Constructing SQL queries

Maarten Fokkinga
DB group, fac EWI, University of Twente, Netherlands

Version of October 12, 2005, 10:59

Abstract. SQL queries can be derived with 100% correctness from a natural language query by a calculation in set and predicate notation. This is particularly useful for queries involving quantifications hidden in the natural language formulation. The calculations lead not only to simple select-from-where formulations but also to formulations with a group-by and having clause.

We recommend the approach in the teaching of SQL, observe the possibility of tools assistance, and call for future work to build tools that support the approach.

ACM Categories and subject descriptors: H.3.3 [Information Storage and Retrieval]: Information Search and Retrieval---Query formulation.
General Terms: Design, Verification.
Additional Key Words and Phrases: SQL, set notation.

Introduction

SQL queries. In the Relational Model [4] a database consists of a collection of relations, each of which is a *set* of tuples. SQL [1, 6] is a language designed to query such databases in a declarative way, that is, using the concepts of the Relational Model an SQL query expresses primarily the *what* and not the *how*, so as to achieve both understandability by the end-user and the possibility for efficient automatic processing.

Constructing SQL queries. In spite of the declarative nature of SQL, the construction of an SQL query can be a tough task: for some queries (especially those involving "for all" and "there exists" assertions hidden in the natural language) there can be a large gap between a query in natural language and a representation of it in SQL. We present an approach in which traditional set and predicate notation is used to bridge that gap. Set and predicate notation is suitable because, firstly, it is very expressive (much more than SQL) so that it can easily represent a query in natural language, and secondly, it lends itself well to a step-by-step transformation (a "calculation") to a form that is close to SQL. Thus the SQL queries obtained in this way are correct by construction (and machine assistance is well possible along the way). We recommend the approach in the teaching of SQL.

Related work. The close correspondence between simple 'select from where' queries and set notation is well-known but hardly exploited in textbooks that teach SQL ([2, 8, 9, 10, 11, 12, 13]). Date [5] *does* promote the use of "relational calculus" for the same purpose, but

in a sense his relational calculus suffers from the same drawback as SQL: the gap between natural language and relational calculus expressions is just too large to overcome in one step. We improve Date's method in the more systematic notation of sets and predicates (borrowed from Z [14]), so that on the one hand the transitions from natural language to sets and from sets to SQL are simpler and on the other hand the calculations with sets and predicates go (much!) smoother than possible in the relational calculus. Our set notation corresponding to 'group-by having' queries differs from the way Date [5] deals with grouping, and is not commonly known (and, to the best of our knowledge, has not been discussed before in the literature).

Practice versus theory. As often, practice deviates from theory. We mention two aspects. First, real databases deviate from the Relational Model in that the relations are represented by tables, that is, by *bags* (also known as *multi-sets*) rather than *sets*. As long as the number of tuples in a table doesn't matter, there is no difference; but sometimes it does matter. In this paper we disregard the problem of bridging the gap between tables and relations in a formal way. An easy way out is to put a costly 'distinct' in each select query, but in order to obtain not too inefficient queries we will leave out such 'distinct' indications when appropriate — without formal justification. A formal justification would use key properties of the tables.

Second, some features of SQL deviate from commonly accepted mathematical standards. For example, in SQL the *sum* of an empty table is defined to be *null* rather than 0. Such features complicate the transition from set notation to SQL, but do not invalidate the principle that we want to show. In this paper we disregard these features too, and in particular we assume that the given tables do not contain nulls.

This paper. In the next section the set and predicate notation is explained, and some calculation rules are given. Then, in section *Select-from-where queries* we demonstrate our approach for the case where we do not strive for group-by clauses. Having done that, we discuss in section *Group-by queries* how group-by and having clauses can be recognized in set and predicate notation, and again give some examples. Some concluding remarks follow in section *Conclusion*.

We present some calculations not until the appendix in order to stimulate the reader to have a try for himself and in order to get at the conclusions quickly.

Sets and predicates

Set and predicate notation. Apart from conventional notation for sets and predicates, we also use the lesser known syntax of the Z notation [14]. Here are some examples:

$$\begin{array}{lllllll} & & \overbrace{\phantom{x:\mathbb{Z};\ y:\mathbb{N}}}^{Decl} & & \overbrace{\phantom{y{>}10 \wedge 2{*}y = x}}^{Pred} & & \overbrace{\phantom{x{+}1}}^{Expr} \\ (1) & \{ & x:\mathbb{Z};\ y:\mathbb{N} & \mid & y{>}10 \wedge 2{*}y = x & \bullet & x{+}1 \ \} \\ (2) & \exists & x:\mathbb{Z};\ y:\mathbb{N} & \mid & y{>}10 \wedge 2{*}y = x & \bullet & x{+}1 \bmod 3 = 0 \\ (3) & \forall & x:\mathbb{Z};\ y:\mathbb{N} & \mid & y{>}10 \wedge 2{*}y = x & \bullet & x{+}1 \bmod 3 = 0 \\ & & \underbrace{\phantom{x:\mathbb{Z};\ y:\mathbb{N}}}_{Decl} & & \underbrace{\phantom{y{>}10 \wedge 2{*}y = x}}_{Pred} & & \underbrace{\phantom{x{+}1 \bmod 3 = 0}}_{Pred'} \end{array}$$

Symbols '|' and '•' are just tokens to separate various parts of the expression. If the *Pred*-part is *true*, the part '| *Pred*' is generally omitted. Notice that the declared variables need not occur in *Expr*: variable y does occur in *Decl* but not in *Expr* and *Pred′*, respectively. In words the expressions mean, in order:

(1): "The set of all values $x{+}1$,
where x varies over $\mathbb{Z}$ and y varies over $\mathbb{N}$ in such a way that $y{>}10 \wedge 2{*}y = x$."

This set consists of all odd numbers greater than 22: $\{23, 25, 27, 29, \ldots\}$.

(2): "There exists an $x : \mathbb{Z}$ and an $y : \mathbb{N}$ such that $y{>}10 \wedge 2{*}y = x$,
for which it is true that $x{+}1$ is divisible by 3."

This assertion is true; for example, take $y = 13$ and $x = 26$; then we have $y{>}10$ and $2{*}y = x$ *and* it holds true that $x{+}1$ is divisible by 3.

(3): "For all $x : \mathbb{Z}$ and $y : \mathbb{N}$ such that $y{>}10 \wedge 2{*}y = x$,
it holds true that $x{+}1$ is divisible by 3."

This assertion is false: a counterexample is $y = 11$ and $x = 22$; then we *do* have $y{>}10$ and $2{*}y = x$ but nevertheless $x{+}1$ is *not* divisible by 3.

The above set (1) and existential predicate (2) are close to SQL; their SQL equivalents read:

$$\begin{array}{lllllll}
 & \overbrace{\phantom{x{+}1}}^{Expr} & & \overbrace{\phantom{\mathbb{Z}\,x,\ \mathbb{N}\,y}}^{Decl} & & \overbrace{\phantom{y{>}10 \wedge 2{*}y = x}}^{Pred} & \\
\text{(1): select} & x{+}1 & \text{from} & \mathbb{Z}\,x,\ \mathbb{N}\,y & \text{where} & y{>}10 \wedge 2{*}y = x & \\
\text{(2): exists (select} * \text{ from} & & & \underbrace{\mathbb{Z}\,x,\ \mathbb{N}\,y}_{Decl} & \text{where} & \underbrace{y{>}10 \wedge 2{*}y = x}_{Pred} & \text{and } \underbrace{x{+}1 \bmod 3 = 0}_{Pred'} \text{)}
\end{array}$$

(In this example we have used the mathematical entity $\mathbb{Z}$ as an SQL table with one anonymous attribute, so that x itself can be used in an expression rather than, say, $x.value$. Similarly for y.) In general, there is no direct SQL equivalent for the universal predicate (at least, not in SQL92, which is the standard we refer to throughout this paper); so, by suitable calculation rules —given below— the $\forall$ has to be transformed into $\exists$, in order to come close to SQL. For brevity we do not discuss the special cases, with SQL terms like 'any' and 'all'.

Set and predicate calculus. There are quite some rewrite rules for sets and predicates that change the form but not the semantics. We list here only the bare minimum, with some explanation following the list:

$$\begin{array}{lllll}
(4) & \exists\, D \mid P \bullet P' & = & \exists\, D \bullet P \wedge P' & \\
(5) & \forall\, D \mid P \bullet P' & = & \forall\, D \bullet P \Rightarrow P' & \\
(6) & \exists\, D \mid P \bullet P' & = & \neg\ \forall\, D \mid P \bullet \neg\ P' & \\
(7) & \forall\, D \mid P \bullet P' & = & \neg\ \exists\, D \mid P \bullet \neg\ P' & \\
(8) & (\exists\, D \mid P \bullet expr = E) & = & expr \in \{D \mid P \bullet E\} & \text{"one-point rule"} \\
(9) & D \mid (\exists\, D' \mid P' \bullet P'') \wedge P & = & D;\, D' \mid P' \wedge P'' \wedge P & \text{"shunting"}
\end{array}$$

Rules (4–5) show that the part '| *Pred*' can be eliminated; note that the elimination gives '$P \wedge$' in case of the $\exists$ predicate, but '$P \Rightarrow$' in case of the $\forall$ predicate. Rules (6–7) show a

well-known duality between $\forall$ and $\exists$. Notice that the part '$\mid P$' stays the same; that is an important, practical, argument for the use of the lesser known extended forms $\forall\, D \mid P \bullet P'$ and $\exists\, D \mid P \bullet P'$ besides the better known forms $\forall\, D \bullet P$ and $\exists\, D \bullet P$. In the "one-point rule" (8) it is assumed that variables declared in D don't occur in *expr*. In the "shunting" rule (9) it is assumed that the variables declared in D are distinct from those in D'; the rule is quite important, applies both to sets and predicates, and *is crucial to avoid subqueries* in SQL formulations. Here are two examples:

$$\begin{array}{ll} & \{x : \mathbb{Z} \mid (\exists\, y : \mathbb{N} \mid y{>}10 \bullet 2 * y = x) \bullet x{+}1\} \\ = & \\ & \{x : \mathbb{Z} \;;\; y : \mathbb{N} \mid y{>}10 \wedge 2 * y = x \bullet x{+}1\} \end{array}$$

and

$$\begin{array}{ll} & \exists\, x : \mathbb{Z} \mid (\exists\, y : \mathbb{N} \mid y{>}10 \bullet 2 * y = x) \bullet x{+}1 \bmod 3 = 0 \\ = & \\ & \exists\, x : \mathbb{Z} \;;\; y : \mathbb{N} \mid y{>}10 \wedge 2 * y = x \bullet x{+}1 \bmod 3 = 0 \end{array}$$

The translation to SQL yields the following equations:

$$\begin{array}{ll} & \text{select } x{+}1 \text{ from } \mathbb{Z}\, x \text{ where exists (select} * \text{ from } \mathbb{N}\, y \text{ where } y{>}10 \wedge 2 * y = x) \\ = & \\ & \text{select } x{+}1 \text{ from } \mathbb{Z}\, x,\, \mathbb{N}\, y \text{ where } y{>}10 \wedge 2 * y = x \end{array}$$

and

$$\begin{array}{ll} & \text{exists (select} * \text{ from } \mathbb{Z}\, x \text{ where} \\ & \qquad \text{exists (select} * \text{ from } \mathbb{N}\, y \text{ where } y{>}10 \wedge 2 * y = x) \text{ and } x{+}1 \bmod 3 = 0) \\ = & \\ & \text{exists (select} * \text{ from } \mathbb{Z}\, x,\, \mathbb{N}\, y \text{ where } y{>}10 \wedge 2 * y = x \text{ and } x{+}1 \bmod 3 = 0) \end{array}$$

More abstractly, consider these two equal sets:

$$\{D \mid (\exists\, D' \mid P' \bullet P'') \wedge P \bullet E\} \quad = \quad \{D;\, D' \mid P' \wedge P'' \wedge P \bullet E\}$$

The transition to SQL gives two statements that are almost equal:

$$\begin{array}{ll} & \text{select } E \text{ from } D \text{ where exists (select} * \text{ from } D' \text{ where } P' \text{ and } P'') \text{ and } P \\ \approx & \\ & \text{select } E \text{ from } D,\, D' \text{ where } P' \text{ and } P'' \text{ and } P \end{array}$$

The almost-equals sign $\approx$ means that the select-results are *possibly different* bags but they do contain the *same* elements: only the multiplicities of the elements can differ. The results can be made equal by inserting appropriate (costly) 'distinct's of which one or both might be redundant on account of key properties. For brevity we do not discuss in this paper the elimination of superfluous 'distinct's.

Select-from-where queries

Database schema. We demonstrate the calculational approach by means of some examples. For these we use the following database schema about sea battles:

$Battle(\underline{name}, ...)$	-- the battles
$Class(\underline{name}, type, ...)$	-- the classes
$Ship(\underline{name}, class, ...)$	-- the ships

Participate(*ship*, *battle*, *result*, ...) -- which ship participates in which battle

In the formulas in the sequel we abbreviate all identifiers of this schema by their first letter. The primary keys have been underlined: battles, classes, and ships are identified by their name, and a ship-battle combination occurs at most once in *Participate*. Further, we assume the inclusion constraints (in fact: foreign key constraints) that each battle and ship mentioned in *Participate* does occur in *Battle* and *Ship*, respectively, and each class mentioned in *Ship* does occur in *Class*:

(10) $\forall\, p : P \bullet \exists\, b : B \bullet p.b = b.n$, i.e., $\{p : P \bullet p.b\} \subseteq \{b : B \bullet b.n\}$

(11) $\forall\, p : P \bullet \exists\, s : S \bullet p.s = s.n$

(12) $\forall\, s : S \bullet \exists\, c : C \bullet s.c = c.n$

Example A. Consider the following query:

Find the ships that participate in all battles.

In the derivation below we proceed by stepwise refinement. The first three steps signify a transition of a piece of natural language (enclosed in double quotes) to formal language; each of the transitions is so small that we feel justified to use an equals symbol. Having obtained line (13), or its preceding line, we realize that SQL has no direct equivalent for $\forall$, so that we rewrite the $\forall$ into $\exists$ according rule (7):

$$
\begin{array}{ll}
 & \text{“the ships that participate in all battles”} \\
= & \\
 & \{s : S \mid \text{“}s\text{ participates in all battles”} \bullet s.n\} \\
= & \\
 & \{s : S \mid (\forall\, b : B \bullet \text{“}s\text{ participates in }b\text{”}) \bullet s.n\} \\
= & \\
(13) & \{s : S \mid (\forall\, b : B \bullet (\exists\, p : P \bullet p.s = s.n \wedge p.b = b.n)) \bullet s.n\} \\
= & \\
 & \{s : S \mid \neg\, (\exists\, b : B \bullet \neg\, (\exists\, p : P \bullet p.s = s.n \wedge p.b = b.n)) \bullet s.n\} \\
= & \\
 & \text{select } s.n \text{ from } S\ s \text{ where not exists} \\
 & \qquad \text{(select } * \text{ from } B\ b \text{ where not exists} \\
 & \qquad\qquad \text{(select } * \text{ from } P\ p \text{ where } p.s = s.n \text{ and } p.b = b.n))
\end{array}
$$

By the way, part $(\exists\, p : P \bullet p.s = s.n \wedge p.b = b.n)$ is, on account of the one-point rule (8), equivalent both to $s.n \in \{p : P \mid p.b = b.n \bullet p.s\}$ and to $b.n \in \{p : P \mid p.s = s.n \bullet p.b\}$, thus giving two more variations that immediately translate to SQL.

Alternative 1. Line (13) has the form $\{s : S \mid \mathit{inAllBattles}\,(s.n) \bullet s.n\}$ where:

(14) $\mathit{inAllBattles}\,(x) \quad = \quad \forall\, b : B \bullet (\exists\, p : P \bullet p.s = x \wedge p.b = b.n)$

Here an experienced set calculator might recognize a common pattern from set theory: a set inclusion, and in this case even set equality and set cardinality equality. Indeed:

$$
\begin{array}{ll}
(15) & \mathit{inAllBattles}\,(x) \\
= & \\
 & \forall\, b : B \bullet (\exists\, p : P \bullet p.s = x \wedge p.b = b.n) \\
= &
\end{array}
$$

$$
\begin{array}{ll}
 & \forall\, b : B \bullet (\exists\, p : P \mid p.s = x \bullet p.b = b.n) \\
= & \quad \text{one-point rule (8)} \\
 & \forall\, b : B \bullet b.n \in \{p : P \mid p.s = x \bullet p.b\} \\
= & \\
 & \{b : B \bullet b.n\} \subseteq \{p : P \mid p.s = x \bullet p.b\} \\
= & \quad \text{inclusion constraint (10) implies } \{b : B \bullet b.n\} \supseteq \{p : P \mid \ldots \bullet p.b\} \\
 & \{b : B \bullet b.n\} \subseteq \{p : P \mid p.s = x \bullet p.b\} \wedge \\
 & \{b : B \bullet b.n\} \supseteq \{p : P \mid p.s = x \bullet p.b\} \\
= & \\
 & \{b : B \bullet b.n\} = \{p : P \mid p.s = x \bullet p.b\} \\
(\dagger) \quad = & \quad \text{inclusion constraint (10) implies } \{b : B \bullet b.n\} \supseteq \{p : P \mid \ldots \bullet p.b\} \\
 & \#\{b : B \bullet b.n\} = \#\{p : P \mid p.s = x \bullet p.b\} \\
(\ddagger) \quad = & \quad n \text{ is a key in } B \\
 & \#\{b : B \bullet b\} = \#\{p : P \mid p.s = x \bullet p.b\} \\
= & \\
 & (\text{select count } (*) \text{ from } B) = (\text{select count (distinct } p.b) \text{ from } P\, s \text{ where } p.s = x)
\end{array}
$$

Step (†) may need some explanation. The downward implication is true since it is an instance of '$x = y \Rightarrow f\, x = f\, y$'. The upward implication has the form '$\#A = \#B \Rightarrow A = B$ and is true, in this case, because by the hint we have that $A \supseteq B$.

Step (‡) is clear enough, and actually is an instance of the following law:

$$
(16) \quad \#\{D \mid P \bullet E\} \quad = \quad \#\{D \mid P \bullet E'\}, \quad \text{provided } \{D \mid P \bullet (E, E')\} \text{ is a bijection}
$$

In words the condition means that in the context of D satisfying P, the value of E functionally determines the value of E' and conversely. In the above application, E is the expression $b.n$, and E' is the expression b. Since n is a key in B, the value of E functionally determines the value of E'; the converse is obvious.

Now, using the above result for *inAllBattles*, we can alternatively calculate from line (13) onwards:

$$
\begin{array}{ll}
 & (13) \\
= & \\
 & \{s : S \mid \mathit{inAllBattles}\,(s.n) \bullet s.n\} \\
= & \\
(17) \quad & \text{select } s.n \text{ from } S\ d \\
 & \text{where (select count } (*) \text{ from } B) = \\
 & \qquad (\text{select count (distinct } p.b) \text{ from } P\, p \text{ where } p.s = s.n)
\end{array}
$$

Alternative 2. Let us denote set subtraction by '\'; the SQL translation is 'except'. This operator facilitates an alternative derivation, namely, expressed in natural language: "the ships that participate in all battles" equals "all ships *except* the ships that do *not* participate in all battles". This is step (∗) below:

$$
\begin{array}{ll}
 & \text{"the ships that participate in all battles"} \\
= & \quad \text{as above} \\
 & \{s : S \mid \mathit{inAllBattles}(s.n) \bullet s.n\} \\
(*) \quad = & \quad \text{see } \textit{Note} \text{ below}
\end{array}
$$

$$
\begin{array}{ll}
 & \{s : S \bullet s.n\} \setminus \{s : S \mid \neg\ inAllBattles(s.n) \bullet s.n\} \\
= & \\
 & \{s : S \bullet s.n\} \setminus \{s : S \mid \neg\ (\forall\, b : B \bullet (\exists\, p : P \bullet p.s = s.n \wedge p.b = b.n)) \bullet s.n\} \\
= & \\
 & \{s : S \bullet s.n\} \setminus \{s : S \mid (\exists\, b : B \bullet \neg\ (\exists\, p : P \bullet p.s = s.n \wedge p.b = b.n)) \bullet s.n\} \\
= & \\
 & \{s : S \bullet s.n\} \setminus \{s : S;\ b : B \mid \neg\ (\exists\, p : P \bullet p.s = s.n \wedge p.b = b.n) \bullet s.n\} \\
= & \\
 & \text{(select } s.n \text{ from } S\ s) \\
 & \text{except} \\
 & \text{(select } s.n \text{ from } S\ s,\ B\ b \\
 & \ \text{where not exists (select} * \text{ from } P\ p \text{ where } p.s = s.n \text{ and } p.b = b.n))
\end{array}
$$

Again, part $(\exists\, p : P \bullet p.s = s.n \wedge p.b = b.n)$ is, on account of the one-point rule (8), equivalent both to $s.n \in \{p : P \mid p.b = b.n \bullet p.s\}$ and to $b.n \in \{p : P \mid p.s = s.n \bullet p.b\}$, thus giving two more variations that immediately translate to SQL.

Note. Step $(*)$ above is clear enough, but viewed more generally it is an instance of the following law for sets:

$$(18)\quad \{D \mid Q \bullet E\} \quad = \quad \{D \bullet E\} \setminus \{D \mid \neg\ Q \bullet E\}, \quad \text{provided } Q \text{ is a function of } E \text{ alone}$$

The condition means that Q can be written as $Q_0(E)$ where $Q_0(\_)$ itself does not depend on the variables introduced by D. In the application $(*)$ above, E is $s.n$ and Q is $inAllBattles(s.n)$, and $inAllBattles(\_)$ itself does not depend on s.

> Fully formally, the condition reads $\forall\, D;\ D' \mid E = E' \bullet Q = Q'$ where $'$ is the substitution of identifiers declared by D into primed identifiers. When the condition is false, the equation need not hold: $\{x : \mathbb{Z} \mid 0{<}x \bullet x^2\} \neq \{x : \mathbb{Z} \bullet x^2\} \setminus \{x : \mathbb{Z} \mid x{\leq}0 \bullet x^2\}$, and, indeed, the condition $\forall\, x, x' : \mathbb{Z} \mid x^2 = x'^2 \bullet (0{<}x) = (0{<}x')$ is false (a counterexample being $x, x' = 7, -7$).

Example B. Here is another example. The query was presented as an exam question to 160 undergraduate computer science students and to a dozen colleagues (assistant professors and PhD students in the field of Databases). No one of the colleagues produced a 100% correct SQL formulation; and only two students did. Yet, by the approach above, one may calculate the SQL formulation with mathematical rigor; in our elaboration all steps are so small that verification is easy. The query reads:

> Find the types of which all ships have sunk.

'The type of' a ship is, of course, the type of the class of the ship. A ship has sunk in a battle if it participated in the battle with *result* = *'sunk'*. We challenge the reader to derive or otherwise invent an SQL formulation. The appendix presents our calculation; it also presents some non-solutions proposed by colleagues and students.

Group-by queries

Group-by in set notation. The SQL semantics of a group-by clause depends on the notion of group. Given a table and some attributes, called the group attributes, a *group* is a nonempty maximal subset of the table, consisting of rows having the same value for the group attributes. Now, given the table formed by the from and where clause of the query, the group-by clause constructs precisely *one* row for each group of the table; the row for a group

is formed from, firstly, the values that the rows in the group take for the group attributes, and secondly, aggregations of some attribute over all rows of the group. SQL knows the aggregates *Max*, *Min*, *Sum*, *Avg*, and *Count*; in the sequel we let F denote an arbitrary aggregate.

In SQL there is no access to the groups themselves other than by the group attributes and aggregations. In set notation, however, we must be able to express the individual groups explicitly, in order to achieve a smooth transition between non-grouped and grouped intermediate results during the calculation from the natural language query to the SQL query. We shall now show how a group is expressed in set notation, and thus what set expression is the direct equivalent of a general SQL group-by query.

Consider an arbitrary SQL group-by query:

(19) select $E(t.b, F(t.c))$ from T t where $P(t.a)$ group by $t.b$ having $Q(t.b, F'(t.c'))$

Actually, T t stands for a *series* of named tables, a stands for *the set* of all attributes of T t, and b for a *subset* of a whereas c and c' are individual members of a. Also, E stands for a *series* of expressions, and each of $F(t.c)$ and $F'(t.c')$ for a *series* of aggregations. Expressed in set notation, the evaluation now proceeds as follows:

$$
\begin{array}{rll}
 & T & 0 \\
\text{from} \downarrow & \{t : T \mid \ \mathit{true} \ \bullet t\} & 1 \\
\text{where} \downarrow & \{t : T \mid P(t.a) \bullet t\} & 2 \\
\text{group-by} \downarrow & \{t : T \mid P(t.a) \bullet G\} & 3 \\
\text{having} \downarrow & \{t : T \mid P(t.a) \wedge Q(t.b, F'\ G_{c'}) \bullet G\} & 4 \\
\text{select} \downarrow & \{t : T \mid P(t.a) \wedge Q(t.b, F'\ G_{c'}) \bullet E(t.b, F\ G_c)\} & 5
\end{array}
\tag{20}
$$

The evaluation starts with the given tables T: line 0. The from clause yields the Cartesian product of these: line 1 (remember, T stands for a *series* of tables). The where clause filters out those rows that do not satisfy the contraining predicate P, as shown in line 2. Then the groups are formed. The group to which row t belongs, is denoted G (not mentioning the dependency on t); the rows belonging to this group all have $t.b$ as values for the group attributes b:

$$G \ = \{t' : T \mid P(t'.a) \wedge t'.b = t.b \bullet t' \ \ \}$$

For future use, note that an aggregation of this group, say with aggregate F working on attribute c, has the form $F\ G_c$ where:

$$G_c = \{t' : T \mid P(t'.a) \wedge t'.b = t.b \bullet t'.c\}$$

Thus the intermediate result yielded by the group-by clause is a set of groups, as shown in line 3. The members of a group are not individually accessible from within SQL; per group only the value of the grouping attribute b and an aggregration on an arbitrary attribute c may be referred to. Line 4 shows the intermediate result, after the having clause has eliminated the groups that do not satisfy predicate Q. Line 5 shows the final result, where the select clause yields for each group G just one row, namely $E(t.b, F\ G_c)$. The conclusion is that an SQL expression in the form of (19) is equivalent to a set expression in the form of (20).

Expressiveness of groups. The equivalence of the general group-by query (19) with the specific set expression (20) leads to an interesting observation. The specific form (20) is just an instance of the general set expression, for which the translation to SQL without a group-by and having clause is immediate. Hence each group-by query can be expressed as a group-less query. In particular, for the general group-by query (19) we find, via (20):

$$\begin{array}{ll} & \text{select } E(t.b, F(t.c)) \text{ from } T\ t \text{ where } P(t.a) \text{ group by } t.b \text{ having } Q(t.b, F'(t.c')) \\ (21)\ \approx & \\ & \text{select } E(t.b,\ FG_c) \text{ from } T\ t \text{ where } P(t.a) \text{ and } Q(t.b,\ FG'_{c'}) \end{array}$$

where FG_c and $FG'_{c'}$, respectively, stand for the following SQL subquery:

$$\begin{array}{l} (\text{select } F\ (\ t'.c\ ) \text{ from } T\ t' \text{ where } P(t'.a) \text{ and } t'.b = t.b) \\ (\text{select } F'(t'.c') \text{ from } T\ t' \text{ where } P(t'.a) \text{ and } t'.b = t.b) \end{array}$$

Each of these subqueries is correlated to the outer main query via variable t. The $\approx$-sign signifies that the resulting tables contain the *same* rows, but possibly with a different amount of duplication.

In general, this elimination of a group-by clause is not to be recommended, since most DBMSs will evaluate the group-by query much more efficiently than the one with the correlated subqueries.

Example C. Take this query:

> Find the ships that participate in all battles, but skip them all if there are no battles.

It differs from example A only in the addition of the directive to "skip them all if there are no battles." Such an adaptation occurs often in practice; the empty set (here: "there are no battles") is made to a special case in which one is not interested. Note that in case there are no battles, *all* ships participate in all battles, but now *no* ship has to be delivered. A seemingly minor adaptation in the query has a major effect on the answer! We shall see such an adaptation again in example D.

> *Note.* In this example, if there are no battles, then the entire table *Battle* is empty and so is the entire table *Participate*; and a simple test on this condition (with the previous query in the else-branch) will suffice. However, such a simple test is not possible if 'the battles' in the query is replaced by 'the battles of type x' or something similar.

In order to construct a query (a group-by query this time), we redo the previous derivation up to formula (13). At that point, we take the skip directive into account (and we might have done it earlier as well, of course). Using abbreviation *inAllBattles* again, the calculation then proceeds as follows. First the extra condition "there are battles" is elaborated. Second, using the shunting rule (9) twice the part $p{:}P$ is shifted outward and the part $s{:}S$ is shifted inward and even disappears; this gives $\{p{:}P \mid \mathit{inAllBattles}(p.s) \bullet p.s\}$ whereas we started out with $\{s{:}S \mid \mathit{inAllBattles}(s.n) \bullet s.n\}$! Third, abbreviation *inAllBattles* is taken into account as before. Finally, in line (22), we recognize the set expression related to the group-by construct.

$$\begin{array}{ll} & \text{"the ships that participate in all battles"} \\ = & \quad \text{as above} \\ & \{s : S \mid \mathit{inAllBattles}(s.n) \bullet s.n\} \\ \supseteq & \quad \text{"but skip them all if there are no battles"} \end{array}$$

$$
\begin{array}{ll}
 & \{s : S \mid \text{“there are battles”} \wedge \mathit{inAllBattles}(s.n) \bullet s.n\} \\
= & \\
 & \{s : S \mid (\exists\, b : B) \wedge \mathit{inAllBattles}(s.n) \bullet s.n\} \\
= & \quad \mathit{inAllBattles}(s.n)\ (14)\ \text{implies} \quad \text{`}(\exists\, b : B) \Rightarrow (\exists\, p : P \bullet p.s = s.n)\text{'} \\
 & \quad \text{inclusion constraint (10) implies `}(\exists\, b : B) \Leftarrow (\exists\, p : P \bullet p.s = s.n)\text{'} \\
 & \{s : S \mid (\exists\, p : P \bullet p.s = s.n) \wedge \mathit{inAllBattles}(s.n) \bullet s.n\} \\
= & \quad \text{shunting} \\
 & \{s : S;\, p : P \mid p.s = s.n \wedge \mathit{inAllBattles}(s.n) \bullet s.n\} \\
= & \\
 & \{s : S;\, p : P \mid p.s = s.n \wedge \mathit{inAllBattles}(p.s) \bullet p.s\} \\
= & \quad \text{shunting} \\
 & \{p : P \mid (\exists\, s : S \bullet p.s = s.n) \wedge \mathit{inAllBattles}(p.s) \bullet p.s\} \\
= & \quad \text{inclusion constraint (11)} \\
 & \{p : P \mid \mathit{inAllBattles}(p.s) \bullet p.s\} \\
= & \quad \text{theorem (15) about } \mathit{inAllBattles} \\
 & \{p : P \mid \#\{b : B \bullet b\} = \#\{p' : P \mid p'.s = p.s \bullet p'.b\} \bullet p.s\} \\
= & \\
(22) \quad & \{p : P \mid \#\{b : B \bullet b\} = \#G_b \bullet p.s\} \\
 & \quad \text{where } G_b \text{ stands for } \{p' : P \mid p'.s = p.s \bullet p'.b\} \\
= & \\
 & \text{select } p.s \text{ from } P\ p \\
 & \text{group by } p.s \\
 & \text{having (select count } (*) \text{ from } B) = \text{count (distinct } p.b)
\end{array}
$$

Note that line (22) has the form of (20) where predicate P is absent, or just *true*, and $Q(x, y)$ is the equality test $\#\{b : B \bullet b\} = y$ without referencing x. The derived SQL formulation is not correct if the directive ‘skip them all if there no battles’ is left out of the query. On account of equation (21) the formulation just derived is equivalent to the following one:

select distinct $p.s$ from $P\ p$
where (select count $(*)$ from B) =
    (select count (distinct $p'.b$) from $P\ p'$ where $p'.s = p.s$)

The main difference with (17) is the from clause: here it is ‘from $P\ p$’ whereas in (17) it is ‘from $S\ s$’. The difference corresponds to the difference in the query: ‘skip them all if there are no battles’.

Example D. Here is another example:

Find for each type the number of its ships; skip types without ships.

Before reading our derivation of an SQL formulation in the appendix, the reader may have a try himself.

Conclusion

A lot of natural language queries on databases are quite simple and need no extensive work for the transition to SQL. However, there do exist seemingly simple but non-trivial natural

language queries, as demonstrated by the failure of quite some students and colleagues to formulate example B correctly in SQL. By means of a few examples we have shown how an SQL formulation may be calculated in a way that guarantees 100% correctness. The success of the approach lays in the fact that set and predicate notation is much more articulate (fine-grained), regular, and logically clean than SQL. Reasoning within set and predicate notation is therefore easier than reasoning within SQL and less error-prone than reasoning within natural language. Moreover, set and predicate notation is also beneficial outside the narrow area of constructing SQL queries. Thus, in our opinion, this approach to the construction of SQL queries may not be lacking in a training at university undergraduate level. The initial experiences at our courses are positive.

The derivations here have demonstrated the principles of the approach. In practice some consecutive steps will be combined as one big step, and several (big) steps will be done in one's head. The example derivations were also simple in the sense that we have used no schema properties like local checks, and global assertions.

In order to make our approach feasible not only in theory but also in practice, three aspects need further elaboration (and future research). First, the calculator needs to know many more laws from set theory and logic, and also *heuristic* rules so as to know in what direction to proceed. Good discrete math textbooks on the practical use of set theory and mathematical logic give calculation rules for set and predicate notation, while Dijkstra [7] brings this to an extreme. Also, some ad-hoc results of the calculations may be generalized to more generally applicable laws, as we did in Alternative 2 of Example A, so as to get a real *calculus*. Second, there need to be rules for dealing with SQL features (like 'distinct', '$\leq$*any*', and '$\leq$*all*'), SQL peculiarities (like *null* values and their consequences), and SQL bugs (like the *sum* of an empty table being *null* rather than 0). Third, the calculator needs to have some machine support that helps in the rewriting steps (copying of the preceding line, verification of the step). In fact, for the notation that we have used, the Z notation [14], there do exist type checkers, pretty printers, and proof helpers and verifiers [3], and semi-automatic theorem proving is nowadays a well-established field. More research is needed to make these tools ready to assist the construction of SQL queries.

What we have demonstrated is the principle of an approach that helps in the construction of SQL formulations for non-trivial queries. The approach may not be lacking in undergraduate training. Tool assistance is possible, but this needs further work for its realization.

Acknowledgment. Many thanks to Henk Blanken for comments that lead to a considerable improvement of the presentation.

References

[1] ISO/IEC 9075. ISO/IEC 9075:1992(E) Information technology - Database languages - SQL. Technical report, ANSI, 1992.

[2] Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, and Riccardo Torlone. *Database Systems*. McGraw-Hill, 2001.

[3] Jonathan Bowen. The Z notation web site. `http://www.comlab.ox.ac.uk/archive/z.html`.

[4] E.F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[5] C. J. Date. Relational calculus as an aid to effective query formulation. In C. J. Date and H. Darwen, editors, *Relational Database: Writings 1989-1991*, pages 115–131. Addison-Wesley, Reading, MA, 1992.

[6] C.J. Date and Hugh Darwen. *A Guide to the SQL Standard (Fourth Edition)*. Addison Wesley, 1996. This work contains much constructive criticism and discussion of the SQL standard, including SQL99.

[7] E.W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics*. Springer Verlag, 1990.

[8] Hector Garcia-Molina, Jeffry D. Ullman, and Jennifer Widom. *Database Systems: the Complete Book*. Prentice Hall, 2002.

[9] Michael Kifer, Arthur Bernstein, and Philip M. Lewis. *Database Systems: An Application-Oriented Approach, 2nd ed.* Addison-Wesley, 2004.

[10] D.M. Kroenke. *Database processing: fundamentals, design, implementation, 9th ed.* Prentice-Hall, Pearson Education, 2004.

[11] Philip M. Lewis, Arthur Bernstein, and Michael Kifer. *Databases and Transaction Processing: An Application-Oriented Approach*. Addison-Wesley, 2001.

[12] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 2000.

[13] A. Silberschatz, H.F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, 2001.

[14] J.M. Spivey. *The Z notation: a reference manual (2nd edition)*. Prentice Hall International, UK, 1992.

Elaboration of example B

Here is a derivation of an SQL formulation for example B: "the types of which all ships have sunk". We discuss the second step separately because, firstly, some people may find this step obvious and not needing a justification, and secondly, the elaboration is typical for an enumeration of the values of a certain attribute (here: the *type*), so that it has greater applicability than in just this calculation: it also occurs in the elaboration of example D further down. The fifth step is justified by the shunting rule (9), and the one-but-last step is an application of the one-point rule (8) with a negation at both sides:

"The types of which all ships have sunk"

=

"yield, for each type t such that all ships of type t have sunk, the value t"

= obvious — or see elaboration below near (23)

$\{c{:}C \mid$ "all ships of type $c.t$ have sunk" $\bullet\ c.t\}$

=

$$
\begin{array}{ll}
 & \{c{:}C \mid (\forall\, s{:}S \mid \text{“}s \text{ is of type } c.t\text{”} \bullet \text{“}s \text{ has sunk”}) \bullet c.t\} \\
= & \{c{:}C \mid (\forall\, s{:}S \mid (\exists\, c'{:}C \bullet s.c{=}c'.n \wedge c'.t{=}c.t) \bullet \text{“}s \text{ has sunk”}) \bullet c.t\} \\
= & \{c{:}C \mid (\forall\, s{:}S;\ c'{:}C \mid s.c{=}c'.n \wedge c'.t{=}c.t \bullet \text{“}s \text{ has sunk”}) \bullet c.t\} \\
= & \{c{:}C \mid (\forall\, s{:}S;\ c'{:}C \mid s.c{=}c'.n \wedge c'.t{=}c.t \bullet (\exists\, p{:}P \mid p.r{=}sunk \bullet p.s{=}s.n)) \bullet c.t\} \\
= & \{c{:}C \mid \neg\, (\exists\, s{:}S;\ c'{:}C \mid s.c{=}c'.n \wedge c'.t{=}c.t \bullet \neg\, (\exists\, p{:}P \mid p.r{=}sunk \bullet p.s{=}s.n)) \bullet c.t\} \\
= & \{c{:}C \mid \neg\, (\exists\, s{:}S;\ c'{:}C \mid s.c{=}c'.n \wedge c'.t{=}c.t \bullet s.n \notin \{p{:}P \mid p.r{=}sunk \bullet p.s\}) \bullet c.t\} \\
= & \text{select distinct } c.t \text{ from } C\ c \text{ where not exists (} \\
 & \quad \text{select} * \text{ from } S\ s,\ C\ c1 \text{ where} \\
 & \qquad s.c = c1.n \text{ and } c1.t = c.t \text{ and} \\
 & \qquad s.n \text{ not in (select } p.s \text{ from } P\ p \text{ where } p.r = sunk))
\end{array}
$$

The one-but-last math line also translates to SQL immediately.

Here is an alternative derivation for the given query. Expressed in natural language, the first step now says that “the type of which all ships have sunk” equals “all types except the types of ships that have not sunk”:

$$
\begin{array}{ll}
 & \{c{:}C \mid \text{“all ships of type } c.t \text{ have sunk”} \bullet c.t\} \\
= & \quad \text{law (18) discussed earlier — the condition is satisfied} \\
 & \{c{:}C \bullet c.t\} \setminus \{c{:}C \mid \neg \text{“all ships of type } c.t \text{ have sunk”} \bullet c.t\} \\
= & \quad \text{as in the previous calculation} \\
 & \{c{:}C \bullet c.t\} \setminus \\
 & \{c{:}C \mid \neg\, \neg\, (\exists\, s{:}S;\ c'{:}C \mid s.c{=}c'.n \wedge c'.t{=}c.t \bullet s.n \notin \{p{:}P | p.r{=}sunk \bullet p.s\}) \bullet c.t\} \\
= & \{c{:}C \bullet c.t\} \setminus \{c,\, c'{:}C;\ s{:}S \mid s.c{=}c'.n \wedge c'.t{=}c.t \wedge s.n \notin \{p{:}P | p.r{=}sunk \bullet p.s\} \bullet c.t\} \\
= & \{c{:}C \bullet c.t\} \setminus \{c'{:}C;\ s{:}S \mid s.c{=}c'.n \wedge (\exists\, c{:}C \bullet c'.t{=}c.t) \wedge s.n \notin \{p{:}P | p.r{=}sunk \bullet p.s\} \bullet c'.t\} \\
= & \{c{:}C \bullet c.t\} \setminus \{c'{:}C;\ s{:}S \mid s.c{=}c'.n \wedge s.n \notin \{p{:}P | p.r{=}sunk \bullet p.s\} \bullet c'.t\} \\
= & \{c{:}C \bullet c.t\} \setminus \{c\,{:}C;\ s{:}S \mid s.c{=}c\ .n \wedge s.n \notin \{p{:}P | p.r{=}sunk \bullet p.s\} \bullet c\ .t\} \\
= & (\text{select } c.t \text{ from } C\ c) \\
 & \text{except} \\
 & (\text{select } c.t \text{ from } S\ s,\ C\ c \\
 & \ \text{where } s.c = c.n \text{ and } s.n \text{ not in (select } p.s \text{ from } P\ p \text{ where } p.r = sunk))
\end{array}
$$

This same result can also be calculated in an ad-hoc way, without calling for law (18). The calculation here starts where the first calculation for the query ends:

$$
\begin{array}{ll}
 & \{c{:}C \mid \neg\, (\exists\, s{:}S;\ c'{:}C \bullet s.c{=}c'.n \wedge c'.t{=}c.t \wedge s.n \notin \{p{:}P \mid p.r{=}sunk \bullet p.s\}) \bullet c.t\} \\
= & \{c{:}C \mid \neg\, (\exists\, s{:}S;\ c'{:}C \bullet s.c{=}c'.n \wedge s.n \notin \{p{:}P \mid p.r{=}sunk \bullet p.s\} \wedge c'.t{=}c.t) \bullet c.t\} \\
= & \{c{:}C \mid \neg\, (\exists\, s{:}S;\ c'{:}C \mid s.c{=}c'.n \wedge s.n \notin \{p{:}P \mid p.r{=}sunk \bullet p.s\} \bullet c'.t{=}c.t) \bullet c.t\} \\
= & \{c{:}C \mid c.t \notin \{s{:}S;\ c'{:}C \mid s.c{=}c'.n \wedge s.n \notin \{p{:}P \mid p.r{=}sunk \bullet p.s\} \bullet c'.t\} \bullet c.t\} \\
= & \{c{:}C \mid c.t\} \setminus \{s{:}S;\ c{:}C \mid s.c{=}c.n \wedge s.n \notin \{p{:}P \mid p.r{=}sunk \bullet p.s\} \bullet c.t\}
\end{array}
$$

It remains to elaborate the second step of the first derivation above. For this, we take *Dom* to be the domain of type attributes. Similarly to the calculation in example C, the shunting

rule (9) is used twice to interchange $t{:}Dom$ and $c{:}C$; this is followed by a disappearance of $t{:}Dom$ on account of the one-point rule (8):

(23)
$$
\begin{array}{ll}
 & \text{“yield, for each type } t \text{ such that } \ldots t \ldots \text{, the value } \ldots t \ldots \text{”} \\
= & \\
 & \{t : Dom \mid \text{“}t \text{ is a type”} \qquad \wedge \ldots\ t\ \ldots \bullet \ldots\ t\ \ldots\} \\
= & \\
 & \{t : Dom \mid (\exists\, c : C \bullet t = c.t) \wedge \ldots\ t\ \ldots \bullet \ldots\ t\ \ldots\} \\
= & \\
 & \{t : Dom;\ c : C \mid t = c.t \qquad \wedge \ldots\ t\ \ldots \bullet \ldots\ t\ \ldots\} \\
= & \\
 & \{t : Dom;\ c : C \mid t = c.t \qquad \wedge \ldots c.t \ldots \bullet \ldots c.t \ldots\} \\
= & \\
 & \{c : C \mid (\exists\, t : Dom \bullet t = c.t) \wedge \ldots c.t \ldots \bullet \ldots c.t \ldots\} \\
= & \\
 & \{c : C \mid c.t \in Dom \qquad \wedge \ldots c.t \ldots \bullet \ldots c.t \ldots\} \\
= & \\
 & \{c : C \mid \qquad\qquad\quad \ldots c.t \ldots \bullet \ldots c.t \ldots\}
\end{array}
$$

The very last step is justified by the assumption that the t attribute of C has domain Dom.

Non-solutions for example B

Many wrong SQL formulations have been proposed for example B. Here we list their equivalent set expressions; for brevity the SQL formulation is omitted (except in the first case; in the other cases it is a direct translation of the first line of a calculation). To understand what the wrong SQL formulation *did* express, we have manipulated the set expression to such a form that there is a direct translation to a concise and clear expression in natural language.

$$
\begin{array}{ll}
 & \text{select } c.t \text{ from } C\ c,\ S\ s,\ P\ p \text{ where } c.n{=}s.c \text{ and } s.n{=}p.s \text{ and } p.r{=}sunk \\
= & \{c : C;\ s : S;\ p : P \mid c.n{=}s.c \wedge s.n{=}p.s \wedge p.r{=}sunk \bullet c.t\} \\
= & \{c : C \mid (\exists\, s : S;\ p : P \bullet c.n{=}s.c \wedge s.n{=}p.s \wedge p.r{=}sunk) \bullet c.t\} \\
= & \text{the types of which } \textit{some} \text{ ships have sunk.}
\end{array}
$$

$$
\begin{array}{ll}
 & \{c : C;\ s : S;\ p : P \mid c.n{=}s.c \wedge s.n{=}p.s \wedge \ldots \bullet c.t\} \\
= & \{c : C \mid (\exists\, s : S;\ p : P \bullet c.n{=}s.c \wedge s.n{=}p.s \wedge \ldots) \bullet c.t\} \\
= & \text{the types with a ship that participated in a battle and } \ldots\ .
\end{array}
$$

$$
\begin{array}{ll}
 & \{c : C;\ s : S \mid c.n{=}s.c \wedge \neg\, (\exists\, p : P \mid p.s{=}s.n \bullet p.r{=}sunk) \bullet c.t\} \\
= & \{c : C;\ s : S \mid c.n{=}s.c \wedge (\forall\, p : P \mid p.s{=}s.n \bullet p.r{\neq}sunk) \bullet c.t\} \\
= & \{c : C \mid (\exists\, s : S \bullet c.n{=}s.c \wedge (\forall\, p : P \mid p.s{=}s.n \bullet p.r{\neq}sunk)) \bullet c.t\} \\
= & \text{the types with a ship that survived each battle in which it participated} \\
= & \text{the types with a ship that has not sunk.}
\end{array}
$$

$$
\begin{array}{ll}
 & \{c : C \mid \neg\, (\exists\, s : S;\ p : P \bullet c.n{=}s.c \wedge s.n{=}p.s \wedge p.r{=}sunk) \bullet c.t\} \\
= & \{c : C \mid \neg\, (\exists\, s : S \mid c.n{=}s.c \bullet \exists\, p : P \bullet s.n{=}p.s \wedge p.r{=}sunk) \bullet c.t\} \\
= & \{c : C \mid (\forall\, s : S \mid c.n{=}s.c \bullet \neg\ \exists\, p : P \bullet s.n{=}p.s \wedge p.r{=}sunk) \bullet c.t\} \\
= & \{c : C \mid (\forall\, s : S \mid c.n{=}s.c \bullet \neg\ \exists\, p : P \mid s.n{=}p.s \bullet p.r{=}sunk) \bullet c.t\} \\
= & \{c : C \mid (\forall\, s : S \mid c.n{=}s.c \bullet \forall\, p : P \mid s.n{=}p.s \bullet p.r \neq sunk) \bullet c.t\} \\
= & \text{the types of classes of which all ships have not sunk.}
\end{array}
$$

$$
\begin{array}{ll}
 & \{c : C \mid \neg\, (\exists\, s : S \mid c.n{=}s.c \bullet \neg\ \exists\, p : P \bullet s.n{=}p.s \wedge p.r{=}sunk) \bullet c.t\} \\
= & \{c : C \mid (\forall\, s : S \mid c.n{=}s.c \bullet \exists\, p : P \bullet s.n{=}p.s \wedge p.r{=}sunk) \bullet c.t\} \\
= & \text{the types of classes of which all ships have sunk.}
\end{array}
$$

$\{c : C \mid \neg\ (\exists\, s : S;\ p : P \bullet c.n{=}s.c \wedge s.n{=}p.s \wedge p.r \neq sunk) \bullet c.t\}$
= the types of classes that have no battle-surviving ships
$= \{c : C \mid \neg\ (\exists\, s : S \mid c.n{=}s.c \bullet \exists\, p : P \bullet s.n{=}p.s \wedge p.r \neq sunk) \bullet c.t\}$
$= \{c : C \mid (\forall\, s : S \mid c.n{=}s.c \bullet \neg\ \exists\, p : P \bullet s.n{=}p.s \wedge p.r \neq sunk) \bullet c.t\}$
$= \{c : C \mid (\forall\, s : S \mid c.n{=}s.c \bullet \neg\ \exists\, p : P \mid s.n{=}p.s \bullet p.r \neq sunk) \bullet c.t\}$
$= \{c : C \mid (\forall\, s : S \mid c.n{=}s.c \bullet \forall\, p : P \mid s.n{=}p.s \bullet p.r{=}sunk) \bullet c.t\}$
= the types of classes of which each ship has sunk in every battle in which it participated.

$\{c{:}C \mid (\forall\, s{:}S \mid (\exists\, c'{:}C \bullet c.t{=}c'.t \wedge c'.n{=}s.c) \bullet \forall\, p{:}P \mid s.n{=}p.s \bullet p.r{=}sunk) \bullet c.t\}$
= the types of which each ship has sunk in every battle in which it participated
$= \{c{:}C \mid \neg\ (\exists\, s{:}S \mid (\exists\, c'{:}C \bullet c.t{=}c'.t \wedge c'.n{=}s.c) \bullet \neg\ \forall\, p{:}P \mid s.n{=}p.s \bullet p.r{=}sunk) \bullet c.t\}$
= the types of which no ship survived a battle
$= \{c : C \mid \neg\ (\exists\, c'{:}C;\ s{:}S \mid c.t{=}c'.t \wedge c'.n{=}s.c \bullet \neg\ \forall\, p{:}P \mid s.n{=}p.s \bullet p.r{=}sunk) \bullet c.t\}$
$= \{c : C \mid \neg\ (\exists\, c' : C;\ s : S \mid c.t{=}c'.t \wedge c'.n{=}s.c \bullet \exists\, p : P \mid s.n{=}p.s \bullet p.r \neq sunk) \bullet c.t\}$
$= \{c : C \mid \neg\ (\exists\, c' : C;\ s : S \bullet c.t{=}c'.t \wedge c'.n{=}s.c \wedge \exists\, p : P \mid s.n{=}p.s \bullet p.r \neq sunk) \bullet c.t\}$
$= \{c : C \mid \neg\ (\exists\, c' : C;\ s : S;\ p : P \bullet c.t{=}c'.t \wedge c'.n{=}s.c \wedge s.n{=}p.s \wedge p.r \neq sunk) \bullet c.t\}$
$= \{c : C \mid c.t \notin \{c' : C;\ s : S;\ p : P \mid c'.n{=}s.c \wedge s.n{=}p.s \wedge p.r \neq sunk \bullet c'.t\} \bullet c.t\}$
$= \{c : C \bullet c.t\} \setminus \{c' : C;\ s : S;\ p : P \mid c'.n{=}s.c \wedge s.n{=}p.s \wedge p.r \neq sunk \bullet c'.t\}$
= all types except for the types with a ship that survived a battle
$\supset$ the types of which each ship has sunk.

$\{c : C \bullet c.t\} \setminus \{c' : C;\ s : S;\ p : P \mid c'.n{=}s.c \wedge s.n{=}p.s \wedge p.r{=}sunk \bullet c'.n \bullet c.t\}$
= all types except for those that have a sunken ship
$= \{c : C \mid c.t \notin \{c' : C;\ s : S;\ p : P \mid c'.n{=}s.c \wedge s.n{=}p.s \wedge p.r{=}sunk \bullet c'.n\} \bullet c.t\}$
$= \{c : C \mid \neg\ (\exists\, c' : C;\ s : S;\ p : P \bullet c.t{=}c'.t \wedge c'.n{=}s.c \wedge s.n{=}p.s \wedge p.r{=}sunk) \bullet c.t\}$
= the types of which no ship has sunk.

Elaboration of example D

Here is a derivation of an SQL formulation for example D: "find for each type the number of its ships; skip types without ships". Up to step (24) the calculation is quite standard, using the shunting rule (9) several times. Then, just before step (24) we recognize the group-by form: the outer set expression and the inner one start similarly. At that point we choose to "optimize" the subexpression $\#\{s'{:}S;\ c'{:}C \mid \ldots \bullet s'\}$ to $\#\{s'{:}S;\ c'{:}C \mid \ldots \bullet (s', c')\}$ in order that the count will appear as 'count $(*)$' in the SQL formulation rather than as 'count $(s'.n)$'. This optimization is possible since "the c' for s'" is uniquely determined.

= "Find for each type the number of its ships; skip types without ships"
"yield, for each type with at least one ship, the type and the number of its ships"
= obvious — or see (23) again
$\{c'' : C \mid$ "there is a ship of type $c''.t$" $\bullet\ (c''.t,\ \#$"ships of type $c''.t$"$)\}$
$= \{c'' : C \mid (\exists\, s : S \bullet$ "s is of type $c''.t$"$) \bullet (c''.t,\ \#$"ships of type $c''.t$"$)\}$
$= \{c'' : C \mid (\exists\, s : S \bullet \exists\, c : C \bullet s.c = c.n \wedge c.t = c''.t) \bullet (c''.t,\ \#$"ships of type $c''.t$"$)\}$
$= \{c'' : C;\ s : S;\ c : C \mid s.c = c.n \wedge c.t = c''.t \bullet (c''.t,\ \#$"ships of type $c''.t$"$)\}$
=

$$
\begin{array}{ll}
 & \{c'' : C;\, s : S;\, c : C \mid s.c = c.n \wedge c.t = c''.t \bullet (\ c.t\ ,\ \#\text{“ships of type }\ c.t\ \text{”})\} \\
= & \\
 & \{s : S;\, c : C \mid s.c = c.n \wedge (\exists\, c'' : C \bullet c.t = c''.t) \bullet (c.t,\ \#\text{“ships of type } c.t\text{”})\} \\
= & \quad c \text{ itself satisfies the condition for } c'', \text{ hence } (\exists\, c'' \ldots) = \mathit{true} \\
 & \{s : S;\, c : C \mid s.c = c.n \bullet (c.t,\ \#\text{“ships of type } c.t\text{”})\} \\
= & \\
 & \{s : S;\, c : C \mid s.c = c.n \bullet (c.t,\ \#\{s' : S \mid (\exists\, c' : C \mid s'.c = c'.n \bullet c'.t = c.t) \bullet s'\})\} \\
= & \\
 & \{s : S;\, c : C \mid s.c = c.n \bullet (c.t,\ \#\{s' : S;\, c' : C \mid s'.c = c'.n \wedge c'.t = c.t \bullet s'\})\} \\
(24) \quad = & \quad \text{each ship } s' \text{ belongs to exactly one class } c' \text{ — see } \mathit{Note} \text{ below} \\
 & \{s : S;\, c : C \mid s.c = c.n \bullet (c.t,\ \#\{s' : S;\, c' : C \mid s'.c = c'.n \wedge c'.t = c.t \bullet (s', c')\})\} \\
= & \\
 & \{s : S;\, c : C \mid s.c = c.n \bullet (c.t,\ \#G)\} \\
 & \qquad\qquad \text{where } G \text{ stands for } \{s' : S;\, c' : C \mid s'.c = c'.n \wedge c'.t = c.t \bullet (s', c')\} \\
= & \\
 & \text{select } c.t, \text{ count } (*) \text{ from } S\ s,\ C\ c \text{ where } s.c = c.n \text{ group by } c.t
\end{array}
$$

Note. Step (24) is an instance of law (16):

$$\#\{D \mid P \bullet E\} \quad = \quad \#\{D \mid P \bullet E'\}, \quad \text{provided } \{D \mid P \bullet (E, E')\} \text{ is a bijection}$$

In the above step, E is the expression s', and E' is the expression (s', c'). Since predicate P says that $s'.c = c'.n$ and since n is a key in C, the value of E functionally determines the value of E'; the converse is obvious.

The derived SQL formulation is not correct if the directive “skip types without ships” is left out of the query. On account of equation (21), the query is equivalent to the following (more expensive) group-less formulation:

```
select distinct
        c.t,  (select count (*) from S s1, C c1 where s1.c = c1.n and c1.t = c.t)
from S s, C c where s.c = c.n
```