ACM SIGMOD RECORD, Vol. 19, No. 1, pp. 18-22 (1990)

A Note on the Translation of SQL to Tuple Calculus

martin gogolla
TU Braunschweig, Informatik, Abt. Datenbanken
Postfach 3329, D-3300 Braunschweig
e-mail: gogolla at infbs.uucp

Abstract

This note presents a translation of a subset of the relational query language SQL into the well known tuple calculus. Roughly speaking, tuple calculus corresponds to first order predicate calculus. The SQL subset is relationally complete and represents a "relational core" of the language. Nevertheless, our translation is simple and elegant. Therefore it is especially well suited as a beginners course into the principles of a formal definition of SQL.

Keywords: Database theory, relational query language, tuple calculus, SQL, formal semantics.

1 Introduction

Several authors have studied the formal semantics of the relational query language SQL. In [CG85] SQL is translated into relational algebra, [Bül87] takes special care of aggregate functions, and in [PBGG89] a translation of SQL into tuple calculus is proposed. A comprehensive description and constructive criticism of the SQL standard can be found in [Dat87]. Nevertheless, up to now no elegant proposal has appeared which especially allows to prove properties of the SQL language easily. This is possible within our approach.

But before discussing the details of the translation we point out the need of formal semantics by means of an example. Consider the following simple cooking book database (keys are underlined).

```
RECIPE ( RName : string;
         Ingred : string;
         UsedQuantity : real )
STORE (  Ingred : string;
         StoreQuantity : real )
ORIGIN ( RName : string;
         Country : string )
```

Let us now look at a typical SQL query looking for some recipes using garlic.

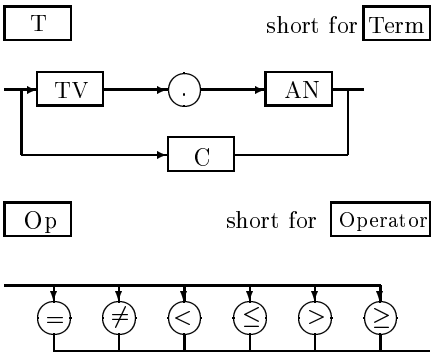
```
SELECT r.RName
FROM   RECIPE r
WHERE  r.Ingred='garlic' AND
r.UsedQuantity >
ANY (  SELECT r'.UsedQuantity
       FROM   RECIPE r',
              ORIGIN o
       WHERE  r'.RName=o.RName AND
              o.Country='italy' AND
              r'.Ingred='garlic' )
```

The subquery computes all garlic quantities in Italian recipes. Therefore, the SQL keywords suggest that the query performs the following task: "Select recipes which use more garlic than any Italian recipe."

In usual colloquial English it is understood that these are recipes using more garlic than the quantity of garlic used in every Italian recipe, i.e., more than the maximal quantity. But a look at the formal semantics shows that the SQL query does not do this job:

$\{ \text{res} : (\text{Res}_1) \mid$

∃ r: RECIPE
res.Res$_1$ =r.RName' ∧
r.Ingred='garlic' ∧
∃ r':RECIPE
(∃ o:ORIGIN
r'.RName'=o.RName' ∧
o.Country='italy' ∧
r'.Ingred='garlic')
∧ r.UsedQuantity>r'.UsedQuantity }

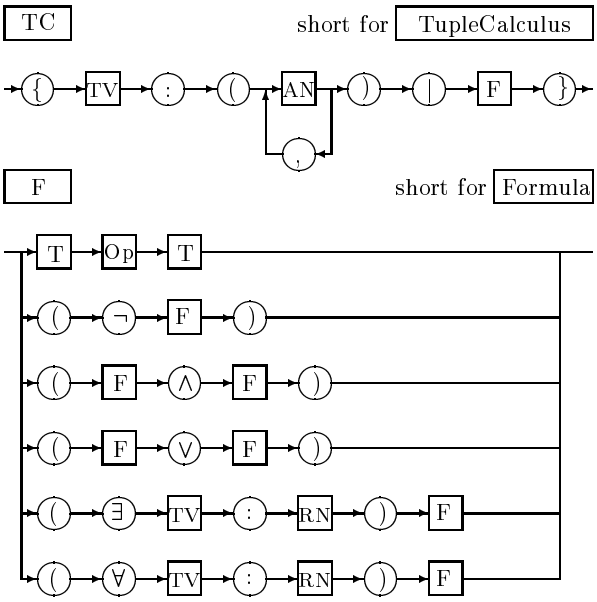


The formula now points out that the SQL query finds recipes which use more garlic than at least one Italian recipe or in other words recipes which use more garlic than the minimal quantity of garlic in Italian recipes. In fact, a similar query to the above one together with the "wrong" description in colloquial English can be found in IBM manuals for SQL [Dat87]. Thus, it is desirable to formally define the semantics of SQL as a basis for implementations. On the other hand, a formal definition can also be employed to prove the equivalence of queries thus making formally well founded optimization techniques even more attractive. Furthermore, one can point out properties of the SQL language.

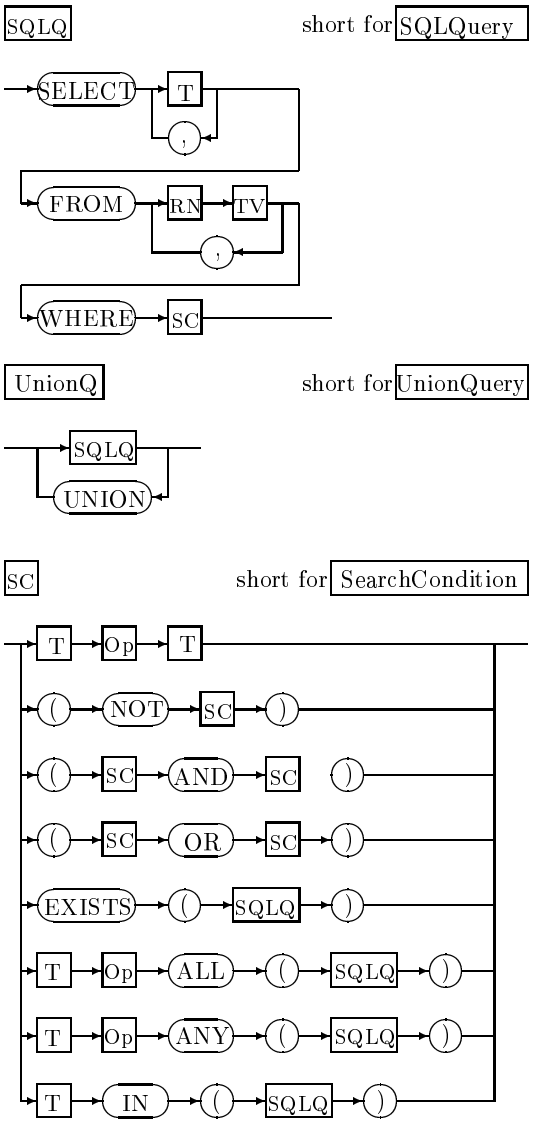
2 Tuple calculus and SQL

We describe the syntax of our versions of tuple calculus and SQL with syntax diagrams. The semantics of tuple calculus can be found in [Mai83]; the semantics of SQL is subject to this note and studied in Chapter 3. Both, tuple calculus and SQL involve terms T (consisting of tuple variables TV and attribute names AN or constants C) and comparison operators Op.

As mentioned before, our version of tuple calculus corresponds to first order predicate calculus. The qualifying Formula F of the tuple calculus query TC has exactly one free variable, namely the specified tuple variable TV whose structure is determined by various attribute names AN. RN stands for the name of a relation.



Our SQL queries SQLQ take into account that subqueries can be built via the operators EXISTS, ALL, ANY and IN specified in the search condition SC. The operator UNION is allowed only on the top level. UnionQ generates these queries.



The nonterminals TV, AN, RN, and C are left unspecified. Additionally, parenthesis may be omitted in accordance with the usual rules and the tuple calculus also employs implication $\Rightarrow$ with the usual semantics.

3 The translation

There are several general requirements the SQL queries have to fulfill (τ, τ_1 and τ_2 stand for terms and φ, φ_1 and φ_2 for formulas):

- We assume that the given SQL query "SELECT τ_1, ..., τ_n FROM R_1 s_1, ..., R_m s_mWHERE φ" and all subqueries are qualified completely via explicitly declared tuple variables: If there are references to an attribute A_j in a result term τ_i or in the formula φ, they are of the form $s_i.A_j$, where s_i is declared in the FROM clause (or in the case of a subquery perhaps in the FROM part of the outer query) and A_j is an attribute of the corresponding relation schema R_i. The names of the tuple variables must be unique ($i \neq j \Rightarrow s_i \neq s_j$) and must be declared only once. The result terms as well as the formula φ use only declared tuple variables.

- Additionally, the result terms τ_i may be constants c_i. The result variable of the tuple calculus expression must be different from all declared variables in the query.

- If terms are compared with an operator like "$\tau_1 \omega \tau_2$" or "τ_1 ω ALL (SELECT τ_2 FROM ... WHERE ...)", then τ_1 and τ_2 have the same data type. For a UNION expression we assume τ_i and τ_i' have the same data type for i$\in$1..n.

The translation is now done by defining a function **sql2tc**, which gives for every SQL expression the corresponding tuple calculus one.

sql2tc$[\![$ SELECT τ_1, ..., τ_n
FROM R_1 s_1, ..., R_m s_m
WHERE φ $]\!]$:=

{ r : (Res_1, ..., Res_n) |
($\exists$ s_1:R_1, ..., s_m:R_m)
(r.Res_1 = τ_1 $\wedge$ ... $\wedge$ r.Res_n = τ_n
$\wedge$ **sql2tc**$[\![$ φ $]\!]$) }

sql2tc$[\![$ (NOT φ) $]\!]$:=
($\neg$ **sql2tc**$[\![$ φ $]\!]$)

sql2tc$[\![$ (φ_1 AND φ_2) $]\!]$:=
(**sql2tc**$[\![$ φ_1 $]\!]$ $\wedge$ **sql2tc**$[\![$ φ_2 $]\!]$)

sql2tc$[\![$ (φ_1 OR φ_2) $]\!]$:=
(**sql2tc**$[\![$ φ_1 $]\!]$ $\vee$ **sql2tc**$[\![$ φ_2 $]\!]$)

sql2tc$[\![$ $\tau_1 \omega \tau_2$ $]\!]$:= $\tau_1 \omega \tau_2$

sql2tc⟦ τ ω ALL
(SELECT s_i.A
FROM R_1 s_1, ..., R_i s_i, ..., R_m s_m
WHERE φ) ⟧ :=

($\forall$ s_i:R_i)
(((∃ s_1:R_1, ..., s_{i-1}:R_{i-1},
s_{i+1}:R_{i+1}, ..., s_m:R_m)
sql2tc⟦ φ ⟧) ⇒ τ ω s_i.A)

sql2tc⟦ τ ω ANY
(SELECT s_i.A
FROM R_1 s_1, ..., R_i s_i, ..., R_m s_m
WHERE φ) ⟧ :=

($\exists$ s_i:R_i)
(((∃ s_1:R_1, ..., s_{i-1}:R_{i-1},
s_{i+1}:R_{i+1}, ..., s_m:R_m)
sql2tc⟦ φ ⟧) ∧ τ ω s_i.A)

sql2tc⟦ τ IN
(SELECT s_i.A
FROM R_1 s_1, ..., R_i s_i, ..., R_m s_m
WHERE φ) ⟧ :=

($\exists$ s_i:R_i)
(((∃ s_1:R_1, ..., s_{i-1}:R_{i-1},
s_{i+1}:R_{i+1}, ..., s_m:R_m)
sql2tc⟦ φ ⟧) ∧ τ = s_i.A)

sql2tc⟦ EXISTS
(SELECT r_1.A_1, ..., r_n.A_n
FROM R_1 s_1, ..., R_m s_m
WHERE φ) ⟧ :=

($\exists$ s_1:R_1, ..., s_m:R_m) **sql2tc**⟦ φ ⟧

sql2tc⟦ SELECT τ_1, ..., τ_n
FROM R_1 s_1, ..., R_m s_m
WHERE φ
UNION
SELECT τ_1', ..., τ_n'
FROM R_1' s_1', ..., R_k' s_k'
WHERE φ' ⟧ :=

{ r : (Res_1, ..., Res_n) |
($\exists$ s_1:R_1, ..., s_m:R_m)
(r.Res_1 = τ_1 ∧ ... ∧ r.Res_n = τ_n
∧ **sql2tc**⟦ φ ⟧)
∨
($\exists$ s_1':R_1', ..., s_k':R_k')
(r.Res_1 = τ_1' ∧ ... ∧ r.Res_n = τ_n'
∧ **sql2tc**⟦ φ' ⟧) }

The rule for UNION can be generalized to the case with more than 2 operands. All used tuple variables are of the form r:R; therefore we only employ safe expressions [Mai83].

4 Some Laws

On the basis of a formally defined language it is now possible to state properties of the language. Here are some laws which can be proved formally.

Theorem:

(I) τ_1 IN (SELECT τ_2
FROM ρ
WHERE φ) ⇔
τ_1 = ANY (SELECT τ_2
FROM ρ
WHERE φ)

(II) NOT (τ_1 ω ALL (SELECT τ_2
FROM ρ
WHERE φ)) ⇔
τ_1 $\overline{\omega}$ ANY (SELECT τ_2
FROM ρ
WHERE φ)

(III) τ_1 ω ANY (SELECT τ_2
FROM ρ
WHERE φ) ⇔
EXISTS (SELECT $\tau(\rho)$
FROM ρ
WHERE (φ) AND $\tau_1 \omega \tau_2$)

(IV) τ_1 ω ALL (SELECT τ_2
FROM ρ
WHERE φ) ⇔
NOT EXISTS (SELECT $\tau(\rho)$

FROM ρ
WHERE (φ) AND $\tau_1 \overline{\omega} \tau_2$)

In law (I) $\overline{\omega}$ is the negation of ω, e.g. $\overline{\leq}$:=>. In law (III) and (IV) $\tau(\rho)$ refers to all attributes occurring in ρ. We assume $\rho \equiv \mathrm{R}_1\ \mathrm{s}_1, \ldots, \mathrm{R}_m\ \mathrm{s}_m$.

Proof: We assume $\rho_i \equiv \mathrm{s}_1{:}\mathrm{R}_1, \ldots, \mathrm{s}_{i-1}{:}\mathrm{R}_{i-1}, \mathrm{s}_{i+1}{:}\mathrm{R}_{i+1}, \ldots, \mathrm{s}_m{:}\mathrm{R}_m$.

(I) **sql2tc**$[\![$ τ_1 IN (SELECT τ_2
FROM ρ
WHERE φ) $]\!] \Leftrightarrow$
($\exists$ $\mathrm{s}_i{:}\mathrm{R}_i$) ((($\exists \rho_i$)
sql2tc$[\![\varphi]\!]$) $\wedge$ $\tau_1 = \tau_2$) $\Leftrightarrow$
sql2tc$[\![$ τ_1 = ANY (SELECT τ_2
FROM ρ
WHERE φ) $]\!]$

(II) **sql2tc**$[\![$ NOT (τ_1 ω ALL (SELECT τ_2
FROM ρ
WHERE φ)) $]\!]$
$\Leftrightarrow$
$\neg$ (($\forall$ $\mathrm{s}_i{:}\mathrm{R}_i$) (($\exists \rho_i$)
sql2tc$[\![\varphi]\!]$) $\Rightarrow$ $\tau_1 \omega \tau_2$) $\Leftrightarrow$
$\neg$ (($\forall$ $\mathrm{s}_i{:}\mathrm{R}_i$) (($\forall \rho_i$)
$\neg$ **sql2tc**$[\![\varphi]\!]$) $\vee$ $\tau_1 \omega \tau_2$) $\Leftrightarrow$
$\neg$ (($\forall$ $\mathrm{s}_i{:}\mathrm{R}_i$) ($\forall \rho_i$)
($\neg$ **sql2tc**$[\![\varphi]\!]$ $\vee$ $\tau_1 \omega \tau_2$)) $\Leftrightarrow$
($\exists$ $\mathrm{s}_i{:}\mathrm{R}_i$) ($\exists \rho_i$)
(**sql2tc**$[\![\varphi]\!]$ $\wedge$ $\tau_1 \overline{\omega} \tau_2$) $\Leftrightarrow$
sql2tc$[\![$ τ_1 $\overline{\omega}$ ANY (SELECT τ_2
FROM ρ
WHERE φ) $]\!]$

(III) and (IV) can be proved analogously. q.e.d.

After looking at the equivalent, but longer translation from SQL to tuple calculus described in [PBGG89], we found it worthwhile to publish our translation. Indeed, it has already been presented in a course on databases at the Technical University Braunschweig in 1987 [Gog87]. The SQL subset considered here does not take into account aggregation (e.g., COUNT, AVG, etc.) nor grouping features (e.g. GROUP BY, HAVING, etc.). These SQL language features will be discussed in a forthcoming paper.

References

[CG85] S. Ceri and G. Gottlob. Translating SQL Into Relational Algebra: Optimization, Semantics, and Equivalence of SQL Queries. IEEE Transactions on Software Engineering, Vol. 11, No. 4, pp. 324–345, 1985.

[Dat87] C.J. Date. A Guide to the SQL Standard. Addison-Wesley, Reading (MA), 1987.

[Gog87] M. Gogolla. Begleitmaterial zur Vorlesung Datenbanksysteme. Technische Universität Braunschweig, Wintersemester 87/88, 1987.

[Mai83] D. Maier. The Theory of Databases. Computer Science Press, Rockville (MD), 1983.

[PBGG89] J. Paredaens, P. de Bra, M. Gyssens, and D. van Gucht. The Structure of the Relational Database Model. Springer, Berlin, EATCS Monographs on Theoretical Computer Science, Vol. 17, 1989.

[Bül87] G. von Bültzingsloewen. Translating and Optimizing SQL Queries Having Aggregates. In: Proc. 13th Int. Conf. on Very Large Data Bases (VLDB), P.M. Stocker, W. Kent (Eds.), VLDB Endowment Press, Saratoga (CA), pp. 235–243, 1987.