Expressive Power of SQL

Leonid Libkin[1]

University of Toronto and Bell Laboratories
Email: `libkin@cs.toronto.edu`

Abstract. It is a folk result in database theory that SQL cannot express recursive queries such as reachability; in fact, a new construct was added to SQL3 to overcome this limitation. However, the evidence for this claim is usually given in the form of a reference to a proof that relational algebra cannot express such queries. SQL, on the other hand, in all its implementations has three features that fundamentally distinguish it from relational algebra: namely, grouping, arithmetic operations, and aggregation.

In the past few years, most questions about the additional power provided by these features have been answered. This paper surveys those results, and presents new simple and self-contained proofs of the main results on the expressive power of SQL. Somewhat surprisingly, tiny differences in the language definition affect the results in a dramatic way: under some very natural assumptions, it can be proved that SQL cannot define recursive queries, no matter what aggregate functions and arithmetic operations are allowed. But relaxing these assumptions just a tiny bit makes the problem of proving expressivity bounds for SQL as hard as some long-standing open problems in complexity theory.

1 Introduction

What queries can one express in SQL? Perhaps more importantly, one would like to know what queries cannot be expressed in SQL – after all, it is the inability to express certain properties that motivates language designers to add new features (at least one hopes that this is the case).

This seems to be a rather basic question that database theoreticians should have produced an answer to by the beginning of the 3rd millennium. After all, we've been studying the expressive power of query languages for some 20 years now (and in fact more than that, if you count earlier papers by logicians on the expressiveness of first-order logic), and SQL is the de-facto standard of the commercial database world – so there surely must be an answer somewhere in the literature.

When one thinks of the limitations of SQL, its inability to express reachability queries comes to mind, as it is well documented in the literature (in fact, in many database books written for very different audiences, e.g. [1, 5, 7, 25]). Let us consider a simple example: suppose that `R(Src,Dest)` is a relation with flight information: `Src` stands for source, and `Dest` for destination. To find pairs of cities (A, B) such that it is possible to fly from A to B with one stop, one would use a self-join:

```
SELECT R1.Src, R2.Dest
FROM   R AS R1, R AS R2
WHERE  R1.Dest=R2.Src
```

What if we want pairs of cities such that one makes two stops on the way? Then we do a more complicated self-join:

```
SELECT R1.Src, R3.Dest
FROM   R AS R1, R AS R2, R AS R3
WHERE  R1.Dest=R2.Src AND R2.Dest=R3.Src
```

Taking the union of these two and the relation `R` itself we would get the pairs of cities such that one can fly from A to B with *at most* two stops. But often one needs a general reachability query in which no a priori bound on the number of stops is known; that is, whether it possible to get to B from A.

Graph-theoretically, this means computing the transitive closure of `R`. It is well known that the transitive closure of a graph is *not* expressible in relational algebra or calculus; in particular, expressions similar to those above (which happen to be unions of conjunctive queries) cannot possibly express it. This appears to be a folk result in the database community; while many papers do refer to [2] or some other source on the expressive power of first-order logic, many texts just state that relational algebra, calculus and SQL cannot express recursive queries such as reachability.

With this limitation in mind, the SQL3 standard introduced recursion explicitly into the language [7, 12]. One would write the reachability query as

```
WITH RECURSIVE TrCl(Src,Dest) AS
      R
    UNION
      SELECT TrCl.Src, R.Dest
      FROM   TrCl, R
      WHERE  TrCl.Dest = R.Src
SELECT * FROM TrCl
```

This simply models the usual datalog rules for transitive closure:

$$\begin{aligned} &\mathit{trcl}(x,y) \text{ :- } r(x,y) \\ &\mathit{trcl}(x,y) \text{ :- } \mathit{trcl}(x,z), r(z,y) \end{aligned}$$

When a new construct is added to a language, a good reason must exist for it, especially if the language is a declarative query language, with a small number of constructs, and with programmers relying heavily on its optimizer. The reason for introducing recursion in the next SQL standard is precisely this folk result stating that it cannot be expressed in the language. But when one looks at what evidence is provided to support this claim, one notices that all the references point to papers in which it is proved that *relational algebra and* *calculus* cannot express recursive queries. Why is this not sufficient? Consider the following query

```
SELECT R1.A
FROM   R1, R2
WHERE (SELECT COUNT(*) FROM R1) >
         (SELECT COUNT(*) FROM R2)
```

This query tests if |R1|>|R2|: in that case, it returns the A attribute of R1, otherwise it returns the empty set. However, logicians proved it long time ago that first-order logic, and thus relational calculus, cannot compare cardinalities of relations, and yet we have a very simple SQL query doing precisely that.

The conclusion, of course, is that SQL has more power than relational algebra, and the main source of this additional power is its *aggregation* and *grouping* constructs, together with *arithmetic* operations on numerical attributes. But then one cannot say that the transitive closure query is not expressible in SQL simply because it is inexpressible in relational algebra. Thus, it might appear that the folk theorem about recursion and SQL is an unproven statement.

Fortunately, this is not the case: the statement was (partially) proved in the past few years; in fact, a series of papers proved progressively stronger results, finally establishing good bounds on the expressiveness of SQL.

My main goal here is twofold:

(a) I give an overview of these recent results on the expressiveness of SQL. We shall see that some tiny differences in the language definition affect the results in a dramatic way: under some assumptions, it can be shown that reachability and many other recursive queries aren't expressible in SQL. However, under a slightly different set of assumptions, the problem of proving expressivity bounds for SQL is as hard as separating some complexity classes.
(b) Due to a variety of reasons, even the simplest proofs of expressivity results for SQL are not easy to follow; partly this is due to the fact that most papers used the setting of their predecessors that had unnecessary complications in the form of nested relations, somewhat unusual (for mainstream database people) languages and infinitary logics. Here I try to get rid of those complications, and present a simple and self-contained proof of expressivity bounds for SQL.

Organization In the next section, we discuss the main features that distinguish SQL from relational algebra, in particular, aggregate functions. We then give a brief overview of the literature on the expressive power of SQL.

Starting with Section 3, we present those results in more detail. We introduce relational algebra with grouping and aggregates, $\mathrm{ALG}_{\mathrm{aggr}}$, that essentially captures basic SQL statements. Section 4 states the main result on the expressive power of SQL, namely that queries it can express are *local*. If one thinks of queries on graphs, it means that the decision whether a tuple $\vec{t}$ belongs to the output is determined by a small neighborhood of $\vec{t}$ in the input graph; the reachability query does not have this property.

Section 5 defines an *aggregate logic* $\mathcal{L}_{\text{aggr}}$ and shows a simple translation of the algebra with aggregates ALG_{aggr} into this logic. Then, in Section 6, we present a self-contained proof of locality of $\mathcal{L}_{\text{aggr}}$ (and thus of ALG_{aggr}).

In Section 7, we consider an extension $\text{ALG}^{<}_{\text{aggr}}$ of ALG_{aggr} in which non-numerical order comparisons are allowed, and show that it is more powerful than the unordered version. Furthermore, no nontrivial bounds on the expressiveness of this language can be proved without answering some deep open problems in complexity theory.

Section 8 gives a summary and concluding remarks.

2 SQL vs. Relational Algebra

What exactly is SQL? There is, of course, a very long standard, that lists numerous features, most of which have very little to do with the expressiveness of queries. As far as expressiveness is concerned, the main features that distinguish SQL from relational algebra, are the following:

– Aggregate functions: one can compute, for example, the average value in a column. The standard aggregates in SQL are `COUNT`, `TOTAL`, `AVG`, `MIN`, `MAX`.
– Grouping: not only can one compute aggregates, one can also group them by values of different attributes. For example, it is possible to compute the average salary for each department.
– Arithmetic: SQL allows one to apply arithmetic operations to numerical values.

For example, for relations `S1(Empl,Dept)` and `S2(Empl,Salary)`, the following query (assuming that `Empl` is a key for both relations) computes the average salary for each department which pays total salary at least 100,000:

```
        SELECT  S1.Dept, AVG(S2.Salary)
        FROM    S1, S2
(*)     WHERE   S1.Empl=S2.Empl
        GROUPBY S1.Dept
        HAVING  TOTAL(S2.Salary) > 100000
```

Next, we address the following question: what is an aggregate function? The first paper to look into this was probably [20]: it defined aggregate functions as $f : \mathcal{R} \to \mathsf{Num}$, where $\mathcal{R}$ is the set of all relations, and Num is a numerical domain. A problem with this approach is that it requires a different aggregate function for each relation and each numerical attribute in it; that is, we do not have just one aggregate `AVG`, but infinitely many of those. This complication arises from dealing with duplicates in a correct manner. However, duplicates can be incorporated in a much more elegant way, as suggested in [14], which we shall follow here. According to [14], an aggregate function $\mathcal{F}$ is a collection

$$\mathcal{F} = \{f_0, f_1, f_2, \ldots, f_\omega\}$$

where f_k is a function that takes a k-element multiset (bag) of elements of Num and produces an element of Num. For technical reasons, we also add a constant

$f_\omega \in \mathsf{Num}$ whose intended meaning is the value of $\mathcal{F}$ on infinite multisets. For example, if Num is $\mathbb{N}$, or $\mathbb{Q}$, or $\mathbb{R}$, we define the aggregate $\sum = \{s_0, s_1, \ldots\}$ by $s_k(\{\!|x_1, \ldots, x_k|\!\}) = \sum_{i=1}^{k} x_i$; furthermore, $s_0 = s_\omega = 0$ (we use the $\{\!|\ |\!\}$ brackets for multisets). This corresponds to SQL's `TOTAL`. For `COUNT`, one defines $\mathcal{C} = \{c_0, c_1, \ldots\}$ with c_k returning k (we may again assume $c_\omega = 0$). The aggregate `AVG` is defined as $\mathcal{A} = \{a_0, a_1, \ldots\}$ with $a_k(X) = \frac{s_k(X)}{c_k(X)}$, $a_0 = a_\omega = 0$.

Languages that model SQL and their expressive power

It is very hard to prove formal statements about a language like SQL: to put it mildly, its syntax is not very easy to reason about. The research community has come up with several proposals of languages that capture the expressiveness of SQL. The earliest one is perhaps Klug's extension of relational algebra by grouping and aggregation [20]: if e is an expression producing a relation with m attributes, $\vec{A}$ is a set of attributes, and f is an aggregate function, then $e\langle \vec{A}, f\rangle$ is a new expression that produces a relation with $m+1$ attributes. Assuming f applies to attribute A', and $\vec{B}$ is the list of all attributes of the output of e, the semantics is best explained by SQL:

`SELECT` $\vec{B}$, $f(A')$
`FROM` e
`GROUPBY` $\vec{A}$

Klug's paper did not analyze the expressive power of this algebra, nor did it show how to incorporate arithmetic operations. The main contribution of [20] is an equivalence result between the algebra and an extension of relational calculus. However, the main focus of that extension is its safety, and the resulting logic is extremely hard to deal with, due to many syntactic restrictions.

To the best of my knowledge, the first paper that directly addressed the problem of the expressive power of SQL, was the paper by Consens and Mendelzon in ICDT'90 [6]. They have a datalog-like language, whose nonrecursive fragment is exactly as expressive as Klug's algebra. Then they show that this language cannot express the transitive closure query under the assumption that DLOGSPACE is properly included in NLOGSPACE. The reason is simple: Klug's algebra (with some simple aggregates) can be evaluated in DLOGSPACE, while transitive closure is complete for NLOGSPACE.

That result can be viewed as a strong evidence that SQL is indeed incapable of expressing reachability queries. However, it is not completely satisfactory for three reasons. First, nobody knows how to separate complexity classes. Second, what if one adds more complex aggregates that increase the complexity of query evaluation? And third, what if the input graph has a very simple structure (for example, no node has outdegree more than 1)? In this case reachability is in DLOGSPACE, and the argument of [6] does not work.

In early 90s, many people were looking into languages for collection types. Functional statically typechecked query languages became quite fashionable, and they were produced in all kinds of flavors, depending on particular collection types they had to support. It turned out that a set language capturing essentially the expressive power of a language for bags, could also model all the essential features of SQL [23]. The problem was that the language dealt with nested relations, or complex objects. But then [23] proved a *conservativity* result, stating that nested relations aren't really needed if the input and output don't have them. That made it possible to use a non-nested fragment of languages inspired by structural recursion [4] and comprehensions [28] as a "theoretical reconstruction of SQL."

Several papers dealt with this language, and proved a number of expressivity bounds. The first one, appearing in PODS'94 [23], showed that the language could not express reachability queries. The proof, however, was very far from ideal. It only proved inexpressibility of transitive closure in a way that was very unlikely to extend to other queries. It relied on a complicated syntactic rewriting that wouldn't work even for a slightly different language. And the proof wouldn't work if one added more aggregate functions.

The first limitation was addressed in [8] where a certain general property of queries expressible in SQL was established. However, the other two problems not only remained, but were exacerbated: the rewriting of queries became particularly unpleasant. In an attempt to remedy this, [21] gave an indirect encoding of a fragment of SQL into first-order logic with counting, FO($\mathbf{C}$) (it will be formally defined later). The restriction was to natural numbers, thus excluding aggregates such as `AVG`. The encoding is bound to be indirect, since SQL is capable of expressing queries that FO($\mathbf{C}$) cannot express. The encoding showed that for any query Q in SQL, there exists a FO($\mathbf{C}$) query Q' that shares some nice properties with Q. Then [21] established some properties of FO($\mathbf{C}$) queries and transferred them to that fragment of SQL. The proof was much cleaner than the proofs of [23, 8], at the expense of a less expressive language.

After that, [24] showed that the coding technique can be extended to SQL with rational numbers and the usual arithmetic operations. The price to pay was the readability of the proof – the encoding part became very unpleasant.

That was a good time to pause and see what must be done differently. How do we prove expressivity bounds for relational algebra? We do it by proving bounds on the expressiveness of first-order logic (FO) over finite structures, since relational algebra has the same power as FO. So perhaps if we could put aggregates and arithmetic directly into logic, we would be able to prove expressivity bounds in a nice and simple way?

That program was carried out in [18], and I'll survey the results below. One problem with [18] is that it inherited too much unnecessary machinery from its predecessors [23, 8, 24, 21, 22]: one had to deal with languages for complex objects and apply conservativity results to get down to SQL; logics were infinitary to start with, although infinitary connectives were not necessary to translate SQL; and expressivity proofs went via a special kind of games invented elsewhere [16].

Here we show that all these complications are completely unnecessary: there is indeed a very simple proof that reachability is not expressible in SQL, and this proof will be presented below. Our language is a slight extension of Klug's algebra (no nesting!). We translate it into an aggregate logic (with no infinitary connectives!) and prove that it has nice locality properties (without using games!)

3 Relational algebra with aggregates

To deal with aggregation, we must distinguish numerical columns (to which aggregates can be applied) from non-numerical ones. We do it by typing: a type of a relation is simply a list of types of its attributes.

We assume that there are two base types: a non-numerical type b with domain Dom, and a numerical type n, whose domain is denoted by Num (it could be $\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{R}$, for example).

A type of a relation is a string over the alphabet $\{\mathsf{b}, \mathsf{n}\}$. A relation R of type $a_1 \ldots a_m$ has m columns, the ith one containing entries of type a_i. In other words, such a relation is a finite subset of

$$\prod_{i=1}^{m} \mathrm{dom}(a_i)$$

where $\mathrm{dom}(\mathsf{b}) = \mathsf{Dom}$ and $\mathrm{dom}(\mathsf{n}) = \mathsf{Num}$. For example, the type of `S2(Empl,Salary)` is bn. For a type t, $t.i$ denotes the ith position in the string. The length of t is denoted by $|t|$.

A *database schema* SC is a collection of relation names R_i and their types t_i; we write $R_i : t_i$ if the type of R_i is t_i.

Next we define expressions of relational algebra with aggregates, parameterized by a collection Ω of functions and predicates on Num, and a collection Θ of aggregates, over a given schema SC. Expressions are divided into three groups: the standard relational algebra, arithmetic, and aggregation/grouping. In what follows, m stands for $|t|$, and $\{i_1, \ldots, i_k\}$ for a sequence $1 \le i_1 < \ldots < i_k \le m$.

Relational Algebra

SCHEMA RELATION If $R : t$ is in SC, then R is an expression of type t.

PERMUTATION If e is an expression of type t and θ is a permutation of $\{1, \ldots, m\}$, then $\rho_\theta(e)$ is an expression of type $\theta(t)$.

BOOLEAN OPERATIONS If e_1, e_2 are expressions of type t, then so are $e_1 \cup e_2, e_1 \cap e_2, e_1 - e_2$.

CARTESIAN PRODUCT For $e_1 : t_1$, $e_2 : t_2$, $e_1 \times e_2$ is an expression of type $t_1 \cdot t_2$.

PROJECTION If e is of type t, then $\pi_{i_1,\ldots,i_k}(e)$ is an expression of type t' where t' is the string composed of $t.i_j$s, in their order.

SELECTION If e is an expression of type t, $i, j \le m$, and $t.i = t.j$, then $\sigma_{i=j}(e)$ is an expression of type t.

Arithmetic

NUMERICAL SELECTION If $P \subseteq \mathsf{Num}^k$ is a k-ary numerical predicate from Ω, and $i_1, \ldots, i_k$ are such that $t.i_j = \mathsf{n}$, then $\sigma[P]_{i_1,\ldots,i_k}(e)$ is an expression of type t for any expression e of type t.

Function Application If $f : \mathsf{Num}^k \to \mathsf{Num}$ is a function from Ω, $i_1, \ldots, i_k$ are such that $t.i_j = \mathsf{n}$, and e is an expression of type t, then $\mathrm{Apply}[f]_{i_1,\ldots,i_k}(e)$ is an expression of type $t \cdot \mathsf{n}$. If $k = 0$ (i.e. f is a constant), then $\mathrm{Apply}[f]_\epsilon(e)$ is an expression of type $t \cdot \mathsf{n}$.

Aggregation and Grouping

Aggregation Let $\mathcal{F}$ be an aggregate from Θ. For any expression e of type t and i such that $t.i = \mathsf{n}$, $\mathrm{Aggr}[i : \mathcal{F}](e)$ is an expression of type $t \cdot \mathsf{n}$.

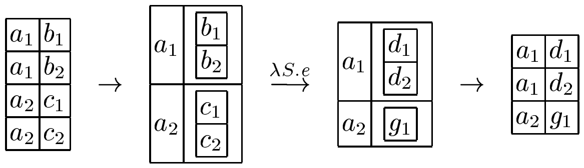
Grouping Assume $e : u$ is an expression over $SC \cup \{S : s\}$. Let e' be an expression of type $t \cdot s$ over SC, where $| t |= l$. Then $\mathrm{Group}_l[\lambda S.e](e')$ is an expression of type $t \cdot u$.

Semantics For the relational algebra operations, this is standard. The operation ρ_θ is permutation: each tuple $(a_1, \ldots, a_m)$ is replaced by $(a_{\theta(1)}, \ldots, a_{\theta(m)})$. The condition $i = j$ in the selection predicate means equality of the ith and the jth attribute: $(a_1, \ldots, a_m)$ is selected if $a_i = a_j$. Note that using Boolean operations we can model arbitrary combinations of equalities and disequalities among attributes.

For numerical selection, $\sigma[P]_{i_1,\ldots,i_k}$ selects $(a_1, \ldots, a_m)$ iff $P(a_{i_1}, \ldots, a_{i_k})$ holds. Function application replaces each $(a_1, \ldots, a_m)$ with $(a_1, \ldots, a_m, f(a_{i_1}, \ldots, a_{i_k}))$.

The aggregate operation is SQL `SELECT` $\vec{A}, \mathcal{F}(A_i)$ `FROM` e, where $\vec{A} = (A_1, \ldots, A_m)$ is the list of attributes. More precisely, if e evaluates to $\vec{a}_1, \ldots, \vec{a}_p$ where $\vec{a}_j = (a_j^1, \ldots, a_j^m)$, then $\mathrm{Aggr}[i : \mathcal{F}](e)$ replaces each $\vec{a}_j$ with $(a_j^1, \ldots, a_j^m, f)$ where $f = \mathcal{F}(\{\!|a_1^i, \ldots, a_p^i|\!\})$.

Finally, $\mathrm{Group}_l[\lambda S.e](e')$ groups the tuples by the values of their first l attributes and applies e to the sets formed by this grouping. For example:



assuming that e returns $\{d_1, d_2\}$ when $S = \{b_1, b_2\}$, and e returns $\{g_1\}$ for $S = \{c_1, c_2\}$.

Formally, let e' evaluate to $\{\vec{a}_1, \ldots, \vec{a}_p\}$. We split each tuple $\vec{a}_j = (a_j^1, \ldots, a_j^m)$ into $\vec{a}'_j = (a_j^1, \ldots, a_j^l)$ that contains the first l attributes, and $\vec{a}''_j = (a_j^{l+1}, \ldots, a_j^m)$ that contains the remaining ones. This defines, for each $\vec{a}_j$, a set $S_j = \{\vec{a}''_r \mid \vec{a}'_r = \vec{a}'_j\}$. Let $T_j = \{\vec{b}_j^1, \ldots, \vec{b}_j^{m_j}\}$ be the result of applying e with S interpreted as S_j. Then $\mathrm{Group}_l[\lambda S.e](e')$ returns the set of tuples of the form $(\vec{a}'_j, \vec{b}_j^i)$, $1 \leq j \leq p$, $1 \leq i \leq m_j$.

Klug's algebra It combines grouping and aggregation in the same operation as follows:

GROUPING & AGGREGATION Let t be of length m. Let $l < i_1 < \ldots < i_k$ with $t.i_j = \mathsf{n}$, and let $\mathcal{F}_1, \ldots, \mathcal{F}_k$ be aggregates from Θ. Then, for e an expression of type t, $\mathrm{Aggr}_l[i_1 : \mathcal{F}_1, \ldots, i_k : \mathcal{F}_k]$ is an expression of type $t \cdot \mathsf{n} \ldots \mathsf{n}$ (t with k ns added at the end).

The semantics is best explained by SQL:

SELECT $\#1, \ldots, \#m, \mathcal{F}_1(\#i_1), \ldots, \mathcal{F}_k(\#i_k)$
FROM E
GROUPBY $\#1, \ldots, \#l$

where E is the result of the expression e. (As presented in [20], the algebra does not have arithmetic operations, and the aggregates are limited to the standard five.)

Note that there are no higher-order operators in Klug's algebra, and that it is expressible in our algebra with aggregates, as $\mathrm{Aggr}_l[i_1 : \mathcal{F}_1, \ldots, i_k : \mathcal{F}_k](e')$ is equivalent to $\mathrm{Group}_l[\lambda S.e](e')$, where e is

$$\mathrm{Aggr}[i_k - l : \mathcal{F}_k](\mathrm{Aggr}[i_{k-1} - l : \mathcal{F}_{k-1}](\cdots(\mathrm{Aggr}[i_1 - l : \mathcal{F}_1](S))\cdots))$$

Example The query $(*)$ from Section 2 is defined by the following expression (which uses the operator combining grouping with aggregation):

$$\pi_{1,4}(\sigma[> 100000]_5((\mathrm{Aggr}_1[3 : \mathcal{A}, 3 : \Sigma](\pi_{2,3,4}(\sigma_{1=3}(S_1 \times S_2))))))$$

where $\mathcal{A}$ is the aggregate AVG, $\sum$ is TOTAL, and > 100000 is a unary predicate on $\mathbb{N}$ which holds of numbers $n > 100000$.

Example The only aggregate that can be applied to non-numerical attributes in SQL is COUNT that returns the cardinality of a column. It can be easily expressed in $\mathrm{ALG}_{\mathrm{aggr}}$ as long as the summation aggregate $\sum$ and constant 1 are present. We show how to define $\mathrm{Count}_m(e)$:

SELECT $\#1, \ldots, \#m-1,$COUNT$(\#m)$
FROM E
GROUPBY $\#1, \ldots, \#m$

First, we add a new column, whose elements are all 1s: $e_1 = \mathrm{Apply}[1]_\epsilon(e)$. Then define an expression $e' = \mathrm{Aggr}[2 : \Sigma](S)$, and use it to produce

$$e_2 \;=\; \mathrm{Group}_{m-1}[\lambda S.e'](e_1).$$

This is almost the answer: there are extra 2 attributes, the mth attribute of e, and those extra 1s. So finally we have

$$\mathrm{Count}_m(e) \;=\; \pi_{1,\ldots,m-1,m+2}(\mathrm{Group}_{m-1}[\lambda S.\mathrm{Aggr}[2 : \Sigma](S)](\mathrm{Apply}[1]_\epsilon(e)))$$

Remark In previous papers on the expressive power of SQL [23, 24, 21, 18], we used languages of a rather different flavor, based on structural recursion [4] and comprehensions [28]. One can show, however, that those language and $\mathrm{ALG}_{\mathrm{aggr}}$ have the same expressiveness, provided they are supplied with the same set of aggregates and arithmetic functions. The proof of this will be given in the full version.

4 Locality of SQL queries

What kind of general statement can one provide that would give us strong evidence that SQL cannot express recursive queries? For that purpose, we shall use the *locality* of queries. Locality was the basis of a number of tools for proving expressivity bounds of first-order logic [15, 13, 11], and it was recently studied on its own and applied to more expressive logics [17, 22].

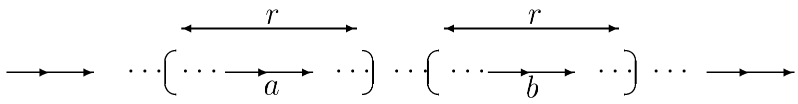


Fig. 1. A local formula cannot distinguish (a, b) from (b, a).

The general idea of this notion is that a query can only look at a small portion of its input. If the input is a graph, "small" means a neighborhood of a fixed radius. For example, Fig. 1 shows that reachability is not local: just take a graph like the one shown in the picture so that there would be two points whose distance from the endpoints and each other is more than $2r$, where r is the fixed radius. Then locality of query says that (a, b) and (b, a) are indistinguishable, as the query can only look at the r-neighborhoods of a and b. Transitive closure, on the other hand, does distinguish between (a, b) and (b, a), since b is reachable from a but not vice versa.

We now define locality formally. We say that a schema SC is *purely relational* if there are no occurrences of the numerical type n in it. Let us first restrict our attention to graph queries. Suppose we have a purely relational schema $R : \mathsf{bb}$; that is, the relation R contains edges of a directed graph. Suppose e is an expression of the same type bb; that is, it returns a directed graph. Given a pair of nodes a, b in R, and a number $r > 0$, the *r-neighborhood of a, b in R*, $N_r^R(a, b)$, is the subgraph on the set of nodes in R whose distance from either a or b is at most r. The distance is measured in the undirected graph corresponding to R, that is, $R \cup R^{-1}$.

We write $(a, b) \approx_r^R (c, d)$ when the two neighborhoods, $N_r^R(a, b)$ and $N_r^R(c, d)$, are isomorphic; that is, when there exists a (graph) isomorphism h between them such that $h(a) = c, h(b) = d$. Finally, we say that e is *local* if there is a number r, depending on e only, such that

$$(a, b) \approx_r^R (c, d) \quad\Rightarrow\quad (a, b) \in e(R) \text{ iff } (c, d) \in e(R).$$

We have seen that reachability is not local. Another example of a non-local query is a typical example of recursive query called *same-generation*:

$$\begin{array}{l} sg(x, x) \text{ :- } \\ sg(x, y) \text{ :- } R(x', x), R(y', y), sg(x', y') \end{array}$$

This query is not local either: consider, for example, a graph consisting of two chains: $(a, b_1), (b_1, b_2), \ldots, (b_{m_1}, b_m)$ and $(a, c_1), (c_1, c_2), \ldots, (c_{m_1}, c_m)$. Assume that *same-generation* is local, and $r > 0$ witnesses that. Take $m > 2r + 3$, and note that the r-neighborhoods of (b_{r+1}, c_{r+1}) and (b_{r+1}, c_{r+2}) are isomorphic. By locality, this would imply that these pairs agree on the same-generation query, but in fact we have $(b_{r+1}, c_{r+1}) \in sg(R)$ and $(b_{r+1}, c_{r+2}) \notin sg(R)$.

We now state our main result on locality of queries, that applies to the language in which no limit is placed on the available arithmetic and aggregate functions – all are available. We denote this language by $\text{ALG}_{\text{aggr}}(\mathsf{All}, \mathsf{All})$.

Theorem 1 (Locality of SQL). *Let e be a pure relational graph query in $\text{ALG}_{\text{aggr}}(\mathsf{All}, \mathsf{All})$, that is, an expression of type* bb *over the scheme of one symbol $R : \mathsf{bb}$. Then e is local.* $\square$

That is, neither reachability, nor same-generation, is expressible in SQL over the base type b, no matter what aggregate functions and arithmetic operations are available. Inexpressibility of many other queries can be derived from this, for example, tests for graph connectivity and acyclicity.

Our next goal is to give an elementary, self-contained proof of this result. The restriction to graph queries used in the theorem is not necessary; the result can be stated in greater generality, but the restriction to graphs makes the definition of locality very easy to understand. The proof will consist of three steps:

1. It is easier to prove expressivity bounds for a logic than for an algebra. We introduce an *aggregate logic* $\mathcal{L}_{\text{aggr}}$, as an extension of first-order logic, and show how ALG_{aggr} queries are translated into it.
2. The logic $\mathcal{L}_{\text{aggr}}$ is still a bit hard to deal with it, because of the aggregate terms. We show that we can replace aggregate terms by *counting quantifiers*, thereby translating $\mathcal{L}_{\text{aggr}}$ into a simpler logic $\mathcal{L}_{\mathbf{C}}$. The price to pay is that $\mathcal{L}_{\mathbf{C}}$ has infinitary connectives.
3. We note that any use of an infinitary connective resulting from translation of $\mathcal{L}_{\text{aggr}}$ into $\mathcal{L}_{\mathbf{C}}$ applies to a rather uniform family of formulae, and use this fact to give a simple inductive proof of locality of $\mathcal{L}_{\mathbf{C}}$ formulae.

5 Aggregate logic and relational algebra

Our goal here is to introduce a logic $\mathcal{L}_{\text{aggr}}$ into which we translate ALG_{aggr} expressions. The structures for this logic are precisely relational databases over two base types with domains Dom and Num; that is, vocabularies are just schemas. This makes the logic *two-sorted*; we shall also refer to Dom as *first-sort* and to Num as *second-sort*.

We now define formulae and terms of $\mathcal{L}_{\text{aggr}}(\Omega, \Theta)$; as before, Ω is a set of predicates and functions on Num, and Θ is a set of aggregates. The logic is just a slight extension of the two-sorted first-order logic.

A SC-structure D is a tuple $\langle A, R_1^D, \ldots, R_k^D \rangle$, where A is a finite subset of Dom, and R_i^D is a finite subset of

$$\prod_{j=1}^{|t_i|} \mathrm{dom}_j(D)$$

where $\mathrm{dom}_j(D) = A$ for $t_i.j = \mathsf{b}$, and $\mathrm{dom}_j(D) = \mathsf{Num}$ for $t_i.j = \mathsf{n}$.

- A variable of sort i is a term of sort i, $i = 1, 2$.
- If $R : t$ is in SC, and $\vec{u}$ is a tuple of terms of type t, then $R(\vec{u})$ is a formula.
- Formulae are closed under the Boolean connectives $\vee, \wedge, \neg$ and quantification (respecting sorts). If x is a first-sort variable, $\exists x$ is interpreted as $\exists x \in A$; if k is a second-sort variable, then $\exists k$ is interpreted as $\exists k \in \mathsf{Num}$.
- If P is an n-ary predicate in Ω and $\tau_1, \ldots, \tau_n$ are second-sort terms, then $P(\tau_1, \ldots, \tau_n)$ is a formula.
- If f is an n-ary function in Ω and $\tau_1, \ldots, \tau_n$ are second-sort terms, then $f(\tau_1, \ldots, \tau_n)$ is a second-sort term.
- If $\mathcal{F}$ is an aggregate in Θ, $\varphi(\vec{x}, \vec{y})$ is a formula and $\tau(\vec{x}, \vec{y})$ a second-sort term, then $\tau'(\vec{x}) = \mathrm{Aggr}_{\mathcal{F}}\vec{y}.\ (\varphi(\vec{x}, \vec{y}), \tau(\vec{x}, \vec{y}))$ is a second-sort term with free variables $\vec{x}$.

The interpretation of all the constructs except the last one is completely standard. The interpretation of the aggregate term-former is as follows: fix an interpretation $\vec{a}$ for $\vec{x}$, and let $B = \{\vec{b} \mid D \models \varphi(\vec{a}, \vec{b})\}$. If B is infinite, then $\tau'(\vec{a})$ is f_ω. If B is finite, say $\{\vec{b}_1, \ldots, \vec{b}_l\}$, then $\tau'(\vec{a})$ is the result of applying f_l to the multiset whose elements are $\tau(\vec{a}, \vec{b}_i)$, $i = 1, \ldots, l$.

It is now possible to translate $\mathrm{ALG}_{\mathrm{aggr}}$ into $\mathcal{L}_{\mathrm{aggr}}$:

Theorem 2. *Let $e : t$ be an expression of $\mathrm{ALG}_{\mathrm{aggr}}(\Omega, \Theta)$. Then there is a formula $\varphi_e(\vec{x})$ of $\mathcal{L}_{\mathrm{aggr}}(\Omega, \Theta)$, with $\vec{x}$ of type t, such that for any SC-database D,*

$$e(D) \quad = \quad \{\vec{a} \mid D \models \varphi_e(\vec{a})\}$$

Proof. For the usual relational algebra operators, this is the same as the standard textbook translation of algebra expressions into calculus expression. So we only show how to translate arithmetic operations, aggregation, and grouping.

- Numerical selection: Let $e' = \sigma[P]_{i_1,\ldots,i_k}(e)$, where P is a k-ary predicate in Ω. Then $\varphi_{e'}(\vec{x})$ is defined as $\varphi_e(\vec{x}) \wedge P(x_{i_1}, \ldots, x_{i_k})$.
- Function application: Let $e' = \mathrm{Apply}[f]_{i_1,\ldots,i_k}(e)$, where $f : \mathsf{Num}^k \to \mathsf{Num}$ is in Ω. Then $\varphi_{e'}(\vec{x}, q) \equiv \varphi_e(\vec{x}) \wedge (q = f(x_{i_1}, \ldots, x_{i_k}))$.
- Aggregation: Let $e' = \mathrm{Aggr}[i : \mathcal{F}](e)$. Then $\varphi_{e'}(\vec{x}, q) \equiv \varphi_e(\vec{x}) \wedge (q = \mathrm{Aggr}_{\mathcal{F}}\vec{y}.\ (\varphi_e(\vec{y}), y_i))$.
- Grouping: Let $e' = \mathrm{Group}_m[\lambda S.e_1](e_2)$, where $e_1 : u$ is an expression over $SC \cup \{S\}$, and e_2 over SC is of type $t \cdot s$. Let $\vec{x}, \vec{y}, \vec{z}$ be of types t, s, u, respectively. Then

$$\varphi_{e'}(\vec{x}, \vec{z}) \;\equiv\; \exists \vec{y}\ \varphi_{e_2}(\vec{x}, \vec{y}) \;\wedge\; \varphi_{e_1}(\vec{z})[\varphi_{e_2}(\vec{x}, \vec{v})/S(\vec{v})]$$

where the second conjunct is $\varphi_{e_1}(\vec{z})$ in which every occurrence of $S(\vec{v})$ is replaced by $\varphi_{e_1}(\vec{x}, \vec{v})$.

The converse does not hold: formulae of $\mathcal{L}_{\text{aggr}}$ need not define safe queries, while all ALG_{aggr} queries are safe. It is possible, however, to prove a partial converse result; see [18] for more details.

6 SQL is local: the proof

We start by stating our main result in greater generality, without restriction to graph queries.

Let SC be pure relational (no occurrences of type n), and D an instance of SC. The *active domain* of D, $adom(D)$, is the set of all elements of Dom that occur in relations of D. The *Gaifman graph* of D is the undirected graph $G(D)$ on $adom(D)$ with $(a, b) \in G(D)$ iff a, b belong to the same tuple of some relation in D. The *r-sphere* of $a \in adom(D)$, $S_r^D(a)$, is the set of all b such that $d(a, b) \leq r$, where the distance $d(\cdot, \cdot)$ is taken in $G(D)$. The r-sphere of $\vec{a} = (a_1, \ldots, a_k)$ is $S_r^D(\vec{a}) = \bigcup_{i \leq k} S_r^D(a_i)$. The *$r$-neighborhood* of $\vec{a}$, $N_r^D(\vec{a})$, is a new database, whose active domain is $S_r^D(\vec{a})$, and whose SC-relations are simply restrictions of those relations in D. We write $\vec{a} \approx_r^D \vec{b}$ when there is an isomorphism of relational structures $h : N_r^D(\vec{a}) \to N_r^D(\vec{b})$ such that in addition $h(\vec{a}) = \vec{b}$. Finally, we say that a query e of type $\mathsf{b} \ldots \mathsf{b}$ is *local* if there exists a number $r > 0$ such that, for any database D, $\vec{a} \approx_r^D \vec{b}$ implies that $\vec{a} \in e(D)$ iff $\vec{b} \in e(D)$. The minimum such r is called the locality rank of e and denoted by $\mathsf{lr}(e)$.

Theorem 3. *Let e be a pure relational query in $\text{ALG}_{\text{aggr}}(\mathsf{All}, \mathsf{All})$, that is, an expression of type $\mathsf{b} \ldots \mathsf{b}$ over a pure relational schema. Then e is local.* □

Since $\text{ALG}_{\text{aggr}}(\mathsf{All}, \mathsf{All})$ can be translated into $\mathcal{L}_{\text{aggr}}(\mathsf{All}, \mathsf{All})$, we must prove that the latter is local. The proof of this is in two steps: we first introduce a simpler counting logic, $\mathcal{L}_{\mathbf{C}}$, and show how to translate $\mathcal{L}_{\text{aggr}}$ into it. We then give a simple proof of locality of $\mathcal{L}_{\mathbf{C}}$.

The logic $\mathcal{L}_{\mathbf{C}}$ is simpler than $\mathcal{L}_{\text{aggr}}$ in that it does not have aggregate terms. There is a price to pay for this – $\mathcal{L}_{\mathbf{C}}$ has infinitary conjunctions and disjunctions. However, the translation ensures that for each infinite conjunction or disjunction, there is a uniform bound on the *rank* of formulae in it (to be defined a bit later), and this property suffices to establish locality.

Logic $\mathcal{L}_{\mathbf{C}}$ The structures for $\mathcal{L}_{\mathbf{C}}$ are the same as the structures for $\mathcal{L}_{\text{aggr}}$. The only terms are variables (of either sort); in addition, every constant $c \in \mathsf{Num}$ is a term of the second sort.

Atomic formulae are $R(\vec{x})$, where $R \in SC$, and $\vec{x}$ is a tuple of terms (that is, variables and perhaps constants from Num) of the appropriate sort, and $x = y$, where x, y are terms of the same sort.

Formulae are closed under the Boolean connectives, and *infinitary connectives*: if φ_i, $i \in I$, is a collection of formulae, then $\bigvee_{i \in I} \varphi_i$ and $\bigwedge_{i \in I} \varphi_i$ are $\mathcal{L}_{\mathbf{C}}$

formulae. Furthermore, they are closed under both first and second-sort quantification.

Finally, for every $i \in \mathbb{N}$, there is a quantifier $\exists i$ that binds one first-sort variable: that is, if $\varphi(x, \vec{y})$ is a formula, then $\exists ix\ \varphi(x, \vec{y})$ is a formula whose free variables are $\vec{y}$. The semantics is as follows: $D \models \exists ix\varphi(x, \vec{a})$ if there are i distinct elements $b_1, \ldots, b_i \in A$ such that $D \models \varphi(b_j, \vec{a})$, $1 \leq j \leq i$. That is, the existential quantifier is witnessed by at least i elements. Note that the first-sort quantification is superfluous as $\exists x\varphi$ is equivalent $\exists 1x\ \varphi$.

We now introduce the notion of a *rank* of a formula, $\mathsf{rk}(\varphi)$, for both $\mathcal{L}_{\mathbf{C}}$ and $\mathcal{L}_{\text{aggr}}$. For $\mathcal{L}_{\mathbf{C}}$, this is the quantifier rank, but the second-sort quantification does not count:

- For each atomic φ, $\mathsf{rk}(\varphi) = 0$.
- For $\varphi = \bigvee_i \varphi$, $\mathsf{rk}(\varphi) = \sup_i \mathsf{rk}(\varphi)$, and likewise for $\bigwedge$.
- $\mathsf{rk}(\neg\varphi) = \mathsf{rk}(\varphi)$.
- $\mathsf{rk}(\exists ix\ \varphi) = \mathsf{rk}(\varphi) + 1$ for x first-sort; $\mathsf{rk}(\exists k\varphi) = \mathsf{rk}(\varphi)$ for k second-sort.

For $\mathcal{L}_{\text{aggr}}$, the definition differs slightly.

- For a variable or a constant term, the rank is 0.
- The rank of an atomic formula is the maximum rank of a term in it.
- $\mathsf{rk}(\varphi_1 * \varphi_2) = \max(\mathsf{rk}(\varphi_1), \mathsf{rk}(\varphi_2))$, for $* \in \{\vee, \wedge\}$; $\mathsf{rk}(\neg\varphi) = \mathsf{rk}(\varphi)$.
- $\mathsf{rk}(f(\tau_1, \ldots, \tau_n)) = \max_{1 \leq i \leq n}\ \mathsf{rk}(\tau_i)$.
- $\mathsf{rk}(\exists x\varphi) = \mathsf{rk}(\varphi) + 1$ if x is first-sort; $\mathsf{rk}(\exists k\varphi) = \mathsf{rk}(\varphi)$ if k is second-sort.
- $\mathsf{rk}(\text{Aggr}_{\mathcal{F}}\vec{y}.\ (\varphi, \tau)) = \max(\mathsf{rk}(\varphi), \mathsf{rk}(\tau)) + m$, where m is the number of first-sort variables in $\vec{y}$.

Translating $\mathcal{L}_{\text{aggr}}$ into $\mathcal{L}_{\mathbf{C}}$ This is the longest step in the proof, but although it is somewhat tedious, conceptually it is quite straightforward.

Proposition 1. *For every formula $\varphi(\vec{x})$ of $\mathcal{L}_{\text{aggr}}(\mathsf{All}, \mathsf{All})$, there exists an equivalent formula $\varphi^\circ(\vec{x})$ of $\mathcal{L}_{\mathbf{C}}$ such that $\mathsf{rk}(\varphi^\circ) \leq \mathsf{rk}(\varphi)$.*

Proof. We start by showing that one can define a formula $\exists i\vec{x}\varphi$ in $\mathcal{L}_{\mathbf{C}}$, whose meaning is that there exist at least i tuples $\vec{x}$ such that φ holds. Moreover, its rank equals $\mathsf{rk}(\varphi)$ plus the number of first-sort variables in $\vec{x}$. The proof is by induction on the length of $\vec{x}$. If $\vec{x}$ is a single first-sort variable, then the counting quantifier is already in $\mathcal{L}_{\mathbf{C}}$. If k is a second-sort variable, then $\exists ik\varphi(k, \cdot)$ is equivalent to $\bigvee_C \bigwedge_{c \in C} \varphi(c, \cdot)$, where C ranges over i-element subsets of Num – this does not increase the rank. Suppose we can define it for $\vec{x}$ being of length n. We now show how to define $\exists i(y, \vec{x})\varphi$ for y of the first sort, and $\exists i(k, \vec{x})\varphi$ for k of the second sort.

1. Let $\psi(\vec{z}) \equiv \exists i(y, \vec{x})\varphi(y, \vec{x}, \vec{z})$ It is the case that there are i tuples $(b_j, \vec{a}_j)$ satisfying $\varphi(y, \vec{x}, \cdot)$ iff one can find an l-tuple of pairs $((n_1, m_1), \ldots, (n_l, m_l))$ with all m_js distinct, such that
   - there are at least n_j tuples $\vec{a}$ for which the number of elements b satisfying $\varphi(b, \vec{a}, \cdot)$ is precisely m_j, and

– $\sum_{j=1}^{l} n_j \cdot m_j \geq i$.

Thus, $\psi(\vec{z})$ is equivalent to

$$\bigvee \bigwedge_{j=1}^{l} \exists n_j \vec{x}\ (\exists! m_j y\ \varphi(y, \vec{x}, \vec{z}))$$

where the disjunction is taken over all the tuples satisfying $n_j, m_j > 0$, m_js distinct, and $\sum_{j=1}^{l} n_j \cdot m_j \geq i$ (it is easy to see that a finite disjunction would suffice), and $\exists! nu\varphi$ abbreviates $\exists nu\varphi \wedge \neg\exists(n+1)u\varphi$.

The rank of this formula equals $\mathsf{rk}(\exists! m_j y\varphi) = \mathsf{rk}(\varphi) + 1$, plus the number of first-sort variables in $\vec{x}$ (by the induction hypothesis) – that is, $\mathsf{rk}(\varphi)$ plus the number of first-sort variables in $(y, \vec{x})$.

2. Let $\psi(\vec{z}) \equiv \exists i(k, \vec{x})\varphi(k, \vec{x}, \vec{z})$. The proof is identical to the proof above up to the point of writing down the quantifier $\exists! m_j k\varphi(k, \cdot)$ – it is replaced by the formula $\bigvee_C(\bigwedge_{c \in C} \varphi(c, \cdot) \wedge \bigwedge_{c \notin C} \neg\varphi(c, \cdot))$ where C ranges over m_j-element subsets of Num. As the rank of this equals $\mathsf{rk}(\varphi)$, we conclude that the rank of the formula equivalent to $\psi(\vec{z})$ equals $\mathsf{rk}(\varphi)$ plus the number of first-sort variables in $\vec{x}$.

This concludes the proof that counting over tuples is definable in $\mathcal{L}_{\mathbf{C}}$. With this, we prove the proposition by induction on the formulae and terms. We also produce, for each second-sort term $\tau(\vec{x})$ of $\mathcal{L}_{\mathrm{aggr}}$, a formula $\psi_\tau(\vec{x}, z)$ of $\mathcal{L}_{\mathbf{C}}$, with z of the second sort, such that $D \models \psi_\tau(\vec{a}, q)$ iff the value of $\tau(\vec{a})$ on D is q.

We may assume, without loss of generality, that parameters of atomic $\mathcal{L}_{\mathrm{aggr}}$ formulae $R(\cdot)$ and $P(\cdot)$ are tuples of variables: indeed, if a second-sort term occurs in $R(\cdot\tau_i\cdot)$, it can be replaced by $\exists k\ (k = \tau_i) \wedge R(\cdot k \cdot)$ without increasing the rank. We now define the translation as follows:

– For a second-sort term t which is a variable q, $\psi_t(q, z) \equiv (z = q)$. If t is a constant c, then $\psi_t(z) \equiv (z = c)$.
– For an atomic φ of the form $x = y$, where x, y are first-sort, $\varphi^\circ = \varphi$.
– For an atomic φ of the form $P(\tau_1(\vec{x}), \ldots, \tau_n(\vec{x}))$, $\varphi^\circ(\vec{x})$ is $\bigvee_{(c_1,\ldots,c_n) \in P} \bigwedge_{i=1}^{n} \psi_{\tau_i}(\vec{x}, c_i)$. Note that $\mathsf{rk}(\varphi^\circ) = \max_i \mathsf{rk}(\psi_{\tau_i}) \leq \max_i \mathsf{rk}(\tau_i) = \mathsf{rk}(\varphi)$.
– $(\varphi_1 \vee \varphi_2)^\circ = \varphi_1^\circ \vee \varphi_2^\circ$, $(\varphi_1 \wedge \varphi_2)^\circ = \varphi_1^\circ \wedge \varphi_2^\circ$, $(\neg\varphi)^\circ = \neg\varphi^\circ$, $(\exists x\varphi)^\circ = \exists x\varphi^\circ$ for x of either sort. Clearly, this does not increase the rank.
– For a term $\tau(\vec{x}) = f(\tau_1(\vec{x}), \ldots, \tau_n(\vec{x}))$, we have

$$\psi_\tau(\vec{x}, z) = \bigvee_{(c, c_1, \ldots, c_n) : c = f(\vec{c})} (z = c) \wedge \bigwedge_{j=1}^{n} \psi_{\tau_j}(\vec{x}, c_j)$$

Again it is easy to see that $\mathsf{rk}(\psi_\tau) \leq \mathsf{rk}(\tau)$.
– For a term $\tau'(\vec{x}) = \mathrm{Aggr}_{\mathcal{F}}\vec{y}.\ (\varphi(\vec{x}, \vec{y}), \tau(\vec{x}, \vec{y}))$, $\psi_{\tau'}(\vec{x}, z)$ is defined as

$$[\varphi_\infty^\circ(\vec{x}) \wedge (z = f_\infty)] \vee [\neg\varphi_\infty^\circ(\vec{x}) \wedge \psi'(\vec{x}, z)]$$

where $\varphi^\circ_\infty(\vec{x})$ tests if the number of $\vec{y}$ satisfying $\varphi(\vec{x},\vec{y})$ is infinite, and ψ' produces the value of the term in the case the number of such $\vec{y}$ is finite.
The formula $\varphi^\circ_\infty(\vec{x})$ can be defined as

$$\bigvee_{i:y_i \text{ of 2nd sort}} \quad \bigvee_{C\subseteq \mathsf{Num},|C|=\infty} \quad \bigwedge_{c\in C} \varphi^\circ_i(\vec{x},c)$$

where $\varphi^\circ_i(\vec{x},y_i) \equiv \exists(y_1,\ldots,y_{i-1},y_{i+1},\ldots,y_m)\varphi^\circ(\vec{x},\vec{y})$.
The formula $\psi'(\vec{x},z)$ is defined as the disjunction of $\neg\exists\vec{y}\varphi^\circ(\vec{x},\vec{y}) \wedge z = f_0$ and

$$\bigvee_{c,(c_1,n_1),\ldots,(c_l,n_l)} \left(\begin{array}{l} \quad z = c \\ \wedge\ \exists! n_1\vec{y}\ (\varphi^\circ(\vec{x},\vec{y}) \wedge \psi_\tau(\vec{x},\vec{y},c_1)) \\ \wedge \cdots \\ \wedge\ \exists! n_l\vec{y}\ (\varphi^\circ(\vec{x},\vec{y}) \wedge \psi_\tau(\vec{x},\vec{y},c_l)) \\ \wedge\ \forall\vec{y} \bigwedge_{a\in\mathsf{Num}}(\varphi^\circ(\vec{x},\vec{y}) \wedge \psi_\tau(\vec{x},\vec{y},a) \to \bigvee_{i=1}^{l}(a=c_i)) \end{array}\right)$$

where the disjunction is taken over all tuples $(c_1,n_1),\ldots,(c_l,n_l)$, $l>0, n_i>0$ and values $c\in\mathsf{Num}$ such that

$$\mathcal{F}(\{\!|\underbrace{c_1,\ldots,c_1}_{n_1 \text{ times}},\ldots,\underbrace{c_l,\ldots,c_l}_{n_l \text{ times}}|\!\}) \quad = \quad c$$

Indeed, this formula asserts that either $\varphi(\vec{x},\cdot)$ does not hold and then $z=f_0$, or that $c_1,\ldots,c_l$ are exactly the values of the term $\tau(\vec{x},\vec{y})$ when $\varphi(\vec{x},\vec{y})$ holds, and that n_is are the multiplicities of the c_is.
A straightforward analysis of the produced formulae shows that $\mathsf{rk}(\psi_{\tau'}) \le \max(\mathsf{rk}(\varphi^\circ),\mathsf{rk}(\psi_\tau))$ plus the number of first-sort variables in $\vec{y}$; that is, $\mathsf{rk}(\psi_{\tau'}) \le \mathsf{rk}(\tau')$. This completes the proof of the proposition.

$\mathcal{L}_{\mathbf{C}}$ is local Formulae of $\mathcal{L}_{\text{aggr}}$ have finite rank; hence they are translated into $\mathcal{L}_{\mathbf{C}}$ formulae of finite rank. We now show by a simple induction argument that those formulae are local. More precisely, we show that for every finite-rank $\mathcal{L}_{\mathbf{C}}$ formula $\varphi(\vec{x},\vec{\imath})$ ($\vec{x}$ of first-sort, $\vec{\imath}$ of second-sort) over pure relational SC, there exists a number $r\ge 0$ such that $\vec{a} \approx^D_r \vec{b}$ implies $D \models \varphi(\vec{a},\vec{\imath}_0) \leftrightarrow \varphi(\vec{b},\vec{\imath}_0)$ for any $\vec{\imath}_0$. The smallest such r will be denoted by $\mathsf{lr}(\varphi)$. The proof is based on:

Lemma 1 (Permutation Lemma). *Let D be first-sort, with $A = adom(D)$, and $r>0$. If $\vec{a} \approx^D_{3r+1} b$, then there exists a permutation $\rho : A \to A$ such that $\vec{a}c \approx^D_r \vec{b}\rho(c)$ for every $c\in A$.*

Proof. Fix an isomorphism $h : N^D_{3r+1}(\vec{a}) \to N_{3r+1}(\vec{b})$ with $h(\vec{a}) = \vec{b}$. For any $c \in S^D_{2r+1}(\vec{a})$, $h(c) \in S^D_{2r+1}(\vec{b})$ has the same isomorphism type of its r-neighborhood. Thus, for any isomorphism type T of an r-neighborhood of a single element, there are equally many elements in $A - S^D_{2r+1}(\vec{a})$ and in $A - S^D_{2r+1}(\vec{b})$ that realize T. Thus, we have a bijection $g : A - S^D_{2r+1}(\vec{a}) \to A - S^D_{2r+1}(\vec{b})$ such that $c \approx^D_r g(c)$. Then ρ can be defined as h on $S^D_{2r+1}(\vec{a})$, and as g on $A - S^D_{2r+1}(\vec{a})$. $\square$

Based on the lemma, we show that every $\mathcal{L}_{\mathbf{C}}$ formula φ of finite rank is local, with $\mathsf{lr}(\varphi) \leq (3^{\mathsf{rk}(\varphi)} - 1)/2$. Note that for the sequence $r_0 = 0, \ldots, r_{i+1} = 3r_i + 1, \ldots$, we have $r_k = (3^k - 1)/2$; we show $\mathsf{lr}(\varphi) \leq r_{\mathsf{rk}(\varphi)}$.

The proof of this is by induction on the formulae, and it is absolutely straightforward for all cases except counting quantifiers. For example, if $\varphi(\vec{x}, \vec{\imath}) = \bigvee_j \varphi_j(\vec{x}, \vec{\imath})$, and $m = \mathsf{rk}(\varphi)$, then by the hypothesis, $\mathsf{lr}(\varphi_j) \leq r_m$, as $\mathsf{rk}(\varphi_j) \leq \mathsf{rk}(\varphi)$. So fix $\vec{\imath}_0$, and let $\vec{a} \approx^D_{r_m} \vec{b}$. Then $D \models \varphi_j(\vec{a}, \vec{\imath}_0) \leftrightarrow \varphi_j(\vec{b}, \vec{\imath}_0)$ for all j by the induction hypothesis, and thus $D \models \varphi(\vec{a}, \vec{\imath}_0) \leftrightarrow \varphi(\vec{b}, \vec{\imath}_0)$.

Now consider the case of the counting quantifier $\psi(\vec{x}, \vec{\imath}) \equiv \exists iz\varphi(\vec{x}, z, \vec{\imath})$. Let $\mathsf{rk}(\varphi) = m$, then $\mathsf{rk}(\psi) = m + 1$ and $r_{m+1} = 3r_m + 1$. Fix $\vec{\imath}_0$, and let $\vec{a} \approx^D_{r_{m+1}} \vec{b}$. By the Permutation Lemma, we get a permutation $\rho : A \to A$ such that $\vec{a}c \approx^D_{r_m} \vec{b}\rho(c)$. By the hypothesis, $\mathsf{lr}(\varphi) \leq r_m$, and thus $D \models \varphi(\vec{a}, c, \vec{\imath}_0) \leftrightarrow \varphi(\vec{b}, \rho(c), \vec{\imath}_0)$. Hence, the number of elements of A satisfying $\varphi(\vec{a}, \cdot, \vec{\imath}_0)$ is exactly the same as the number of elements satisfying $\varphi(\vec{b}, \cdot, \vec{\imath}_0)$, which implies $D \models \psi(\vec{a}, \vec{\imath}_0) \leftrightarrow \psi(\vec{b}, \vec{\imath}_0)$. This concludes the proof of locality of $\mathcal{L}_{\mathbf{C}}$.

Putting everything together, let e be a pure relational expression of $\mathrm{ALG}_{\mathrm{aggr}}(\mathsf{All}, \mathsf{All})$. By Theorem 2, it is expressible in $\mathcal{L}_{\mathrm{aggr}}(\mathsf{All}, \mathsf{All})$, and by Proposition 1, by a $\mathcal{L}_{\mathbf{C}}$ formula of finite rank. Hence, it is local.

7 SQL over ordered domains

So far the only nonnumerical selection was of the form $\sigma_{i=j}$, testing equality of two attributes. We now extend the language to $\mathrm{ALG}^{<}_{\mathrm{aggr}}$ by allowing selections of the form $\sigma_{i<j}(e)$, where both i and j are of the type b, and $<$ is some fixed linear ordering on the domain Dom.

This small addition changes the situation dramatically, and furthermore in this case we can't make blanket statements like "queries are local" – a lot will depend on the numerical domain Dom and available arithmetic operations.

7.1 Natural numbers

Let $\mathsf{Num} = \mathbb{N}$. We consider a version of $\mathrm{ALG}_{\mathrm{aggr}}$ that has the most usual set of arithmetic and aggregate operators: namely, $+, \cdot, <$ and constants for arithmetic, and the aggregate $\sum$. This suffices to express aggregates `MIN`, `MAX`, `COUNT`, `TOTAL`, but certainly not `AVG`, which produces rational numbers.

We shall use the notations:

- $\mathrm{SQL}_{\mathbb{N}}$ for $\mathrm{ALG}_{\mathrm{aggr}}(\{+, \cdot, <, 0, 1\}, \{\Sigma\})$, and
- $\mathrm{SQL}^{<}_{\mathbb{N}}$ for $\mathrm{ALG}^{<}_{\mathrm{aggr}}(\{+, \cdot, <, 0, 1\}, \{\Sigma\})$.

It is sufficient to have constants just for 0 and 1, as all other numbers are definable with $+$.

We show how a well-known counting logic $\mathrm{FO}(\mathbf{C})$ [3] can be embedded into $\mathrm{SQL}^{<}_{\mathbb{N}}$. The importance of this lies in the fact that $\mathrm{FO}(\mathbf{C})$ over ordered structures captures a complexity class, called TC^0 [3, 26], for which no nontrivial general lower bounds are known. In fact, although TC^0 is contained in DLOGSPACE, the containment is not known to be proper, and to this day we don't even know if $\mathrm{TC}^0 \neq \mathrm{NP}$. Moreover, there are indications that proving such a separation result, at least by traditional methods, is either impossible, or would have some very unexpected cryptographic consequences [27].

Definition of $\mathrm{FO}(\mathbf{C})$ (see [3, 10, 19]) It is a two-sorted logic, with second sort being the sort of natural numbers. That is, a structure D is of the form

$$\langle \{a_1, \ldots, a_n\}, \{1, \ldots, n\}, <, +, \cdot, \underline{1}, \underline{n}, R_1, \ldots, R_l \rangle,$$

where the relations R_i are defined on the domain $\{a_1, \ldots, a_n\}$, while on the numerical domain $\{1, \ldots, n\}$ one has $\underline{1}, \underline{n}, <$ and $+, \cdot$ interpreted as ternary predicates (e.g., $+(x, y, z)$ holds iff $x + y = z$). This logic extends first-order by counting quantifiers $\exists ix\ \varphi(x)$, meaning that at least i elements satisfy φ; here i refers to the numerical domain $\{1, \ldots, n\}$ and x to the domain $\{a_1, \ldots, a_n\}$. These quantifiers bind x but not i.

Theorem 4. *Over ordered structures,* $\mathrm{FO}(\mathbf{C}) \subseteq \mathrm{SQL}_{\mathbb{N}}^{<}$. *In particular,*

$$\text{uniform } \mathrm{TC}^0 \subseteq \mathrm{SQL}_{\mathbb{N}}^{<}.$$

Proof sketch. With order and aggregate `TOTAL`, one can define the set $\{1, \ldots, m\}$ where $m = \mid adom(D) \mid$ (by counting the number of elements not greater than each element in the active domain). On this set, one defines $+, \cdot, <$, and then uses the standard translation of calculus into algebra, except for using the aggregate $\sum$ to translate counting quantifiers. □

Corollary 1. *Assume that reachability is not expressible in* $\mathrm{SQL}_{\mathbb{N}}^{<}$. *Then uniform* TC^0 *is properly contained in NLOGSPACE.*

As separation of complexity classes is currently beyond reach, so is proving expressivity bounds for $\mathrm{SQL}_{\mathbb{N}}^{<}$.

One can also show a closely-related upper bound on the class of decision problems expressible in $\mathrm{SQL}_{\mathbb{N}}^{<}$:

Proposition 2. *Every Boolean query in* $\mathrm{SQL}_{\mathbb{N}}^{<}$ *is contained in P-uniform* TC^0.

Notice that the reachability query, even over ordered domains of nodes, is *order-independent*; that is, the result does not depend on a particular ordering on the nodes, just on the graph structure. Could it be that order-independent queries in $\mathrm{SQL}_{\mathbb{N}}$ and $\mathrm{SQL}_{\mathbb{N}}^{<}$ are the same? Of course, such a result would imply that TC^0 is properly contained in DLOGSPACE, and several papers suggested this approach towards separating complexity classes. Unfortunately, it does not work, as shown in [17]:

Proposition 3. *There exist order-independent non-local queries expressible in* $\mathrm{SQL}_{\mathbb{N}}^{<}$. *Thus, there are order-independent* $\mathrm{SQL}_{\mathbb{N}}^{<}$ *queries not expressible in* $\mathrm{SQL}_{\mathbb{N}}$.

Proof sketch. On the graph of an n-element successor relation with an extra predicate P interpreted as the first $\lfloor \log_2 n \rfloor$ elements, one can define the reachability query restricted to the elements of P. □

Counting abilities of $\mathrm{SQL}_{\mathbb{N}}$ are essential for this result, as its analog for relational calculus does not hold [9].

7.2 Rational Numbers

The language $\mathrm{SQL}^{<}_{\mathbb{N}}$ falls short of the class of queries real SQL can define, as it only uses natural numbers. To deal with rational arithmetic (and thus to permit aggregates such as `AVG`), we extend the numerical domain Num to that of rational numbers $\mathbb{Q}$, and introduce the language

$$\mathrm{SQL}^{<}_{\mathbb{Q}} \text{ as } \mathrm{ALG}^{<}_{\mathrm{aggr}}(\{+,-,\cdot,\div,<,0,1\},\{\Sigma\}).$$

This is a stronger language than $\mathrm{SQL}^{<}_{\mathbb{N}}$ (and thus than $\mathrm{FO}(\mathbf{C})$) – to see this, note that it can define rational numbers, and if one represents those by pairs of natural numbers, in some queries these numbers may grow exponentially with the size of the database: something that cannot happen in the context of $\mathrm{SQL}^{<}_{\mathbb{N}}$.

The most interesting feature of $\mathrm{SQL}^{<}_{\mathbb{Q}}$ is perhaps that it is capable of coding inputs with numbers:

Theorem 5. *Let SC be a pure relational schema. Then there is an $\mathrm{SQL}^{<}_{\mathbb{Q}}$ expression e_{SC} of type n such that for every SC-database D, $e_{SC}(D)$ is a single rational number, and*

$$D_1 \neq D_2 \quad \Rightarrow \quad e_{SC}(D_1) \neq e_{SC}(D_2)$$

Proof sketch. The proof is based on the following: if P_1 and P_2 are two distinct nonempty sets of prime numbers, then $\sum_{p \in P_1} \frac{1}{p} \neq \sum_{p \in P_2} \frac{1}{p}$. We then code tuples with prime numbers (at most polynomial in the size of the input) and add up inverses of those codes. □

Thus, with the addition of some arithmetic operations, $\mathrm{SQL}^{<}_{\mathbb{Q}}$ can express many queries; in particular, $\mathrm{SQL}^{<}_{\mathbb{Q}}$ extended with all computable numerical functions expresses all computable queries over pure relational schemas! In fact, to express all computable Boolean queries over such schemas, it suffices to add all computable functions from $\mathbb{Q}$ to $\{0,1\}$. In contrast, one can show that adding all computable functions from $\mathbb{N}$ to $\{0,1\}$ to $\mathrm{SQL}^{<}_{\mathbb{N}}$ does not give us the same power, as the resulting queries can be coded by non-uniform TC^0 circuits. Still, the coding is just of theoretical interest; even for graphs with 20 nodes it can produces codes of the form $\frac{p}{q}$ with p,q relatively prime, and $q > 10^{1000}$; for $q > 10^{10000}$ one needs only 60 nodes.

8 Conclusion

Did SQL3 designers really have to introduce recursion, or is it expressible with what's already there? Our results show that they clearly had a good reason for adding a new construct, because:

1. Over unordered types, reachability queries cannot be expressed by the basic SQL `SELECT-FROM-WHERE-GROUPBY-HAVING` statements; in fact, all queries expressible by such statements are local.
2. Over ordered domains, with limited arithmetic, reachability queries are most likely inexpressible, but proving this is hard as separating some complexity classes (and perhaps as hard as refuting some cryptographic assumptions). Adding more arithmetic operations might help, but only at the expense of encodings which are several thousand digits long – so the new construct is clearly justified.

Being a theoretician, I like to see proofs of theorems (even folk theorems!), hence writing all those papers [23, 21, 24, 18] on the expressiveness of SQL. Having finished [18] just over a year ago, I felt that the whole story can be presented in a nice and clean fashion, without asking the reader to spend days studying the prerequisites. I've attempted to give such a presentation here. I hope I convinced you that next-generation database theory texts shouldn't just state that certain queries are inexpressible in SQL, they should also include simple proofs of these results.

Acknowledgements Although the presentation here is new, it is based entirely on previous results obtained jointly with other people. Special thanks to Limsoon Wong, with whom many of those papers were coauthored, and who in fact suggested back in '93 that we look at the expressiveness of aggregation. The aggregate logic was developed jointly with Limsoon, Lauri Hella, and Juha Nurmonen, who also collaborated with me on various aspects of locality of logics. Simple proofs of locality of logics were discovered in an attempt to answer some questions posed by Moshe Vardi. For their comments on the paper I thank Limsoon, Lauri, Juha, Martin Grohe, Thomas Schwentick, and Luc Segoufin. Part of this work was done while I was visiting the Verso group at INRIA-Rocquencourt.

References

1. S. Abiteboul, R. Hull and V. Vianu. *Foundations of Databases*, Addison Wesley, 1995.
2. A. V. Aho and J. D. Ullman. Universality of data retrieval languages. In *POPL'79*, pages 110–120.
3. D.M. Barrington, N. Immerman, H. Straubing. On uniformity within NC^1. *JCSS*, 41:274–306, 1990.
4. P. Buneman, S. Naqvi, V. Tannen, L. Wong. Principles of programming with complex objects and collection types. *TCS*, 149 (1995), 3–48.
5. J. Celko. *SQL for Smarties: Advanced SQL Programming.* Morgan Kaufmann, 2000.
6. M. Consens and A. Mendelzon. Low complexity aggregation in GraphLog and Datalog, *TCS* 116 (1993), 95–116. Extended abstract in *ICDT'90.*
7. C. J. Date and H. Darwen. *A Guide to the SQL Standard.* Addison Wesley, 1997.
8. G. Dong, L. Libkin and L. Wong. Local properties of query languages. *TCS* 239 (2000), 277–308. Extended abstract in *ICDT'97.*

9. M. Grohe and T. Schwentick. Locality of order-invariant first-order formulas. *ACM TOCL*, 1 (2000), 112–130.
10. K. Etessami. Counting quantifiers, successor relations, and logarithmic space, *JCSS*, 54 (1997), 400–411.
11. R. Fagin, L. Stockmeyer and M. Vardi, On monadic NP vs monadic co-NP, *Information and Computation*, 120 (1995), 78–92.
12. S. Finkelstein, N. Mattos, I.S. Mumick, and H. Pirahesh. Expressing recursive queries in SQL. ANSI Document X3H2-96-075r1, 1996.
13. H. Gaifman. On local and non-local properties, *Proceedings of the Herbrand Symposium, Logic Colloquium '81*, North Holland, 1982.
14. E. Grädel and Y. Gurevich. Metafinite model theory. *Information and Computation* 140 (1998), 26–81.
15. W. Hanf. Model-theoretic methods in the study of elementary logic. In J.W. Addison et al, eds, *The Theory of Models*, North Holland, 1965, pages 132–145.
16. L. Hella. Logical hierarchies in PTIME. *Information and Computation*, 129 (1996), 1–19.
17. L. Hella, L. Libkin and J. Nurmonen. Notions of locality and their logical characterizations over finite models. *J. Symb. Logic*, 64 (1999), 1751-1773.
18. L. Hella, L. Libkin, J. Nurmonen and L. Wong. Logics with aggregate operators. In *LICS'99*, pages 35–44.
19. N. Immerman. *Descriptive Complexity*. Springer Verlag, 1998.
20. A. Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *J. ACM* 29 (1982), 699–717.
21. L. Libkin. On the forms of locality over finite models. In *LICS'97*, pages 204–215.
22. L. Libkin. Logics with counting and local properties. *ACM TOCL*, 1 (2000), 33–59. Extended abstract in *LICS'98*.
23. L. Libkin, L. Wong. Query languages for bags and aggregate functions. *JCSS* 55 (1997), 241–272. Extended abstract in *PODS'94*.
24. L. Libkin and L. Wong. On the power of aggregation in relational query languages. In *DBPL'97*, pages 260–280.
25. P. O'Neil. *Database: Principles, Programming, Performance*. Morgan Kaufmann, 1994.
26. I. Parberry and G. Schnitger. Parallel computation and threshold functions. *JCSS* 36 (1988), 278–302.
27. A. Razborov and S. Rudich. Natural proofs. *JCSS* 55 (1997), 24–35.
28. P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science* 2 (1992), 461–493.