Manifesto for SIR SQL

WITOLD LITWIN

University Paris Dauphine, PSL

SIR SQL stands for SQL generalized to Stored and Inherited Relations (SIRs). The benefit is the logical navigation free (LNF) and calculated attribute free (CAF) typical queries to base tables being SIRs, while the same outcome typical queries to base tables at present require LN or CAs specs within. Recall that LN means cumbersome procedural equijoins on foreign and referenced keys, in Codd's original meaning of these terms, while CA specs are also procedural, value expressions with LN or with sub-queries with aggregate functions. A SIR R is any base table defined as presently and referred to implicitly as R_, enlarged with some inherited attributes (IAs). The latter are defined as if they were in a specific view of R_, including LN or CA specs that typical queries to R_ has to contain at present, at least partly. Same output queries to R may address IAs instead, remaining LNF or CAF consequently.

Below, we first illustrate the problem with LN or CA specs in typical SQL queries more in depth. We then describe step-by-step the solution brought by SIR SQL. Afterwards, we outline the design of SIR SQL front-end to an SQL DBS. We claim that a few months of work should suffice for any popular SQL DBS. We point to the proof-of-concept prototype in Python reusing SQLite3. We conclude by claiming that courses and textbooks on relational DBs should take notice of SIR SQL. Also, - that every popular DBS should provide for SIR SQL "better sooner than later". Making more productive apparently at least 7-million present SQL clients. Representing, estimated 31B+ US$ market size.

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

All present SQL DBSs bother the clients, users and developers, with unnecessary procedurality of queries to the base tables. The 1st culprit is the logical navigation (LN) in typical queries addressing base tables with foreign keys and referenced tables. Recall that LN means equijoins on foreign and referenced keys, as Codd originally defined these terms in [1] and result from his sheer idea of a foreign key (FK). The LN should be therefore a part of every base table with FKs scheme, not of queries to such a table. The procedurality that the LN implies, i.e., the necessary length of the SQL join clauses defining it, may easily double the one of the query without. Not surprisingly, clients usually at least dislike the LN. Especially, - when it includes outer joins, [4]. In short, queries to base tables requiring LN at present should 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 was in the base table, the query could address it by name only, i.e., could

9th Int. Conf. on Information & Education Innovations (ICIEI 2024), Verbania Italy. Preprint.

be CAF. Since it cannot be the case at present, SQL clients must define such CAs in the queries. The increase to the query procedurality may be substantial, e.g., may double the query. The sheer complexity of some CAs, those defined by sub-queries especially, also bothers many.

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 also the typical ones of any DBs at present and our examples related to generalize accordingly. We refer to Supplier-Part in short as S-P DB. S, P, SP are stored relations, (SRs), also called base tables. For Codd, SR means that none of its stored attributes, (SAs), can be calculated from the DB 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]. That is, each is the implicit “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.

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 should be in fact also SAs of SP. They are not, since normalization anomalies for SP that would result are unacceptable for the relational model as well known. E.g. consider a query searching for the basic data of smaller supplies, say Q1: “For every supply, select S# and, possibly, SNAME as well as P# with, possibly, PNAME, and QTY, whenever QTY <= 200”. 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. It would suffice if SNAME and PNAME were attributes of SP. However, they are not. Hence, Q1 formulates at present as Q2 below regardless of SQL dialect used or with an equivalent From clause:

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;

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

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.

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 more than two times more procedural than Q3.

2 OUR SOLUTION

Presented in several papers indexed on our home page, [9], by title and usually by abstract and for some in [5], this one consisted of the following steps.

1. Add to the relational model the construct we called *Stored and Inherited Relation* (SIR). As the name hints to, a SIR, say R, has both stored and inherited attributes (SAs and IAs). One declares the SAs as if they were within a base table at present, named R_ by default. This includes any table constraints and options, the definition of the primary key (PK) especially and, perhaps, of any FKs of R. IAs are basically defined and calculated for each SIR R tuple as if they were at present within specific view R. This one should have, (i), the same list of attributes as SIR R, with the scheme of every SA stripped to its name only, prefixed with R_ if the need occurs, and (ii) the same From clause. The latter should be furthermore such that (iii) for every stored tuple that one could insert into the table formed by the SAs, given any table constraints, there should be a view tuple with the same values for every IA named upon an SA and vice versa. We qualified every such view of *C-view R*, C standing for *canonical*. For every tuple of SIR R, the value of every IA is then the one of the same name IA in C-view R. The difference between SIR R and C-view R is thus physical only: every SA in SIR R is the same name and values IA in C-view R. In particular, we consider that the outcome of any query involving SIR R is the one of the same query involving C-view R instead at present.

2. For practical use, obviously one should define a SIR through extensions to Create Table of some existing SQL dialect and alter SIRs through extensions to Alter Table of the dialect. Call *kernel* the dialect chosen. Call *SIR SQL* (dialect) any kernel (dialect) with these statements extended. Accordingly, consider that any SIR SQL Create Table provides for any attributes, table constraints and options definable in kernel's Create Table and for any attributes definable in kernel's Create View. For every SIR R, call accordingly *base of R* and name it by default R_, the projection of R on all and only attributes that could be in Create Table R_ of the kernel. Basically, R_ is all and only SAs of R. For some kernels, R_ may also include *virtual* (dynamic, computed, generated….) *attributes* (VAs). Recall, , e.g. from our papers on SIRs, that every VA is in fact an IA inherited from R_ through arithmetic value expression with, perhaps, scalar functions over SAs or other VAs of R_, provided within the VA scheme at present, and through an implicit 'From R_' clause. Whether R_ has VAs or not, suppose R_ name usable by queries, the update queries in particular, as discussed in Step 7 and C-view R as we will show soon.

The attributes in SIR SQL Create Table R defined as if they were in a Create View R of the kernel are all the IAs of R other than VAs, if there is any VA. From clause of Create View R should follow the entire attribute list of Create Table R that can mix SAs, VAs and (other) IAs of R. The clause should precede in contrast any eventual R_ table constraints or options. From clause should in particular make view R it defines through (i) extended to any VAs if there are any and through (ii), should make it a C-view, i.e., should make it conform also to (iii). We say that IAs other than perhaps VAs in SIR R and all their values *enlarge* (every tuple of) R_.

Finally, operationally, we put any consecutives IAs among SAs and all consecutive IAs followed by From clause, in < { } brackets, replacing the usual SQL ',' separators. The convention proved facilitating the implementation.

For any SIR R, we call *Inheritance Expression* (IE) any scheme defining IAs of R other than eventual VAs. An *explicit* IE defines every IA or uses SQL '*' for some and defines the From clause. A *valid* From clause defines, through (i) and (ii), C-view R or is *invalid*. An IE can alternatively be *implicit*. It then omits some or even all IAs or parts of, or even entire From clause. An implicit IE may be *empty*, i.e., without any IA and From clause. Any implicit IE is pre-processed into the explicit one for any further processing.

The obvious advantage of any implicit IE, whenever such an IE is possible, is to be less procedural than the explicit one. Besides, we have shown that for every SIR R, even the explicit IE can be less procedural to define and maintain than Create View R for C-view R. Regardless of why a C-view R could be wished, to define and maintain SIR R should be always less procedural than Create Table R_ and Create View R. The reason is that for every SA of SIR R, C-view R must redefine it as an IA, as above outlined, unlike the IE. In present terms, every SIR R is thus a *view saver* for C-view R. Like every VA is a view saver for the equivalent view at present.

E.g., consider the following SP scheme instead of the original one:

(1) SP (S#, P#, QTY {SNAME, STATUS, S.CITY, PNAME, COLOR, WEIGHT, P.CITY From SP_ Left Join S On SP.S#=S.S# Left Join P On SP.P#=P.P#};

Here, (S#,P#,QTY) is the base of SP, i.e. SP_. It is the original S-P.SP renamed by default. The brackets delimit the IE, in only one part here. View SP it would lead to through (i) and (ii) could be C-view SP within S-P, if view SP was added to. Renaming of the original SP, by default to SP_, would be accordingly necessary. No SQL dialect allows for two tables with the same (proper) name in a DB. We leave as exercise the Create View SP for C-view SP, given IE in (1). Same, - for the calculus of how much more procedural that one would be than the IE.

From clause in (1) is a valid one, i.e. view SP formed according to (i) - (ii) is C-view SP. Indeed, as (iii) requires, for every tuple one could possibly insert to SP_, it provides for view SP tuple with the same S#, P# and QTY values. E.g. for SP_ tuple (S1, P1, 300), there is one and only one view SP tuple (S1, P1, 300, Smith, 20, London, Nut, Red, 12, London). It is then also the logically the same SIR SP tuple, except that, physically, S#,P#,QTY are SAs and not IAs of view SP. Likewise, for the tuple (S6, P1, 200), possible for SP_ as no table constraint prohibits it, with respect to SP content at Fig. 1, SIR SP, as well as C-view SP, would each have the tuple (S6, P1, 200, null…null), with every IA being null. But if one replaced, e.g., the left Join S with Inner Join S, then the latter tuple would not be in view SP anymore. Hence, view SP would not be C-view SP neither and the envisaged From clause would be invalid. It would be valid iff one declares SP.S# as FK referencing S.S#. The referential integrity (table) constraint would prohibit the latter tuple. Notice, that the original From clause from (1) would remain valid.

Let us call S-P1 the S-P DB with SIR SP (1). Observe that for S-P1, LNF Q1 becomes possible. The obvious reason is that it may address in SP the IAs defined through the LN. For S-P, Q2 needs in contrast such LN, as it must address the sources of the IAs. Moreover, S-P1 clients could similarly issue any LNF query selecting any SP tuple so and so…, while also addressing any attributes of S or P inherited by SIR SP. Unlike any clients of S-P could do. By extrapolation, the example clearly hints on the practical potential of SIRs.

Observe finally that no IA in (1) introduces normalization anomalies, i.e., storage, insert, update or delete anomaly. Indeed 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, inserts or updates if a source value varies, e.g., S1 changes the name to John. Likewise, no need for any additional deletes, if the referential integrity implies the cascading and any source tuple gets deleted, e.g., S.S1. These anomalies would in contrast necessarily occur, contradicting thus the relational model, if every IA of S-P1.SP was, trivially, an SA instead. They would lead in particular, to the risk of inconsistencies in the DB, if any of the redundant manipulations went awry. The discussed absence of normalization anomalies provided by IAs of S_P1.SP, generalizes obviously to any SIRs hence, potentially, to DBs with billions of tuples in practice.

3. Enlarge the present SQL meaning of FK concept so to include our perception of Codd's original idea, [1], [3]. We called the result *Primary Key Named FK* (PKN FK), [5]. One may declare a PKN FK as in SQL at present, including the details of the referential integrity. It is then a *declared* PKN FK. Otherwise, a non-PK atomic SA R.A such that (a) the DB has only one PK named A and (b) A is not within a declared PKN FK in R, is a *natural* PKN FK in R. A natural PKN FK does not imply any referential integrity. In SIR SQL, for a PKN FK, the referential integrity is thus possible, but not mandatory. Unlike it is for any SQL dialects at present. Recall that Codd considered the referential integrity as we do, [1,2,3].

Besides, we consider of course that one may declare for SIRs FKs that are not PKN ones. These are dealt with in SIR SQL as SQL FKs at present, unlike any declared PKN FKs, as it will appear. Finally, recall that, in any case, every FK in a SIR has to be an SA, as at present.

E.g., whether in S-P scheme or in S-P1 one, SP.S# and SP.P# are natural FKs, as there is no FK clause in any SP scheme. They do not imply any referential integrity. Alternatively one could declare them FKs as at present. This would imply the referential integrity with respect to S and P. Finally, as required for any FK, whether natural or declared within S-P1.SP scheme, both FKs are SAs.

4. Consider any PKN FK Fi ; i = 1,..,k in a base table R scheme defined as at present in any SQL dialect, as the shorthand for all the non-PK attributes of the table, say R'i, it references as the "logical pointer", in already recalled Codd's original vocabulary. It means that R scheme does not only includes the name and the selected by the client values of the "foreign" PK, but also inherits the name and, for every Fi value all the associated values of every non-PK attribute of R'i. Namely, for every R.Fi value f, the tuple with f inherits also every non-PK value in R'i tuple with the PK = f, if R'i has such a tuple. These are the values of the attributes we spoke about for S-P.SP, i.e., that should be in S-P.SP if the present relational model allowed for. If R'i has no such tuple for a value of a natural PKN FK, then the R-tuple inherits a null for every IA named upon a non-PK attribute of R'j.

For every Fi in R, we called such IAs, *natural inheritance* (NI) *in* R *from* R' *through* Fi. Accordingly, the term NI in R designates all the NIs through Fi in R. Likewise, we qualify every IA in NI of *naturally* inherited (NIA). We called also *implicit* the SIR R scheme not yet completed with the NI. In particular, every base table scheme with PKN FKs, defined as at present is thus an implicit scheme for SIR SQL. We then qualified of *explicit* the scheme with the NI added to, considered of course as the actual one of R.

5. More formally, suppose that by analogy to SQL '*', for some Create Table R with some PKN FKs F1…Fk and, perhaps, with some table constraints and options, one denotes as R_.* all the attributes schemes defined as they could be at present in some SQL dialect. Next, let us continue to denote for every Fi, the base

table referenced by Fi as R'i and, as R'i.#, all the non-primary key attributes of R'i. Then, consider that the implicit Create Table R has the usual form:

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

The explicit one would be as follows then, with NI 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>];

Observe that From clause above defines in fact the LN from R towards every referenced base table. Also, the NIAs are named upon all the referenced attributes. Unlike presently thus, no typical query to R, i.e., searching for every tuple of R so and so…, and addressing one or more R'I as well, requires any LN. It may indeed address R with NIAs instead. Every such query becomes LNF in consequence. An obvious practical benefit from SIRs with NI thus.

Ex. For S-P, we may consider the following Create Table SP as the present one, with '…' meaning the data type and Primary (S#, P#) being the only table constraint:

Create Table SP(S#..., P#..., QTY… Primary Key (S#, P#));

For SIR SQL, since S# and P# are (natural) PKN FKs, this would be the implicit Create Table with empty IE for the explicit one as follows:

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

The statement creates clearly the already discussed scheme (1) of S-P1.SP. Accordingly, Q1 becomes possible. More generally, any query selecting every SP tuple so and so, while addressing also any attributes of S or P, may be an LNF one. Unlike their present equivalents, we recall. Observe also that the implicit Create Table SP is the one the S-P DBA would use for S-P.SP at present. In other words, for SIR SQL, present S-P scheme defines in fact S-P1. Altogether, in SIR SQL, the LNF queries, likely making every (presumed) present S-P client happier and surely more productive, become standard, without any additional data definition work.

6. Furthermore, consider now that some SIR R has IAs not in NI, e.g., some CAs that are not VAs for a kernel providing for the latter. Suppose accordingly that the IE may have From clause referring to R_ only or defining also (explicitly) LN to tables other than those of NI. Next, consider that R.* stands for all the attribute schemes in Create Table R and that [<explicit From>] designates the just discussed optional From clause. Then R should have Create Table R in the form:

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

This Create Table is the explicit one if R does not have any PKN FK. Otherwise, it is an implicit one, with implicit NI and IE. The latter would not be however empty anymore, although it might have no From clause. As before, SIR SQL Create Table processing would pre-process the scheme to the explicit one, with the (explicit) NI thus, as follows this time:

Create Table R (R.*, {R'1.#,…,R'k.# [<explicit 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 idea is of course to let the DBA to issue Create Table R as non-procedural as reasonably possible. E.g., for SP with T-WEIGHT one could accordingly state:

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

Observe that for WEIGHT-T, we have declared only the value expression. Within SQL specs for SAs and for IAs (in views), it is not possible to have any less procedural Create Table for SP. Since SP has PKN FKs, the scheme would be the implicit one for:

Create Table SP (S#...,P#...,QTY {QTY*WEIGHT As T-WEIGHT From SP_ Left Join S On SP.S#=S.S# Left Join P On SP.P#=P.P#} Primary Key (S#, P#));

Any further Create Table SP processing would concern the latter. Notice that the explicit IE appears more than three times more procedural than the implicit one. C-view would be even more procedural, 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 in 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 in S-P, and CAF for T-WEIGHT.

7. 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 developer's effort, 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. That one becomes the same name DB in the kernel.

- SIR-layer forwards to the kernel every Create Table R submitted without PKN FKs and without any IA other than, perhaps, those declared as if they were VAs for the kernel. Any Create Table R with PKN FKs is for SIR-layer an implicit scheme, pre-processed 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.

- 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 R scheme in kernel's meta-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 SIR papers 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 the SIR SQL data manipulation statements, SIR-layer simply forwards any submitted query to the kernel. For any update queries to SIRs, safe policy for every kernel and every SIR R is then to address R_. E.g., Insert To SP_..., Update SP_... and Delete From SP_... for S-P1.SP. Update queries addressing SIR R directly instead, 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 CA updates. Again, see the SIR papers for more.

8. Finally, validate the canonical implementation through the proof-of-concept prototype, e.g. with SIR-layer in Python and SQLite3 as the kernel. The actual prototype available at present provides also for a 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 with 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 their inception, i.e., five decades for early birds, every RDBS requires SQL clients with typical queries to base tables, to specify the LN and the CAs 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 and bothers many. Our proposal gets rid of this annoyance by substantially simplifying such queries to the LNF and CAF ones. The LNF queries become possible for the present base table schemes. For CAF queries, it may suffice to only add to Create Table the value expressions defining the CAs. Finally, the prototype SIR DBS proved simple.

Given all this, courses and textbooks on relational DBs should take notice of SIR SQL. Also, popular DBSs should provide SIR SQL, "better sooner than later". Making more productive, apparently, 7+ million SQL clients worldwide, [6], [7], including 40% of all developers and providing for, estimated, 31B US$ market 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 .