SIR SQL for Logical Navigation and Calculated Attribute Free Queries to Base Tables.

WITOLD LITWIN

University Paris Dauphine, PSL

SIR SQL stands for SQL with Stored and Inherited Relations (SIRs). Every SIR has attributes defined as in an SQL Create Table. In addition, it has some *Inherited Attributes* (IAs), definable in SQL Create View or queries only at present. SIRs may also have foreign keys (FKs), generalizing the SQL FKs. For SIRs with FKs, IAs provide for logical navigation free (LNF) queries to base tables, i.e., free of equijoins on SIR ,SQL foreign and referenced keys. The same outcome SQL queries to the same base tables, but without IAs, must involve the LN avoided.

IAs may also be the *calculated attributes* (CAs). One defines these through value expressions impossible for any SQL Create Table. A CA may accordingly involve, e.g., attributes from different tables or aggregate functions, or sub-queries. SIRs with CAs provide for CAF queries, involving these CAs by names only. In contrast, SQL queries to base tables needing any of these CAs have to define them. The end result is that typical SQL queries to base tables requiring LN or CAs at present become LNF or CAF in SIR SQL. This makes them usually substantially less procedural and quasi natural select-project only queries. SIR SQL should accordingly significantly boost SQL client productivity. Although the problem of LNF and CAF queries is anything but new, our solution is the only of the kind, to our best knowledge.

Below, we first illustrate the problem of LN and of CAs in queries to SQL base tables using Codd's Supplier-Part original example. We then present SIR SQL in depth. We show how LNF and CAF queries to base tables become possible. Then, we discuss the front-end intended to provide SIR SQL on any SQL DBS through a few months of developer's work. We point to the proof-of-concept prototype for SQLite3 in Python. We finally postulate that courses and textbooks on relational DBs from now on take notice of SIR SQL. Also, - that every popular SQL DBS upgrades to SIR SQL "better sooner than later". 7+ million SQL clients worldwide, of the most used DB language, providing for 31B+ US$ market size of SQL DBSs, will benefit from.

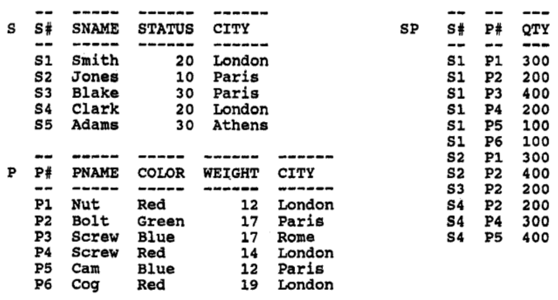
CCS Concepts •Information systems~Data management systems~Database design and models~Relational database model

Additional Keywords and Phrases: SQL, Stored and Inherited Relations

1 THE PROBLEM

Since their inception, five decades ago for the pioneers, all present SQL DBSs bother the clients, users and developers, with parts of typical queries to the base tables, necessary beyond the otherwise natural or almost, formulation of such queries. The 1st culprit is the logical navigation (LN) in queries to base tables with foreign keys and to the referenced tables. Recall that LN means equijoins on foreign and referenced keys, as Codd originally defined these terms in [1] and results from his sheer idea of a foreign key (FK). Actually, the latter seems implying that for every FK, the LN involving (or *from*) the FK, in particular preserves every value of the FK, i.e., always expresses as a semi-outer join, reducing to the inner one, if one wishes so, iff FK respects the referential integrity. The procedurality that the LN implies, i.e., the necessary length (number of characters) of the SQL join clauses defining it, may often double the one of the query without. Not surprisingly, clients usually at least dislike the LN. Especially, - when it necessarily expresses as the outer joins, [4]. In short, queries to base tables requiring LN at present should possibly be LNF instead.

The 2nd culprit is the impossibility for any SQL dialect at present, to declare base tables with the calculated attributes (CAs), defined by value expressions with, e.g., aggregate functions or sub-queries or sourced in other tables. If a CA a query needs could be in the base table, the query could address it by name only, i.e., the query could be CAF. Since it cannot be so for any CAs at present, SQL clients must define the specs of any of those in the queries. The increase to the query procedurality may be substantial, e.g., again may often double the query. The sheer complexity of some CAs, of those defined by sub-queries especially, also bothers many.



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

E.g., consider Supplier-Part DB of Codd, Fig. 1, the “mother of all the relational DBs”, [1], [2]. In other words, Supplier-Part design principles are the typical ones of any DBs at present and property shown by our examples below generalize accordingly. We refer to Supplier-Part as to S-P DB in short. S, P, SP are 1NF *stored* relations, (SRs), also called *base* tables. For Codd, SR means that none of its stored attributes, (SAs), can be calculated through the DB schema and content. Next, S.S#, P.P# and SP(S#,P#) are the *primary keys* (PKs). Finally SP.S# and SP.P# are *foreign keys* (FKs) for Codd, originally, [1]. I.e., each is the “logical pointer” to the (unique in S-P) PK with the same name and, for every FK value, to the, unique in the referenced table and thus in S-P, tuple with the same PK value, whenever such a tuple exists.

A typical query to SP, i.e., searching for every supply so and so…, would address some of SP attributes together with some attributes of S or P. The rationale is that all the non-key SAs of S and P are obviously conceptual attributes of SP as well. They should thus be also SAs of SP. They are actually not. The normalization anomalies for SP that would follow and that we discuss more below, are unacceptable for the relational model.

E.g. consider a query searching for the basic data of smaller supplies, say Q1: “For every supply in QTY <= 200”, select S#, with SNAME whenever known, then P# with, also whenever known, PNAME, and QTY. Q1 could simply formulate in SQL as:

Select S#, SNAME, P#, PNAME, QTY From SP Where QTY < 200;

Q1 expresses only the necessary projection and restriction and is, for many, a telegraphic style, but natural language query. It would suffice if SNAME and PNAME were attributes of SP. However, they are not. Hence, Q1 formulates at present as Q2 below or with an equivalent From clause, regardless of SQL dialect used:

Select S#, SNAME, P#, PNAME, QTY From SP Left Join S On SP.S#=S.S# Left Join P On SP.P#=P.P# Where QTY < 200;

The reason is that whatever SP tuple Q1 selects, nothing in S-P scheme indicates SNAME & PNAME values Q1 should reference through the foreign keys, when these values exist. The LN in Q2 does it therefore instead. The “price” is that Q2 becomes twice as procedural and anything, but a natural language query.

Next, every supply has obviously some weight, say T-WEIGHT, defined as QTY * WEIGHT, where WEIGHT value is the one referenced through SP.P# value of the supply, if it is in P. If T-WEIGHT was of interest to clients and obviously it would often be in practice, it should be a CA of SP. Then, e.g., query Q3 providing the ID and T-WEIGHT of every supply could simply be:

Select S#,P#,T-WEIGHT From SP;

Q3 would be a CAF query, with respect to T-WEIGHT and LNF query with respect to P. However, as even SQL beginners know, T-WEIGHT cannot be a CA of SP for any popular SQL dialect. Hence one has to express Q3 as Q4 with the T-WEIGHT scheme in it, e.g.:

Select S#,P#, QTY * WEIGHT As T-WEIGHT From SP Left Join P On SP.P# = P.P#;

As one can see, Q4 is twice+ more procedural than Q3.

Recall finally that the problematic of LNF and of CAF queries to base tables is anything but new. Already in early 80ties, Meyer & Ullman proposed the, so-called, universal relation as a solution for the LNF queries. However, despite its initial popularity, the concept did not prove practical as yet. For the CAF queries, Sybase SQL dialect introduced, also in early 80ties, the *virtual* (dynamic, computed, generated….) *attributes* (VAs). Several other SQL dialect adopted VAs since. Nevertheless, the result was and remains only a partial solution, E.g., T-WEIGHT cannot be a VA in any SQL dialect we are aware of. We discuss VAs more later on.

2 OUR SOLUTION

The idea is that since the queries to base tables should be LNF and CAF, for every base table R with (Codd’s) FKs, for which queries could address some attributes of R or some referenced through an LN, or some CAs, Create Table R should predefine the name of every such attribute and every LN, as well as every CA. Likewise, for every table without any FKs, Create Table should predefine every CA. This said, everything that follows is mere technical details to make the solution the most practical.

Notice that trivial SAs cannot be the solution, as pointed out earlier for S-P. Observe also that all the names of the predefined attributes, as well as all the LN clauses should possibly be implicit in Create Table R as issued by the database administrator (DBA). For every FK, all these names should indeed be already in the meta-tables and one can easily infer the LN clause. Every statement should then be reasonably, i.e., within the general SQL framework, the least procedural for DBA. In particular, - avoiding in this way errors in otherwise copied names or in LN clauses. Dedicated pre-processing may then add to the Create Table every missing attribute name and the LN, for any further processing. Notice finally that if all the attributes to be predefined and all such LN clauses are implicit in a Create Table issued by DBA without any CAs, then any such Create Table formulates simply as some present one. In other words, DBA creates then base tables supporting LNF queries without any additional work with respect to the one required from DBA at present to create "only" base tables without that capability.

Several details that follow were presented separately in depth impossible here within the space limits. The related papers are indexed at our home page, [9], by title and at least by abstract, whenever the pdf is copyrighted. Also, [5] refers to some. Below, we systemize the details into the following steps, introduced in "… for Dummies" way.

1. Extend SQL to *Stored and Inherited Relations* (SIRs). Call *SIR SQL* SQL extended accordingly. SIR construct in general was abundantly presented in our previous papers. For SIR SQL, any SIR R is simply a 1NF base table R consisting of some SQL base table enlarged with IAs definable as in an SQL view or query. The SQL table within R bears its own implicit (default) name R_ and constitutes the *base* of R. The name R_ is available to any SIR SQL statement, as if R_ was stand-alone.

We refer to the attributes of R_ as to *base attributes* (BAs) of R. We also refer to the definition of the IAs within any SIR R as to *Inheritance Expression* (IE). With respect to SQL Create Table, SIR SQL Create Table provides accordingly the usual SQL Create Table capabilities for R_ and additional ones for the IE. The latter are basically as in SQL Create View or queries only at present. Likewise, SIR SQL provides for a more general Alter Table statement. All the other SIR SQL statements are simply the SQL statement. For most of the latter, the processing differs however from their SQL counterparts, as it will appear. The From clause in any SIR SQL Create Table R should be further as follows, qualified then of *valid*. Let *t* denote an R_ tuple. Let also R* be the table defined by From clause and *t'* an R* tuple. Then, for every *t* that one could insert to R_ given the table constraints in Create Table R, R* should have exactly one tuple with *t* as sub-tuple and should have only such tuples.

For any further processing, SIR SQL Create Table R contains every IA scheme intended for R and the valid From clause, i.e., the entire IE, specified as these IAs and From clause could be for an SQL view. We qualify of *explicit*, every such IA, the From clause, the IE, as well as the Create Table. Besides, as it will appear, SIR SQL Create Table R may miss some IE parts or even the entire IE. We speak accordingly about *implicit* IAs, From clause, IE and Create Table. We qualify of *empty* an entirely omitted IE.

Rules we discuss later provide for the explicit Create Table given any implicit Create Table. Actually, we presume every SIR SQL Create Table submitted to be implicit. It is not iff it turns out to contain only BAs and to be (i) without any SIR SQL FK qualified below of *primary key named* (*PKN*) FK and (ii) without any IE or if there is any, then it is an explicit one. The submitted explicit SIR SQL Create Table R without any IAs reduces to SQL Create Table R. R is then (one could say: only) an SQL base table, not a SIR.

If the submitted Create Table R turns out to be implicit, the pre-processing we detail later completes the IE, even empty, to the explicit one. In particular thus, for SIR SQL, any submitted Create Table R with PKN FKs creates SIR R. R has then all the BAs of the implicit scheme, of R_ thus, and the explicit IE. Every tuple of R is a tuple of R_, enlarged with the IA values or nulls in the same name IAs of R*-tuple with the same PK. Create Table R attribute list may however place IAs in R differently of their R* (same name) sources.

Finally, whenever R is a SIR, BAs together with the table constraints and options form the R_ scheme. Then, whenever Create Table R defines also IE elements, i.e., some IA schemes and, perhaps, partial or entire From clause, all these elements should be within minimal necessary number of { } brackets separating the IE from R_ scheme. Each bracket replaces then a usual SQL separator, i.e., ‘,’ or space. As IAs may be spread among BAs or separated by BAs from From clause, more than one pair of { } may be necessary. The convention facilitates the parsing of the SIR SQL Create Table statement, as it appeared.

Ex. Suppose S-P.SP declared through the SQL Create Table of some SQL dialect, e.g., SQLite SQl:

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

Consider then the explicit SIR SQL Create Table SP:

(2) Create Table SP (S# TEXT, P# TEXT, QTY INT {WEIGHT*QTY As T_WEIGHT, 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#));

(2) defines SIR SP enlarging SP (1) with the IAs defined within. The attributes and the (only) table constraint of (1) within (2), define the base SP_. IE is entirely within single pair of { }. Let us call S-P1 the DB with S, P and SP (2). Fig. 2 shows S-P1.SP content for SIR SQL clients, given Fig. 1.From clause in (2) is valid. Indeed it is first clearly so for any SP tuple at Fig. 2. SP tuples in Fig. 1 however implicitly respect the referential integrity (RI) between SP and S, and P. So does every tuple at Fig. 2. However, neither in (1) nor in (2) there are no FK table constraints, as for SP in [1] besides. Hence, RI is not enforced. One may thus insert to SP_ (S6, P1, 200). Since the LN in From clause of (2) consists of left outer joins, R* would contain one and only one tuple (S6, P1, 200, null, null, null, null, P1, Nut…). One may easily see also that this property of R* generalizes to any tuple breaking the IR with respect to S or P. The From clause in (2) is thus a valid one. In

contrast, any From clause with any inner join instead of the outer one, would not fit. The latter would make From clause valid iff the RI was enforced.

Next, observe that the LNF Q1 applies to SP (2). It is also so for the CAF Q3. The rationale is of course the presence of the IAs in SP. As calculated only, the latter never create any normalization anomaly, i.e., insert, update, delete or storage anomalies. These anomalies would in contrast necessarily occur, if every IA of S-P1.SP was trivially, an SA, as Codd's model requires for any BA. Recall that the relational model, prohibites those because of the annoying side-effects. E.g., in SIR SP, the redundant with respect to S and P IA values in SP, e.g., in 6 tuples for S1 there, Fig. 1, do not cost any additional storage, while they would obviously do, if they were SAs. Likewise, SP does not need any updates if a source value varies, e.g., S1 name changes to 'John', again unlike for the "trivial" choice. Finally, the latter could in particular lead to hidden inconsistencies, if a redundant data manipulation goes awry. E.g., if WEIGHT changes, but (SA) T-WEIGHT does not for any reasons. Or, if one inserts tuple (S2, P3, Bolt, Green…), (guess why?). All these properties of IAs generalize to any DBs with SIRs. "Better late than never", through IAs in base tables, the SIR construct lifts an intriguing limitation in Codd's model, [1].

Table S

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

Table P

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

Table SP

S#	P#	QTY	*T-WEIGHT*	*SNAME*	*STATUS*	*S.CITY*	*PNAME*	*COLOR*	*WEIGHT*	*P.CITY*
S1	P1	300	*3600*	*Smith*	*20*	*London*	*Nut*	*Red*	*12*	*London*
S1	P2	200	*3400*	*Smith*	*20*	*London*	*Bolt*	*Green*	*17*	*Paris*
S1	P3	400	*6800*	*Smith*	*20*	*London*	*Screw*	*Blue*	*17*	*Oslo*
S1	P4	200	2800	*Smith*	*20*	*London*	*Screw*	*Red*	*14*	*London*
S1	P5	100	1200	*Smith*	*20*	*London*	*Cam*	*Blue*	*12*	*Paris*
S1	P6	100	1900	*Smith*	*20*	*London*	*Cog*	*Red*	*19*	*London*
S2	P1	300	3600	*Jones*	*10*	*Paris*	*Nut*	*Red*	*12*	*London*
S2	P2	400	6800	*Jones*	*10*	*Paris*	*Bolt*	*Green*	*17*	*Paris*
S3	P2	200	3400	*Blake*	*30*	*Paris*	*Bolt*	*Green*	*17*	*Paris*
S4	P2	200	3400	*Clark*	*20*	*London*	*Bolt*	*Green*	*17*	*Paris*
S4	P4	300	4200	*Clark*	*20*	*London*	*Screw*	*Red*	*14*	*London*
S4	P5	400	4800	*Clark*	*20*	*London*	*Cam*	*Blue*	*12*	*Paris*

Fig. 2. S-P1 base table schemes and content. IAs in SP are *Italic*. S and P are the same as in SP at Fig. 1.

Observe next that S-P1.SP defined by (2), contains by name and value with respect to S.S# or P.P#, i.e., the source PKs, every attribute of S-P. Easy to see thus that not only Q1 formulates as the substantially less procedural LNF Q2, but that, more generally, any query Q addressing any attributes of S or of P through some LN with, perhaps, any attributes of S-P.SP, formulates as a substantially less procedural LNF Q' to S-P1.SP. As for Q1 and Q2, Q' consists simply of Q without LN, with, perhaps, CITY prefixed with S or P, instead of the non-prefixed one in Q.

We show soon furthermore that one may create S-P1.SP through the implicit Create Table containing only SP_ scheme and the value expression defining T-WEIGHT. Namely, instead of (2), DBA could issue:

(3) Create Table SP (S# TEXT, P# TEXT, QTY INT {WEIGHT*QTY As T_WEIGHT} Primary Key (S#, P#));

The obvious advantage of (3) is to be almost 2.5 times less procedural. If SP did not have T-WEIGHT or any CA more generally, it will appear that (3) would reduce simply to (1), i.e., to the SQL Create Table SP for S-P. For SIR SQL however, the latter would be the implicit Create Table for S-P1.SP, with empty IE. In other terms, for SIR SQL, present S-P scheme defines in fact S-P1. Yet in other words, S-P scheme suffices in fact for the LNF queries, unlike presently.@

Next, for every SIR R, there is an SQL view R, hence with IAs only, defining logically the same relation as SIR R. We qualify the latter of *canonical* view of SIR R and of *C-view* R, in short. C-view R results from Create View R with the same attribute list as in SIR R, except that every R_ attribute is stripped to its name only, followed by the same From clause. The difference between SIR R and C-view R is thus only physical: every SA in SIR R becomes the IA with the same name and value in every tuple within C-view R and vice versa. Adding a C-view R to an SQL DB with R_ as stand-alone base table, provides then for the same LNF or CAF queries as SIR R. Provided however that these queries address the C-view R, instead of the base table R, necessarily renamed somehow, i.e., to R_. The rather easy to see drawback of any C-view R with respect to SIR R is that the former must be more procedural to specify and to maintain than even the explicit IE in SIR R. C-view R has to indeed redefine every SA of R_ as an IA and it constitutes a separate table to maintain in sync with R_. The implicit SIR schemes whenever possible, with possibly an even empty IE, are obviously even more advantageous. As we just stated, it would be so, e.g., for S-P1.SP (3) and of course, even more for (1). In present terms, every SIR R is thus a *view saver* for C-view R. Recall also that quest for less procedural and more natural data definition and manipulation always was and still is the driving force for the DB research, as well as for the CS generally. Remember that it is why in particular the relational DBs succeeded to Codasyl and IMS ones.

Finally, as hinted to in the example above, in practice, every SIR SQL Create Table will extend to SIRs Create Table of some existing SQL dialect. Likewise, SIR SQL Alter Table will extend Alter Table of the dialect. Call *kernel* (SQL) the dialect chosen. Some kernels, provide for base tables with SAs only, as Codd proposed. Any SIR R defined in SIR SQL extending the dialect will then have the base R_ with SAs only. Other kernels provide for the already mentioned VAs as BAs as well. Recall then, e.g. from our papers on SIRs, that every base table R with VAs is in fact a limited SIR R. There, every VA is an IA inherited only from R_ and only through arithmetic value expression with, perhaps, scalar functions over SAs or other VAs of R_ and with implicit 'From R_' clause. Accordingly, e.g., as inheriting also from P, T-WEIGHT could not be a VA for any present kernels. Nevertheless, despite their limitations, VAs became popular view-savers, as we already hinted to.

For SIR SQL consequently, there is no need for a kernel providing for VAs, although one can still define any VAs the kernel provides for. Indeed, regardless of any such kernel and any VAs it could provide for, SIR SQL dialect for the kernel would always provide for an equivalent IA with the same value expression and the implicit 'From R_'. In practice, the only syntactical difference would be that while any VAs define the attribute name first and the value expression after, the IA scheme would be the other way around and somewhere within { } brackets, instead of usual ',' or ' '. Somehow consequently, as one could already observe, for SIR SQL, if VAs are present in a Create Table of SIR R, they are not considered IAs, since they are not added to R_ scheme, but are within. In other words, for SIR SQL, for any SIR R, IAs are only the attributes defined in explicit Create Table R as if they were in C-View R. Unlike we assumed for the general definition of the SIR construct in our previous papers. That one was designed for Codd's relational model, proposing SAs only in base tables, [1], to recall. Consequently, VAs were there specifically defined IAs as well.

2. Generalize SQL FK concept to SIRs. That one, introduced by the SQL designers, in our opinion, differs from Codd's original one, [1], [3]. For SIR SQL, it appeared useful to merge both. Fig. 3 sums up the result. First, one may specify any *declared* FK using the familiar SQL Foreign Key clause or 'References' keyword etc. in an SA scheme, provided by some SQL kernels, e.g., SQLite SQL. Any declared FK implies accordingly the referential integrity (RI). The only semantic difference to any SQL FK is that a declared FK may reference a SIR. Next, for SIR SQL, any declared FK referencing PK with the same proper name is a (declared) *primary key named* FK, (PKN FK). Besides, in SIR SQL, for any SIR R, a *natural* (PKN) FK is any non-PK atomic SA A such that (a) there is only one PK with proper name A and (b) A is not within a declared PKN FK in R.

Next, in SIR SQL, the characteristic property of every PKN FK, say F, in submitted Create Table R is the *natural inheritance* (NI) *through* F. The term designates mandatory IAs in the explicit Create Table R, given F. We qualified of *natural* or of *NIA* every IA in an NI. NIAs forming any NI through F are named upon every non-PK attribute in the referenced table. For every F, they are placed in R in the source order. Finally, we talk simply about NI, to designate all the NIAs of R. It appeared useful for the proof-of-concept prototype we address later, to place NI at the end of R attribute list. The NIs through PKN FKs are placed in NI in the (left-to-right) order of the FKs within Create Table R.

With respect to NI values for every R-tuple, we consider that these result from Codd's "logical pointer" calculus, supposed for any FKs in [1]. Namely, for every PKN FK, one calculate these values through the equijoin, FK=PK, i.e., through the LN clause. The possibility of such calculus for queries selecting values of some attributes of R and of some of the referenced tables was novel by Codd's times. It apparently motivated the "logical" qualifier. The rationale for Codd was indeed that in "well designed" DB, all the non-key attributes referenced by any FK of R, are, conceptually, also attributes of R. E.g., these were the attributes we spoke about for S-P.SP, i.e., every non-key attribute of S or of P. For every tuple of R, if FK had some value, then the referenced values were all these in the sole eventual tuple with PK=FK.

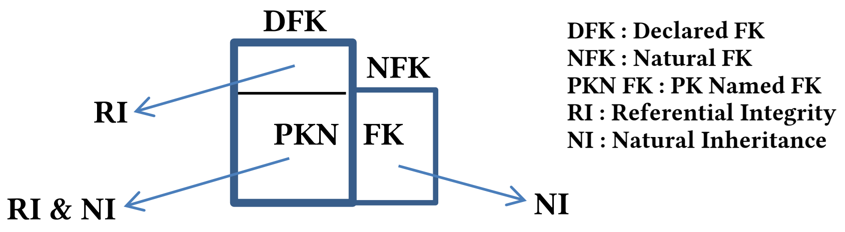


Fig. 3 SIR SQL Foreign Keys

However, for any base table R, none of such conceptual attributes could be among the actual ones of R. As above discussed, they would necessarily be SAs in Codd's model. Hence, they would always imply the normalization anomalies. The side-effects of the latter, hinted to above, would typically offset any practical interest of the LNF queries with respect the same outcome queries with LN to the normalized tables. However, as also discussed already, when all these attributes are NIAs instead, none can ever imply any normalization anomaly. The LNF queries to any R with NI become attractive again, as it will appear more below. It is our rationale for NI concept in SIR SQL.

For SIR SQL furthermore, for any declared PKN FK, the referenced table and PK attribute names for the LN are provided as usual in SQL. Recall that for a dialect, this may be simply the 'References' keyword in a BA scheme. For any natural PKN FK, the LN is in contrast implicit through FK=PK name equality. Notice that for SIR SQL, for any base table R with PKN FKs, the mandatory presence of NIAs in the explicit scheme means that R has to be a SIR. As Fig. 3 illustrates, any natural FK F implies only the NI through F. Any declared PKN FK implies both: NI and RI for the FK. Thus for a SIR SQL FK, RI is not mandatory but optional only, while NI is the characteristic property of any PKN FK. Unlike it is of course for any SQL FK. Codd apparently originally considered RI as we do, [1]. Also, NI was for him the characteristic property of any FK, as e.g., in S-P. It remained however for the actual use in queries and views only, being implicit only for any base table with FK. With this difference, Codd FKs were, apparently SIR SQL PKN FKs, [1,2,3].

Notice further that for a natural FK, there may be tuples with FK values for which there is no referenced PK values. According to just discussed Codd's original idea of FK, NIAs are then null, i.e., the LN (clause) consists of an outer semi-join. Recall also that in some SQL dialects, an FK may reference a candidate key instead of the PK. Possibly, even with the same proper name as the FK. Observe that no such (declared) FK can be a PKN FK. For SIR SQL, any non-PKN FK, hence any such FK in particular, implies only RI for the base table, as at present, Fig. 3. If any LN should be predefined, it has to be explicitly in the presumed implicit Create Table. Likewise, any IAs defined through have to be explicit in the attribute list there as well. These can have same name as attributes selected in the referenced table, or may be CAs defined using the LN or can be aliased IAs (assimilated below to simplest CAs). All these IAs can be anywhere in the attribute list of the Create Table. Finally, observe that, also as at present, any SIR SQL FK has to be an SA.

E.g., For SIR SQL, SP.S# and SP.P# are natural FKs whether in (1) or (2). There is indeed no FK clause in these schemes. These attributes are FKs for Codd in [1] as well, but, as any SQL clients know, not for SQL. None implies any RI, but, for SIR SQL, both do imply NI through SP.S# and SP.P#. Accordingly, they imply (2) without T-WEIGHT as the explicit SP scheme given (1) as the implicit one, as well as every corresponding IA with all its values in Fig. 2. Alternatively one could declare these FKs through Foreign Key clauses, as they would have to be if one wished them to be SQL FKs. For SIR SQL, this would imply not only the NI, but also RI for each FK. Finally, whether natural or declared, both FKs are SAs, as SIR SQL requires for any FK.@

Next, we consider that every presumed implicit SIR SQL Create Table R with any (SIR SQL) PKN FKs is effectively an implicit one, as we already mentioned. The explicit Create Table R is the implicit one with NI added to. In other terms, for SIR SQL, in every presumed implicit Create Table with PKN FKs, every PKN FK in is not only the reference to PK named as FK, but is also the shorthand for the NI from the referenced table. One infers then the NIA names and the LN clauses through the SYS meta-tables. Easy algorithms for, discussed in our previous articles, were prototyped for the SQLite kernel. Altogether, e.g., as somehow stated already, if one submits Create Table (1), (2) without T-WEIGHT will result as the explicit one.

Accordingly, for SIR SQL, any SQL Create Table R with SIR SQL FKs, does not define “only” an SQL base table as at present, i.e., only with the BAs, table constraints and options. Instead, it defines SIR R with the same BAs, table constraints and options, forming base R_ but also with the NI. E.g., rephrasing what was just stated, while for SQL presently, (1) defines “only” SP with attributes as at Fig. 1, for SIR SQL, it defines *in finale* SP (2) without T-WEIGHT, i.e. with the attributes at Fig. 2, without the latter. Accordingly for SIR SQL, for S-P content in Fig.1, (1) specifies the SP content in Fig. 2 without T-WEIGHT column.

Finally, as hinted to in the S-P1 example, the rationale for implicit schemes is (i) that they are obviously always less procedural than the explicit ones. Then, (ii), whenever base tables of SIR SQL DB do not contain any CAs, the DB Administrator (DBA) may simply issue the present Create Table for each of these tables. DBA provides accordingly for the LNF queries without any additional work to define the IE. Recall, - as it was wished for our solution, Without “moving a finger” as one says, DBA makes accordingly SIR SQL clients likely happier and for sure more productive than their present SQL counterparts.

3. Define formally SIR SQL Create Table. Accordingly, suppose now that some implicit Create Table R defines, in the left-to-right order, PKN FKs F1…Fk ; k > 0. Next, by analogy to the SQL ‘*’, suppose that one denotes as R_.* all the attributes defined as they could be at present in the kernel SQL. Next, for every Fi, let us denote the base table referenced by Fi as R’i and, as R’i.#, all the non-primary key attributes of R’i. Also, consider that the presumed implicit Create Table R has the usual form of an SQL Create Table, i.e.:

```
Create Table R (R_.* [<Table constraints>)  <Table options>];
```

The explicit Create Table R would be then as follows, with NI scheme in { } brackets:

Create Table R (R_.* {R'1.#,…,R'k.# From R_ Left Join R'1 On R_.F1 = R'1.F1 … Left Join R'k On R_.Fk = R'k.Fk} [<Table constraints>]) [<Table options>];

E.g., for (1) being the implicit Create Table for S-P1.SP, (2) without T-WEIGHT, already indicated as the explicit one, conforms to this definition.

Furthermore, consider that the presumed implicit Create Table R explicitly defines some IAs. These can be CAs that are not VAs for the kernel providing for the latter or, simply, IAs named upon the same name attributes in a base table not referenced through a PKN FK. Accordingly, we consider that the presumed implicit IE may have no From clause or may have From clause that (explicitly) defines some equijoins on some R_ attributes and, for each such attribute A, on a key attribute named differently of A, within some base table other than NI. We consider that this clause does not contain any Where clause. No practical need for the latter appeared as yet. Next, consider that R.* stands for all the attributes in Create Table R, BAs and IAs thus. Finally, suppose that [<optional From>] designates the just discussed From clause. If it follows a BA, then it should start with '{' bracket. Also, it should always terminate with '}' bracket. The presumed implicit Create Table R should then be in the form:

Create Table R (R.* [{] [<optional From>]} [<Table constraints>]) [<Table options>];

This Create Table is the explicit one if R has the optional From clause and does not have any PKN FK. Otherwise, it is an implicit Create Table. IE is not empty anymore however, although it might be without From clause. As before, SIR SQL Create Table processing would preprocess the scheme to the explicit one, with the (explicit) IE as follows:

Create Table R (R.*, [{] [From R_}] | [{]R'1.#,…,R'k.# [< optional From>] Left Join R'1 on R_.F1 = R'1.F1 … Left Join R'k on R_.Fk = R'k.Fk}] [<Table constraints>) <Table options>];

The From R_ clause is for the earlier discussed case of CAs sourced, in R_ or in other such CAs only. Recall, that these CAs are similar to VAs in SQL, sourced only in some SAs or other VAs of R being defined. Both cases here generate explicit From clauses compatible with SQL Create View syntax. The general goal of the proposed implicit Create Table is, of course, to let the DBA to issue Create Table as non-procedural as reasonably possible within the general SQL framework for data definition. E.g., for SP (1) enlarged with T-WEIGHT, DBA could accordingly issue:

Create Table SP (S# TEXT,P# TEXT, QTY INT {QTY*WEIGHT As T-WEIGHT} Primary Key (S#,P#));

Observe that for T-WEIGHT, DBA declared only the value expression. Within SQL framework for SA or IA definition, it is not possible to have any less procedural Create Table for SP. Since SP has PKN FKs and is without From clause, Create Table (2) will result from as the explicit one. Any further Create Table SP processing would concern the latter.

Notice that the explicit IE, in (2) thus, appears more than three times more procedural than the implicit one above. C-view SP would be even more procedural than the explicit IE, as it's easy to find out. Observe

especially that Q3 is now possible, unlike for the original S-P.SP. In the same time the LNF queries like Q1 remain valid. Moreover, LNF queries may now also address T-WEIGHT. E.g., as the following one:
Select S#, SNAME, P#, PNAME, QTY From SP Where T-WEIGHT > 2000;

In other words, a query to SP may now be both: LNF addressing any attributes in S-P1, hence also all these in S-P, and CAF for T-WEIGHT specifically.

The result actually generalizes nicely to any SIR R with NI and CAs. Namely, let us call any such R *naturally dependent,* (ND), *on* its NI source tables. E.g. S-P1.SP is ND on S and P. Observe that an NI source can be ND in turn and so on. Under some theoretical conditions, too advanced to discuss them here, but actually rather typical, NI in R may then directly or indirectly naturally inherit from every non-PK attribute of every base table in the DB other than R. SIR SQL query addressing some R attributes, possibly CAs among these, together with some attributes inherited through the NI, may be LNF and, possibly, CAF, for directly or indirectly any attributes of base tables other than R. In contrast, any equivalent query to the SQL DB with the same base table schemes as the bases of SIRs with the same names, has to include the LN and perhaps CA schemes. It has then to be always more procedural than the former, perhaps several times. E.g., as it was for Q3 and for Q4.

All this nicely leads furthermore to the general framework for SQL queries with LN reducing to LNF ones in SIR SQL. Namely consider any (typical) SQL DB D1 with some base tables R1…Rk each with FKs and with some base tables B1…Bj without any FK. Consider then also a SIR SQL DB D2 named upon D1 and with B1…Bj, as well as with SIRs named upon R1…Rk and such that for every i = 1..k, D1.Ri = D2.Ri_. Next, consider query Q^1 to D1 addressing any attributes among those of some Rj ; j = 1..k. Or addressing, using LN through any of PKN FKs in Rj, any attributes among all those of every table R referenced by these FKs. Or, for every of the latter tables with PKN FKs, addressing using LN through any of these FKs, any attributes among those of every table referenced by the latter. Etc. Then, every such Q^1 reduces to LNF query Q^2 to D2, with the same select-project clauses as Q^1 and with From Rj clause only. Except that some proper names of the IAs addressed by select-project clauses not qualified in Q^1 might get qualified in Q^2. This, - since they had to be qualified in Rj.

Notice that there may be then D2.Rj, where there is Q^2 for any choice of D1 attributes for Q^1. Next, suppose some CAs in D2, enlarging any B tables, each becoming thus a SIR, or enlarging any D2.R. Then any query Q^3 to D1 with schemes of some of such CAs within, becomes the evident CAF or LNF and CAF Q^4 to D2.

Finally, one gets similarly the simple condition for every SQL Create Table R to be the implicit one of SIR R in SIR SQL. It is so, iff R has any PKN FKs. Then, every present SQL D1 scheme as above is also, for SIR SQL, the one of D2 where every Create Table R with PKN FKs is the implicit one for SIR R.

Suggestion: check-out the above framework for S-P and S-P1, supposing that Create Table S and Create Table P precede Create Table SP (why in fact one should suppose so?). Besides, observe that any LNF Q^2

above results only from PKN FKs in D1. A DBA has however also the possibility of non-PKN FKs, Fig. 3. Such an FK, say F, may provide then for RI only, without any IAs (using LN) through F. Or, in practice F may be useful to provide (through LN) for every IA that could be NIA if F was a PKN FK, but with all such IAs being placed elsewhere in R. Or, through such F, one may select for IAs only some of the attributes of the referenced table, e.g. for security reason. Or, one can define some CAs instead of some of these IAs.

DBA has to define every such IA explicitly then within Create Table. The latter may be, perhaps, implicit, given presence of PKN FKs as well. Explicit IAs mean further that Create Table with, is only SIR SQL one. Finally, if one introduces accordingly a non-PKN FK to Create Table R in D2 for any of the above reasons and creates explicit IAs accordingly, R will provide for LNF queries addressing these IAs as well. Unlike could do D1, of course.

E.g., suppose that S-P1 DBA does not wish every attribute of P as NIA in SP, available to LNF queries there thus. Instead, from P, DBA wishes for SP only PNAME aliased to PART_NAME, COLOR and WEIGHT-KG expressing part weight in KGs rounded to grams, instead of US pounds in P.WEIGHT. P.CITY should be invisible to queries to SP. Instead of (1), the following implicit (why?), SP could do:
Create Table SP (S# TEXT, X# TEXT {PNAME AS PART_NAME, COLOR, WEIGHT-KG AS ROUND (WEIGHT/4.53, 3)} QTY INT {From SP_ Left Join P On X# = P# }) Primary Key (S#, P#));
We leave as exercise the explicit Create Table SP resulting from.   Also, - the query to S-P1.SP searching for all data about every supply by Smith. As well as the equivalent query to S-P.SP. Not surprisingly, expect the latter being substantially more procedural than the former and anything but quasi-natural query.

4. <u>SIR SQL Canonical Implementation.</u> Let us call SIR (enabled) DBS, any relational DBS (RDBS) providing for SIR SQL. To implement a SIR DBS "simply", i.e. through a couple of months of programming only, stick to the *canonical* implementation. In the nutshell, SIR DBS consists then from the front-end, called *SIR-layer*, reusing an existing *kernel* RDBS, e.g., SQLite3. The tandem works as follows:

- SIR-layer takes care of every SIR SQL dialect statement and returns any outcomes. Every SIR SQL dialect extends to SIRs the kernel SQL dialect.

- The kernel is the actual storage for every SIR SQL DB, becoming the same name DB for the kernel SQL.

- SIR-layer forwards to the kernel every Create Table R submitted without PKN FKs and without any IA, but perhaps with VAs declared as if they were intended for the kernel. Any Create Table R with PKN FKs is for SIR-layer an implicit scheme, preprocessed accordingly to the explicit one with the (explicit) NI. Besides, SIR-layer parses every explicit Create Table R to Create Table R_ and Create View R defining C-view R. It then forwards both statements as an atomic transaction to the kernel. Fig. 4 illustrates the result for S-P1.SP processing.

- SIR-layer also forwards to the kernel every (SIR SQL) Alter Table R that does not contain SIR-specific clause termed IE clause. It is indeed supposed kernel SQL Alter Table, addressing thus base table R that is not a SIR and should remain so. SIR-layer also forwards any Alter Table R_. IE clause may be explicit or implicit, even empty. It always means that R is or should become a SIR. If R is a SIR already, SIR-layer issues to the kernel Alter View R with new C-view R produced from IE clause and, for an implicit IE clause, from the altered R_ scheme and from view R scheme in kernel SYS-tables. If R is not yet a SIR, SIR-layer similarly produces and sends to the kernel as an atomic transaction: Alter Table R renaming R to R_ and Create View R with C-view R. See [5] for more.

- Furthermore, SIR-layer forwards to the kernel any Drop Table R if R is not a SIR. Otherwise it issues an atomic transaction with Drop Table R_ and Drop View R.

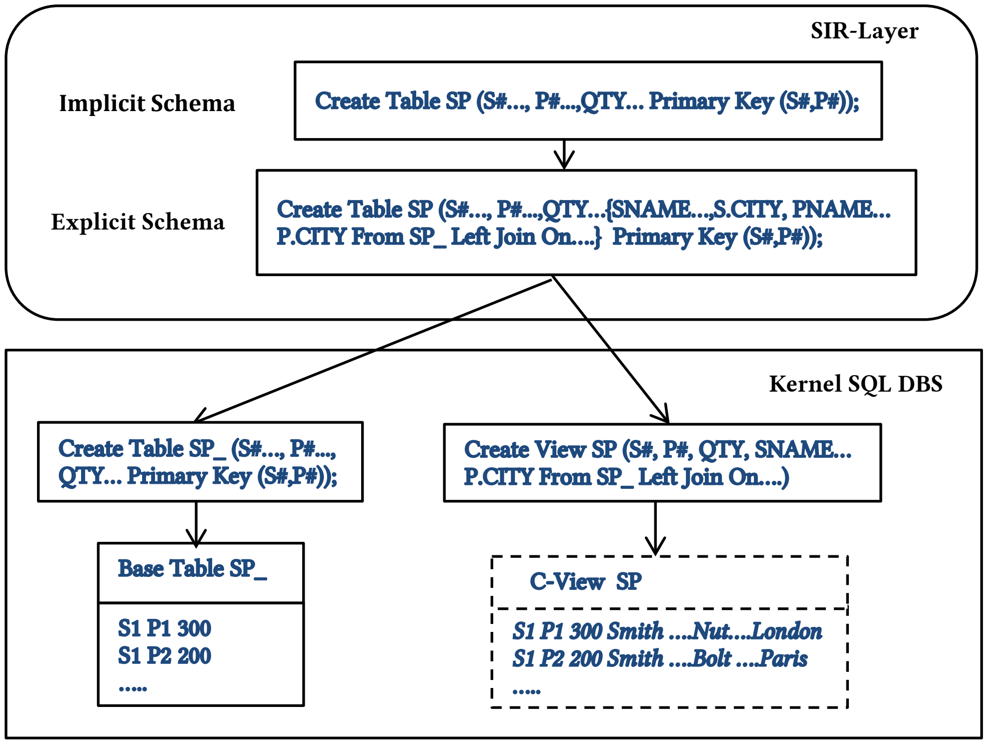

SIR-Layer
Implicit Schema
Create Table SP (S#..., P#...,QTY... Primary Key (S#,P#));
Explicit Schema
Create Table SP (S#..., P#...,QTY...{SNAME...,S.CITY, PNAME... P.CITY From SP_ Left Join On....} Primary Key (S#,P#));
Kernel SQL DBS
Create Table SP_ (S#..., P#..., QTY... Primary Key (S#,P#));
Create View SP (S#, P#, QTY, SNAME... P.CITY From SP_ Left Join On....)
Base Table SP_
S1 P1 300
S1 P2 200
.....
C-View SP
S1 P1 300 Smith ....Nut....London
S1 P2 200 Smith ....Bolt ....Paris
.....

Fig. 4. **Canonical Implementation of SIR SP with the actual content with the kernel SQL DBS. C-view SP content is virtual only, as usual for any views.**

- For the SIR SQL data manipulation statements, SIR-layer simply forwards any submitted query to the kernel. For any SIR SQL update statement, safe policy for every kernel and every SIR R is to address R_. E.g., Insert To SP_..., Update SP_... and Delete From SP_... for S-P1.SP. An update statement addressing SIR R directly, e.g., Insert To SP…, may or may not work. It depends on kernel's view update capabilities.

The kernel would indeed address any such queries to view R. In particular, no present kernel provides for any CA updates.

6. Finally, validate the canonical implementation through the proof-of-concept prototype. SIR-layer in Python and SQLite3 as the kernel appeared the most suitable for this goal. The actual prototype available at present provides also for self-running demo. The overall effort was 2-3 months of makeshift Python's developer, i.e., the effort conform to expectations. The demo creates S-P1, either from the explicit SP scheme or from the S-P.SP scheme. The latter is assimilated to SIR SQL implicit scheme with empty IE, resulting from the natural PKN FKs S# and P#. Then, one manipulates S-P1, through LNF queries or, after adding T-WEIGHT CA, through LNF and CAF queries. Users familiar with Python may easily alter the demo. E.g., to prepare their own SIR DBS reusing another kernel: DB2, SQL Server, PostgreSQL, MySQL… you name it. See [9] for more on the prototype.

3. CONCLUSION

Since five decades i.e., since 1974 when IBM introduced SQL, every SQL DBS requires the clients with typical queries to base tables, to specify the LN and the CAs, at least other than VAs for some dialects, in the queries. The procedurality of the LN and of CA specs is usually substantial, i.e., can easily double or triple the query size, bothering then many. SIR SQL gets rid of this annoyance, simplifying the typical queries to the LNF and CAF ones. In particular, the LNF queries to the base table defined as generally at present become possible. For CAF queries, it may suffice to add to Create Table only the value expressions defining the CAs. Finally, the proof-of-concept prototype SIR DBS with SQLite as the kernel proved simple to realize. Although the problem of LNF and CAF queries is anything but new, our solution is the only of the kind, to our best knowledge.

We postulate accordingly that DB courses and textbooks from now take notice of SIR SQL. Also, that every popular SQL DBS upgrades its dialect to SIR SQL one, "better sooner than later". 7+ million clients worldwide of the most used DB language by far, [6], [7], [10], including 70% of all developers and providing for estimated 31B US$ market size of SQL DBSs, [8], will benefit from.

REFERENCES

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

[2] Date, C., J. & al. An Introduction to Database Systems, 8th ed. Pearson Education Inc. ISBN 9788177585568, 2006, 968p.

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

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

[5] Litwin, W. Stored and Inherited Relations with PKN Foreign Keys. 26th European Conf. on Advances in Databases and Information Systems (ADBIS 22), 12p. Springer (publ.).

[6] Gaffney, K., P., Prammer, M., Brasfield, L., Hipp, D., R., Kennedy,D., Jignesh, M., P. SQLite: Past, Present and Future. PVLDB, 15(12):3535-3547,2022.

[7] How Many SQL Developers Is Out There: A JetBrains Report, Dec. 23, 2015.

[8] ZipDo. Essential Sql Statistics in 2024. https://zipdo.co.

[9] Litwin, W. Home Page. https://www.lamsade.dauphine.fr/~litwin/witold.html .

[10] Stonebraker, M. Pavlo, A. What Goes Around Comes Around… And Around… SIGMOD Record, June 2024 (Vol. 53, No. 2).