Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 198 (2022) 171–178

Procedia Computer Science

www.elsevier.com/locate/procedia

The 11th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH 2021)
November 1-4, 2021, Leuven, Belgium

Natural Stored and Inherited Relations

Witold Litwin*

Université Paris Dauphine PSL, Paris 75016, France

Abstract

A *stored and inherited relation* (SIR) is a 1NF stored relation (SR) enlarged with *inherited* attributes (IAs). We show that one may consider a usual scheme of an SR R with foreign keys as implicitly defining a *natural* SIR R. IAs do not introduce normalization anomalies, while a usual select-project-join query to SR R with foreign keys becomes select-project only towards natural SIR R. Such queries become thus *logical navigation free* (LNF), without any additional data definition work, unlike at present. We show how the usual SR schemes provide for natural SIRs. We show also that making a popular DBS SIR-enabled should be simple. Pre-existing relations with foreign keys could be trivially converted to SIRs, preserving existing applications, while providing for LNF queries for new ones. We postulate every major DBS becoming SIR-enabled "better sooner than later".

Keywords: Relational Model; Logical Navigation ; Stored and Inherited Relations ; SQL Databases; Cloud Databases; Big Data

1. Introduction

The relational model defined by Codd has two 1NF constructs, [6]. A *stored* relation (SR) consists of *stored* attributes (SAs). One cannot calculate values of these from the DB scheme. A view consists of *inherited* attributes, (IAs). These are calculable through a stored query, called *view scheme*, from SAs or previously defined IAs. Originally, one supposed IAs calculated only. Later, it appeared sometimes practical to also store view snapshot, refreshed whenever needed. Such views were named *materialized*, [14], [13], [16]. We proposed recently a 3rd construct, called *stored and inherited relation* (SIR) [1],[2]. The concept roots in [5], part of the trend to harness

* Corresponding author.
Email address: Witold.litwin@dauphine.fr

1877-0509
10.1016/j.procs.2021.12.225

inheritance in relational DBs, [S96], [P], [11]… One may see every SIR R as enlarging some SR R with some IAs. More precisely, every tuple of SR R gets then additional values or nulls, calculated as if they were in a specific view R detailed below. Besides, the primary key of SR R remains that of SIR R and SIR R does not have any tuples with key value not in SR R. IAs and SA in SIR R may get intermixed. We suppose finally the original meaning of IAs, i.e., as calculable only, unless we state otherwise. Hence, basically, the enlargement is a virtual one only. Note that no SIR is a materialized view since SAs are not calculable from the scheme, by definition.

Next, we refer to SR R within SIR R, as to R_ and call R_ *base* of SIR R. Then, we operationally define view R as mathematically (logically…) equal to SIR R. The only difference is then physical. SIR R preserves every SA S of SR R, qualified then also of R_.S. It also preserves every tuple of SR R. View R has instead an IA S, inherited from SR R. This one is qualified of R_.S as well. We suppose indeed for view R that SR R is renamed by default to R_ as well. To define any view named R, some renaming of SR R is anyhow necessary, as no SQL DB can have relations sharing a name. For every SIR R tuple, there is then an equal view R tuple and vice versa. View R as just defined always exists. We termed it *C-view*, for *conceptually equal* (to SIR R) view, [1].

We called *inheritance expression* (IE) the scheme of IAs in SIR R. An IE is basically the scheme of all the IAs in C-view R that are also IAs in SIR R. IE includes thus in particular the From clause. In [1], we proposed SIR-specific extensions to SQL. To create any SIR R accordingly, one may issue Create Table R defining as usual every SA in R_ and the IE. The IAs defined so in Create Table R may intermix with the SA-schemes there, accordingly to what we already stated for SIRs in general. We termed such an IE *explicit*. We also termed *SIR-enabled* DBSs the (yet hypothetical) DBS supporting SIRs.

In particular, the IE in Create Table precedes all the table options if there is any. As usual for SQL, these options define the composite primary key and every foreign key (FK). They may also define any atomic primary key, although one rather declares the latter within its attribute scheme. Additional rules in [1] may make an IE *implicit*. A less procedural, i.e., with fewer required keystrokes, IE results from. Every implicit IE is transparently transformed into the explicit one, e.g., as one transforms '*' at present. The general advantage of SIRs over views is that for every Create View R for C-view R, there can be a less procedural IE in SIR R. Likewise, any C-view is more procedural to maintain, [1].

Recall that SRs of an SQL DB suffice for every SQL query to the DB. That is likely why SRs were also called *base tables*. DBA may nevertheless create additional views of these, making queries to some SRs less procedural. Such a view typically presents an SR let it be R, as if R got additional attributes, enlarging every R-tuple accordingly. Observe that this is actually what every C-view does. The additional attributes, becoming IAs in the view, as does also every SA of R, serve the goal basically in two ways:

(i) The IAs make queries usually select-project only, instead of select-project-join over different SRs otherwise. The term *logical navigation* (LN) customarily designates the join clause. Hence, one says also that queries to the view become *logical navigation free* (LNF) and one talks about *LNF-view*. The SRs other than R are sources of the additional IAs in the LNF-view. A usual query to an LNF-view can simply project the view on values selected in some IAs sourced in R or in some additional IAs. Without the join clause, in practice, the query becomes always less procedural and felt more natural by the clients. Especially, - if the LN involves outer joins [8], [15].

(ii) The IAs avoid value expressions defining otherwise the same attributes in queries. (a) These expressions can be arithmetic with scalar functions. (b) They can be also with aggregate functions, perhaps defined through sub-queries. In both cases, the view defines every such IA. Queries may then simply invoke IAs names, becoming less procedural, necessarily.

The additional attributes would be usually inconvenient to manage if they were within the SRs, hence SAs. For (i) the normalization anomalies would about always appear. For (ii), frequent updates would often be needed. Both drawbacks do not occur for any views with IAs calculable only.

Recall that by definition, every SIR R is mathematically C-view R. SIRs clearly may thus also attain goals (i) or (ii). As said also, the IE can always be less procedural then the C-view. **This is our rationale for SIRs**. Notice that this rationale already made some SIRs popular since decades. These are SIRs providing for case (a) in (ii) above. An implicit IE defines then every IA so that it could be a *computed* (dynamic, virtual…) attribute on a present DBS with that capability, e.g., MySQL, Oracle…., starting from Sybase in eighties. It was evidently observed that it is always less procedural to define the computed attributes than the equivalent view. In particular, the former are often qualified then as *view savers* and usually as base tables as well. We extend the qualifiers to any SIRs accordingly. For more on computed attributes see countless tutorials or early research on, e.g., [17]. Of course no one realized by then that every relation with computed attributes is a specific SIR.

In [2] we pointed out that one may attain (i) in particular for SIRs with FKs. The IAs should inherit then basically all the SAs and, perhaps, also all the IAs, in the referenced relations. It appeared also that for some such SIRs, qualified of *natural,* one should be able to infer the explicit IEs only from the present SR schemes. In other words, the implicit IE is then empty, i.e., has zero procedurality. Consequently, not only queries become usually LNF, but, also, to define the IE does not bother the DBA whatsoever. In popular terms, this makes every natural SIR, a perfect *LNF-view saver*. Recall that procedurality gains were already the rationale for the relational model with respect to the Codasyl model, dominant then, [6], [12]. They also motivated the research on LNF queries already in eighties. They are finally the reason for the lasting popularity of computed attributes with respect to equivalent views (ii), we recall.

We now analyse natural SIRs more in depth. We term *natural* every IA of a natural SIR as well. A referenced attribute can be a natural IA itself, etc. The exceptions are IAs that would result from a circular, i.e., recursive, referencing. We prohibit this one in general for any SIRs, as SQL does so already for any views.

We first revisit the concept of FK. Next, we formally define natural SIRs. We then state the rules for generating from usual schemes of SRs with FKs the explicit ones of natural SIRs. Next, we show more in depth why natural SIRs are perfect LNF-view savers. We show also that making a popular DBS SIR-enabled so to provide for natural SIRs could be simple. Pre-existing relations with FKs could be trivially converted to SIRs, preserving existing applications, while providing for LNF queries for new ones. We conclude that every popular DBS should become SIR-enabled "better sooner than later".

2. Foreign Keys Revisited

Despite being fundamental to the relational model, the FK concept seems imprecise, [12]. For our purpose, we call *foreign key* an attribute, perhaps composed, such that every of its values could be that of some uniquely chosen (stored) primary key, unless the Foreign Key constraint references a (stored) candidate key. The former is the definition by Codd, maintained in all his papers of our knowledge. The latter is a possibility on some popular DBSs, e.g., MySQL. It follows that both, the foreign and, basically, the primary key, share a domain, perhaps composed as well. The latter key is usually called *referenced* key (RK). The FK can implicitly share the proper name with the RK. This is usual for atomic keys, less usual for composite ones. For the former, we speak about *natural* FK and RK. Composite FKs seem less "natural". They are by far less frequent and, when they are the only choice with existing data, many prefer to add an atomic primary key, often called *surrogate*. Alternatively to implicit name sharing, one can explicitly designate the RK. This is necessary if more than one primary key with the proper name of the FK exists or the RK has a different proper name. We talk then about *declared* key. For the above discussed reasons, we also consider that every composite FK or not referencing a primary key has to be a declared one. Finally, FK as just defined does not imply the referential integrity. Neither did the original relational model, claiming it definable only, [6]. But for SQL, hence generally at present, every FK must be a declared one, through Foreign Key clause of course. This one guarantees then also the referential integrity somehow.

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. "Biblical" S-P database.

Ex. 1. Consider the "biblical" S-P DB at Fig. 1, with suppliers and parts in M:N relationship, allegedly the original DB of the relational model, [7]. S-P is the mould for about every practical DB and serves most of our examples. Here, SP.S# and SP.P# have the same proper names as S.S# and P.P# respectively. The original verbal description of S-P supposes also that each pair shares the domain. Each pair, i.e., (SP.S#, S.S#) and (SP.P#, P.P#), constitutes thus natural FK and RK. Both pair provides for inheritance by SP that we detail below. At the figure, both pairs respect the referential integrity. However, the constraint is not declared. Hence, e.g., an insertion of (S1,

P7 200) destroying that integrity for P and SP remains valid. Also, P6 may change to P60 in P if a need occurs, without impacting or being impacted by use of P6 in SP. Otherwise, e.g., if such a change required or implied same change to SP, a client searching for QTY of P6 supplied by S1, could get heart attack learning that such supply never existed.

If the referential integrity is nevertheless required for a pair, e.g., (SP.S#, S.S#), one should declare in Create Table SP through the usual Foreign Key (S#) References (S.S#), with usual On Delete and On Update options perhaps. The clause would provide then both: inheritance and referential integrity. Likewise, if DBA changes SP.P# to Part#, the usual Foreign Key (Part#) References (P.P#) would provide both: inheritance and referential integrity.

The inheritance we mentioned above follows the obvious conceptual one through the natural FKs in SP, of every attribute of S and of P other than RKs. This was apparently Codd's motivation under the term of "cross-referencing" in [6]. E.g., for every supply, its supplier identified with SP.S# has also a name, status and city. Which are in fact those it has as supplier within S for S.S# = SP.S#, i.e. the same natural FK and RK. We show in next example how this inheritance impacts explicit Create Table SP for natural SIR SP. Similar observations apply to parts. This is what makes sense at present of usual queries to SP and S or P with the LN through S.S# = SP.S# or P.P# = SP.P# equijoins. E.g., consider the query for the name of every supplier of screws.

3. Natural SIRs

3.1. The Concept

Consider a base R_ with every FK referencing a primary key, each in a different 'referenced) relation R', R being none of them. Actually, a SIR with FKs is usually so. Suppose that SIR R enlarges R_ only with IAs sourced in every R' and as follows. First, for every R', for every attribute other than the RK, there is one and only one IA in R, with the same proper name as in R'. IA name can be qualified if needed. Next, for an SQL DB, where the order of attributes matters, for every R', all the IAs in R sourced in R', are placed right after the last attribute of the FK in their source order. Then, for every FK value *v*, perhaps composed, if RK *v* exists, then make every same-name-IA value in the R-tuple with *v* equal to that in R'-tuple. If any of the referenced attributes is null in the tuple, then set to null the IA. Finally if the R'-tuple with v does not exist, set to null in the R-tuple every IA sourced in R'.

In less technical words, one may say that for such SIR R, every FK value stands as an abbreviation of all the values it references. However, every such value other than that of the RK is implicit in SR R. In the natural SIR R, each such value becomes explicitly an IA value. IE defines how every such value should be calculated. Somehow like in 'LNF' for 'Logical Navigation Free', every letter stands for the FK value logically pointing towards the entire word it implicitly references. We "calculate" the remaining letters upon the need.

For every natural SIR R, we term it also *natural* for SR R, hence also for the base R_. Finally, we recall that we qualify of *natural* every IA of SIR R and its IE.

It's easy to see that for every SR R with FKs hence for every base R_, we have only one natural SIR R. Next, suppose that we determine somehow the explicit scheme of R from that of SR R renamed in the process to R_. We consider then SR R scheme as the implicit one of R. The rules we propose below produce the explicit Create Table R for the natural SIR R accordingly, from every Create Table for SR R scheme with FKs, hence also from every present SR R scheme. We formulate these rules after the next motivating example.

Ex. 2. In the wake of Ex. 1, recall that in S-P, each pair: (SP.S#, and S.S#) and (SP.P#, and P.P#) constitutes the natural FK and RK. The discussed inheritance by SP from S and from P follows. Suppose now that we enlarge the original SP to SIR SP with the following IAs, designated in *Italics*:

(1) SP (S#, *SNAME, STATUS, S.CITY*, P#, *PNAME, COLOR, WEIGHT, P.CITY*, QTY)

In (1), every IA is uniquely named as some non-key attribute of S or of P. There is also one IA per every such attribute and SP has no other IAs. Furthermore, suppose that for every SP.S# value from S, for every IA sourced in S, the value is as for S.S# = SP.S#. Likewise, consider that for each SP.P# value, every IA value from P, is as for the same P.P# there. If there is no referenced S.S#, then suppose we set all IAs from S to nulls. We do the same for every missing P.P# value. Finally, observe that SIR SP (1) has every IA placed with respect to its FK as its source is with respect to the referenced one. All these properties make (1) a natural SIR SP. It is thus also the natural SIR SP for its base SP_ and for the original SR SP. By the same token, every attribute in *Italics* in (1) is a natural IA. Fig. 3 shows the content of SP (1), given that of S-P at Fig. 1. IAs are also in *Italics* there.

We now operationally define the explicit SQL scheme of SIR SP (1), i.e., Create Table SP, valid for SIR-enabled

DBSs only, of course. We follow the general rules we proposed earlier for every Create Table for a SIR, recalled in the Introduction. As said, one may start with Create View SP for C-view SP. Recall that in SQL, no Create View SP can refer to a relation named SP. Hence, suppose that we rename the original SR SP to SP_. Then suppose the following statement to define C-view SP:

(2) Create View SP AS (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#));

It is easy to see, from values within Fig. 1 in particular, that (2) effectively creates logically the same relation as intended for SIR SP (1). Next, suppose the following Create Table SP for the original SP:

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

The following Create Table SP is then the explicit scheme for SIR SP (1), in SQL extended for SIRs as proposed:

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

Scheme (4) indeed merges (2) and (3) to get every SA and IA in (1) in order. IE consists of every IA in (1) and of From clause in (2). In what follows, we designate DB with SIR SP (4) as S-P1. Fig. 2 shows S-P1 schemes, with the implicit one, i.e., (3), highlighted in **bold** within the explicit SP. Fig. 3 shows S-P1.SP content for S-P.SP at Fig. 1.

S-P1 Scheme

Create Table S (	**Create Table P (**	**Create Table SP (**
S# Char 5,	P# Char 5,	**S# Char 5**, SNAME, STATUS, S.CITY,
SNAME Char 30,	PNAME Char 30,	**P# Char 5**, PNAME, COLOR, WEIGHT, P.CITY
STATUS Int,	COLOR Char 30,	**QTY Int**
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#));**

Fig. 2. S-P1 scheme with explicit SP scheme. The implicit one would be that of S-P.SP, bold for SP here.

Notice that as said generally before, (3) became also in (4) SQL scheme of the base SP_, except for the relation name. We suppose From clause in (4) to refer to that one. The goal of (4) is to enlarge SP_, after all. In (2), SP_ designates instead "classically" a separate relation, i.e., S-P.SP renamed to SP_. Not any part of the relation being created, unlike in (4). We allow for such referencing in SIRs in general. It avoids sometimes, like in (4) in fact, a (prohibited) circular referencing among SIRs, [1]. Since R_ is an SR, it inherits indeed from nothing, so, unlike perhaps R, it cannot participate in any recursive inheritance. Except for this formal difference, SP.SP_ scheme referred to in (2) and SP_ referred to in (4), define the same relation, including the physical implementation that consists of data types only here. For every tuple of SIR SP (4), From clause there calculates then the same value or null for every IA enlarging every stored tuple of base SP_ of SIR SP, as the same clause calculates for the same IA in the (only) tuple containing the same sub-tuple (S#, P#, QTY) in view SP (2). Notice also that IE effectively precedes the table options in (4). Those define the primary key only, since both SP.S# and SP.P# are natural FKs. As an exercise, calculate finally every value defined by IE (4) to check that all those are effectively the intended one for SIR SP through view SP (2). Compare finally to Fig. 3.

Observe that IE in (4) is less procedural than Create View (2), as claimed for every SIR R with respect to C-view R, [1]. The easy to grab rationale is that (2) had to redefine every IA that remains an SA in S-P1.SP_. More formally, suppose the *procedurality p* of (2) measured as the minimal number of characters to type-in (keystrokes). We have $p = 152$ for view SP (2) and $p = 112$ for IE in (4). So C-view SP scheme is 30% more procedural than IE. On a SIR-enabled DBS, SP (4) would provide for LNF queries as we show soon. C-view SP (2) would do as well. At any present DBS, not supporting SIR SP thus, (2) would be the only choice. Time to type is in general at least linear with p. As a view saver, our SIR SP (4) would thus save 30% of the time necessary for keystroking Create View SP. In popular vocabulary, to create view SP (2) when SIR SP (4) could exist would be just a waste of time. As we signalled, the property of an IE with lesser p extends to every SIR R with respect to C-view R and, even more generally, with respect to every eventual view R, mathematically or in practice only, equivalent to C-view R, [1].

Finally, observe that SIR SP (4) is effectively natural one for SP_, whether the base of S-P1.SP or renamed S-

P.SP. First, SP_.S# and SP_.P# remain FKs in (4), naturally referencing S and P. SP_.S# brings therefore to the explicit SP scheme (4) *SNAME…S.CITY* at the positions of all these IAs in (4). Also From clause with the Left Join S On… in (4), provides for every SP_.S# value and every IA in (4) either the expected value or makes the IA null. Likewise P brings *PNAME…P.CITY* at their places in (4) and Left Join P On….Finally, (4) does not define any other IA.

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

Fig. 3. S-P1.SP content. IA names and values are *Italics*.

3.2. Inferring Explicit Natural Schemes

As usual, we suppose every referenced relation already declared when the referencing one is being created. Suppose then that a SIR-enabled DBS gets Create Table R that defines stored attributes only, hence may be an implicit scheme of a natural SIR. DBS processes it towards the explicit natural scheme as follows.

1. *Add the IAs for the declared FKs*. For every declared RK, perhaps composite, retrieve from meta-tables, e.g., SYSTABLES etc. for DB2, the name of every attribute other than the name of any attribute within the RK. Place each of these names, qualified if needed, in Create Table R, following the order on IAs defined for a natural SIR.

2. *Add the IAs for the natural keys*. For every (atomic) attribute C of R, other than any attribute of a declared FK, retrieve from the meta-tables the name of every R' with C as primary key. Iff there is exactly one such R', complete Create Table R with the unique in R names of every IA sourced in R' and required for the natural SIR R. Follow the ordering of IAs of a natural SIR.

3. *Create From clause.* To start, after last SA and before the table options, append: From R_. Then, for every R'1…R'n above processed; $n \geq 1$; let K1…Kn be the keys, referenced accordingly by F1..Fn, the numbering following the order of FKs within R. For every atomic Fi ; i=1...n ; append to the current form of From clause: Left Join R'i On (Fi = Ki). For a composite Fi, say (Fi_1,..,Fi_k) append Left Join On (Fi_1=Ki_1 And…And Fi_j=Ki_j).

Ex. 3. We skip the easy but tedious proof of the rules. We only show that they make Create Table SP (3) the implicit one for Create Table SP (4). Suppose indeed S-P1.S and S-P1.P already created. There is no declared FK in (3), hence rule (1) does not apply. Rule 2 finds S.S#, and P.P#. For the former, Rule 2 inserts to (3) *SNAME…S.CITY*, right after S#. Likewise, it inserts *PNAME...P.CITY* right after SP.P#. Since SP.S# precedes SP.P# in SP, Rule 3 builds From clause as in (4) and terminates the build-up. Reverse order of FKs in SP would reverse Left Joins. Result would be mathematically the same, as these joins are commutative.@

For DBA, the outcome is simply $p(IE) = 0$. In other words, e.g., to create the natural SIR SP instead of S-P.SP costs DBA *zero* additional work and time. As free bonus, DBA saves also 100% of procedural cost of view SP. Simply put, for a DBA of SIR-enabled DBS, to create view SP instead of the natural SIR SP to provide usually substantially less procedural LNF queries for clients, would be a total waste of time. We now detail these bonuses for both clients and DBA.

3.3. Natural SIRs as Perfect LNF-view Savers

We detail the bonus for SP only, but the outcome applies to any natural SIRs. The IAs in (4), are the attributes that S-P.SP conceptually has as well. But, as said, operationally it does not. As an SR, S-P.SP, it is indeed constrained to its normal form by the model. The well-known goal is to avoid the (normalization) anomalies, [9].

Anomalies in contrast do not occur for an IA, [2]. E.g., recall that if SNAME was within stored SP, then, first, it would denormalize it. Indeed, since we have FD S# -> SNAME, SP would no more be in 2NF even. Then, the denormalized SP would require, e.g., six times more storage for SNAME = 'Smith' than the normalized S for its single 'Smith', Fig. 1. Next, to update Smith to, say, Smit, would require 1 update only in S, but 6 in the denormalized SP. If even one of these failed, FD S# -> SNAME would be violated with obvious drawbacks for queries. If SP.SNAME is in contrast a natural IA, none of these anomalies occurs. Similarly, for every IA sourced in any other non-key attribute of S and P. SP can have thus all the attributes it should, but could not have with the current model.

A query to S-P.SP, i.e., searching for attributes of every supply so and so, usually formulates as a select-project-join query. It searches indeed for attributes of SP, but also for some conceptual attributes of a supply that ended up elsewhere. E.g., in real life, a query about a supply usually requires not only some SP data, but also the name of supplier(s) and those of part(s) supplied, at least. The join clause nests joins over each FK and RK, e.g., as in (2) or (4). As said, the term LN usually refers to such joins since the eighties, coined by the universal relation idea fans. Same query formulated towards S-P1.SP becomes a select-project one only. Every attribute value in S or P possibly needed is now indeed also within SP. LN not only always increases *p* of the query, but often is even more procedural than the rest. That is why, since decades, clients usually dislike LN, at best.

E.g., consider a usual query: for every supply by 'Smith', retrieve the quantity and the name of the part supplied, if known. The formulation for S-P would be query Q as follows or an equivalent one:

(Q) Select QTY, PNAME From SP Left Join S On (SP.S# = S.S#) LEFT JOIN P On (SP.P# = P.P#) Where SNAME = "Smith";

(Q) includes the LN through Left Join of SP with S, then the Left Join with P. On SIR-enabled DBSs with S-P1, hence with SIR SP instead of SR SP, the same query would formulate as the LNF part of (Q) only:

(Q') Select PNAME, QTY From SP Where SNAME = "Smith";

Q' is visibly at least twice less procedural. Besides, most clients would have no trouble with it. No way to say it for Q, [8], [15].

As already mentioned, C-view SP (2) conceptually serving (4), is the LNF-view for the same (LNF) queries. As also shown however, (2) is already more procedural than the explicit IE in (4). It becomes a total waste of time when one infers (4) from (3) as the implicit S-P1.SP scheme and the LNF-view saver for the discussed queries to S-P1.

4. Implementing Natural SIRs

In [1] we proposed the implementation we qualified of *canonical*, for SIR-enabled DBS. In sum, the latter consists of a (canonical) layer above an existing DBS, called *SIR-layer*, interfacing every SQL statement. Internally, it calls services of the DBS, referred to as *kernel* (DBS). Above SIR-layer, the relational constructs are thus: SRs, SIRs and views. Underneath, i.e., for the kernel, there are necessarily SRs, perhaps with computed attributes and views only. For every SR or a SIR with IAs declared as computed attributes or a view, SIR-layer simply forwards Create statement to the kernel. Then, it also forwards every query to these constructs only. In contrast, for every SIR R other than above, SIR-layer atomically creates within the kernel the couple: SR R_ and C-view R. Create Table R_ and Create View R are easily extracted by SIR-layer from the explicit Create Table R. SIR-layer transparently forwards then every Select query addressing SIR R to view R. Kernel DBS may internally optimize somehow such queries, e.g., by materializing some IAs within view R for faster joins, [13], [14] or [V87], SIR-layer passes gains to SIR R in this way. An update query to SIR R goes to view R, but the client may need to issue it rather to base relation R_, depending on *view updating* capabilities of kernels, [10]. E.g., MySQL and MS Access provide for updates of outer join views hence would allow for some updates of view SP. SQL Server does not allow for, hence the S-P1 client should issue updates to base tables only. Same for SQLite that flatly refuses every view update, etc.

With respect to the above sketched capabilities, the only new one the natural SIRs require for SIR-layer is the processing of the proposed rules. This appears simple to add. The canonical implementation itself appears also simple to conceive, [1]. See [3] as well, for a simulated implementation of the latter. Altogether, to reuse the present schemes of SRs conform to the requirement for natural SIRs appears simple, almost trivial. The old dream of the LNF queries comes to life as welcome bonus. One should also easily see that every SIR-enabled DBS is backward

compatible with every pre-existing DB on the kernel used. Likewise, it appears simple to convert on demand a usual pre-existing SR R with FKs to a natural one through an Alter Table R extended to hold such a need, [4].

Ex. 4. Suppose that SIR-layer gets Create Table SP (3). As in Ex. 3, it transforms it into Create Table SP (4). Then, supposing the canonical implementation, it issues to the kernel from (4) the statements: (i) Create Table SP_ that is (3) renamed to SP_ and (ii) Create View SP (2). From now on, SIR-layer passes to the kernel every incoming LNF query to SP. The kernel transparently processes it using view SP and returns the outcome. Likewise, SIR-layer passes to the kernel any other select query. For update queries that ideally could or should address SIR SP, the client actually needs to decide where to address those as just discussed.

Altogether, one can realize from the above that the processing time overhead due to SIR-layer, with respect to the one within the kernel, supposed a popular DBS, should be about zero for any query. For the SQL DDL statements one can expect also about zero for Drop Table and at most about a millisecond for Create Table or Alter Table, [1], hence negligible. The SIR-layer processing consists indeed only of the statement's parsing.

5. Conclusion

A scheme of an SR with FKs at any present DBS, usually implicitly defines the natural SIR in fact. No need for additional data definition work for DBA to create the latter. LNF queries to base tables for usual needs are the bonus for clients. By the same token, DBA saves the hassle of creating and maintaining an LNF-view, necessary for the same LNF queries at present and even more procedural than the explicit natural SIR. Decades of burden for SQL clients or DBAs are over.

To add the above processing of FKs to the canonical implementation appears simple. The canonical implementation appeared simple to develop and efficient already. Altogether, the processing overhead due to SIR-layer should be negligible. One may conclude that since inception of the model, usual schemes with FKs have not been read as they should be, i.e., as implicitly defining natural SIRs. LN within otherwise simple SQL queries bothered generations uselessly. Major DBs should become SIR-enabled "better sooner than later". It would be a long overdue service to, likely, millions of clients of SQL DBs of all kinds and flavours at present.

Future research should provide for the proof-of-concept implementation of SIR-layer supporting natural SIRs. SQLite seems the best kernel for at present. Besides, one should generalize our rules for natural SIRs to other SIRs with FKs. This seems particularly easy for SIRs only with IAs that could be computed attributes at present. Early results are in [4], as are details on natural SIRs that the size limit on this paper prohibited to include.

Acknowledgments

Our thanks go to Ron Fagin for invitation to present this material at IBM Almaden Research Cntr., March 2020. We also thank Darrell Long for March 2020 invitation to talk about at UCSC Eng. as well. We thank finally Peter Scheuermann for helpful comments.

References

[1] Litwin, W. "SQL for Stored and Inherited Relations." *21st Conf. on Enterprise Inf. Syst., (ICEIS 2019)*, http://www.iceis.org/?y=2019 , 12.
[2] Litwin, W. "Manifesto for Improved Foundations of Relational Model." *EICN-2019. Procedia Comp. Sc., 160, (2019), 624-628, Elsevier,* 6.
[3] Litwin, W. Supplier-Part Databases with Stored and Inherited Relations Simulated on MS Access. *Lamsade Tech. E-Note*, 2016. pdf
[4] Litwin, W. "Stored and Inherited Relations with Foreign Keys." *Lamsade e-report. June, 2020, 14,* pdf.
[5] Litwin, W. Ketabchi, M., Risch, T., 1992. "Relations with Inherited Attributes." *HPL. Palo Alto. Tech. Rep. HPL-DTD-92-45,* 30.
[6] Codd, E., F., 1970. "A Relational Model of Data for Large Shared Data Banks." *CACM*, 13,6.
[7] Date, C., J. 2004. "An Introduction to Database Systems." *Pearson Education Inc. ISBN 0-321-18956-6.*
[8] Date, C., J., & Darwen, H. "Watch out for outer join." *Date and Darwen Relational Database Writings*. 1991.
[9] Date, C., J. "Database Design and Relational Theory, Normal Forms and All That Jazz." *O'Reilly, 2012.*
[10] Date, C., J. "View Updating and Relational Theory." *O'Reilly, 2012.*
[11] Date, C., J. "Type Inheritance & Relational Theory." *O'Reilly, 2016.*
[12] Date, C., J. "E.F. Codd and Relational Theory." *Lulu. 2019.*
[13] Goldstein, J. Larson, P. "Optimizing Queries Using Materialized Views: A Practical, Scalable Solution." *ACM SIGMOD -2001.*
[14] Halevy, A.,Y. "Answering queries using views: A survey." *VLDB Journal* 10, 2001, 270–294.
[15] Jajodia, S., Springsteel, F., N., 1990. "Lossless outer joins with incomplete information." *BIT*, 30, 1, 34-41.
[16] Larson, P., Zhou J. "Efficient Maintenance of Materialized Outer-Join Views." *IEEE ICDE-2007.*
[17] Vigier, Ph., Litwin, W. "Dynamic attributes in the multidatabase system MRDSM". *IEEE ICDE-1986.*