1

QUERY LANGUAGES WITH GENERALIZED QUANTIFIERS

Antonio Badia*, Dirk Van Gucht*, and Marc Gyssens**

**Department of Computer Science, Indiana University, Bloomington, Indiana 47405-4101, USA*
***Department WNI, University of Limburg (LUC), B-3590 Diepenbeek, Belgium*

ABSTRACT

Various existing query languages allow queries with embedded sub-queries as well as sub-query comparison statements. It is often argued that these features enhance the declarativeness of the query language. We establish a link between between the phenomenon of sub-query syntax in query languages and the theory of *generalized quantifiers* as it was introduced by Barwise and Cooper in linguistics. Having established this link, we formulate and defend a thesis which states that most natural queries can be formulated as a conjunction of first-order predicate statements and set-predicate statements over sub-queries. We also introduce the language $\mathcal{QLGQ}$ which is designed in accordance with this thesis. We conclude by showing how various existing query languages have implicitly adopted generalized quantifiers and discuss some related features of these languages.

1 INTRODUCTION

During the last two decades, several relational database query languages have been introduced that model the relational calculus or algebra [3, 4, 5], or extensions thereof. Typical about most of these languages is that, apart from the constructs necessary to model the features in the theoretical language on which they are based, they also contain constructs that do not necessarily enhance their expressive power, but make them more user-friendly. One such construct found in several of these query languages is the ability to express queries with embedded sub-queries as well as sub-query comparison statements.

For example, consider unary relations `company`(`cname`) and `product`(`pname`) and binary relations `type`(`pname`, `ptype`) and `sells`(`cname`, `pname`) the meaning of which is obvious. Now consider the query *find all companies that sell products of type* `'knife'` over the database consisting of those four relations. One way of expressing this simple query in SQL (e.g., [6]) is shown in Figure 1.

```
SELECT C.cname
FROM   company C, product P, sells S, type T
WHERE  C.cname = S.cname AND P.pname = S.pname
AND    S.pname = T.pname AND T.ptype = 'knife'
```

Figure 1 The query *find all companies that sell products of type* `'knife'` expressed in SQL using join syntax.

The solution in Figure 1 simulates the way in which this query would be formulated in the relational algebra, using join, selection, and projection. An alternative way of expressing the above query is shown in Figure 2.

```
SELECT C.cname
FROM   company C
WHERE  EXISTS
      (SELECT P.pname
       FROM   product P, type T
       WHERE  P.pname = T.pname AND T.ptype = 'knife'
       AND    EXISTS
             (SELECT *
              FROM   sells S
              WHERE  S.pname = P.pname AND S.cname = C.cname))
```

Figure 2 The query *find all companies that sell products of type* `'knife'` expressed in SQL using sub-query syntax.

The solution in Figure 2 uses sub-query syntax. Although the solution in Figure 1 is more concise than the solution in Figure 2, many non-specialist users will find the latter solution easier to understand, because it better highlights the quantification in the query as it is formulated in natural language. Preference of the sub-query syntax over the join syntax will be even more outspoken when more complex queries are considered.

Apart from SQL, several other relational query languages support sub-query syntax. An example is the relational calculus with set operators (denoted RC/S) of Özsoyoğlu and Wang [10]. Another good example of sub-query syntax can be found in deductive database query languages such as LDL [9] and CORAL [12].

The principal motivation for the inclusion of sub-query syntax is the enhanced *declarativeness* of the resulting query language, especially when it involves the formulation of complex queries. In the case of SQL and RC/S, the inclusion of sub-query syntax does *not* extend the *expressiveness* of the language, however.

The problem of quantification in natural language has also been studied in linguistics. In particular, Barwise and Cooper [2] introduced the previously defined concept of *generalized quantifiers* in natural language formalization (see also [13]).

In this paper, we establish a link between the phenomenon of sub-query syntax in query languages and the theory of generalized quantifiers in linguistics. Having established this link, we put forth the *conjunctive formulation thesis*. This thesis states that the natural way to formulate a real-world query is as a conjunctions of simple, first-order predicates and set predicates over sub-queries. We show that, with the help of generalized quantifiers, query languages can be designed wherein queries are formulated in accordance with this thesis.

The paper is organized as follows. In Section 2 we discuss the work of Barwise and Cooper on generalized quantifiers. In Section 3 we propose the language $\mathcal{QLGQ}$ (Query Language with Generalized Quantifiers), which is a variation of $\mathcal{LGQ}$ (Language with Generalized Quantifiers) introduced by Barwise and Cooper. In Section 4, we state and argue the conjunctive formulation thesis, basing ourselves in part on some expressiveness results about $\mathcal{QLGQ}$. In Section 5, we discuss the extent to which several existing query languages have (implicitly) adopted generalized quantifiers. We also discuss some related features of these languages. We end with a conclusions section (Section 6).

2 GENERALIZED QUANTIFIERS

2.1 Generalized quantifiers in linguistics

Historically, *generalized quantifiers* were introduced by mathematical logicians in the late 1950's who wanted to study logical properties of quantifiers that could not be expressed within first-order logic. [8] The body of literature on generalized quantifiers in mathematical logic—and, more recently, in finite model theory—is extensive. Only in the early 1980's, generalized quantifiers became also prominent in linguistics. Building on techniques proposed by Montague [7], Barwise and Cooper [2] published a seminal paper concerning the adoption of the logical concept of generalized quantifiers in natural language formalization. The best way to illustrate their approach is through an example.

Consider the unary predicates `company` and `product` and the binary predicates `type` and `sells`, which have the same meaning as the corresponding unary and binary relations in the Introduction. Next, consider the following classes of natural language sentences:

- *Class 1*:
  1. Some products are sold.
  2. All products of type `'knife'` are sold.
  3. No products of type `'fork'` are sold.
  4. At least two products are sold.
  5. Most products are sold.
- *Class 2*:
  1. Some product of type `'knife'` is sold by all companies.
  2. Exactly one product is sold by no company.
  3. Some company sells an even number of products.

The grammatical structure of each sentence in Class 1 is of the form

$$\langle\text{noun phrase}\rangle\langle\text{verb phrase}\rangle.$$

For example, the grammatical structure of the first sentence in Class 1 is

$$\underbrace{\text{Some products}}_{\text{noun phrase}}\ \underbrace{\text{are sold}}_{\text{verb phrase}},$$

and that of the second sentence is

$$\underbrace{\text{All products of type } \texttt{'knife'}}_{\text{noun phrase}}\ \underbrace{\text{are sold}}_{\text{verb phrase}}.$$

The grammatical structure of the sentences in Class 2 is also of the same form, but with verb phrases that are more complex. Unfortunately, this grammatical structure is often lost when trying to formalize these natural-language sentences as first-order-logic sentences:

- *Class 1*:
  1. $(\exists \mathtt{p})(\mathtt{product(p)} \land (\exists \mathtt{c})(\mathtt{company(c)} \land \mathtt{sells(c,p)}))$.
  2. $(\forall \mathtt{p})(\mathtt{product(p)} \land \mathtt{type(p,\text{'knife'})} \Rightarrow$
     $(\exists \mathtt{c})(\mathtt{company(c)} \land \mathtt{sells(c,p)}))$.
  3. $\neg(\exists \mathtt{p})(\mathtt{product(p)} \land \mathtt{type(p,\text{'fork'})} \land$
     $(\exists \mathtt{c})(\mathtt{company(c)} \land \mathtt{sells(c,p)}))$.
  4. $(\exists \mathtt{p}_1)(\exists \mathtt{p}_2)(\mathtt{product}(\mathtt{p}_1) \land \mathtt{product}(\mathtt{p}_2) \land \mathtt{p}_1 \neq \mathtt{p}_2 \land$
     $(\exists \mathtt{c}_1)(\exists \mathtt{c}_2)(\mathtt{company}(\mathtt{c}_1) \land \mathtt{company}(\mathtt{c}_2) \land$
     $\mathtt{sells}(\mathtt{c}_1,\mathtt{p}_1) \land \mathtt{sells}(\mathtt{c}_2,\mathtt{p}_2)))$.
  5. Not expressible in first-order logic.
- *Class 2*:
  1. $(\exists \mathtt{p})(\mathtt{product(p)} \land (\forall \mathtt{c})(\mathtt{company(c)} \Rightarrow \mathtt{sells(c,p)}))$.
  2. $(\exists \mathtt{p})(\mathtt{product(p)} \land \neg(\exists \mathtt{c})(\mathtt{company(c)} \land \mathtt{sells(c,p)}) \land$
     $(\forall \mathtt{p}')(\mathtt{product}(\mathtt{p}') \land \mathtt{p}' \neq \mathtt{p} \Rightarrow (\exists \mathtt{c}')(\mathtt{company}(\mathtt{c}') \land \mathtt{sells}(\mathtt{c}',\mathtt{p}'))))$.
  3. Not expressible in first-order logic.

This lack of grammatical faithfulness of first-order logic with respect to natural language quantification prompted Barwise and Cooper to pursue a drastically different route to formalize natural language. Instead of identifying quantifiers with *determiners* such as `some`, `all`, `no`, `at least two`, `most`, etc., they identified quantifiers with *noun phrases*. For example, in Class 1, the quantifiers are, respectively, the following noun phrases:

- *Class 1*:

1. `some`	*products*,
2. `all`	*products of type* `'knife'`,
3. `no`	*products of type* `'fork'`,
4. `at least two`	*products*, and
5. `most`	*products*.

Notice that each of the above noun phrases consists of a determiner and an expression which can be interpreted as a set. For Class 1, these *set expressions* are *products*, *products of type* `'knife'`, and *products of type* `'fork'`. These set expressions can be formalized using the standard set abstraction mechanism from logic, wherein $\widehat{x}\,[\varphi(x)]$, with $\varphi(x)$ some well-formed formula, represents the set $\{x \mid \varphi(x)\}$:

$$\begin{array}{lcl} \textit{products} & = & \widehat{\mathtt{p}}\,[\mathtt{product(p)}], \\ \textit{products of type } \mathtt{'knife'} & = & \widehat{\mathtt{p}}\,[\mathtt{product(p)} \wedge \mathtt{type(p,'knife')}], \quad \text{and} \\ \textit{products of type } \mathtt{'fork'} & = & \widehat{\mathtt{p}}\,[\mathtt{product(p)} \wedge \mathtt{type(p,'fork')}]. \end{array}$$

Thus, according to Barwise and Cooper, a *generalized quantifier* from a *syntactical* point of view is a *pair* consisting of a determiner *and* a set expression. To explain the *semantics* that Barwise and Cooper associated to generalized quantifiers, we reconsider the five sentences in Class 1 and observe that their common verb phrase, *are sold*, can also be seen as a set expression:

$$\textit{are sold} \quad = \quad \widehat{\mathtt{p}}\,[\mathtt{product(p)} \wedge (\exists \mathtt{c})(\mathtt{sells(c,p)})].$$

Obviously, a sentence of Class 1 is true if the set represented by the above expression satisfies the condition expressed by the respective general quantifier. Therefore, Barwise and Cooper argued that the semantics of a generalized quantifier can be described by the *family of* all *sets* that satisfy the condition expressed by the generalized quantifier in the current world of discourse. (We shall assume that the domain of discourse is some finite set D). For each generalized quantifiers in Class 1, Table 1 shows the family of sets describing its semantics. In Table 1, $[\![\langle \text{set expression} \rangle]\!]$ denotes the set represented by $\langle \text{set expression} \rangle$ in the current world of discourse.

Following Barwise and Cooper, the sentences of Class 1 can now be rewritten as follows:

Generalized quantifier	**Family of sets**
`some` *products*	$\{S \subseteq D \mid S \cap [\![\mathit{products}]\!] \neq \emptyset\}$
`all` *products of type* `'knife'`	$\{S \subseteq D \mid [\![\mathit{products\ of\ type}\ \texttt{'knife'}]\!] \subseteq S\}$
`no` *products of type* `'fork'`	$\{S \subseteq D \mid S \cap [\![\mathit{products\ of\ type}\ \texttt{'fork'}]\!] = \emptyset\}$
`at least two` *products*	$\{S \subseteq D \mid \lvert S \cap [\![\mathit{products}]\!]\rvert \geq 2\}$
`most` *products*	$\{S \subseteq D \mid \lvert S \cap [\![\mathit{products}]\!]\rvert > \lvert S - [\![\mathit{products}]\!]\rvert\}$

Table 1 Families of sets describing the semantics of the generalized quantifiers in Class 1.

- *Class 1*:

  1. (`some` *products*)(*are sold*).
  2. (`all` *products of type* `'knife'`)(*are sold*).
  3. (`no` *products of type* `'fork'`)(*are sold*).
  4. (`at least two` *products*)(*are sold*).
  5. (`most` *products*)(*are sold*).

Observe how the grammatical structure of the original natural language sentences is preserved. The semantics of these sentences is as explained above; for example, the first sentence of Class 1 is true precisely if

$$[\![\mathit{are\ sold}]\!] \in [\![\texttt{some}\ \mathit{products}]\!],$$

where $[\![\langle\text{generalized quantifier}\rangle]\!]$ denotes the family of sets describing the semantics of $\langle$generalized quantifier$\rangle$. Using this semantics, it is possible to re-formalize the set expression *are sold* within the framework of generalized quantifiers:

$$\mathit{sold} \quad = \quad \widehat{\texttt{p}}\,[(\texttt{some}\ \textbf{thing})(\widehat{\texttt{c}}\,[\texttt{sells(c,p)}])].$$

(Notice that the original formalization made use of traditional quantifiers.) The constant **thing** in the re-formalization represents the domain of discourse ("all possible things"), i.e., $[\![\textbf{thing}]\!] = D$. Hence all the sentences in Class 1 can be specified formally completely within the Barwise-Cooper syntax. The sentences in Class 2 can be formulated similarly, as follows:

- *Class 2*:

  1. $(\texttt{some}\ \widehat{\texttt{p}}\,[\texttt{product(p)}])(\widehat{\texttt{p}}\,[(\texttt{all}\ \widehat{\texttt{c}}\,[\texttt{company(c)}])(\widehat{\texttt{c}}\,[\texttt{sells(c,p)}])])$.
     Read: Some product is a product such that each company is a company that sells that product.
  2. $(\texttt{exactly one}\ \widehat{\texttt{p}}\,[\texttt{product(p)}])$
     $(\widehat{\texttt{p}}\,[(\texttt{no}\ \widehat{\texttt{c}}\,[\texttt{company(c)}])(\widehat{\texttt{c}}\,[\texttt{sells(c,p)}])])$.
     Read: Exactly one product is a product such that no company is a company that sells that product.
  3. $(\texttt{some}\ \widehat{\texttt{c}}\,[\texttt{company(c)}])$
     $(\widehat{\texttt{c}}\,[(\texttt{even nr of}\ \widehat{\texttt{p}}\,[\texttt{product(p)}])(\widehat{\texttt{p}}\,[\texttt{sells(c,p)}])])$.
     Read: Some company is a company such that an even number of products are a product that is sold by that company.

Barwise and Cooper also introduced a language, called $\mathcal{LGQ}$ (Language with Generalized Quantifiers), which admits generalized quantifier syntax. The formalized sentences shown above either implicitly (in the case of the Class 1 sentences) or explicitly (in the case of the Class 2 sentences) are formulated in $\mathcal{LGQ}$.

2.2 Generalized quantifiers as set predicates

In the next section, we shall present $\mathcal{QLGQ}$, an adaptation of $\mathcal{LGQ}$, the language of Barwise and Cooper, and propose it as a query language for relational databases. Thereto, we first discuss an alternative view of generalized quantifiers which better suits or purposes, namely as predicates over sets [13].

To see that generalized quantifiers can be viewed as predicates over sets, reconsider as an example the first sentence of Class 1 in Subsection 2.1 written in the syntax of Barwise and Cooper:

$$(\texttt{some}\ \ \textit{products})(\textit{are sold}).$$

We also recall the semantics given to this sentence:

$$[\![\textit{are sold}]\!] \in [\![\texttt{some}\ \ \textit{products}]\!]$$

Taking into account the semantics of the generalized quantifier in the above sentence as shown in Table 1, the semantics of the entire sentence can alternatively be written as follows:

$$[\![\textit{are sold}]\!] \cap [\![\textit{products}]\!] \neq \emptyset.$$

Hence, an alternative way to explain the semantics of the first sentence of Class 1 is to view the determiner `some` as expressing a (binary) relationship between the sets $[\![\textit{products}]\!]$ and $[\![\textit{are sold}]\!]$. More generally, we can think of `some` as a binary relationship over subsets of the domain of discourse, D:

$$\texttt{some} \quad = \quad \{(U,V) \in 2^D \times 2^D \mid V \cap U \neq \emptyset\}.$$

Following this latter approach, it is syntactically more appropriate to formulate the sentence (`some` *products*)(*are sold*) as the set-predicate formula

$$\texttt{some}(\textit{products}, \textit{are sold}).$$

From now on, we will take this approach and interpret generalized quantifiers as set predicates. In Table 2, we give the definitions of some frequently occurring generalized quantifiers specified as binary set predicates.

Generalized quantifier	**Family of sets**
`some`	$\{(U,V) \in 2^D \times 2^D \mid V \cap U \neq \emptyset\}$
`all`	$\{(U,V) \in 2^D \times 2^D \mid U \subseteq V\}$
`only`	$\{(U,V) \in 2^D \times 2^D \mid V \subseteq U\}$
`all and only`	$\{(U,V) \in 2^D \times 2^D \mid V = U\}$
`all but not only`	$\{(U,V) \in 2^D \times 2^D \mid U \subset V\}$
`not all`	$\{(U,V) \in 2^D \times 2^D \mid \neg U \subseteq V\}$
`no`	$\{(U,V) \in 2^D \times 2^D \mid V \cap U = \emptyset\}$
`at least` k	$\{(U,V) \in 2^D \times 2^D \mid \|V \cap U\| \geq k\}$
`at most` k	$\{(U,V) \in 2^D \times 2^D \mid \|V \cap U\| \leq k\}$
`precisely` k	$\{(U,V) \in 2^D \times 2^D \mid \|V \cap U\| = k\}$
`most`	$\{(U,V) \in 2^D \times 2^D \mid \|V \cap U\| > \|V - U\|\}$

Table 2 Some frequently encountered binary generalized quantifiers viewed as set predicates.

Most of the examples of generalized quantifiers we have encountered thus far are based on *binary* set predicates. There exist, however, many natural *unary* generalized quantifiers and also some natural *ternary* generalized quantifiers. A good example of a unary generalized quantifier is SQL's `EXISTS`. In SQL, `EXISTS`(S) is true if and only if the set S is non-empty. An example of a ternary generalized quantifier is contained in the sentence *more computer science students than math students take courses in logic.* Observe that in this sentence there are three relevant sets: (1) the set of computer science students, (2) the set of math students, and (3) the set of students taking courses. Thus, `more than` is a ternary generalized quantifier.

Therefore, we view a generalized quantifier as a set predicate with arbitrary arity rather than a binary set predicate. As it turns out, this is still not quite general enough for the purposes of this paper:

Definition 2.1 *A* generalized quantifier *of type* $[k, l]$*,* $k, l \geq 1$*, over the (finite) domain* D *is a* k*-ary relation over* 2^{D^l}*.*

Thus, we view a generalized quantifier as predicate of arbitrary arity ranging over relations of arbitrary arity rather than over sets.

3 THE QUERY LANGUAGE $\mathcal{QLGQ}$

3.1 Syntax of $\mathcal{QLGQ}$

We are now ready to specify the syntax of $\mathcal{QLGQ}$.

We implicitly assume there are infinitely enumerable sets of first-order predicate names of each arity, first-order variables, and first-order constants. Let $\mathbf{Q}$ be an enumerable set of generalized quantifier names, typed in the sense of Definition 2.1. The expressions of the language $\mathcal{QLGQ}(\mathbf{Q})$ are basic terms, set terms, basic formulae, and formulae, defined as follows:

- *Basic terms*:
    1. If x is a variable, then x is a basic term.
    2. If c is a constant, then c is a basic term.

- *Set terms*:

  1. If R is a k-ary predicate then R is a k-ary set term.
  2. If $\vec{x}$ is a finite sequence of k *distinct* variables, and φ is a formula, then the set abstraction $\widehat{\vec{x}}[\varphi]$ is a k-ary set term.
  3. If S_1 and S_2 are k-ary set terms, then the union $(S_1 \cup S_2)$ is a k-ary set term.
  4. The expression $\mathbf{thing}^k$ is a k-ary set term.
  5. If $t_1, \ldots, t_m$ are k-ary tuples of basic terms, then the set $\{t_1, \ldots, t_m\}$ is a k-ary set term.

- *Basic formulae*:

  1. If x and y are variables, then $x = y$ is a basic formula.
  2. If S is a k-ary set term, and $b_1, \ldots, b_k$ are basic terms, then both $S(b_1, \ldots, b_k)$ and $\neg\, S(b_1, \ldots, b_k)$ are basic formulae.
  3. If $Q \in \mathbf{Q}$ is a generalized quantifier name of type $[k, l]$, and $S_1, \ldots, S_k$ are l-ary set terms, then both $Q(S_1, \ldots, S_k)$ and $\neg Q(S_1, \ldots, S_k)$ are basic formulae.

- *Formulae*:

  1. If φ_1 and φ_2 are formulae, then the conjunction $\varphi_1 \wedge \varphi_2$ is a formula.

We finally define $\mathcal{QLGQ}(\mathbf{Q})$ queries:

Definition 3.1 *A $\mathcal{QLGQ}(\mathbf{Q})$* query *is a set term of $\mathcal{QLGQ}(\mathbf{Q})$ of the form $\widehat{\vec{x}}\,[\varphi]$, where φ a $\mathcal{QLGQ}(\mathbf{Q})$ formula with free variables $\vec{x}$.*

3.2 Semantics of $\mathcal{QLGQ}$

Let D be a finite set, and let $\mathcal{M}$ be a first-order model of the predicates under consideration. In order to attach semantics to the expressions of $\mathcal{QLGQ}(\mathbf{Q})$, we need in addition a *generalized-quantifier model*, $\mathcal{Q}$, of $\mathbf{Q}$ with domain D, which is defined below:

Definition 3.2 *Let D be a finite set. A* generalized quantifier model *of a enumerable set of quantifier $\mathbf{Q}$ with domain D, denoted in the sequel as $\mathcal{Q}$, is a mapping which assigns to each generalized quantifier name $Q \in \mathbf{Q}$ a generalized quantifier of the same type over D, in the sense of Definition 2.1.*

Finally, let ϑ be a variable-assignment mapping, assigning to each first-order variable an element of D. The semantics of an expression e of $\mathcal{QLGQ}(\mathbf{Q})$ relative to $\mathcal{M}$, $\mathcal{Q}$ and ϑ, denoted as $[\![e]\!]$, is recursively defined, as follows:

- *Basic terms*:
  1. $[\![x]\!] = \vartheta(x)$.
  2. The value $[\![c]\!]$ is the constant value assigned to c in the model $\mathcal{M}$.
- *Set terms*:
  1. The set $[\![R]\!]$, R being a k-ary predicate, is the k-ary relation assigned to R in the model $\mathcal{M}$.
  2. The set $[\![\widehat{\vec{x}}\,[\varphi]]\!]$, $\vec{x} = x_1, \ldots, x_k$ being a sequence of k different variables, is the set of all tuples $(u_1, \ldots, u_k)$ in D^k such that the value of $[\![\varphi[x_1 \leftarrow u_1, \ldots, x_k \leftarrow u_k]]\!]$ is true, where $\varphi\,[x_1 \leftarrow u_1, \ldots, x_k \leftarrow u_k]$ denotes the formula φ in which the variables $x_1, \ldots, x_k$ have been substituted by the constant values $u_1, \ldots, u_k$, respectively.
  3. $[\![(S_1 \cup S_2)]\!] = [\![S_1]\!] \cup [\![S_2]\!]$.
  4. $[\![\mathbf{thing}^k]\!] = D^k$.
  5. $[\![\{t_1, \ldots, t_m\}]\!] = \{[\![t_1]\!], \ldots, [\![t_m]\!]\}$.
- *Basic formulae*:
  1. The truth value of $[\![S(b_1, \ldots, b_k)]\!]$ is the truth value of
     $([\![b_1]\!], \ldots, [\![b_k]\!]) \in [\![S]\!]$.
     The truth value of $[\![\neg\, S(b_1, \ldots, b_k)]\!]$ is the truth value of
     $([\![b_1]\!], \ldots, [\![b_k]\!]) \notin [\![S]\!]$.
  2. The truth value of $[\![Q(S_1, \ldots, S_k)]\!]$ is the truth value of
     $([\![S_1]\!], \ldots, [\![S_k]\!]) \in \mathcal{Q}(Q)$.
     The truth value of $[\![\neg\, Q(S_1, \ldots, S_k)]\!]$ is the truth value of
     $([\![S_1]\!], \ldots, [\![S_k]\!]) \notin \mathcal{Q}(Q)$.
- *Formulae*:
  1. $[\![\varphi_1 \wedge \varphi_2]\!] = [\![\varphi_1]\!] \wedge [\![\varphi_2]\!]$.

3.3 Examples of $\mathcal{QLGQ}$ queries

We shall add one binary relation, `buys(cname, pname)`, to the example database used in the previous sections. Consider the following queries, some of which are quite complex:

- *Class 3*:

  1. Find each product of type `'knife'`.
  2. Find each product that is not of type `'knife'`.
  3. Find each product that is of type `'knife'` or type `'fork'`.
  4. Find each company that sells each product that it buys.
  5. Find each pair of companies such that the first company sells no products that are bought by the second company.
  6. Find each company that buys products but does not sell products.
  7. Find each company for which there are at least ten products of which that company is the only seller.

To give the reader a better feeling of the language $\mathcal{QLGQ}$ and the use of generalized quantifiers therein, we express the above queries in $\mathcal{QLGQ}$:

- *Class 3*:

  1. $\widehat{\mathtt{p}}\,[\mathtt{product(p)} \wedge \mathtt{type(p, 'knife')}]$
  2. $\widehat{\mathtt{p}}\,[\mathtt{product(p)} \wedge \neg\,\mathtt{type(p, 'knife')}]$
  3. $\widehat{\mathtt{p}}\,[\mathtt{product(p)} \wedge (\widehat{\mathtt{p'}}\,[\mathtt{product(p')} \wedge \mathtt{type(p', 'knife')}] \cup$
     $\widehat{\mathtt{p'}}\,[\mathtt{product(p')} \wedge \mathtt{type(p', 'fork')}])\mathtt{(p)}]$
  4. $\widehat{\mathtt{c}}\,[\mathtt{company(c)} \wedge \mathtt{all}(\widehat{\mathtt{p}}\,[\mathtt{product(p)} \wedge \mathtt{sells(c,p)}],$
     $\widehat{\mathtt{p}}\,[\mathtt{product(p)} \wedge \mathtt{buys(c,p)}])]$
  5. $\widehat{\mathtt{c_1, c_2}}\,[\mathtt{company(c_1)} \wedge \mathtt{company(c_2)} \wedge$
     $\mathtt{no}(\widehat{\mathtt{p}}\,[\mathtt{product(p)} \wedge \mathtt{sells(c_1,p)}], \widehat{\mathtt{p}}\,[\mathtt{product(p)} \wedge \mathtt{buys(c_2,p)}])]$
  6. $\widehat{\mathtt{c}}\,[\mathtt{company(c)} \wedge$
     $\mathtt{some}(\widehat{\mathtt{p}}\,[\mathtt{product(p)}], \widehat{\mathtt{p}}\,[\mathtt{product(p)} \wedge \mathtt{buys(c,p)}]) \wedge$
     $\mathtt{no}(\widehat{\mathtt{p}}\,[\mathtt{product(p)}], \widehat{\mathtt{p}}\,[\mathtt{product(p)} \wedge \mathtt{sells(c,p)}])]$
  7. $\widehat{\mathtt{c}}\,[\mathtt{company(c)} \wedge \mathtt{at\ least\ ten}(\widehat{\mathtt{p}}\,[\mathtt{product(p)}],$
     $\widehat{\mathtt{p}}\,[\mathtt{product(p)} \wedge \mathtt{only}(\{\mathtt{(c)}\}, \widehat{\mathtt{c'}}\,[\mathtt{company(c')} \wedge \mathtt{sells(c',p)}])])]$

4 THE CONJUNCTIVE FORMULATION THESIS

Before we present the conjunctive formulation thesis, we state, without proof, three expressiveness results regarding $\mathcal{QLGQ}$.

The first result is about $\mathcal{QLGQ}$ without generalized quantifiers:

Theorem 4.1 *The language* $\mathcal{QLGQ}()$*, the relational calculus without quantifiers, and the relational algebra without projection are equivalent in expressive power.*

Let $\texttt{exists}^{(1)}$ be the unary generalized quantifier of type $[1, 1]$ the semantics of which is the unary relation $\{(U) \in 2^{D^1} \mid U \neq \emptyset\}$. We have the following:

Theorem 4.2 *The language* $\mathcal{QLGQ}(\{\texttt{exists}^{(1)}\})$*, the relational calculus, and the relational algebra are equivalent in expressive power.*

Let **FOQ** be the set of generalized quantifiers consisting of `some`, `all`, `only`, `all but not only`, `not all`, `no`, `at least` k, `at most` k, `precisely` k, and $\texttt{exists}^{(1)}$. We have the following:

Theorem 4.3 *The language* $\mathcal{QLGQ}(\mathbf{FOQ})$ *and first-order logic are equivalent in expressive power.*

Now reconsider the syntax of $\mathcal{QLGQ}$ and observe that the negation and disjunction connectors of first-order logic do not appear as boolean connectors of formulas in the syntax of $\mathcal{QLGQ}$: $\mathcal{QLGQ}$ queries are formulated *conjunctively*. Nevertheless, in combination with a reasonable set of generalized quantifiers, $\mathcal{QLGQ}$ can express all of first-order-logic (Theorem 4.3). Furthermore, as $\mathcal{QLGQ}$ is a variation of $\mathcal{LGQ}$, the formulation of a query in $\mathcal{QLGQ}$ closely resembles the way in which this query would be formulated in natural language. All these observations combined with our experience with $\mathcal{QLGQ}$ prompt us to propose the *conjunctive formulation thesis*:

> *The natural way to formulate a real-world queries is as a conjunction of statements. These statements may either be*
>
> - *first-order predicate statements (or their negation) , or*
> - *set predicate statements (or their negation) over (potentially complex) sub-queries.*

Notice that, by Theorem 4.2, SQL allows the user to formulate each relational calculus or algebra query in accordance with the conjunctive formulation thesis. This observation lends further credence to the conjunctive formulation thesis.

5 GENERALIZED QUANTIFIERS IN EXISTING QUERY LANGUAGES

In this section, we discuss to which extent several logic-based query languages—without seeking completeness—have incorporated generalized quantifiers (implicitly) and related features that are also present in $\mathcal{QLGQ}$ or can easily be added.

5.1 SQL

In Figure 1 we already demonstrated SQL's implicit incorporation of generalized quantification. In particular, SQL has the unary generalized quantifier `EXISTS`. The `EXISTS` predicate of SQL corresponds to an infinite class of generalized quantifiers in $\mathcal{QLGQ}$, namely the class $\{\texttt{exists}^{(l)} \mid l \geq 1\}$, where $\texttt{exists}^{(l)}$ is the generalized quantifier of type $[1, l]$ the semantics of which is the unary relation $\{(U) \in 2^{D^l} \mid U \neq \emptyset\}$. (Actually, $\texttt{exists}^{(1)}$ suffices to simulate SQL's `EXISTS`, as follows from Theorem 4.2.) Theorem 4.2 also shows that the presence of the `EXISTS` predicate in SQL makes it possible to formulate queries expressible in the relational calculus or algebra in accordance to the conjunctive formulation thesis. It is unfortunate that SQL was not designed with a larger variety of generalized quantifiers, or with an extension mechanism that would allow to create this variety, so that (1) queries expressible in the relational calculus or algebra could be formulated even more naturally, and (2) certain queries not expressible in the relational calculus or algebra could also be formulated.

Other SQL constructs that allow the use of sub-queries are `ALL`, `ANY`, and `IN`.

The constructs `ALL` and `ANY` can be seen as computing the maximum respectively the minimum of the set of numbers returned by the sub-query and are therefore really aggregate functions. This feature will be discussed further in the context of the language CORAL in Subsection 5.3.

Here, we focus on the `IN` construct. As an example, reconsider the query *find all companies that sell products of type* `'knife'`. from the Introduction. A third alternative to express this query in SQL is shown in Figure 3.

```
SELECT C.cname
FROM   company C
WHERE  C.cname IN (SELECT S.cname
                   FROM   sells S
                   WHERE  S.pname IN (SELECT P.pname
                                      FROM   product P, type T
                                      WHERE  P.pname = T.pname
                                      AND    T.ptype = 'knife'))
```

Figure 3 The query *find all companies that sell products of type* `'knife'` expressed in SQL using the `IN` construct.

In this query, the sub-query

```
SELECT P.pname
FROM   product P, type T
WHERE  P.pname = T.pname
AND    T.ptype = 'knife'
```

represents an "unnamed" unary predicate which is true for an object precisely if that object is a product of type `'knife'`. In the same spirit,

```
SELECT S.cname
FROM   sells S
WHERE  S.pname IN (SELECT P.pname
                   FROM   product P, type T
                   WHERE  P.pname = T.pname
                   AND    T.ptype = 'knife')
```

represents an unnamed unary predicate which is true for an object precisely if that object sells a product of type `'knife'`. The ability to use unnamed predicates has the advantage that temporary names do not have to be introduced to formulate syntactically correct queries with natural sub-queries. This feature is also present in, e.g., λ-calculus [], but not in the traditional relational calculus.

In $\mathcal{QLGQ}$, SQL's `IN` expression correspond to basic formulae of the second kind, whereas generalized quantifiers correspond to basic formulae of the third kind (see Subsections 3.1 and 3.2). The example query as formulated in Figure 3 can be simulated in $\mathcal{QLGQ}$ as follows:

$\widehat{\mathtt{c}}\,[\mathtt{company(c)} \wedge \widehat{\mathtt{c'}}\,[\mathtt{sells(c',p)} \wedge \widehat{\mathtt{p'}}\,[\mathtt{product(p')} \wedge \mathtt{type(p', 'knife')}](\mathtt{p})](\mathtt{c})]$.

Notice that unnamed predicates can also be used in $\mathcal{QLGQ}$.

5.2 Relational calculus with operators (RC/S)

The query language RC/S was introduced by Özsoyoğlu and Wang to enhance the declarativeness of the relational calculus as a language to formulate queries involving universal quantifiers and negations [10].[1]

To get a flavor of the syntax of RC/S, consider as an example the two queries *find the companies that buy all products* and *find the companies that do not sell products of type* `'knife'`. In the relational calculus, these queries can be formulated as

$$\{\mathtt{c} \mid \mathtt{company(c)} \wedge (\forall \mathtt{p})(\mathtt{product(p)} \Rightarrow \mathtt{buys(c,p)})\}$$

and

$$\{\mathtt{c} \mid \mathtt{company(c)} \wedge \neg\,(\exists \mathtt{p})(\mathtt{product(p)} \wedge \mathtt{type(p, 'knife')} \wedge \mathtt{sells(c,p)})\},$$

respectively. Consider the subformula of the form $(\forall x)(\varphi(x,y) \Rightarrow \psi(x,z))$ in the first expression and the subformula of the form $\neg\,(\exists x)(\varphi(x,y) \wedge \psi(x,z))$ in the second expression. Özsoyoğlu and Wang observed that these expressions can be replaced by the expressions

$$\{x \mid \varphi(x,y)\} \subseteq \{x \mid \psi(x,z)\}$$

and

$$\{x \mid \varphi(x,y)\} \cap \{x \mid \psi(x,y)\} = \emptyset,$$

respectively. In RC/S, the example queries are expressed as

$$\{\mathtt{c} \mid \mathtt{company(c)} \wedge \{\mathtt{p} \mid \mathtt{product(p)}\} \subseteq \{\mathtt{p} \mid \mathtt{buys(c,p)}\}\}$$

and

$$\begin{aligned}&\{\mathtt{c} \mid \mathtt{company(c)} \wedge \\ &\quad \{\mathtt{p} \mid \mathtt{product(p)} \wedge \mathtt{type(p, 'knife')}\} \cap \{\mathtt{p} \mid \mathtt{sells(c,p)}\} = \emptyset\},\end{aligned}$$

[1] Actually, many of the ideas occurring in [10] can be traced back to a paper by Pirotte [11].

respectively. Obviously, both examples are classic cases of generalized quantification. In $\mathcal{QLGQ}$, they can be expressed using the `all` and the `no` generalized quantifier, respectively.

The query language RC/S has other features related to generalized quantification. This can be seen in the syntactic structure of the basic formulae of RC/S. Let S and T denote two comparable RC/S sub-queries. In Table 3, we show the most relevant basic formulae of RC/S together with their corresponding $\mathcal{QLGQ}(\mathbf{FOQ})$ formulae.

RC/S	$\mathcal{QLGQ}(\mathbf{FOQ})$
$\vec{x} \in S$	$S(\vec{x})$
$S \subseteq T$	$\texttt{all}(S, T)$
$S \subset T$	$\texttt{all but not only}(S, T)$
$S = T$	$\texttt{all and only}(S, T)$
$S = \emptyset$	$\neg\, \texttt{exists}^{(l)}(S)$
$S = D^k$	$\texttt{all}(\mathbf{thing}^k, S)$
$S \cap T = \emptyset$	$\texttt{no}(S, T)$
$S \cup T = \emptyset$	$\neg\, \texttt{exists}((S \cup T))$
$S \cap T = D^k$	$\texttt{all}(\mathbf{thing}^k, S) \wedge \texttt{all}(\mathbf{thing}^k, T)$
$S \cup T = D^k$	$\texttt{all}(\mathbf{thing}^k, (S \cup T))$

Table 3 The basic formulae of RC/S with the corresponding formulae of $\mathcal{QLGQ}$.

There are also some differences between RC/S and $\mathcal{QLGQ}$. In $\mathcal{QLGQ}$, any enumerable set of generalized quantifiers can be considered. In RC/S, however, only the set of generalized quantifiers under consideration is part of the syntax. Unlike $\mathcal{QLGQ}$, RC/S also allows conventional existential quantification. A further difference between RC/S and $\mathcal{QLGQ}$ is that disjunction in RC/S is handled via the logical connector $\vee$. In $\mathcal{QLGQ}$, disjunction is handled through the incorporation of set terms of the form $(S \cup T)$. The reason to specify $\mathcal{QLGQ}$ differently from RC/S on these last two points was to formulate the conjunctive formulation thesis as strongly and as simply as possible. Therefore, we believe that most queries involving disjunction are more appropriately handled in $\mathcal{QLGQ}$ syntax.

5.3 CORAL

The language CORAL [12] is a deductive database query language extending datalog with features such as complex objects, negation, aggregation functions, program modules, and an interface to the object-oriented programming language C++. We only discuss CORAL features related to set construction and set comparison.

To get a flavor of the syntax of CORAL, consider as an example the query *find the companies that buy all products* introduced in Subsection 5.1. This query can be expressed in CORAL as shown in Figure 4.

```
psetbought(C,set(<P>)) :- buys(C,P).
pset(set<P>)           :- product(P).
result(C)              :- company(C),
                               psetbought(C,Psetbought),pset(Pset),
                               subset(Pset,Psetbought).
```

Figure 4 The query *find the companies that buy all products* expressed in CORAL.

In general, a CORAL rule of the form

$$\mathtt{p}(\mathtt{X},\mathtt{set}(<\mathtt{Y}>)) \;\; :- \;\; \varphi(\mathtt{X},\mathtt{Y}),$$

where $\varphi(X,Y)$ is a conjunction of CORAL literals, corresponds to the $\mathcal{QLGQ}$ set abstraction

$$\widehat{\mathtt{Y}}\,[\varphi(\mathtt{X},\mathtt{Y})].$$

Unlike $\mathcal{QLGQ}$ or SQL (or RC/S, for that matter), CORAL does *not* support unnamed predicates. Like $\mathcal{QLGQ}$, CORAL supports set predicates. This feature is illustrated in the third rule of the CORAL program shown Figure 4. Observe that, e.g, the set predicate `subset(Pset, Psetbought)` corresponds to the $\mathcal{QLGQ}$ basic formula `all(Pset, Psetbought)`. In addition to the `subset` predicate, CORAL offers the set predicates `member`, `union`, `intersection`, `difference`, and the aggregate function `count`, `max`, and `min`. (The predicate `member` corresponds to SQL's `IN` construct discussed at length in Subsection 5.1

and will therefore not be considered here.) For example, the rule

$$\mathtt{r(X,Uset)}\ :-\ \mathtt{p(X,Sset),q(X,Tset),union(Sset,Tset,Uset).}$$

associates in `r` to each value of `X` occurring in both `p` and `q` as `Uset` the union of the set values `Sset` and `Tset` associated with `X` in `p` and `q`, respectively. With these built-in set predicates, it is possible to derive numerous other generalized quantifiers in CORAL, as shown in Figure 5, where `s` is a unary set predicate representing 2^{D^l} for some fixed l (cfr. Definition 2.1), and `empty` is a unary set predicate representing the empty set.

```
some(Uset,Vset)    :- s(Uset),s(Vset),
                         intersection(Vset,Uset,Wset),
                         not empty(Wset).
all(Uset,Vset)     :- s(Uset),s(Vset),subset(Uset,Vset).
only(Uset,Vset)    :- all(Vset,Uset).
not_all(Uset,Vset) :- s(Uset),s(Vset),not all(Uset,Vset).
no(Uset,Vset)      :- s(Uset),s(Vset),not some(uset,Vset).
most(Uset,Vset)    :- s(Uset),s(Vset),
                         intersection(Vset,Uset,Sset),
                         difference(Vset,Uset,Tset),
                         count(Sset) > count(Tset).
exists(Uset)       :- s(Uset),empty(Uset).
```

Figure 5 A CORAL program defining various generalized quantifiers.

We thus conclude that, in CORAL, the combination of the set-abstraction mechanism, predicate naming, and the availability of some well chosen built-in set predicates, allow for a strong support of generalized quantification. Unlike in $\mathcal{QLGQ}$, new generalized quantifiers can be constructed from existing ones. However, there is also an important drawback to this combination of CORAL features: it is possible to specify CORAL programs that are inherently intractable. For example, the CORAL program in Figure 6 defines the powerset of a predicate p. In particular, `powerset(Uset)` will be true if `Uset` is a subset of the relation represented by p.

Besides set predicates, CORAL also supports aggregate functions. The aggregate function `count` was used in the program in Figure 5 to define the `most` generalized quantifier. In this program, we used the aggregate function in "predicative mode." Aggregate functions can also be used in "generative

```
psingleton(X,set(<Y>)) :- p(X),X=Y.
psingletons(Sset)      :- singleton(X,Sset).
powerset(Uset)         :- empty(Uset).
powerset(Uset)         :- psingletons(Uset).
powerset(Uset)         :- powerset(Vset),powerset(Wset),
                             union(Vset,Wset,Uset).
```

Figure 6 A CORAL program computing the powerset of the relation represented by some predicate.

mode," however. For example, in the CORAL program in Figure 7, the count function generates for each company the number of products sold by that company.

```
cnrofproducts(C,count(set(<P>))) :- company(C),product(P),
                                        sells(C,P).
```

Figure 7 A CORAL program in which the aggregate function `count` is used in generative mode.

The combination of generative and predicative usage of aggregate functions allows for elegant, and often succinct, query formulation. For example, the CORAL program in Figure 8 computes the companies that sell the most products. In the second rule of this program, the `count` and `max` aggregate functions are used in generative mode, and, in the third rule, the `count` aggregate function it is used in predicative mode.

```
cproducts(C,set(<P>))             :- company(C),product(P),
                                        sells(C,P).
maxpsold(max(set(<count(Pset)>))) :- cproducts(C,Pset).
sellsmostp(C)                     :- cproducts(C,Pset),
                                        maxpsold(count(Pset)).
```

Figure 8 A CORAL program in which aggregate functions are used both in predicative and generative mode.

6 CONCLUSION

In this paper we have established a link between generalized quantification, as it was conceived in mathematical logic and subsequently used in linguistics, and the phenomenon of sub-query syntax in query languages. We introduced the query language $\mathcal{QLGQ}$ as a prototype formal language to accommodate generalized quantification, and argued that its properties support the conjunctive formulation thesis. This thesis states that real-world queries are most naturally formulated conjunctively in terms of simple, first-order predicate statements and (possibly complex) set predicates over sub-queries.

REFERENCES

[1] Barendregt, H. P., "The λ-Calculus: Its Syntax and Semantics," North-Holland, Amsterdam, New York, 1984.

[2] Barwise, J. and Cooper, R., "Generalized quantifiers and natural language," Linguistic and Philosophy, 4, 1981, pp. 159–219.

[3] Codd, E.F., "Relational model of data for large shared data banks," Communications of the ACM, 13, 1970, pp. 377–387.

[4] Codd, E.F., "Further normalizations of the relational model," in "Data Base Systems," R. Rustin, Ed., Prentice-Hall, Englewood Cliffs, N.J., 1972, pp. 33–64.

[5] Codd, E.F., "Relational completeness of database sublanguages," in "Data Base Systems," R. Rustin, Ed., Prentice-Hall, Englewood Cliffs, N.J., 1972, pp. 65–98.

[6] Date, C., "A Guide to the SQL Standard," Reading, Mass., Addison Wesley, 1988.

[7] Montague, R., "Formal Philosophy: Selected Papers," R. H. Thomason, Ed., Yale University Press, New Haven, 1974.

[8] Mostowski, A., "On a generalization of quantifiers," Fundamenta Mathematica, 44, 1957, pp. 12–36.

[9] Naqvi, S. and Tsur, S., "A Logical Language for Data and Knowledge Bases," Computer Science Press, New York, 1989.

[10] Özsoyoğlu, G. and Wang, H., "A relational calculus with set operators, its safety, and equivalent graphical languages," IEEE Transactions on Software Engineering, 15, 1989, pp. 1038–1052.

[11] Pirotte, A., "High level data base query languages, in "Logic and Databases," H. Gallaire and J. Minker, Eds., Plenum Press, New York, 1978, pp. 409–436.

[12] R. Ramakrishnan, D. Srivastava and S. Sudarshan, "CORAL: control, relations, and logic," in Proceedings of the 18th International Conference on Very Large Data Bases, 1992.

[13] van Benthem, J., "Questions about quantifiers," Journal of Symbolic Logic, 49, 1984, pp. 443–466.