Table SP (S# Char 5, P# Char 5, QTY Int

{Round(100*Qty/(select sum(X.qty) from SP_ X where X.p# = SP_.p#), 3) as PERCENTAGE, SNAME, S.CITY, STATUS, PNAME, COLOR, WEIGHT, P.CITY From SP_ Left Join S On (SP_.S# = S.S#) LEFT JOIN P On (SP_.P# = P.P#)} Primary Key (S#, P#));

One may easily double check that IE above would be again less procedural than Create View SP for C-view SP. Recall also that PERCENTAGE cannot be a VA for any relational DBS at present. Hence SIR SP with and, more generally, any SIRs with CAs that cannot be VAs, are the only known view-savers for ‘their’ C-views at present. Hence, again, they would always be better choices for the DBA as well. @

II.B Foreign Keys for SIRs

Despite being fundamental to the relational model, the concept of the foreign key appears still surprisingly imprecise. The original definition is in [7]. Codd amended it later several times, [13]. The present definitions in textbooks or for popular DBSs differ from the original and are not all equivalent. For SIRs, we merge the original and SQL concepts. We thus call *foreign key* (FK) an SA and an SA only, perhaps composite, with qualified name, say, R.F, if (i) F is declared so through the familiar SQL Foreign Key table option, being then a *declared* FK. If not, (ii) a non-key attribute R.F is a *natural* FK iff (a) R.F is atomic and is not a part of a declared so-called below PKN FK and if (b) while SIR DBS processes Create Table R or Alter Table R, there is a unique in the DB PK named F and with qualified name R’.F, where R’ ≠ R, sharing the domain of R.F. In case (i), recall that one often qualifies R.F of *referencing* some key named, say, R’.C, primary or candidate on some DBSs. One often qualifies then also R’.C of the *referenced* key (RK). Likewise, one qualifies so R’, while R is the *referencing* relation. We retain this whole terminology for the natural FKs, with C = F and R’ the base table with.

Recall that at present, every FK must be declared as in (i), the referential integrity being mandatory. Actually, the Foreign Key constraint (clause, option…) was not in [7], but was introduced later by the SQL Standard Group. In contrast, the original FK concept is, likely, including the one we called the natural FK. Indeed Codd neither required to declare every FK nor considered the referential integrity mandatory for every FK, [7], [13]. Also, the requirement of sharing the domain in [7] implied the same proper name for the FK and the PK. The distinction between attributes and domain was introduced only later as well.

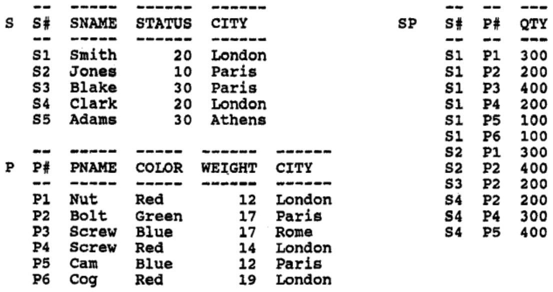
Besides, the central original idea for any FK A in [7] seems the “cross-referencing” from R to R’ realized by R.A and some R’.A as the *logical* pointers, [7]. In addition of having the same proper name as R.A, R’.A had to be a key, qualified then of *primary*. The qualifier indicated R’.A as the key for the referencing. Codd indeed expected R’ to often have more than one key (this origin of the *primary* qualifier seems largely forgotten). More in depth, the cross-referencing meant that every R-tuple *t* with *t*.A = *v*, references the only R’-tuple *t’* where *t’*.A = *v*, provided that such *t’* exists. Consequently for every *t* with A = *v*, one may determine *t’* through the relational calculus, an equijoin namely, regardless of the underlying physical representation. Codd conjectured the logical pointers more practical then the physical ones. The latter were the basic mode for referencing by the times of [7].

As known, Codd’s conjecture turned out right, despite vigorous opposition for years, if not decades. The benefit claimed by the logical referencing was later called the one of the *logical/physical data independence*. In particular, as also known, if a query needs some values in R together with some referenced ones in R’, then the (*left*) *FK-join*: R left outer join R’ on FK = RK in the query expresses the referencing, regardless of underlying physical data structures and changes to these. Likewise does the equivalent *right FK-join*, or, sometimes, the equivalent *inner FK-join* if the referential integrity is enforced. Recall in particular that RK can be composite, say declared as (C1, C2…Ck) with FK composite then as well, say declared as (F1, F2…Fk). Then, FK = RK means F1 = C1 and F2 = C2…and Fk = Ck. An RK may in particular be the PK, which means that (C1, C2…Ck) is also the content of the Primary Key constraint of R’. Recall finally that the FK-joins in queries constitute the already mentioned LN. See oldies for more on the theme.

Another consequence of Codd’s proposal was that one can formulate every above discussed query as if the attributes of R’ not in RK, were in R, except for the additional LN in the join clause of the query, i.e., the FK-join. While these R’ attributes typically could not be in R, since would create normalization anomalies, denormalizing R in consequence. Nevertheless, despite the additional procedurality due to the LN, countless examples devised after [7], has shown that any equivalent queries using the physical pointers implied by any navigational DBS in use by then, especially the Codasyl or IMS DBS would typically be several times more procedural. See the oldies again.

Recall that for a declared FK, RK may be a candidate key on some DBSs, i.e., any key not declared PK. This is nevertheless at best, a debatable possibility, since error-prone in the absence of the table option for candidate keys in SQL, unlike for the PK. Also FK and RK may have the same or different (proper) names. Recall that in case of a composite FK and RK, say F and C above, the same name means that every couple (Fi, Ci) ; i = 1…k shares a name. The rationale is that the referencing FK -> RK within an FK constraint is by attribute position at present, not by name sharing. Finally, we call below an FK *primary key named* (PKN), whenever RK is a PK and FK and RK share the name. Observe that a PKN FK can thus be natural or declared, while every natural FK is PKN by definition. It is so also for any FK fitting the original Codd’s definition. In other words, PKN FK concept qualifies (names) adequately Codd’s FK initial idea, to our best understanding of his wording. Non PKN FKs are a later possibility, introduced apparently, as we recalled, by the SQL designers.

Besides, the natural FKs appear the most popular, perhaps surprisingly for some. The rationale is the least procedural FK-joins in queries. Atomic declared FKs do the same, but require the Foreign Key constraint, cumbersome for many. Also, the referential integrity they impose, may sometimes contradict the application requirements.



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

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

SP	S#	P#	QTY
	S1	P1	300
	S1	P2	200
	S1	P3	400
	S1	P4	200
	S1	P5	100
	S1	P6	100
	S2	P1	300
	S2	P2	400
	S3	P2	200
	S4	P2	200
	S4	P4	300
	S4	P5	400

Fig. 1 S-P database

Furthermore, we suppose that, in every Create Table R with PKN FKs submitted to a SIR DBS, every PKN FK F and only such F implies specific IAs in the intended actual Create Table R. The latter is the explicit SIR R scheme, we recall. We call these IAs *Natural Inheritance,* (*NI*), in R *from* R’ or *through* F and define them as follows. Let A’ denote the ordered set of all the non-PK attributes of R’. Then, the NI through F in R consists of the set A’ of IAs (i) defined by the pseudo SQL query: select A’ From R_ FK-join R’; and (ii) either placed in R immediately after the last attribute in Create Table R that is not in any other NI through some PKN FK of R or placed after some such NIs. We also qualify then of *naturally inherited* every IA within the NI, the NIA, in short. Finally, we consider that any non-PKN FK in R scheme implies in contrast the referential integrity only.

Furthermore, observe that the above discussed obstacle of the denormalization of R imposing then the LN in the typical queries to base tables, disappears with SIR DBs. No IA can indeed create any normalization anomalies. The practical interest of the NI is to provide then for the LNF queries for present queries requiring the LN. E.g, as for Ex. 4, one may create every R with PKN FKs as SIR R with the explicit IE including the NA. This is clearly the case of (3). Operationally, as there, one can specify every NIA, as one would do for C-view R, e.g., for C-view SP (2) for (3). However, in practice any NI will rather be inferred from the implicit R scheme, as we will show soon. Then, more precisely, any query addressing any SAs of R reduced to R_, as at present and (ii) any non-PK attribute R'.A through some LN, may address instead NIA R.A, without LN then, e.g. as in Ex. 4. Any SIR R with the NI, e.g., (3) again, acts accordingly as a the view-saver for C-view R.

It follows from the above that every declared PKN FK F implies both the referential integrity and the NI. If one does not want the former, but still wishes the latter, one should not declare such F. Provided that, as usual, there is only one relation with PK F in the DB, for any atomic PKN FK F a natural FK F will result and fit the goal. Otherwise, one should change in addition the name of every PK F other than R'.F. Notice that such need should be rare, as schemes with several relations sharing a PK name seem infrequent. Anyhow, all this cannot work for any composite PKN FK F. One solution is to (a) add to R' a *surrogate* that is, we recall, makeshift atomic PK and (b) name it, let us say, C, uniquely for a PK in the DB. The composite RK becomes consequently a candidate key. Then, it suffices to replace F with C in R. The latter will be a natural FK hence will provide for the NI only, as wished. The classical Ex. 14 illustrates the case later on.

Finally, we suppose that no SA F of some R can become implicitly a natural FK because one issued some Create Table R" with PK named F or Alter Table R" that ended up with the PK R".F. In practice, it means that no such statement can enlarge R with the NI from R". A dedicated Alter Table R we discuss in Section IV is necessary.

Ex. 7. Natural FKs are in S-P, assuming S and P created before SP. SP.S# and SP.P# are the natural PKN FKs then, with S.S# and P.P# being the respective RKs. The original verbal description of S-P scheme indicates indeed that each pair has a common domain. Finally, as the natural FK, SP.S# in SP scheme (1), supposed now the implicit SP scheme for S-P1.SP, will imply the NI from S, consisting of {SNAME…S.CITY} in (3). Likewise, SP.P# will imply the NI from P. As it will appear formally in next section, SP scheme (1) will lead then to SP scheme (3) as the explicit scheme, with (1) as an implicit one.

The original description of S-P also does not mention any referential integrity. Nevertheless, at Fig. 1, every SP tuple respects this constraint for both FKs. Regardless, one may insert, e.g., P7 into SP, without the presence of P7 in P. The feature can be useful, e.g., if DBA allows for the later insert of P7 data into P. If the referential integrity was in contrast required for a pair, e.g., (SP.S#, S.S#), one should declare in Create Table SP or Alter Table SP the usual: Foreign Key (S#) References S(S#).., including the On Delete and On Update options perhaps. SP.S# would become the declared (atomic) PKN FK. On the other hand, if in S-P as on Fig. 1, DBA created SP before S and P, then, for Create Table SP, neither SP.S# nor SP.P# would be natural FKs anymore. Consequently, there would not be both NIs in the explicit SIR SP scheme (1). This would make S-P1.SP = S-P.SP in fact, until one perhaps alters S-P1.SP as we discuss later.@

II.B *Basic Natural SIRs*

We will now show that Create SP (1), can be an implicit scheme for SP (3). Recall that (1) defines all and only SAs of SIR SP (3), hence, we have *p*(IE) = 0 there. The property frees thus DBA in need to create (3), from any additional procedurality otherwise required. We will show that the property generalizes in fact to any SIR R qualified of *natural* in [2].

Def. 1. Suppose that SIR R has PKN FKs $F_1 \ldots F_K$ and for every F_k ; k=1..K ; R contains the NI through F_k denoted as $\underline{A'}_k$. Suppose also that for every NI, referencing some base table R', one qualifies with R' name every attribute in NI in name conflict otherwise with any other attribute, including an SA. Let *R_* denotes all the SAs of R_, including every VA, we recall, if there are any. Then R is *natural* iff the explicit Create Table R has the following or any equivalent form:

Create Table R (*R_* {$\underline{A'}_1 \ldots, \underline{A'}_k$ From R_ Left Join R'_1 On R_.F_1 = R'.F_1 Left Join R'_2 On R_.F_2 = R'_2.F_2… Left Join R'_k On R_.F_k = R'_k.F_k};.

We refer to all the NIs in any SIR R simply as to *NI in R* or as to *NI in R through $F_1 \ldots F_k$*. Besides, the following obvious proposition follows.

Prop. 1. C-view R for a natural SIR R has the following pseudo SQL scheme:

Select R_.*, $\underline{A'}_1 \ldots, \underline{A'}_k$ From R_ Left Join R'_1 On R_.F_1 = R'.F_1 Left Join R'_2 On R_.F_2 = R'_2.F_2… Left Join R'_k On R_.F_k = R'_k.F_k

Def. 2. We say that a natural R is a *basic* one iff every R' is an SR or is SIR R' with every IA declared as VA.

Ex. 8. On a SIR-enabled DBS, S-P1.SP illustrated at Fig. 2 is a natural SIR. First, S and P referenced each through a natural, hence PKN, FKs S# and P#, obviously differ from each other, as required. Then, the IAs: SNAME, STATUS, S.CITY constituting the NI through S# from S, follow QTY. The same occurs for SP.P# and P, except that they follow the NI through S#. Both IAs CITY are qualified for obvious reasons. Finally From clause is conform to Def. 1. In contrast, S-P1.SP enlarged further with PERCENTAGE would not be natural, since the latter would be an IA outside SP_ and outside both NIs. In any case both NIs in SP constitute the NI in SP.

Accordingly, in our terminology, every IA following S# till SA P# is sourced in S. Next, SP naturally inherits each and all of them. Respectively, same is true for every IA sourced in P. All these IAs together constitute for SP its NI through the foreign keys and they naturally enlarge SP_. Finally, SP is a basic natural SIR.@

Observe now the following easy properties of natural SIRs:

Prop. 2. Suppose that Create Table R in some DB defines at present an SR R with PKN FKs. Accordingly, consider the following generic formula for such Create Table R, where '…' designates some SAs or VAs or none:

(7) Create Table R (…, F_1,… F_K…. <table options>);

Then, (7) can be an implicit scheme for natural SIR R with the NI through F_1,… F_K and R_ defined by (7).

Proof. To prove (7), one should provide a deterministic algorithm for the explicit IE as in Def. 1. We sketched the latter in [3]. In the next section, we provide a more complete formulation.@

Accordingly, given (7), for every natural SIR R, we say sometimes that R is so *for* R_ or *for* base table R with (7) as the actual (explicit) Create Table R.

Ex. 9. Create Table SP (1) and Create Table SP (3) are clearly conform to Prop 1. Hence S-P1.SP (3) is the natural SIR for S-P.SP (1) and for the base SP_ of SP (3). Finally, it is the unique natural SIR for both.@

Recall furthermore that, given the quest for non-procedurality, the empty IE of (7) is a definitive advantage over the explicit one in Def. 1 and in C-view R, by the same token. Also, on every SIR-enabled DBS, providing for the natural SIRs in particular, DBA could therefore always create a natural SIR R with no more work than for sole R_ as a stand-alone SR R at present.

Ex. 10. On a SIR DBS, Create Table SP (1) would suffice for SP (3). The S-P scheme would define S-P1. The DBA would have no additional work to define S-P1.SP.@

In the same time, as we already hinted for S-P1, the clients would gain LNF queries to SP, regardless of whether the DBA uses

(3) or, obviously better, (1) for S-P1.SP. We have hinted that this may be true more generally for every natural SIR R. We will prove it now for basic natural SIRs.

Prop. 3. Suppose that a DB has a base SR R with the scheme that could also be the implicit one of a natural SIR R. Let us denote a base table that R references as R'. Next, consider a select-project-join query Q_1 (i) projecting on some SAs of R or on some non-key SAs in one or more of R's, and (ii) where R and each R' are joined through some of the PKN FK-joins, i.e., forming thus, likely, a typical LN at present. Next, consider the LNF query Q_2 addressing SIR R only, through the same select-project clauses as Q_1, except, perhaps, that some IAs are qualified. Then, for every SR R and every Q_1, Q_2 is equivalent to Q_1. SIR R is consequently a view-saver for such LNF queries. It is also possibly the least procedural one, in the sense of possibly empty IE.

Proof. For every Q_1, suppose that one renames SR R to R_ and replaces all the FK-joins of Q_1 with the FK-joins in (5). The relation defined by these joins contains the same attributes as the natural SIR R explicitly defined by (5), except, perhaps, that (i) some attribute names became qualified or (ii) the order of the attributes is different. Given the properties of left outer equi-joins, the modified Q_1 is equivalent to the original, provided that if select-project clauses of original Q_1 referred to an attribute name that became qualified, then every such name in modified Q_1 is qualified as well.

The latter From clause can be equivalently modified to the nested one, where the inner query (a) has both the select clause and the FK-joins as in (5) and (b) it is named R within the outer From. Whether the latter R designates SIR R on SIR-enabled DBS, (as we tacitly suppose here), or designates "only" view R at present, does not matter. The overall result is that for every Q_1, query Q_2 with the same select-project clause as the latter query but referring to R only in From clause, instead of containing the inner query defining R, is the LNF query to R equivalent to Q_1. Accordingly, view R is a view for LNF queries for any such Q_1. SIR R is consequently a view-saver for any such LNF queries. R scheme is possibly the least procedural one, since DBA may choose the implicit one with empty IE (what every DBA will likely do in practice then).@

Ex. 11. Consider the need for SP.S#, SNAME, CITY for every supply. Every Q_1 to S-P must have then the LN through the FK-join SP Left Join S. E.g., one may issue Q_1 as:

(8) Select SP.S#, SNAME, CITY From SP Left Join S On SP# = S.S#;

After renaming SP to SP_, one can equivalently replace Q_1 with:

(9) Select SP_.S#, SNAME, S.CITY From SP_ Left Join S On SP# = S.S# Left Join P On SP.P# = P.P#;

Notice that CITY became qualified. The latter query is in turn equivalent to the following one with nested From:

(10) Select SP.S#, SNAME, S.CITY From (Select SP_.S#, SNAME, STATUS, S.CITY, SP_.P#, PNAME, COLOR, WEIGHT, P.CITY, QTY From SP_ Left Join S On SP_.S#=S.S# Left Join P On SP_.P#=P.P#) As SP;

Finally, whether SP designates now in fact view SP (2) or SIR SP (3), then the latter query becomes simply Q_2 as follows:

(11) Select S#, SNAME, S.CITY From SP;

Observe that p (Q_1) = 56 and p (Q_2) = 31. Thus the LN alone in Q_1 is almost as procedural as Q_2. Hard to see why an S-P client having choice, could prefer Q_1 to Q_2. @

Notice that, usually, DBS would process Q_2 by the standard query modification approach that would walk backward the above steps towards Q_1. Similar conclusions will hold more generally for every select-project-join query to S-P.SP and S or P, with LN through the FK joins over FKs of SP. Accordingly, view SP is a view for LNF queries, requiring the above FK-joins otherwise. SIR SP is then a view-saver for the same LNF queries. It is also possibly the least procedural one, since one can define the SAs only.@

Recall also that every IA A of a natural SIR R, is a natural one itself. By definition, it thus has the same name as an attribute of some base table R' that R references, called also *source* of R.A. Thus one may consider that for every query Q to R only that we qualified of an LNF one, for every IA A that Q perhaps addresses, Q addresses then in fact some R'.A. One may say then that Q is an LNF query not only to R, but also, indirectly through every R.A addressed, to every base table R' that is the source of. For some, that meaning of an LNF query is perhaps the primary one even, [24].

Accordingly to this terminology, one can formulate Prop. 3 in an alternative way that some may find more appealing:

Prop. 3bis. Suppose that a DB has a base table R with the scheme of the SAs that makes it the implicit one of a natural SIR R. Let R'_1… be every base table that R references. Next, let Q_1 be a select-project-join query addressing some SAs in R scheme and some attributes of R'_1 or of R'_2 or…, with every join being a left FK-join preserving R or with any join equivalent to. Then, for every possible Q_1, there is a query Q_2 addressing SAs in R scheme and such that (i) Q_2 is equivalent to Q_1, (ii) Q_2 is an LNF query also to every R' that Q_1 addresses (through the joins) and (iii) Q_2 is the select-project part of Q_1 with From R clause only, except that some attribute names in Q_1 may end up qualified. R is consequently a possibly the least procedural view-saver for such LNF queries.@

Observe finally that if Create Table R defining at present an SR R only, may define the natural SIR R instead, then it provides for the discussed attractive LNF queries, at no additional data definition cost for DBA. We now describe the algorithm effectively providing for that capability, i.e., of inferring the explicit natural SIR R scheme from the one of the SR R, on any popular DBS.

II.C Inferring Explicit Schemes of Basic Natural SIRs

As already stressed, suppose every referenced relation to preexist the referencing one. Suppose also that SIR-enabled DBS gets Create Table R with SAs only and, may be, with some of these being declared FKs. The scheme may be thus an implicit scheme of the natural SIR R. SIR-enabled DBS processes then the Create Table as follows. The algorithm mainly generalizes our motivating examples. The outcome is the explicit Create Table R. We specify the rules only verbally, omitting easy details the actual implementation would require. We take for granted that the implicit Create Table R defines SAs only. We also consider only the *canonical* implementation of a SIR-enabled DBS in [1], we recall in Section VI below. The SIR-enabled DBS creates and manages then every SIR R as base table R_ and C-view R within the kernel DBS.

Alg. 1. 1. (Determine every natural FK). For every (atomic) SA R.A that is neither a primary key nor a declared FK or within such FK, check in the meta-tables, provided by every popular SQL DBS at present and often named SYSTABLES for base tables and SYSVIEWS for views, e.g., in DB2, whether there is a unique relation (named) R", with the primary key sharing the name and the domain of R.A. If so, R.A is a natural FK. Next, (i) if R" (name) does not end up with '_', then R' := R". Else (ii) if R"/'_' is not in SYSVIEWS etc., then R':= R". Else, R' is not a basic natural SIR. Then, check the rules for compound natural SIRs we outline later.

2. (Processing every PKN FK). For every PKN FK with R' determined in (1) or analogously for any declared FK (recall that, on some popular DBSs, this may require to check in SYSTABLES that RK is a PK, since it could be a candidate key), retrieve from SYSTABLES the name of every non-PK attribute of R'. Then (iii), place all these names, qualified if needed, in Create Table R, as in (a) in Def. 1.

3. (Create From clause). Let F_1… be the PKN FKs enumerated in the left-to-right order in Create Table R and R'_1… be the referenced relations. Suppose for the form of the string below, for every composite F, the simplified notation we indicated before.

Then, after the last SA and before the table options, insert the string in the form of: From R_ Left Join R'$_1$ On (R_.F$_1$ = R'$_1$.F$_1$) Left Join R'$_2$ On (R_.F$_2$ = R'$_2$.F$_2$)….@

Ex. 12. We skip the easy but tedious proof of the rules. We only show that they build the explicit Create Table SP (3) from (1). Suppose thus S-P1.S and S-P1.P already created. Rule 1 produces names (S, S#) and (P, P#). For the former, Rule 2 finds S# in (1). It thus inserts SNAME…S.CITY, right after S#. Likewise, it inserts PNAME...P.CITY right after SP.P#. Finally, Rule 3 builds From clause in (3) and terminates the explicit IE.@

For the DBA, as already hinted to, the rules mean simply *p*(IE) = 0. They thus mean free bonus of *zero* additional time for creating, instead of S-P.SP, the natural SIR SP (3), with its *p* = 112 (explicit) procedurality of the IE. This, to provide the clients with also free then bonus of typically far less procedural LNF queries. For the DBA again, an even bigger bonus is with respect to the present situation. One saves indeed 100% of procedurality *p* = 152 of Create View SP (2) for the same purpose.

II.E *Compound Natural SIRs*

A *compound* natural SIR R inherits through some FKs from SIRs. These can be natural perhaps compound themselves, or others. In other words, a non-PK attribute of an R' can now be an SA or an IA. Operationally, as usual today, we suppose again every R' being created before R. By the same token, we suppose that later alterations of any R' schemes do not cascade to R. Here are motivating examples of compound natural SIRs. They seem framework for frequent practical cases.

Ex. 13. Suppose one alters S-P scheme as follows. An additional relation CG (CITY, GPS) stores uniquely for each city the GPS location. Suppose further that on a SIR-enabled DBS, one creates CG first, then S and P with their S-P schemes, Fig. 1 and SP through its scheme (1), at last. CG has no FKs, hence its scheme above defines an SR. Then, the S scheme has only one FK that is the natural FK S.CITY, referencing CG.CITY. S is now therefore a basic natural SIR, with S-P.S scheme as the implicit one and the following explicit scheme, inferred through the rules above:

(12) S (S#, SNAME, STATUS, CITY {GPS From S Left Join CG On S.CITY = CG.CITY})

The explicit Create Table S scheme for (13) is obvious to figure out. P becomes a basic natural SIR analogously. But, the NI for SP defined by Prop. 1 includes now also two IAs: S.GPS and P.GPS. Hence, SP becomes the compound natural SIR. Its explicit Create Table evolves to:

(13) Create Table SP (S# Char 5 {SNAME, STATUS, S.CITY, S.GPS} P# Char 5 {PNAME, COLOR, WEIGHT, P.CITY, P.GPS} QTY INT {From SP_ Left Join S On (SP_.S# = S.S#) LEFT JOIN P On (SP_.P# = P.P#)} Primary Key (S#, P#));

LNF queries to SP may now address both P.GPS and S.GPS. But, it is easy to see that Alg. 1 does not let to infer (13) from (1) anymore. It needs the completion we show soon.

Finally, it's instructive to appreciate the procedurality gain with LNF queries searching for GPS data. E.g., suppose the search for SP.S#, SNAME, S.CITY, S.GPS and SP.P#, PNAME, P.CITY, P.GPS, as well as QTY, for every supply with QTY > 100. For S-P with SR CG added as base table, every SQL query Q$_1$ expressing the search would need some LN, e.g., the nested one as follows:

(14) Select SP.S#, SNAME, S.CITY, S.GPS, SP.P#, PNAME, P.CITY, P.GPS, QTY From SP_ Left Join (S Left Join CG On S.CITY = CG.CITY) On SP_.S# = S.S# Left Join (P Left Join CG P.CITY = CG.CITY) On SP_.P# = P.P# Where Qty > 100;

The LNF Q$_2$ to S-P1 would be in contrast simply:

(15) Select SP.S#, SNAME, S.CITY, S.GPS, SP.P#, PNAME, P.CITY, P.GPS, QTY From SP Where Qty > 100;

We have *p* (Q$_1$) =203 and *p* (Q$_2$) = 83. Thus Q$_1$ is almost 2.5 times more procedural than Q$_2$. Besides, no wonder that the complexity of LN through nested joins in (15) is not what most clients like best. The result would not change much if, e.g., one familiar with properties of joins unnested these while formulating Q$_1$ or replaced some with the left natural ones etc. Recall finally that all these advantages of Q$_1$ come for free at the data definition level for DBA. I.e., if looked upon as SIR implicit schemes, the "classic" Create Table S, Create Table P and Create Table SP of S-P DB, could make instantly possible for Q$_1$, instead of forcing Q$_2$ only at present.@

Ex. 14. Suppose that the DBA defines for S-P also the well-known base table providing the allocations of the supplies in SP to jobs:

(16) Create Table SPJ (S#..., P#..., J#..., ALLOC…, Primary Key (S#, P#, J#), Foreign Key (S#, P#) Referencing SP (S#, P#));

Here, (S#, P#) is a declared composite PKN FK. Suppose also that J# is not a natural FK for some reason. For a SIR-enabled DBS, the above scheme is the implicit one of SIR SPJ. Neither S# nor P# is a natural FK in SPJ, since both are within a declared PKN FK. Hence, SPJ would be a natural compound SIR where the explicit scheme naturally inherits every non-key SA and IA of SP:

(17) Create Table SPJ (S#..., P#..., J#..., ALLOC, {QTY, SNAME,..S.CITY, PNAME…P.CITY From SP_ Left Join SP On SP_.S# = S.S# And SP_.P# = P.P#} Primary Key (S#..., P#..., J#), Foreign Key (S#, P#) Referencing SP (S#, P#));

Notice that the referential integrity between SP and SPJ would be enforced, as for every declared PKN FK and as for any FK at present. If it is not desired, then, as said generally before, one should not declare (S#, P#) as an FK. The natural attributes in NI through (S#, P#) that one wishes to preserve for LNF queries and which are neither in the NI through S# nor through P#, should be then explicit. Actually, this amounts to QTY only. A better approach to inherit QTY instead implicitly as well is through the already discussed technique of a surrogate, say SP# here, added to SP, i.e., enlarging the implicit scheme of SP to SP (SP#, S#, P#, QTY). The composed key (S#, P#) is no more the primary one. SPJ may become SPJ (SP#, J#, ALLOC), with SP# being the PKN FK and QTY becoming an implicit IA, since in the NI of SP#.

Anyway, whether QTY is implicit or explicit in SPJ, a SIR-enabled DBS will provide for the LNF queries addressing any SAs of SPJ and through the IAs, any SA or IA of SP. Hence, through IAs of SPJ, such query will be also able to, transitively and transparently, address every attribute of S and of P. Thus, again at no cost for the DBA, the client could, e.g., search for SNAME, PNAME and available QTY of every part allocated to job 'J1' through the select-project only LNF Q$_1$:

Select SNAME,PNAME,QTY From SPJ Where J#='J1'.

In contrast, supposing SPJ (16) and the original SP, the necessary LN in Q$_2$ below, would make the latter more than three times more procedural and dreadful for many:

Select SNAME,PNAME,QTY From SPJ Left Join (SP Left Join S On SP.S#=S.S# Left Join P On SP.P#=P.P#) On (SPJ.S#=SP.S# And SPJ.P#=SP.P.#) Where J#='J1'. @

Ex. 15. Suppose now for S-P1 that one enlarges S-P.P with the calculated attribute WEIGHT_KG as in Example 5. Suppose further that DBA again creates S and P before SP, with P with WEIGHT_KG becoming SIR P, we recall. Then SP implicitly defined through SP_ scheme named Create Table SP, would remain a natural SIR. However, it will be a compound one this time, regardless one defined WEIGHT_KG as the VA or as if it was an IA of C-view P. The explicit Create Table SP would become:

(18) Create Table SP (S# Char 5, P# Char 5, QTY INT {SNAME, STATUS, S.CITY, PNAME, COLOR, WEIGHT, WEIGHT_KG, P.CITY From SP_ Left Join S On SP_.S# = S.S# LEFT JOIN P On SP_.P# = P.P#} Primary Key (S#, P#));

SP provides now for the LNF queries to WEIGHT_KG as well.@

Given these examples, the enhancement to the rules for basic natural SIRs in the previous section can be as follows. The new need is to recognize for every R' whether itself it is not a SIR.

<u>Alg. 2</u>. (i) - Rule (1) in Alg. 1 states that R can be a compound natural R if R"/'_' is in SYSVIEWS etc. Consider so now. Then, R is effectively a compound natural SIR, since R"/'_' is a SIR. Hence set R' to R' := R"/'_'.

(ii) - The NI in R from R' is now defined through SYSVIEWS etc. This one should be, as usual, every IA of view R' defined there other than every IA inherited from RK in R'_. The latter is to be found through SYSTABLES etc.

(iii) - Perform finally rule (3).

We skip the easy proof in favor of the motivating example.

<u>Ex. 16</u>. Consider again S-P altered as in Ex. 13. Then rule (1) above will find for S# that S_ table in SYSTABLES etc. is R" with S as R"/_ in SYSVIEWS etc. The control will pass to rule (i) that will set S as R'. Likewise, for P#, it will find R' := P. Rule (ii) will then determine for S from SYSTABLES etc. that S.S# is the RK, hence it will find out from SYSVIEWS etc. that {SNAME…S.GPS} is the NI for SP.S# in SP. Likewise, it will determine {PNAME…P.GPS} as the NI for SP.P#. Then, after applying rule (3) again, the end result for From clause would be the one in (13) and, altogether, Create Table SP (13) will be the explicit one in our case.

Likewise, for S-P altered as in Ex. 15, Alg. 1 for the basic natural SIR and SYSTABLES etc. alone will determine for SP, the NI from S. For P in turn, to find it out, Alg. 1 will call Alg. 2 for the compound natural SIR. The explicit Create Table SP (18) will be the overall result.@

III Other SIRs with PKN FKs

One can define implicit schemes also for SIRs other than natural ones, provided they have PKN FKs. The following proposition shows it. We denote the explicit Create Table R as R_E and as R_I the implicit one derived as follows.

Prop. 4. Consider SIR R with PKN FKs defined by R_E in the form that follows. We denote as A any attribute other than any of those in NI. We suppose that at least one A is an IA other than a VA. Notice that R cannot be then a natural one. Next, we denote as $\underline{A}$ all such IAs. Then, we denote as NI all the NIAs and as NI-joins all the FK-join clauses of the NI. We further denote as <$\underline{A}$ def. clause> the part of From clause defining $\underline{A}$ and not containing any of joins in <NI-joins>, if there is any such part. Next, brackets [] denote optional parts of the statement, as usual. Finally, we omit all the {} brackets around IAs, except, perhaps, of the initial '{', if used instead of the usual SQL '(' and of the final '}' terminating the From clause. Supposing now that R_E is as follows:

R_E = Create Table R (|{ A_1…,A_2…A_K…,NI From R_ [<$\underline{A}$ def. clause>] NI-joins} [<Table constraints>]) [<Table options>];

Then, R_I is R_E without NI and NI-joins.@

Proof. R_I suffices, as one can complete it towards R_E through the pre-processing obviously almost the same as the one for natural SIRs. The only difference is the eventual completion of the <$\underline{A}$ def. clause>. We therefore skip the easy, but tedious completion specs.

We also skip the trivial generalization of Alg. 1 & 2 providing for a unique R_E inference algorithm for both: natural SIRs and the ones dealt with here. @

Notice that (implicit) IE of R_I resulting from Prop. 4 cannot be empty. Unlike was the implicit IE for any natural SIR, recall.

The rationale for R_I's here is obviously $p\ (R_I) < p\ (R_E)$, again. The gain is thus even greater with respect to p(C-view R). The typical needs for the explicit IAs seem as follows. (i) R has one or more calculated IAs (CAs), each defined through a value expression inheriting from SAs or from other IAs in R or from attributes of some R'≠R, or defined by a sub-query. Then, (ii) for some FK F, R may have for privacy, only some or even none of the natural IAs through F. Or, (iii), F may have the same IAs as in NI, but, for query convenience, displaced within R or renamed for some. Finally, (iv) F_1 and F_2 in R may share R'. This is contrary to the assumptions of NI, we recall. Here are motivating examples, illustrating these needs. The procedurality savings that appear are always about or above 50%, sometimes with R_E being several times more procedural. These are clearly substantial saving, by any practical meaning of the term.

<u>Ex. 17</u>. (i) Consider SIR P from Ex. 13 enlarged with WEIGHT_KG placed after WEIGHT. Supposing that the kernel does not provide for VAs, the implicit Create Table P would be:

(19) P_I = Create Table P As (P# Char 4, PNAME Char 20, COLOR Char 10, WEIGHT Int {INT(WEIGHT * 0.454) As WEIGHT_KG} CITY Char (30), Primary Key (P#));

SIR-enabled DBS would enlarge then (19) to the explicit Create Table as follows:

(20) P_E = Create Table P As (P# Char 4, PNAME Char 20, COLOR Char 10, WEIGHT Int {INT(WEIGHT * 0.454) As WEIGHT_KG} CITY Char (30) {GPS From P_ Left Join CG On P_.CITY = CG.CITY} Primary Key (P#));

The explicit IE is visibly more than twice procedural than the implicit one. Alternatively, suppose now that MySQL is the kernel. The implicit Create Table P could contain the VA WEIGHT_KG:

(21) P_I = Create Table P As (P# Char 4, PNAME Char 20, COLOR Char 10, WEIGHT Int, WEIGHT_KG As INT(WEIGHT * 0.454), CITY Char (30), Primary Key (P#));

P_E would keep then the definition of WEIGHT_KG as the VA, but will remain obviously as procedural as (22).@

<u>Ex. 18</u>. (i) Suppose that for some security reasons, no attribute of Supplier should be in SP available for LNF queries, except for S#, somehow renamed so to hide its relationship to S. SP_I could simply be:

(23) SP_I = Create Table SP (X Char 5, P# Char 5, QTY INT Primary Key (S#, P#));

Since SP.P# remains a natural FK, SP_E would be:

(24) SP_E = Create Table SP (X Char 5, P# Char 5, QTY INT {PNAME, COLOR, WEIGHT, P.CITY From SP_ Left Join P On SP_.P# = P.P#} Primary Key (S#, P#));

SP_I is again visibly about a half of the SP_E..

(ii) Suppose now that only the attributes SNAME, CITY of S-P.S should be visible to LNF queries to SP. SP_I could be:

(25) SP_I =,Create Table SP (X Char 5, P# Char 5, QTY INT {SNAME, CITY From SP_ Left Join S On SP_.X = S.S#} Primary Key (X, P#));

For both (i) and (ii), the renaming of SP.S# was necessary, as it would be a PKN FK otherwise. It would imply then NI in the explicit scheme, obviously contradicting the specs. SP_E for (26) would be again substantially more procedural, by about 50% visibly again:

(26) SP_E =,Create Table SP (S≠ Char 5, P# Char 5 {PNAME, COLOR, WEIGHT, CITY} QTY Int {SNAME, CITY From SP_ Left Join S On SP_.S≠ = S.S# Left Join P On SP_.P# = P.P#} Primary Key (S≠, P#));

<u>Ex. 19.</u> Suppose that SP should be as the natural one, but with the additional IA named WEIGHT_T. This one should indicate for every supply, its total weight, supposed to be WEIGHT * QTY, whenever one knows WEIGHT of the supplied part. One wishes also WEIGHT_T to follow QTY in SP. WEIGHT_T is not a natural

IA, since defined through a value expression. Neither, it could be a VA, since WEIGHT is not in SP. Nevertheless the following SP_I is OK:

(27) SP_I = Create Table SP (S# Char 5, P# Char 5, QTY INT {WEIGHT*QTY AS WEIGHT_T} Primary Key (S#, P#));

SP_E would be:

(28) SP_E = Create Table SP (S# Char 5, P# Char 5, QTY INT, {WEIGHT * QTY AS WEIGHT_T, SNAME, S.CITY, STATUS, PNAME, COLOR, WEIGHT, P.CITY From SP_ Left Join S On (SP_.S# = S.S#) LEFT JOIN P On (SP_.P# = P.P#)} Primary Key (S#, P#));

Indeed, both S# and P# in SP_I continue to represent all their natural IAs. Next, WEIGHT_T would need FK-join between SP_ and P in SP_I if it should become an explicit one as is. But, this clause can be omitted otherwise, as defining also the NI from P. The procedurality of the implicit IE, say p_1, in SP_I is $p_1 = 23$. For the explicit IE, we have $p_2 = 134$. Thus, the explicit IE is almost six times more procedural. The procedurality ratio between entire SP_E and SP_I is obviously smaller, but still again SP_I is visibly about twice less procedural than SP_E. Notice finally that the procedurality, say p_3, of Create View SP for C-view SP that is the only practical possibility for WEIGHT_T at present, is $p_3 = 171$. Hence, it is more than seven times greater than p_1, making the implicitly defined SP quite a view-saver.

Ex. 20. SP should get as the last attribute, a calculated IA, say PERCENTAGE. For every supply, the latter should be the percentage that the QTY of that supply constitutes with respect to the entire supply of the part supplied. Sub-query below defines PERCENTAGE, leading to SP_I:

(29) SP_I = Create Table SP (s# Char 5, p# Char 5, qty Int {(select Round (100*Qty / (select sum(X.qty) from SP_ X where X.[p#] = SP_.[p#]), 3)) as PERCENTAGE} Primary Key (S#, P#));

Without the NI inferred, the From SP_ clause should follow the attribute list. But, it is not here, since would be redundant with the FK-joins referencing S and P. SP_E would therefore be:

(30) SP_E = Create Table SP (s# Char 5 {SNAME, S.CITY, STATUS} p# Char 5 {PNAME, COLOR, WEIGHT, P.CITY} qty Int, {Round(100*Qty/(select sum(X.qty) from SP_ X where X.p# = SP_.p#), 3) as PERCENTAGE From SP_ Left Join S On (SP_.S# = S.S#) LEFT JOIN P On (SP_.P# = P.P#)} Primary Key (S#, P#));

It is easy to calculate that the implicit IE is 2.1 times less procedural than the explicit one. In other words, the implicit IE saves 53% of the explicit one. Hence, it is even more efficient as the view-saver, (how much?).

Ex. 21. Consider the following DB named E-M providing data on employees. Some employees are managers. Each employee has an ID named E#. Each manager has the ID M# that is some E# renamed. An employee may share work time among several managers. An SA FRC indicates the fraction of work time spent by employee E# for manager M#. A manager may get M# and start to manage some employees before all the other data of those or of her/himself are in E-M. One could accordingly define E-M as:

(31) EMP (E#..., NAME…, TEL…, DEP…, Primary Key (E#));

(32) EM (E#.., M#..., {M.NAME, M.TEL, M.DEP} FRC… {From EM_ Left Join Emp M On EM_.M# = M.E#} Primary Key (E#, M#);

EMP is an SR and EM is visibly a SIR. Suppose EMP created first. E# is then a natural FK. M# in contrast is neither a natural nor a declared FK. For our meaning of the FK concept, it is so just an SA in EM useful for the DBA to define the IAs with selected data about every manager, when already in EMP. Besides, as E# is a PKN FK, EM should contain the NI through E#, while it does not. (32) shows thus EM_I. The preprocessing would expand it to the (substantially more procedural) EM_E that we leave as an exercise. EM would provide for LNF queries on every attribute of employees or of managers in E-M.@

IV Altering SIRs with FKs

One can alter every SIR R with FKs through the Alter Table R, specified for SIRs in [1]. In particular, one can define a new IE, through the SIR-specific clause, termed *IE clause*. The clause defines for any R, the new IE, regardless of the existing one, if any. In fact, the clause basically defines the C-view R scheme, as one would define it through Create View R or Alter View R. For this reason, as those that will appear in Section VI, the IE clause is mandatory whenever one adds, renames or drops an SA in R.

We call *explicit* the IE clause defining the C-view as just discussed. It has to contain the explicit IE in particular. The rationale for such IE clause is that the IAs within intermix with the SAs of R_, as Ex. 22 below illustrates. The names of the latter serve as placeholders. One specifies the explicit IE clause for any new IE in this way. Besides, the clause can be sometimes *implicit*, i.e., without the entire content of the explicit one. A SIR DBS pre-processes every implicit IE clause into an explicit one for any further processing.

For even lesser procedurality, we now consider that for a relation with PKN FKs, the implicit IE clause avoids to enumerate any NIs. It may happen then that these NIs are the only IAs to add. We suppose that one may write then the (implicit) IE clause simply as IE {}.

An IE clause may in particular add IAs other than VAs to an SR R or to SIR R with CAs declared as VAs only. Such R may pre-exist the upgrade of a DBS to a SIR-enabled one. Observe furthermore that for every SR R with PKN FKs, the clause IE {} makes R a basic natural SIR, without affecting any pre-existing data. The rewrite rules for pre-processing implicit IE clauses into the explicit ones with NIs are easy figure out.

Recall also that every natural SIR R, hence also the one resulting from Alter Table R for an SR R, brings to R the free bonus of the LNF queries to any SA of R and to any of the non-PK attributes of any R'. For every preexisting query, the outcome remains the same, except for every query referring to R through '*' or R.*. Recall that such queries are not recommended for applications, hence rare. Notice also that for every preexisting SR R, in the absence of any of the discussed alterations, the DBS upgrade would not affect any preexisting queries.

Ex. 22. Consider S-P in use on some present DBS. Suppose this DBS becomes SIR DBS. Every preexisting query to S-P will provide the same outcome. Then, Alter Table SP IE {}; will upgrade SP to S-P1.SP. The rewrite rules will process the Alter into the explicit one, [1], visibly more procedural by far:

(33) Alter Table SP IE {S#, SNAME, S.CITY, STATUS, P#, PNAME, COLOR, WEIGHT, P.CITY, QTY From SP_ Left Join S On SP_.S# = S.S# LEFT JOIN P On SP_.P# = P.P# };

After (34), SP would be a natural SIR. New queries to SP may also be now LNF. Every existing one will provide for the same outcome, except for queries with: '*' or 'SP.*' obviously. The alteration will not affect any existing S-P.SP (stored) content, the one at Fig. 1 especially.@

Observe finally, that, as already discussed for the creation of SIRs with FKs, for every existing SR R with PKN FKs, one should perform IE (), only if (i) every R' preexists and, (ii) if R' also has PKN FKs, then (i) was already applied to R' etc. E.g., if all the relations of Ex. 13 are preexisting SRs, one should alter CG first, S and P after and SP, at last. Otherwise for a declared FK, an error could appear, while a natural FK could silently miss an IA. R would provide then for fewer LNF queries, obviously.

V Relational Design for SIR DBs

The relational design has for goal the "best" collection of base table schemes for a DB. Usually, it means the smallest possible number of 4NF SRs. About always in practice then, each of these is also in 5NF and, even, in less popular 6NF, [10]. Several methods for are known. Whatever is one's favor, let us refer to the result as $\underline{S}$. In practice, every $\underline{S}$ contains SRs with (declared) FKs. Then, $\underline{S}$ is accompanied with some scheme, say O, of the base tables creation order, so that for every referencing table R, every referenced one, say R' as before, exists when R is being created. A run-time error occurs otherwise on every DBS of our knowledge at present.

We call *relational design for SIR DBs*, any methods similarly aiming at "best" collection of base table schemes that can be SRs or SIRs. "Best" means here for every SIR, first the least procedural schemes, hence the implicit ones whenever possible. Next, "best" continues to aim, for every base table, at the NF making it free of anomalies. But a new general issue is then, as we'll show now, that the concepts of i-th NF for i > 1 and of BCNF need a revision. Then, any method has to take to the account the natural FKs as well. We now address successively both issues.

V. A SIR DB Specific Normal Forms

Observe from our example that, e.g., before one adds WEIGHT_KG, P is BCNF (and so on). It's no more after. Indeed, FD: WEIGHT -> WEIGHT_KG, makes P in 2NF at best. But, as an IA, WEIGHT_KG, does not create any normalization anomalies. Unlike SA WEIGHT_KG would do. The lossless decomposition through Heath theorem, [18], making P without IA WEIGHT_KG, hence in 3NF again, would be senseless. Hence, P with WEIGHT_KG as IA should not lose its BCNF "status". Similar situation occurs for SP with WEIGHT_T, given FD: (WEIGHT, QTY) -> WEIGHT_T there. The issue was already observed by some clients of VAs, [28].The practical way out is to expand the definition of the normal forms so to take to the account that no IA may introduce a normalization anomaly. Our proposal is thus as follows:

Def. 3. A 1NF relation, SR or SIR, is *i*NF ; *i* = 2..6 ; or in BCNF iff the relation formed by all its SAs is in *i*NF or BCNF.

Then, both P without WEIGHT_KG and with WEIGHT_KG as an IA are in BCNF. In contrast, P with WEIGHT_KG declared as an SA, i.e., without the value expression, but only enumerated for each value of WEIGHT, would not be in 3NF even. Like P would not be also for the usual definition of 3NF and BCNF. Similarly, - for SP and WEIGHT_T. Notice that the definition applies to present relations with VAs. We recall that all these are specific SIRs. Finally, it is backward compatible for SRs Altogether Def. 3 is long overdue thus.

V.B SIR DB Specific Design Steps

For a SIR DB, some SRs in $\underline{S}$ may become SIR bases. Let us call $\underline{S}^f$ the schemes forming the intended DB possibly with SIRs. One basically seeks $\underline{S}^f$ where every SIR scheme is the least procedural possible, i.e., the least procedural implicit one. As it appeared, e.g., for S-P and S-P1, we may typically expect $\underline{S} = \underline{S}^f$ with every SIR being a natural one, basic or compound. On the other hand, as already abundantly discussed, DBA may wish some SRs enlarged with some explicit IAs, CAs especially. Also with respect to O's schemes, if O^f, is the creation order scheme for a SIR DB with all SRs and SIR bases in $\underline{S}$, it may happen that O acceptable for $\underline{S}$ is not for $\underline{S}^f$. E.g., O = (SP, S, P) is OK for S-P, but, as one could see, not for S-P1. Besides, if the goal is a SIR DB upfront, then it is obviously not useful to define any O, just perhaps to alter it later to some O^f anyhow. Altogether, beyond every present method for $\underline{S}$, designing a SIR DB may require some of the following SIR-specific steps.

(i) For every R in $\underline{S}$ with PKN FKs, (a) rename every PKN FK that should not be so anymore in $\underline{S}^f$, in case there is any such FK.

(ii) For every R in $\underline{S}$, for every explicit IA A that should enlarge R, if there is any such A, every CA in particular, add A scheme. Generally, - as an explicit scheme, including the explicit LN, if the latter will not be in in the explicit From clause of R scheme. Alternatively, for a CA, add it as the (implicit) VA scheme, whenever more appropriate.

(iii) Choose O^f such that for every R (a) for every natural FK, R' preexists Create Table R, (b) for every IA in R, every source relation preexists Create Table R as well.

Step (i) may occur for some popular design methods. E.g. it would be the case if one aims at SIR DB from Ex. 18(ii), while obtaining $\underline{S}$ of S-P at Fig. 1 as the result of the lossless decompositions of the universal relation, [24], through Heath Theorem, [18], until every base table is in BCNF at least. Step (ii) is optional as well. One would actually need it for SIR DB resulting from Ex. 18(ii), but not, e.g., for S-P1.SP, with $\underline{S} = \underline{S}^f$. Also, in this step, DBA adds every CA A scheme. Notice that the DBA may then have the choice of more than one R. The only necessary condition is indeed that A is functionally dependent on the primary key of chosen R. E.g., the DBA may add WEIGHT_KG scheme either to P or to SP only. However, it is easy to figure out that choosing for A the relation that is the earliest in O^f among all those possible for A, can make some queries searching for A values less procedural. Thus, e.g., WEIGHT_KG should rather be in P.

Step (iii) is obviously mandatory for every practical SIR DB. The choice of O^f is obviously more restrictive than of O for the same SRs, including those forming the bases of SIRs. Every O^f has to indeed to create, for every natural FK and every explicit IA, CAs especially, every referenced base table. Recall finally that for every DB, hence for SIR DBs as well if the universal relation for $\underline{S}$ has non-trivial multivalued-valued dependencies (MVDs), the decomposition should start with the Fagin's Theorem, [14]. The Heath Theorem applies then to the resulting intermediate SRs with FDs only. More details would be out of scope here.

Besides, our prior various examples abundantly motivate the above steps. E.g., for S-P and S-P1 DBs, we have $S = S^f$ as at Fig. 2 and only step (iii) applies. O^f could be (S, P, SP) or (P, S SP). The final result for SP was the natural basic SIR at Fig. 3, we recall. Likewise, to design S-P and S-P1 variants in Ex. 13, again $S = S^f$ and only step (iii) applies. O^f is either (CG, S, P, SP) or (CG, P, S, SP) obviously. Whatever the choice is, in the SIR DB, CG would remain an SR, S and P would be basic natural SIRs and SP would end up a compound natural one, we recall.

Next, in Ex. 18 (ii) again, although SP.P# is the only PKN FN, one still needs O^f = (S, P, SP) or O^f = (P, S SP). The rationale is the LN towards S in From clause of SP, making S referenced by SP.SNAME and of SP.CITY. Thus all three steps apply to this DB. Finally, the design of S-P1 variants with WEIGHT_KG or PERCENTAGE, or WEIGHT_T, will apply any favorite DBA's design method, followed by steps (ii) and (iii) only.

VI IMPLEMENTING SIRS

The canonical implementation consists of the, so-called, *SIR-layer*, interfacing every client and the DBA, [1]. Fig. 4. Every SIR-layer uses SQL kernel of the already discussed some kernel (DBS). Together, this creates a SIR DBS, in our terminology. Above SIR-layer, the relational constructs for any clients and DBA are: SRs, SIRs and views. We suppose the SIR SQL dialect for these constructs to be the kernel one, with the DDL syntax extended as above amply discussed. Underneath, i.e., for the kernel, there are necessarily only two constructs: (i) SRs, perhaps with VAs and (ii) views. We suppose then that (a) for every SIR DB, say D_S, there is in the kernel a DB, say D_K, termed *kernel* DB with the same name as D_S, (b) for every SR or SIR with every IA declared as a VA and for every view in D_S, SIR-layer creates the same table in D_K. In other words, SIR-layer simply forwards to the kernel every Create Table or Create View statement for such tables. For the Foreign Key constraint, this may however imply the naming rule for D_S we discuss soon. In contrast, for every SIR R with some or all IAs declared otherwise than VAs for D_S, SIR-layer creates atomically within D_K the following *canonical representation* of R, *CR* in short:

(i) base table R_. This can thus be an SR or an SR with all the VAs. R_ has then also every table constraint and option declared for SIR R. Except perhaps for the Foreign Key constraint modified as above described.

(ii) C-view R. In particular, one may create the latter with only the proper attribute names in conflict being qualified (prefixed), e.g., as in (3). The conflicting SAs are then prefixed with R_. Alternatively, one may simply qualify every name, with no consequences for the queries to C-view, at any kernel we are aware of.

Fig. 4 illustrates the creation of S-P1 DB as D_S and D_K, with CR of SP. SP_I scheme defines SP in D_S. SIR-layer first preprocess (PP) SP_I to SP_E scheme, Then, it processes (P) the whole S-P1 schema to D_K scheme. That one defines base tables S, P, SP_ and C-view SP, canonically implementing S-P1. (SP_, SP) is there the canonical representation (CR) of SP.

The naming rule for FK constraints in D_S referred to above, is that whenever for an FK, one intends to reference SIR R' with IAs other than VAs, then the FK constraint should actually reference R'_. The rule results from the CR and from both Codd's and SQL definitions of FKs. These require indeed every RK to be an SA. For any SIR R', RK is thus within the base of R'_ in D_S, hence within the base table R'_ in D_K. In contrast, no RK can be declared as an attribute of R'. This would make indeed the constraint to reference C-view R' in the D_K. The attempt to create R with such an FK constraint would fail.

Note as future work that it is, however, possible to use the CR without that rule. Namely, one can design the variant of the canonical implementation where even if R' is a SIR in D_K, the FK constraint in D_S names R' anyway. The advantage would be only one referencing rule for any FK constraint in D_S, i.e., one always references the base table name only. The price would be a more complex implementation. The latter should indeed include additional preprocessing of every Create Table for D_S, finding for every declared FK; whether R' is a SIR. SIR-layer could determine this from SYSTABLES and SYSVIEWS. If R' designates a C-view there and that there is also base table R'_, then SIR-layer should replace R' with R'_. SIR-layer would generate then the Create Table R_ and Create View R statements for D_K only after processing in this way every FK in the statement. At present, all this does not seem however worth the gain.

Whatever is the variant, SIR-layer obviously easily extracts Create Table R_ and Create View R for the CR, i.e., for D_K, from the explicit Create Table R for D_S, i.e., for SIR-layer. The Create Table R_ contains indeed every attribute scheme outside {} and every table option. Some '}' may then get replaced with ',' in Create Table R_. The Create View R in turn, copies the name of every attribute in Create Table R outside some {} and every scheme of an IA, together with every element '*' or in the form of 'R'.*, if there is any, as well as the From clause terminating the IE, necessarily within {}. The last '}' ends up the IE, also necessarily. It also terminates Create View R, being replaced with ';' there. The rationale for the simplicity and correctness of all this parsing is our Section II assumption on the {} brackets that neither bracket can be in a relation name in the SQL dialect supposed for the SIR DBS. To appreciate how the absence of such brackets complicates the implementation of SIR-layer, practice our examples.

Likewise, one can easily see how SIR-layer should process for the CR, every Alter Table and Drop Table, [1]. Finally, SIR-layer simply forwards every Create View R to the kernel, whether over SRs, SIRs or views, E.g., it would do so for any views of S-P1.SP.

Next, we suppose the SIR-layer to pass through every Select query for D_S. The kernel with D_K processes it then towards the tables with the same names. For any SIR R within the query, the kernel will thus process it towards C-view R. It may in particular internally optimize the execution time of such queries, e.g., by materializing some IAs of view R for faster joins, [16], [17], [27]… SIR-layer also forwards unchanged to the kernel, every updating query, i.e., Insert, Update, or Delete. However, since an IA may be not updatable, we suppose as "safe" policy for the canonical implementation that any such queries name only tables for SIR DB that are base tables for the kernel's DB under the CR. These have thus only SAs and, perhaps, some VAs, for the kernel providing for those. If, for a SIR DB, R is an SR or a SIR where every IA is a VA, the query should name R. Thus, e.g., an update should start as: Update R…., as at present. If, in contrast, R is a SIR with IAs other than VAs, the query should refer to R_.

Beyond that rule, the correctness of an updating query to the canonically implemented SIR-layer would depend on kernel's *view update* capabilities, [11]. E.g., the LNF query to S-P1, say, Q_1: Delete From SP Where SNAME = 'Smith'; would be directed thus by every kernel towards view SP. It then would be correct for MS Access and MySQL. Both provide indeed for updates of outer join views, of view SP thus. In contrast Q_1 would fail on SQL server kernel and SQLite. None provides indeed that capability, forcing the updating of SRs instead. Instead of Q_1; the correct query Q_2 could then be: Delete From SP_ Where S# In (Select SP.S# from SP Where SNAME = 'Smith');. Notice that Q_2 would be correct for MS Access and SQL Server as well. The visible drawback with respect to Q_1 is greater procedurality, because of the LN through the 'In' clause.

The above functions of SIR-layer, where outlined already in [1] and [5]. Their canonical implementation appeared simple. The only new function here is the preprocessing of FKs. It appears simple to integrate. As stated within Section II.C, to find whether an attribute is a natural FK, kernel's meta-tables, e.g. SYSTABLES, should suffice. Same is valid for determining whether a declared FK is PKN and for locating for every PKN FK, every SA or VA to become the source for the NI within the basic natural SIR. Likewise, exploring the meta-table(s) for views, e.g., SYSVIEWS, should suffice for every compound natural SIR. Both meta-tables should suffice consequently for any other SIR with FKs, with CAs in particular.

Altogether, to reuse typical present schemes of SRs with PKN FKs as the implicit schemes of natural SIRs appears simple, perhaps surprisingly simple. Simple means here a couple of months of programming at most. Recall that, unlike today, any such schemes become view-savers for LNF queries. It appears similarly simple to extend this processing to the implicit schemes of SIRs with FKs and explicit IAs, the CAs other than VAs especially. Recall that SIRs with the latter become view-savers for LNF & CAF queries involving such CAs as well.

VII CONCLUSION

On a SIR-enabled DBS, a typical present scheme of a stored relation R with FKs, defines a natural SIR R. LNF queries to base tables at no cost for the DBA are the bonus. Also, SIRs with only some PKN FKs or with CAs, still provide for LNF or CAF queries, through Create Table R usually substantially less procedural than possible at present. This is also a bonus for the DBA.

Next, it appears easy to generalize the present relational DB design to SIR. In particular, an overhaul of the Normal Forms emerges then. The decades old practice of VAs makes this overhaul long overdue. Notice to DB textbook authors.

Finally, it looks simple to add the preprocessing of PKN FKs to the previously proposed canonical implementation of SIR-layer. We plan the proof-of-concept prototype as the next step. The Python's beta version for SQLite3 kernel is actually already there. Recall that the SQLite3 apparently serves an estimated trillion+ DBs (VLDB 22). More generally, any embedded kernel SQL should suffice for the canonically implemented SIR DBS. The road to make dough is wide open.

Altogether, relational schemes with FKs were visibly not read as they should be, from the very inception of the relational model. LN with its often felt dreadful joins, within otherwise simple queries to base tables, was the penalty for generations. Likewise was the alternate need of views to offset the shortcoming. Same for the CAF queries, either presently limited to VAs or requiring dedicated views as well. Relational DBSs should become SIR-enabled "better sooner than later". Making LNF & CAF queries to the base tables the standard, at last. It will be a long overdue service to SQL clients, likely in many millions these days.

ACKNOWLEDGMENTS

We are grateful to Ron Fagin for invitation to present this material at IBM Almaden Research Cntr., March 2020. We thank also Berthold Reinwald and C. Mohan for helpful comments. Likewise, we thank Darrell Long for his March 2020 invitation to talk about at UCSC Eng. as well and to Sheldon Finkelstein for inspiring remarks.

REFERENCES

[1] Litwin, W. SQL for Stored and Inherited Relations. 21st Intl. Conf. on Enterprise Information Systems, (ICEIS 2019), http://. www iceis.org/?y=2019, 12p.

[2] Litwin, W. Manifesto for Improved Foundations of Relational Model. EICN-2019. Procedia Computer Science, 160, (2019), 624-628, Elsevier, (publ.).

[3] Litwin, W. Natural Stored and Inherited Relations. EUSPN-ICTH 2021, Procedia Computer Science, Elsevier (publ.), 8p.

[4] Litwin, W. Stored and Inherited Relations. arXiv:1703.09574 [cs.DB]. March 2017.

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

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

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

[8] Date, C., J. 2004. *An Introduction to Database Systems*. Pearson Education Inc. ISBN 0-321-18956-6.

[9] Date, C., J., & Darwen, H., 1991. Watch out for outer join. *Date and Darwen Relational Database Writings*.

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

[11] Date, C., J. View Updating and Relational Theory. O'Reilly, 2012.

[12] Date, C., J. Type Inheritance & Relational Theory. O'Reilly, 2016.

[13] Date, C., J. E.F. Codd and Relational Theory. Lulu. 2019.

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

[15] Beeri, C., R. Fagin, J. H. Howard. A Complete Axiomatization for Functional and Multi-valued Dependencies in Database Relations", SIGMOD-77, 47-61. 4.

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

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

[18] Heath, I., J., 1971. Unacceptable file operations in a relational data base. *ACM SIGFIDET '71 Workshop on Data Description, Access and Control*, 19-33.

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

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

[21] Litwin, W. Ketabchi, M., Risch, T., 1992. Relations with Inherited Attributes. HPL. Palo Alto, CA. Tech. Rep. HPL-DTD-92-45, 30.

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

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

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

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

[26] Stonebraker, M. Moore, 1996. D. *Object-Relational DBMSs: The next Great Wave*. Morgan Kaufmann. 2nd Ed. 1998.

[27] Valduriez P., 1987. Join indices. *ACM TODS*, 12(2), 218–246.

[28] Does a computed column break 3NF ? https://dba.stackexchange.com/questions/247095/does-a-computed-column-break-3nf-third-normal-form , 2019.

```
Create Table S (        Create Table P (        Create Table SP (
S#  Char 5,             P# Char 5,              S# Char 5
SNAME Char 30,          PNAME Char 30,          P# Char 5
STATUS Int,             COLOR  Char 30,         QTY Int {SNAME, STATUS, S.CITY, {PNAME, COLOR, WEIGHT, P.CITY
CITY Char 30,           WEIGHT Int,             From SP_ Left Join S On SP_.S#=S.S# Left Join P On SP.P#=P.P#}
Primary Key (S#));      CITY Char 30,           Primary Key (S#, P#));
                        Primary Key (P#));
```

Fig. 2: S-P1 scheme with explicit SP scheme. The implicit one would be that of S-P.SP, outside the brackets {}.

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	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Par
P6	Cog	Red	19	London

Table SP

S#	P#	QTY	*SNAME*	*STATUS*	*S.CITY*	*PNAME*	*COLOR*	*WEIGHT*	*P.CITY*
S1	P1	100	*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*	*Rome*
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*	*Athens*	*Cam*	*Blue*	*12*	*Paris*

Fig. 3: S-P1 content. IAs are *Italic*. S-P1.SP is the natural SIR for S-P.SP.

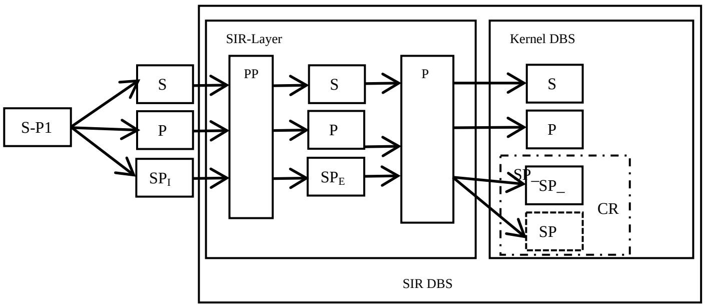


Fig. 4: Creation of S-P1 SIR DB.