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

Witold Litwin
University Paris Dauphine, PSL

Abstract — **SIR SQL stands for SQL with Stored and Inherited Relations (SIRs). Every SIR SQL Create Table makes definable any *base* attributes one could have in an SQL Create Table at present. But one can also define *inherited* attributes (IAs), definable only in SQL queries or views up to now. One may also define foreign keys (FKs) that are SQL ones or logical pointers in Codd's original sense. IAs in SIRs with Codd's FKs usually provide for logical navigation free (LNF) queries, i.e., without equijoins on FKs and referenced keys. The same outcome SQL queries to the same base tables without IAs, must include LN avoided.**

SIR SQL Create Table may in particular include IAs definable through value expressions also possible in SQL queries or views only up to now, usually referred to as *calculated attributes* (CAs). CAs may involve, e.g., attributes from different tables or aggregate functions, or sub-queries. CAs in SIRs provide for CAF queries, addressing any CAs in SIRs by name only. In contrast, every SQL query to base tables needing CAs has to fully define each of these.

The end result is that most of SQL queries to base tables, requiring LN or CAs schemes at present, become LNF or CAF in SIR SQL. The latter queries are usually substantially less procedural, i.e., by dozens of characters at least. They become also quasi-natural, i.e., with Select clause only naming the selected attributes, From clause naming a single base table and Where clause, if any, with, at worst, short Boolean formulae over usual constraints on some attribute values. SIR SQL should accordingly significantly boost SQL clients' productivity. Especially, since most clients are data analysts or application developers, not SQL geeks. While the problematic of LNF and CAF queries is four decades old, our solution is the first practical one, to our best knowledge.

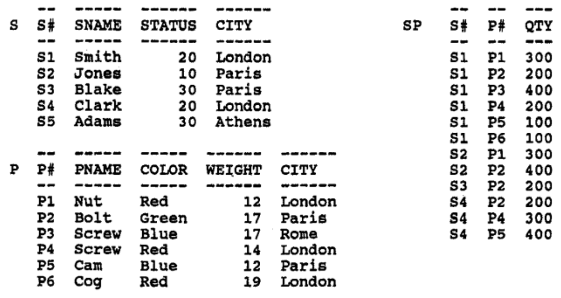
Below, we illustrate the problem of LN and of CAs in queries to SQL base tables using Codd's original Supplier-Part DB. We then present SIR SQL. We show in depth how SIR SQL LNF and CAF queries to base tables become possible. We show in particular that Create Table statements defining an SQL DB at present, usually define also a SIR SQL DB, providing for LNF queries to base tables as free bonus. We discuss the front-end for SIR SQL that should require, for any popular SQL DBS, a few month implementation efforts only, validated by proof-of-concept prototype for SQLite3. We accordingly postulate to upgrade every popular SQL DBS to SIR SQL. 7+ million SQL clients worldwide, of the dominant DB language, providing for 31B+ US$ market size of SQL apps, will benefit from.

***Keywords*—Relational database model, *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 most of queries to the base tables, necessary beyond the otherwise quasi-natural formulation of such queries. The latter consists of Select with some attributes of a single base table named in From clause, called *addressed* by the *query* and of Where clause, if any, with, at worst, short Boolean formulae over usual constraints on some attributes, e.g., $A < 100$ And $A \geq 50$. The 1st culprit for those cumbersome parts is the logical navigation (LN) in queries to base tables with foreign keys and to the referenced tables. Recall that the term *LN* or *LN joins* means equijoins on foreign and referenced keys, as Codd originally defined these terms in [1] and results from Codd's 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., LN always expresses a semi join, reducing to the inner one, if one wishes so, iff FK respects the referential integrity (RI). The procedurality that the LN implies, i.e., the necessary length (number of characters) of the SQL join clauses defining it, usually adds at least dozens of characters to the query without. This makes accordingly, at least linearly, longer query writing and debugging times. Especially, - when the joins are the outer ones, [4]. Not surprisingly, clients usually at least dislike the LN. 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., by dozens of characters to type-in at least. The sheer complexity of some CAs, those defined by sub-queries especially, also bothers many, implying particularly careful debugging.



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 ones of most of DBs at present and properties 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 query searching for every supply so and so… in practice, would most of time address some of SP attributes together with some attributes of S or P in its Select and Where clauses. The rationale is that all the non-key SAs of S and P are conceptual attributes of SP as well. They should be also SAs of SP. They are actually not. The normalization anomalies for SP that would follow and that we discuss more below are indeed 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 quasi-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 quasi-natural language query.

Next, in practice, 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, Maier & 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.

Besides, for decades there were sporadic proposals for DBs with 1NF only base tables, including all the conceptual base table attributes we spoke about, hence providing for LNF queries. None made to practice, obviously outweighed by inconveniences of normalization anomalies. Likewise, it was always known that one may hide normalized base tables behind multiple denormalized views providing for LNF and CAF queries. This approach did not make it neither. Sheer number of Create View statements necessary to type-in with multiple replications of base attribute names, as well as problematic maintenance of views in sync with alterations to base tables, about always outweigh the advantages to queries. Altogether, the problem of a DB supporting LNF and CAF queries to base tables, remained open.

2 OUR SOLUTION

Overview. The idea is that, since 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 defining its values. Likewise, every Create Table R should predefine every CA considered as (conceptual) attribute of R. The tricky issue is that none of these additional (pre)definitions should imply any normalization anomalies in R, with respect to the existing NF of R. This said, everything that follows is mere technical details, intended to make the proposed solution the most practical.

Notice upfront that trivial SAs cannot help, 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 in Codd's sense, all the names logically pointed to should indeed be already in SQL meta-tables. One can also easily infer the LN clause whenever an FK is an SQL one or, is a so-called *primary key named* SIR SQL (specific) FK, as we'll show. 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 manually copied names or in LN clauses and the waste of time for their eventual debugging. 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.

From now on, we introduce SIR SQL through the following steps, described in "for dummies" form. We published some details separately already, in depth impossible here within the space limits. Especially in [5] that references in turn main earlier related papers. Our home page, [9], indexes all our related papers, by title and abstract at least. The pdf is the bonus, whenever not copyrighted. There is also ppt for conference papers.

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 Create View or query only up to now. 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.

An IE defines every IA A in the attribute list of SIR SQL Create Table as one can do for A in the attribute least of an SQL query or view. I.e., A can have the name of an attribute in the table defined by From clause or can alias such an attribute or can be a value expression over some such names or can be a sub-query… In every case except the 1st one, for SIR SQL, we qualify A of CA. Some IAs in the list may also result from the generic SQL '*' character. While IE attribute list can thus be as in any queries or views, IE is in contrast, limited with respect to the two other clauses of an SQL query or view, i.e., From and Where clauses. Every From clause should indeed either be simple From R_ or a sequence of left or right or inner joins, each on some BA, perhaps composite, and a key attribute of usually other table. These joins should further be such that (i) for every R_ tuple *t*, there should be in the table defined by From clause, say T, exactly one tuple *t'* with *t* as a sub-tuple and (ii) T should not have any other tuples. We will recall the rationale for these assumptions in what follows.

We qualify From clause in SIR SQL Create Table formed as above of *valid*. Otherwise it is *invalid*. We also refer to the joins expressed within as to SIR SQL *(predefined) LN (joins)*. In practice these joins should be indeed the ones we spoke about. I.e., they predefine SIR SQL LN (joins) that would be typically required by SQL queries (at present) if they addressed the same base tables without IE. Furthermore, whenever R is a SIR, BAs together with the table constraints and options form R_ scheme.

Next, in every SIR SQL Create Table R, if there is any From clause, it then follows the (entire) attribute list with BAs and IAs and precedes every eventual table constraint or option. Also, whenever Create Table R defines also any part of IE, i.e., some consecutive IAs or From clause, every such part should be separated from any of R_ SQL specs by { } brackets. Each bracket replaces a usual SQL separator, i.e., ',' or space. IAs may be spread among BAs or separated by BAs from From clause, hence 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 Create Table SP formulated as follows:

(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#));

Create Table (2) is a SIR SQL one only, i.e. impossible to formulate in any present SQL dialect. It defines SIR SP enlarging SP (1) with IAs defined within. The attributes and the (only) table constraint of (1) within (2), define the base SP_. IE is entirely within a single pair of { }. Also as required, From clause follows the entire attribute list and precedes the only SP table constraints that is the Primary Key constraint. Let us call S_P1 the DB with S, P and SP (2). Fig. 2, placed at the end of the text, shows S_P1.SP scheme and content for SIR SQL clients, given Fig. 1. We underlined every key attribute (and only such attributes), as usual. IAs are in italics. 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, the table in SP (2) From clause would contain one and only one tuple (S6, P1, 200, null, null, null, null, P1, Nut…). One may easily see also that this property generalizes to any tuple breaking the IR with respect to S or P. The discussed table would not contain further any more tuples than these resulting from inserts to SP_. From clause in (2) is thus a valid one. In contrast, any From clause for SP (2) with any inner join instead of the outer one, would not fit. The latter would make From clause valid iff the RI was enforced. Notice that the outer join expression would remain valid then anyhow.

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 (2). As the values of these are calculated only, none of these IAs can ever create any normalization anomalies, i.e., insert, update, delete or storage anomalies. These anomalies would in contrast necessarily occur, if any of IAs of S_P1.SP was trivially, an SA, as Codd's model requires for every BA. Recall that the relational model prohibits the anomalies 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 SP.S# = '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].

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 qualify of *explicit* every SIR SQL Create Table R defining every IA and From intended for R as above discussed. E.g., (2) can be the explicit Create Table SP for S_P1. We also call then *explicit*, IE, IAs, IA list and From clause, e.g., within (2). Besides, as it will appear, SIR SQL Create Table lets DBA to omit parts of even entire IE. We speak then about *implicit* Create Table, IE…. We qualify of *empty* an entirely omitted IE. As it will appear, in practice, an implicit Create Table should be always substantially less procedural than the explicit one.

More in depth, a Create Table is implicit iff it (i) contains any SIR SQL FK qualified in next section of already mentioned PKN FK or (ii) contains only some CAs, including IAs simply aliased, but neither has From clause nor PKN FKs, or (iii) contains SIR SQL specific generic character '#'. Whenever there is an implicit Create Table for an explicit one, the DBA is (obviously) expected to take advantage of the former. Accordingly, every submitted SIR SQL Create Table is subject to SIR SQL specific pre-processing. This one 1st finds whether the statement is effectively implicit. Iff it turns out so, the pre-processing rewrites the statement to the explicit one. Every rewriting keeps every BA, table constraint and option of the submitted Create Table, as well as every IA and From clause, if there are any. It adds IAs or parts of, or even entire, From clause, so to form the explicit Create Table R. If the submitted Create Table does not turn out to be implicit, the pre-processing considers it explicit, hence not needing any rewriting. The further SIR SQL processing we discuss later on works on the explicit statements only.

We discuss later the rewriting rules for PKN FKs. Notice for now only, with respect to SIR SQL FKs, that every (present) SQL FK in some SIR R, is SIR SQL one as well, by default. Besides, a BA in R can be a SIR SQL FK, without being the former. With respect to '#', we recall here only that while modeled on SQL '*', whenever qualified with a base table name, e.g., R.#, it designates only every non-PK attribute of R. Likewise, '#' alone designates only every non-PK attribute of every base table in From clause. We discussed the rewriting of a Create Table with '#' previously, [5], and will not come back to here.

E.g., we show soon that DBA may create S_P1.SP through the implicit Create Table containing only SP_ scheme and the value expression of T-WEIGHT. Instead of submitting (2), one expects therefore DBA to submit only:

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

The procedurality difference is 110 characters in favor of (3). I.e., 86 characters are necessary in SQL for (3) versus196 for (2), as one can easily double-check. Such lesser procedurality is an obvious practical advantage of (3). 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.

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, for SIR SQL, S_P scheme suffices for the LNF queries that have to be with LN in any present SQL dialect.@

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, discussed in detail in our previous papers, 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.

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. In our opinion, SQL FK concept differs from Codd's original one, [1], [3]. Codd defined an FK as a "logical pointer" (LP) to some table. For SIR SQL, it appeared useful to merge both concepts. We called the result *SIR SQL FK.*

Overview. Accordingly to our intention, an SA in submitted Create Table R can be SIR SQL FK along following dimensions, Fig. 3. Along one dimension, FK may enforce RI. We speak accordingly about *RI dimension* and *RI* FK. The other dimension defines FK as LP. We speak about *LP dimension* and *LP* FK. LP dimension is our perception of Codd's FK original concept. Besides, from now on, FK means SIR SQL FK, unless we talk specifically about (present) SQL FKs.

FK specs along LP dimension in Create Table R contribute to IE. One defines accordingly in explicit and perhaps also implicit Create Table R, LN through FK and every IA calculated using this LN. In the wake, it is LP dimension that predefines in explicit SIR SQL Create Table R, IAs and LN making LNF and CAF typical queries addressing R.

With respect to RI FKs, one defines every such FK as if it was an (SQL) FK for kernel SQL dialect. Whatever is the kernel, one may always define every such FK through kernel's dialect for SQL Foreign Key constraint. Some kernels provide also for References keyword in FK specs as SA, e.g., SQLite. In SIR SQL Create Table, one should place every Foreign Key constraint after IE, hence after the last '}' in practice. The only semantic difference to kernel's specs of the same name SQL FK may be that RI FK references a SIR.

An SA F in submitted SIR SQL Create Table R is LP FK, iff in explicit Create Table R there is LN on F and there are some IAs defined using this LN. Every FK F can be IR FK only or LP FK only or both. Every LP FK is either *primary key named* (PKN) FK, [5] or *LN defined* (LND) FK. PKN FKs do not require LN and IAs in submitted Create Table R. LND FKs do. For every PKN FK in submitted Create Table R, the pre-processing adds LN and IAs transparently. PKN FKs appear in fact the common practice already, except that they are defined only as RI FKs. LND FKs should serve less frequent specific needs we discuss soon.

PKN FKs. An FK F in base table R is PKN iff (i) F is not PK of R, (ii) there is a base table R_1 with PK named F as well and sharing the domain of R.F, (iii) F is RI FK and R_1 is the referenced table or (iv) F is atomic and there is only one R_1. In the latter case, if FK is not RI FK, we speak about *natural* (PKN) FK, or NFK in short.

Observe that, by definition then, NFKs do not enforce RI, i.e., are LP FKs only. RI is thus optional for PKN FKs. Recall that Codd apparently considered RI optional for FKs as well. Unlike did the SQL designers. Observe also that NFKs do not require any specific declarations in submitted SIR SQL Create Table, unlike PKN RI FKs. On the other hand, observe that R_1 is always necessarily different from R. Recall finally that some kernels, e.g., MySQL provide for referenced keys that are not PKs. Even if a latter (candidate) key shares the name with FK, it is not PKN FK.

E.g., consider (3) as submitted SIR SQL Create Table. Suppose that DBA already created S and P. Then, S# and P# are PKN FKs. None is RI FK, i.e., enforces RI, e.g., through the familiar Foreign Key SQL table constraint. Each is thus NFK. Hence, e.g., insert of (S6, P6, 300) would go through.

A submitted Create Table R is implicit if it contains PKN FKs. For every PKN FK F, F defines so-called *natural* (NI) *through* F. The term designates (i) so-called *natural* IAs (NIAs) and (ii) LN on F through which the values of every NIA are computed for queries. Both NIAs and LN are in the explicit Create Table R only. In other words, the pre-processing adds these to submitted Create Table R whenever it finds PKN FKs within.

More in depth, for every PKN FK F1,F2…, numbered in the (left-to-right) order in R and referencing respectively R1, R2,…, NIAs constituting NI through Fi ; i = 1,2… ; have names and values of all and only non PK attributes of Ri. In particular, whenever needed, an NIA name can be the qualified Ri attribute name. Then, for every NIA, LN on R.F = Ri.F, with vector equality for every composite F defines its every value and every null. Also, all the NIAs constituting NI through F1 follow the last BA or IA specified in implicit Create Table R. Then, all the NIAs of NI through F2 follow NI through F1 etc. Finally, for every Fi, NIAs in NI through Fi are in their source order in Ri.

With respect to every LN defining NI through Fi, expressed only in the explicit Create Table R, we recall, if implicit Create Table R has no From clause, then pre-processing expresses LN on F1 as: From R_ Left Join R1 on R_.F1 = R1.F1. Else, it expresses LN on F1 as: Left Join R1 on R_.F1 = R1.F1, appending it to From clause in the implicit Create Table R. Next it appends to From clause being built up LN for F2, if there is any, expressed as: Left Join R2 on R_.F2 = R2.F2 etc. We qualify of, simply, *NI*, all the resulting NIAs and LN joins added.

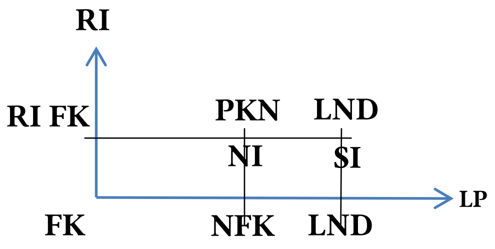


Fig. 3 SIR SQL Foreign Keys. RI FK enforces RI. PKN FKs imply NI, NFKs in particular. LND FKs specify SI. An FK can be RI FK only or LP FK only, or both.

E.g. Let us follow up on previous example. The pre-processing of (3), creates (2) as explicit Create Table SP. IAs following T_WEIGHT together with From clause form NI. All these IAs are NIAs. IAs named upon S attributes and LN on S# form NI through S#. Likewise, IAs sourced in P and LN on P# form NI through P#.@

LND FKs. An SA F, perhaps composed, is LND FK iff (i) in submitted Create Table R, for some R_1 with key C not named F or not a PK, there is LN (join) on F = C and (ii) in the attribute list, there are IAs either named upon attributes of R_1 or being CAs addressing the latter through value expressions or simple aliasing. The former IAs may, in particular, result from R_1.# clause we discussed earlier. SA F that is LND FK does not enforce RI, unless one declares F also RI FK, Fig. 3.

For every LND FK F, we qualify of *specific* inheritance (expression) or of SI in short, through F, the just described IAs and LN, (in submitted Create Table thus). By the same token, the term SI designates the result for all the LND FKs in R and we qualify of *specific* every IA of SI, SIA in short.

E.g. Suppose that SP DBA prefers IAs sourced in S named and placed in SP differently from NIAs in (2). Namely, these IAs should be as in the following submitted Create Table SP:

Create Table SP (SN TEXT {SNAME, S.CITY As SCITY} P# TEXT, QTY INT {WEIGHT*QTY As T_WEIGHT From SP_ Left Join On SP_.SN = S.S#} Primary Key (S#, P#));

Here, SNAME and SCITY are SIAs. SN is an LND FK only, hence LP only, i.e., does not enforce RI. Together, LN and the attributes sourced in S constitute SI through SN. It is (entire) SI in fact, since P# provides for NIAs only. Also, STATUS remains private to S, in the sense that only SP clients knowing that SN and S# share a domain (in Codd's vocabulary) may select it and only through the LN in the query then. We leave the explicit Create Table SP resulting from the above submitted one as exercise. @

Summing up on SIR SQL FKs and Codd's ones. As we already hinted to, for every SIR R with some LP FKs, NI or SI values in every R-tuple reflect Codd's "logical pointer" idea, [1]. Namely, for every PKN or LND FK, one calculates all these values using LN. 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 that in a "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. Some of these attributes in R may be furthermore subject to aliasing or a value expression. For every tuple of R, if FK has then value matching a value of the referenced key C, usually PK, then the value of every such conceptual R attribute, hence of an IA in R, is the one in the sole eventual tuple with PK=FK for NI or, following the notation above, with F=C for SI. For an IA being CA, such a value can further be subject to a value expression. If FK value in R tuple does not match any C value, as it can happen for Codd's FK, then every such IA is null in the tuple. In other words, the general form of LN is an outer semi join. As we have already hinted to, such form remains valid even if LP FK is also RI FK, although an inner join does then as well.

However, in Codd's model, for any base table R, none of such conceptual attributes could be among the actual ones of R. As already discussed, they would necessarily be SAs. Hence, they would always imply normalization anomalies. The side-effects of the latter, hinted to above, would, most of time, offset any practical interest of the LNF queries with respect the same outcome queries with LN to the normalized tables. But, as also discussed already, whenever all these attributes are NIAs instead, none can ever imply any normalization anomalies. LNF queries to any R with NI or SI become attractive again, as it will appear more below. Especially, since as S_P illustrates as well, most of time in practice, a base table needs FKs, since most of its conceptual attributes have to be in referenced tables. Accordingly, most of queries addressing a table with FKs will address NI or SI and formulate as quasi-natural LNF ones. All this is our rationale for FKs in SIR SQL defined as discussed.

In other words yet, for Codd apparently, at least originally, it's not RI, as later for SQL, but NI or SI that were the characteristic property of any FK, as e.g., in S_P. They were used in queries and views only. For any base tables with FKs, NI or SI remained in Codd's model implicit only, [1,2,3]. In contrast, while also following on Codd's intentions, SIR SQL LP FKs provide for the explicit predefinition of NI or SI in the base tables with. This frees accordingly the queries to these tables and to referenced ones from LN formulae.

Next, observe that for SIR SQL, any SQL Create Table R with PKN 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 NI. E.g., for SQL presently, (1) defines "only" SP with attributes as at Fig. 1. But, for SIR SQL, (1) defines in fact SP (2) without T-WEIGHT. I.e. it defines SP as at Fig. 2, without T-WEIGHT column. Accordingly for SIR SQL, given S_P content in Fig.1, (1) specifies SP content in Fig. 2 without the latter one.

Observe finally that, as illustrated above by SP with T-WEIGHT example, the rationale for implicit schemes is that they are always less procedural than the explicit ones. Furthermore, whenever base tables of SIR SQL DB do not contain any CAs and any SI, DBA may simply issue the present Create Table for each of these tables. DBA provides in this way for LNF queries without any additional procedurality 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 at present.

3. State formally SIR SQL Create Table for SIRs. Suppose now that some implicit Create Table R defines, in the left-to-right order, some 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, as we spoke about earlier. 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>];

Finally, consider that '=' in LN join on F = C designates the usual equality if F is atomic and a vector one otherwise. The explicit Create Table R would be 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 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 provided by an SI. Accordingly, we consider that the implicit IE may have no From clause or the one of the SI. Next, consider that R.* stands for all the attributes in the implicit Create Table R, BAs and IAs thus, all the latter being within at least one pair of { } brackets. Finally, suppose that < SI From > denotes the From clause of SI. If it follows a BA, then it should start with '{' bracket. Also, it should always terminate with '}' bracket. Finally, consider that that the notation [A] means as usual that A is optional. Every submitted SIR SQL Create Table R should then fit the form:

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

Form (a) leads to several forms of explicit Create Table R. First, (a) is itself the explicit one, iff R has the < SI From > clause and does not define any PKN FK. Next, R may have all the CAs defined without any LND FK, hence without < SI From> and be also without any PKN FK. Then, the explicit Create Table fits:

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

Next, (a) may define also *k* PKN FKs. Let us denote as <A | B > the usual syntactical clause that either A or B is mandatory. The resulting explicit Create Table R should accordingly have the form:

Create Table R (R.* [{] R'1.#,…,R'k.# < < SI From> | 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>];

As before, for every above case of the submitted implicit scheme, SIR SQL Create Table pre-processing should produce the explicit one. Observe that, in particular, every produced From clause conforms to SQL syntax for From clause for queries and views. Altogether, e.g., that is why DBA could issue the already discussed implicit Create Table SP (3), where, for T-WEIGHT, DBA could declare the value expression only. Since SP has PKN FKs and is without < SI From > clause, the explicit Create Table (2) with full definition of T-WEIGHT results from. Recall that this one is by 110 characters more procedural than the 86 characters long implicit (3), i.e., more than twice.

Notice also that the explicit IE in (2), is over three times more procedural than the implicit one. 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: CAF for T-WEIGHT specifically and LNF addressing any BAs of SP or, through the LN predefined in S_P1.SP, any BAs in S_P1 with T-WEIGHT. Recall that these are the same BAs as in S_P.

4. State more formally the LNF & CAs capabilities of SIR SQL. First, let us sum up the classical terminology we use. Thus, the term LN designates for an SQL query or view, any expression of LN joins. If R is a table with an FK F, in SQL or SIR SQL sense, and R1 is the referenced table with key C, then LN join is an equijoin on F = C that preserves every tuple of R and only every matching tuple of R1. C can be composite, say (c1…cj) in which case F has to be (f1…fj) as well and the equality is a vector one, i.e., f1=c1 And… And fj=cj, to recall. Given an LN join, one speaks also about LN *from* R *to* R1 *through* F.

As known, there are various ways to express LN in From and Where clauses. For practical reasons, SQL standard recommends expressing any LN in From clause only, through an SQL-conform sequences of Left Join or Right Join or Inner Join verbs, followed by join attribute equality clauses. E.g., to express the above LN join as: R Left Join On R.F = R1.C or as R … Left Join On R.f1 = R1.c1 And… And R.fj = Rcj. It is then always possible in particular to express every LN join in From clause as a left join. If R has several FKs, this leads to a sequence of SQL left join expressions only, basically not needing any parenthesis. This is the rationale for our definition of the explicit Create Table above and, consequently, for our algorithm rewriting any implicit Create Table R to the implicit one.

Accordingly, one says that a query Q selecting attributes of some base tables is LNF, iff Q does not express any LN join. One formulates then typically LNF Q as Select… From R Where…, for some base table R and usually qualifies Q of query *to* R. Also,. Q is in contrast a query *with LN* if it expresses any LN joins. Likewise, one may call Q *with* CAs if it contains any named value expressions or named sub-queries or renames (aliases) some attributes. One qualifies Q of CAF if it only addresses (names) some CAs.

Consider now a SIR SQL dialect reusing some popular SQL dialect, say SQLite, as before referred to as *kernel* dialect. Suppose that DBA creates a SIR SQL DB D1 using a sequence S1, allowed by kernel dialect, of Create Table statements. Recall that many but perhaps not all kernels require that for every base table R referencing some R1, Create Table R1 precedes Create Table R. Consider in addition that for every S1, if any R should reference some R1 through a natural FK F, then Create Table R1 precedes Create Table R in S1 (why?). Suppose further that every submitted Create Table R in S1 predefines every CA considered as part of the conceptual scheme of R. E.g., as (3) predefines T_WEIGHT. Finally, let D be an SQL DB created according to kernel's rules through sequence S that is the same as S1, except that one shortens every S1 Create Table R to the one defining the attribute list of D1.R_ only.

Now, consider an LNF and perhaps CAF query Q1 to D1 with From R clause only and selecting some IAs and, perhaps, some BAs of R. Then, for every Q1, there is an equivalent, i.e., with the same output, Q to D. If S1 does not define any CAs, then case studies and S_P especially, easily show that Q can have Select and Where clauses of Q1, except that one may qualify less some of attributes. Unlike Q1, Q has to express however always also some LN. The LN in Q may formulate in particular as: From R <LN exp>, where the latter designates those of the LN joins predefined in D1 that are necessary and sufficient to formulate Q. Likewise, while Q1 may only address predefined CAs, i.e., may be CAF, Q would need to define every CA it needs instead. The latter are at least all the CAs addressed in Q1. In addition, they can be every CA scheme of a CA in some D1.R1 that a CA in Q1 and in D1.R perhaps addresses in turn and so on, recursively.

The exact rules for the equivalence of Q1 and Q are beyond the scope here. The practical result is that LN and, eventually, CA specs within any Q are about always substantially procedural, i.e., require dozens of characters (keystrokes). Implying accordingly the already mentioned typing and debugging time overhead. E.g., observe that to express even a single LN join requires e.g., in (2), over 20 characters and only two LN clauses in Q2 require at least 51 characters. Likewise, for any Q, e.g., just T_WEIGHT defined within would require at least 13 characters, while any CA defined as a sub-query would need several times more. In this sense, any Q1 is substantially more advantageous than equivalent Q.

Case studies and S_P show furthermore that most of SQL DBs has PKN FKs only. Suppose now the creation of D through an S1 sequence, as well as of D1 through the same sequence. Then D1 supports LNF Q1 queries for which (equivalent) Q queries to D always require the LN defined through the above discussed <LN exp>. Furthermore, it appears that LNF queries to D1 suffice for most of practical queries. Such query Q to D selects indeed BAs of some table R and some non-PK BAs of any R1 referenced by a SIR SQL PKN FK of R or even only such BAs, and so on, recursively. Recall that every such attribute in a, directly or indirectly, referenced table is also conceptually one of R that could not be in R actually, given the normalization anomalies that would follow. In every latter case above, every referenced BA will be also NIA of R. Permitting, for any discussed Q, for substantially less procedural LNF Q1, hence also substantially more advantageous in this sense for the clients. The end result at data definition side is that, for every discussed S1 sequence, while SQL DBA could only provide for D requiring LN in most of queries, SIR SQL DBA will provide, without "moving a finger", for D1 permitting for the LNF ones instead,

Furthermore, suppose that every D1.R predefines every eventual LN join through a SIR SQL PKN FK or LND FK and predefines every eventual CA A that is conceptually attribute of D.R, but not actually, as already discussed. Any (SQL) query Q to base tables of D that needs any A, has to define A within Q, also necessarily. Case studies and SP especially, show that most often, any such Q formulates as LNF and CAF Q1 to D1. As already said, the definition of a CA through an SQL value expression requires typically at least a dozen of characters and many more if it requires a sub-query. Whether Q1 is thus CAF only or CAF and LNF or LNF only, Q1 should always be significantly more advantageous for the clients than Q. In contrast, on data definition side, the presence of any CA in D1 base tables imposes on DBA, for every CA, at least the above procedurality of the value expression or of the sub-query. Recall that DBA has to define every such CA in some implicit Create Table. Notice that, as we already pointed out, if D1 DBA defines a CA A in D1.R that could be a VA A in D.R at present, then the procedurality would be the same in both cases. I.e., even if D cannot have VAs, D1 DBA would not spend any more work on A than D DBA would need if D supported VAs.

Finally, DBA of D1 may choose for some R, a non-PK SA F other than every PKN FK in R, as an LND FK referencing some R1, with the LN join and SI in the implicit Create Table R then, we recall, Fig. 3. Unlike NI, SI may filter some of the R1 attributes from becoming IAs with names and values of NIAs in R. Also nay hide some by creating CAs from, perhaps even simple aliases. Every filtered IA becomes unavailable to LNF queries. This may be convenient to some clients. Finally DBA may order IAs differently from NI. This may make happier some clients liking the most famous SQL query; Select * From R. In every case, there is some procedurality overhead for DBA with respect to the creation of any such LND FK as PKN FK instead. E.g., as exercise, one may calculate this overhead for our example of S_P1 with LND FK, while we discussed SIR SQL FKs.

5. Propose a 'simple' SIR SQL Implementation. 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, we overview now and illustrate at Fig. 4. In the nutshell, SIR DBS consists then from the front-end, called *SIR SQL layer* or *SIR-layer* in short, reusing as follows any popular *kernel* SQL DBS, e.g., SQLite3:

- 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. The preprocessing infers every NIA name and LN from kernel's SYS meta-tables. Likewise, for every base table R' referenced through LND FK spec, the preprocessing infers every IA name resulting from R'.#. Easy algorithms for all this, discussed in our previous articles, were prototyped for SQLite kernel.

Besides, SIR-layer parses every explicit Create Table R to Create Table R_ and Create View R defining C-view R. To prevent any name conflicts in C-view, for the latter, every attribute is qualified with its source table name, including R_ for every BA. SIR-layer then forwards both statements as an atomic transaction to the kernel. Fig. 4, at the end of the article, 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.

- For 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 to specify most of time in queries to base tables, LN and CAs, at least other than VAs for some dialects. The procedurality of LN and of CA specs is usually substantial, requiring dozens of characters to type-in and debug, bothering many. SIR SQL gets rid of this annoyance, most of time reducing the queries to the LNF and CAF ones. In particular, the LNF queries to the base table created through Create Table as generally at present only become possible. In fact we expect most of SIR SQL DBAs to define DBs in this way, providing LNF queries to clients as free bonus, without "moving a finger".

For CAF queries, it may suffice to add to Create Table only the value expressions defining the CAs. Recall also that SQL base tables at present simply do not provide for CAs possible with SIR SQL. We expect future SIR SQL DBs used for data analysis, e.g., Big Data DBs, to routinely profit from this capability. 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.

In sum, we have shown that if present SQL DBs had SIR-layers, LNF and CAF queries to base tables would be the standard and their present equivalents a bad dream. Also, as it appeared for SQLite, to provide SIR-layer for a present SQL DBS, should cost no more than a few months effort, i.e., - peanuts by industrial standards. We postulate consequently that DB courses and textbooks take notice of SIR SQL from now on. This, despite the lack of any SIR SQL layer fully implemented as yet. After all, same happened to present relational DBSs.

By the same token, we postulate that popular SQL DBSs upgrade to SIR SQL, "better sooner than later". 7+ million SQL clients worldwide will benefit from. Bear in mind also that this spread out makes SQL the most used DB language, [6], [7], [10]. Recall finally that SQL crowd makes 70% of all developers and provides for estimated 31B US$ market size of SQL DBSs, [8].

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).

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 tables and content. IAs are in italics. IAs content is virtual only. S and P are as in Fig. 1.

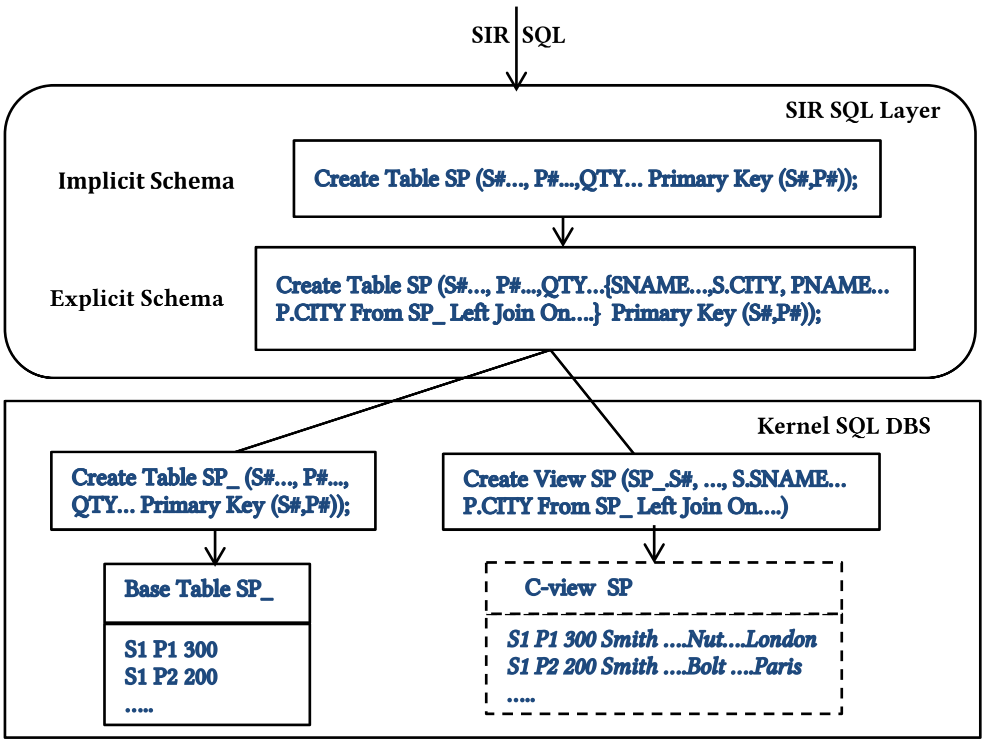

SIR SQL
SIR SQL Layer
Implicit Schema
Create Table SP (S#..., P#...,QTY... Primary Key (S#,P#));
Explicit Schema
Create Table SP (S#..., P#...,QTY...{SNAME...,S.CITY, PNAME... P.CITY From SP_ Left Join On....} Primary Key (S#,P#));
Kernel SQL DBS
Create Table SP_ (S#..., P#..., QTY... Primary Key (S#,P#));
Create View SP (SP_.S#, ..., S.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 within kernel SQL DBS. As for any SQL view, C-view SP content is basically virtual.