Basic SQL and Relational Calculus

Michael Benedikt

1 Comments on Lecture Notes

These lecture notes are about the query language portion of SQL, which is covered at great length both in every single database textbook and in millions of Web tutorials. The notes introduce the most essential bits of SQL syntax, but the focus is not on how to use SQL; instead it is supplementary theoretical material on SQL that is not in the textbook. The notes will not consider all the features of SQL that we cover in lecture, but will focus on explaining the "relational part" of SQL, comparing it to relational algebra and relational calculus. The results we give are standard, but much of the notation (e.g. "basic selects") is unique to these notes. The main results of this note – e.g. the fact that SQL without aggregation is "the same as" relational calculus – are mentioned in every database textbook. But they are not proven in any of the well-known database books. This note is for those of you who would like to know exactly what being "the same as" means formally, and who would like to see at least a sketch of the proofs.

These are a draft version of the notes, and may contain typos – please report any that you notice to me. They are not meant as a replacement for lecture!

2 SQL and its relationship to logical and algebraic languages

We have seen three "mathematical" languages with equivalent expressive power, safe relational calculus, active-domain relational calculus and relational algebra. We now introduce the query language of the database standards suite SQL, and compare it with the prior languages.

SQL has many features that go beyond relational calculus or algebra – e.g. aggregates. In lecture we will discuss many of these features, but in this note we will investigate the part that is equivalent in expressive power to relational calculus.

2.1 SQL SELECT queries

SQL SELECT queries are of the form:

SELECT [DISTINCT]? $V_{m_1}.A_1$ $[AS\ B_1]?\ldots$ $V_{m_k}.A_k$ $[ASB_{m_k}]?$
FROM Rel_1 $[AS?\ V_1]?\ldots$ ", " Rel_n $[AS?\ V_n]?$
WHERE ϕ

The *WHERE clause* ϕ is built up from conditions V_1 RelOp C_1, V_1 RelOp V_2 using AND and OR, where RelOp is a comparison operator $=, \leq \ldots$.

Here the ? indicates an optional part of the grammar.

Example 1 *Consider a schema with tables* Student(Studid, LastName...) *and* Enrollment(Studid, Courseid, ...).

An example of an SQL SELECT *query on this schema is:*

SELECT S.Studid FROM Student S, Enrollment E WHERE S.Studid = E.Studid

This query returns the ids of all students who are taking some course.

Informally, the semantics of an SQL SELECT query is as follows: start with an empty output O. Iterate variables over the relations $Rel_1 \ldots Rel_n$, with each iteration putting tuples into the variables named $V_1 \ldots V_n$ if these are specified, otherwise the tuples are put into variables that are named after the tables. Thus in each iteration, we have particular tuples $t_1 \ldots t_n$. We check the conditions in the WHERE clause ϕ on $t_1 \ldots t_n$, using the usual semantics for boolean operations. If the check passes, we add a tuple to the output O with k attributes whose values are $t_{m_1}.A_1 \ldots t_{m_k}.A_k$, renamed as $B_1 \ldots B_k$ if these optional column names are present. If the keyword DISTINCT is present, then duplicates are removed.

Note that the language above, when given a relational input, may produce as an output a *multi-set* or *bag*. For example, consider the query

SELECT S.LastName FROM Student S

working on a schema where the Student table has attributes Studid and LastName.

If given as input a table

Studid	LastName
21	"Jones"
23	"Smith"
31	"Jones"
42	"Jones"

this query will output the multi-set of tuples where {Studid = "Jones"} occurs three times and the tuple {Studid = "Smith"} occurs one time.

SQL has a good reason for making multi-set semantics the default. But to measure its expressiveness we need to "make SQL into a relational language". We can make SQL SELECT queries into a relational language simply by requiring the term DISTINCT to be in the SELECT clause [1].

[1] Full SQL has many other non-relational features, including comparison with NULLs and aggregation – we will discuss some of these in lecture, but not further in the notes

The set of *SQL basic select queries* (denoted SQLBasSel) contains all queries of the form:

$$\begin{aligned}\textsf{SELECT DISTINCT } & V_1.A_1 \ AS \ B_1 \ldots V_j.A_j \ AS \ B_n \\ \textsf{FROM } & Rel_1 \ AS \ V_1 \ldots Rel_n \ AS \ V_n \\ \textsf{WHERE } & \phi\end{aligned}$$

where ϕ is built up from conditions $V_i.A_i$ RelOp C_1, $V_i.A_i$ RelOp $V_j.A_j$ or $V_i.A_i$ RelOp c using AND, where RelOp is one of $=, <, \leq, \ldots$.

Note some differences from regular SELECT queries defined above:

- No arithmetic operations or OR
- We assume every table or column is given an alias
- The keyword DISTINCT is present, so output will always be duplicate free.

We can give an alternative semantics for SQL basic selects, by translation to relational algebra. Informally, we i) take a cross product of the tables in the FROM clause, then ii) we apply selections corresponding to the WHERE clause, and finally iii) we project according to the SELECT clause and iv) rename according to aliases in the SELECT clause. Formally, we proceed as follows. Consider an SQLBasSel query:

$$\begin{aligned}\textsf{SELECT DISTINCT } & V_1.A_1 \ AS \ B_1 \ldots V_j.A_j \ AS \ B_j \\ \textsf{FROM } & Rel_1 \ AS \ V_1 \ldots Rel_n \ AS \ V_n \\ \textsf{WHERE } & \phi_0\end{aligned}$$

We let τ_i be a relational algebra operation that renames each attribute a of Rel_i as $V_i.a$, and let $\tau = \tau_1(Rel_1) \times \ldots \times \tau_n(Rel_n)$. That is, τ does a rename followed by a cross product. Thus τ performs the product operation referred to in part i) of the informal algorithm above.

We define a function $T'(\phi)$ that converts an SQL WHERE clause ϕ into a sequence of selections inductively:

- $T'(V_1.a_1 \ \textsf{RelOp} \ V_2.a_2) = \sigma_{V_1.a_1 \ \textsf{RelOp} \ V_2.a_2}$
- $T'(V_1.a_1 \ \textsf{RelOp} \ c) = \sigma_{V_1.a_1 \ \textsf{RelOp} \ c}$
- $T'(\phi_1 \text{ AND } \phi_2) = T'(\phi_1) \circ T'(\phi_2)$ where $\circ$ is operator composition.

Now let $\tau' = T'(\phi_0)$, where ϕ_0 is the WHERE clause of the query. That is, τ' performs the sequence of selection operations corresponding to part ii) of the informal algorithm above.

We let τ'' be the sequence of projections $\pi_{V_1.A_1} \circ \ldots \pi_{V_n.A_k}$. That is, τ'' performs the projection operation in part iii) of the informal algorithm above.

Finally, we let τ''' be the operation that for each $i \leq j$ renames $V_i.A_i$ as B_i.

We can translate Q' as $\tau''' \circ \tau'' \circ \tau' \circ \tau$, thus performing i) – iv) in succession.

Example 2 *Consider a fragment of our university schema:*

- *A table* Student *with attributes* Studid, LastName, *and* FirstName
- *A table* Enrollment *with attributes* Year, Term, Courseid *and* Studid

And consider an SQL basic select query:

SELECT DISTINCT E.Year FROM Student S, Enrollment E
WHERE S.Studid = E.Studid *AND* S.LastName = "$Smith$"

The translation above will give the relational algebra query:

$$\pi_{\mathsf{E.year}}\sigma_{\mathsf{S.studid}=\mathsf{E.studid}}\sigma_{\mathsf{LastName}=\text{“}Smith''}(\rho(\mathsf{Studid} \mapsto \mathsf{S.studid}, \mathsf{Student}) \times \rho(\mathsf{Studid} \mapsto \mathsf{E.studid}\ \mathsf{Year} \mapsto \mathsf{E.year}, \mathsf{Enrollment})$$

2.2 Unions of SQL basic selects

The language SQLBasSel misses the ability to express queries that use the relational algebra UNION operation. In fact, even SQL SELECT queries (without the "basic" restriction, allowing the use of OR in the WHERE clause) cannot express relational algebra union. To handle this, SQL adds a UNION construct, which behaves exactly as relational algebra union does.

USQLBasSel queries are just built up from SQLBasSel queries using the SQL UNION operator.

UNION has the obvious meaning: Q_1 UNION Q_2 performs Q_1 and Q_2 and eliminates duplicates before unioning the result. Thus the query:

(SELECT S.LastName FROM Student S) UNION (SELECT P.LastName FROM Professor P)

would always return a table with no duplicates.

It is clear that we can extend the translation in the previous subsection to translate USQLBasSel queries to relational algebra, just mapping UNION to relational algebras union operator.

So now we have a translation mapping UNIONs of SQL basic selects to relational algebra. But note that we did not need all of relational algebra, but rather a subset of the relational operators. The positive relational algebra RA^+ is the algebra built up from the rename, projection, selection, product, and union operators. The SPJ algebra SPJ is the subset built up from the projection, selection, rename, and product operators. Then our translation above shows:

Proposition 1 *Referring to the definitions above, we have:*

- *Every* SQLBasSel *query is equivalent to some* SPJ *query*
- *Every* USQLBasSel *query is equivalent to some* RA^+ *query.*

Practical impact: *The algorithm above is a basic but important step in query evaluation, since it allows the query engine to deal with relational algebra, which is much easier to manipulate than the SQL source language.*

Above we translated SQL into Relational Algebra. What can we say about the expressiveness of SQL basic SELECTS compared to relational calculus? SQL is much more similar to relational calculus than to relational algebra – the WHERE clause is analogous to the formula part of a relational calculus query, while the SELECT clause is analogous to the returned variables. The FROM clause is similar to an active domain restriction, although it allows the query rewriter to be much more precise about where the tuples come from. We could have translated SQL basic selects directly into relational calculus, but we can get a translation by combining the mapping to relational algebra above with the translation (discussed in prior notes) from algebra to calculus.

If we look at the kind of relational calculus queries we need to capture basic SELECTs, we see that the main feature is that we do not need negation or universal quantification. We now state this property precisely. Positive existential first-order logic, abbreviated $\exists FO^+$, is built-up from atomic predicates $R(x_1 \ldots x_n)$, x_1 RelOp x_2 and x_1 RelOp c by $\wedge, \vee, \exists$, omitting negation. Say that a $\exists FO^+$ formula is *variable-bounded* if it is of the form $\bigvee_i \phi_i$, where each ϕ_i does not use $\vee$, every free variable of the formula is contained in some relation symbol within each ϕ_i, and every bound variable in ϕ_i occurs in some relation symbol. So $\exists y R(x, y, z) \wedge z > 5 \vee \exists w S(x, w)$ is variable-bounded. Notice that the condition on free variables guarantees that variable-bounded formulas are safe, in that the number of solutions if finite whenever the input relations are finite.

The variable-bounded positive relational calculus RC^+ is defined as in the relation calculus, but requiring all formulas to be variable-bounded $\exists FO^+$ formulas.

The following result establishes exactly what you can express with an SQL basic SELECT query:

Theorem 1 *The following languages are equivalent in expressiveness:*

1. USQLBasSel
2. RA^+
3. RC^+

Similarly, SQLBasSel, SPJ *queries and variable-bounded* RC^+ *queries that do not use* $\vee$ *all have the same expressiveness.*

Same expressiveness means that they can express the same queries, up to mapping between positional and named notation.

Proof We will only prove the first statement, although the second statement follows from the proof.

The direction RA^+ to RC^+ uses the same translation given in the earlier notes – one can just observe that this translation does not make use of negation except for handling the difference operator, and can also enforce the variable-bounded property. The direction from USQLBasSel to RA^+ is given above.

It remains to show that every RC^+ query can be expressed as a union of SQL basic SELECTs.

Suppose we have $Q = \{\langle x_{j_1} \ldots x_{j_k}\rangle | \phi\}$, where ϕ is a variable-bounded $\exists FO^+$ formula.

We can write Q as the union of queries $Q_1 \ldots Q_n$ where $Q_i = \{\langle x_{j_1} \ldots x_{j_k}\rangle | \phi_i\}$ and each ϕ_i does not use $\vee$. Since USQLBasSel has a union operator, we will be done as long as we can translate each Q_i into an SQL basic SELECT. Thus we will assume that we are have $Q = \{\langle x_{j_1} \ldots x_{j_k}\rangle | \phi\}$, where ϕ is a variable-bounded $\exists FO^+$ formula without $\vee$.

We first normalize the formula ϕ in the body of the query so that it is of the form $\exists y_1 \ldots y_n \bigwedge_j \rho_j$, where ρ_j does not contain the binary boolean connectives $\wedge$ or $\vee$. We can also arrange that there are only variables in each relation symbol, and no repeated variables, by introducing additional equality clauses (e.g. $R(x,y) \wedge S(w,y)$ becomes $R(x,y) \wedge S(w,z) \wedge y = z$).

We define a named schema where the attributes of relation R are exactly the numbers $1 \ldots arity(R)$. For the i^{th} occurrence of a relation R in ϕ, we will have an alias R^i in our query. Because ϕ is variable-bounded, every variable must participate in some occurrence of some relation. For each variable x, let $RelOcc(x)$ be the alias corresponding to one of the occurrences of a relation that contains x, and $Pos(x)$ be the corresponding position where x occurs – if x occurs in multiple relations, the choice of which one to use is arbitrary.

We first give a translation function $T(\phi)$ from normalized ϕ's of the form $\exists y_1 \ldots y_n \bigwedge_j \rho_j$ to an USQLBasSel WHERE clause. The output will be an AND of atomic WHERE clauses. The components ANDed together consist of the following:

For each ρ_j in ϕ of the form x_1 RelOp x_2, $T(\phi)$ will contain one basic WHERE clause

$RelOcc(x_1).Pos(x_1)$ RelOp $RelOcc(x_2).Pos(x_2)$.

Notice that the function $T(\phi)$ ignores the leading existential quantifiers, and also ignores any relation symbols.

Our final translation function transforms the query Q to $T(Q)$ defined as:

SELECT DISTINCT $RelOcc(x_{j_1}).pos(x_{j_1}) \ldots RelOcc(x_{j_k}).pos(x_{j_k})$
FROM $R_1\ R_1^1, \ldots\ R_1\ R_1^{m_1} \ldots R_n\ R_n^1, \ldots R_n\ R_n^{m_n}$
WHERE $T(\phi)$

Above we assume R_1 has m_1 occurrences in ϕ, R_1 m_2 occurrences, and so forth.

One can now verify that relative to the input renaming of position i of relation R as attribute $R.i$, and the output renaming of $RelOcc(x_{j_1}).pos(x_{j_m})$ by m, Q is equivalent to $T(Q)$.

□

Example 3 *We consider how the above translation would work on the RC^+ query:*

$$\{\langle x\rangle |\ \exists y R(x,y,x)\}$$

We would first normalize the query to to:

$$\{\langle x\rangle |\ \exists y\ \exists z\ R(x,y,z) \wedge z = x\}$$

There is only one occurrence of R, so we will have one alias variable, call it R^1. We apply the function T to the right hand side $\exists y\ \exists z\ R(x,y,z) \wedge z = x$. Looking at the definition of T, this reduces to applying the function $T(z = x)$ to get $R^1.3 = R^1.1$. Thus the translation $T(Q)$ will be:

SELECT DISTINCT $R^1.1$ FROM R R^1 WHERE $R^1.3 = R^1.1$

3 Larger fragments of SQL

Basic SQL SELECTs represent a powerful language. But Basic SELECTs are limited since they do not have the ability to mimic relational calculus *negation* or (in relational algebra terms), the difference operator.

We now consider extending SQL with *subqueries*. Subqueries allow one to build larger queries from smaller ones. SQL allows many ways of doing this, but in this note we will focus on IN, NOT IN, EXISTS, NOT EXISTS, which do allow one to express negation.

We first show some examples of SQL with subqueries.

Example 4 *Consider the student/enrollment database from the previous examples.*

A query asking for the ids of students who are not enrolled in any class could be written:

SELECT S.Studid FROM Student S WHERE S.Studid *NOT IN*
(SELECT E.Studid FROM Enrollment$E$)

The expression SELECT E.Studid FROM Enrollment E *is a subquery. It is an "uncorrelated subquery" in that it can be run as a stand-alone SQL query in itself.*

The same query can be written:

SELECT S.Studid FROM Student S WHERE *NOT EXISTS*
(SELECT E.Studid FROM Enrollment E WHERE E.Studid = S.Studid)

The expression SELECT E.Studid FROM Enrollment E WHERE E.Studid = S.Studid *is a "correlated subquery" – it cannot be evaluated stand-alone because of the presence of the variable S that is not "bound" in the* FROM *clause.*

Note that for SQL basic SELECTs, we assumed that every alias mentioned in the SELECT or WHERE clause was introduced in the FROM clause. After all, a query without this property could not be evaluated. To define the syntax of our larger fragment, we will build up from components that might have aliases that do not occur in a FROM clause. These are *free aliases*. For example, S is a free alias in the correlated subquery

SELECT E.Studid FROM Enrollment E WHERE E.Studid = S.Studid

of Example 4. Free aliases are analogous to free variables in relational calculus formulas.

Our syntax will define two classes, $\mathsf{SQL}_{\mathsf{Subquery}}$ queries and $\mathsf{SQL}_{\mathsf{Subquery}}$ WHERE clauses. For example,

SELECT E.Studid FROM Enrollment E WHERE E.Studid = S.Studid

is a query, but

X.Studid = 5 OR X.Studid NOT IN SELECT E.Studid FROM Enrollment E WHERE E.Studid = S.Studid

is a $\mathsf{SQL}_{\mathsf{Subquery}}$ WHERE clause.

$\mathsf{SQL}_{\mathsf{Subquery}}$ queries and $\mathsf{SQL}_{\mathsf{Subquery}}$ WHERE clauses have a mutually recursive definition:

- If Q_1 is a $\mathsf{SQL}_{\mathsf{Subquery}}$ query returning tuples with a single column, then
  $X.a$ NOT IN Q, C NOT IN Q, $X.a$ IN Q, and C IN Q
  are valid $\mathsf{SQL}_{\mathsf{Subquery}}$ WHERE clauses, where X is an alias, a is an attribute, and C is a constant. [2] [3]

- If X and Y are aliases, a and b are attributes, RelOp is a comparison operator, then $X.a$ RelOp $Y.b$ and $X.a$ RelOp c are $\mathsf{SQL}_{\mathsf{Subquery}}$ WHERE clauses. That is, we have every WHERE clause that a basic SELECT can have.

- $\mathsf{SQL}_{\mathsf{Subquery}}$ WHERE clauses are closed under AND, OR, and NOT.

[2] more complex uses of IN, NOT IN are possible, using multiple-columns and arithmetic, but the semantics is more complex

[3] There are additional sanity conditions that we omit: for example, the data type of attribute a in $X.a$ must match the type of the single column returned by Q_1

- If Q_1 is a $\mathsf{SQL}_{\mathsf{Subquery}}$ query, then EXISTS Q_1 and NOT EXISTS Q_1 are valid $\mathsf{SQL}_{\mathsf{Subquery}}$ WHERE clauses.
- If ϕ is an $\mathsf{SQL}_{\mathsf{Subquery}}$ WHERE clause then the following is an $\mathsf{SQL}_{\mathsf{Subquery}}$ query:

$$\begin{aligned}\mathsf{SELECT}\ [\mathsf{DISTINCT?}]\ V_{m_1}.A_1\ [AS\ B_1?]\ldots\ V_{m_k}.A_k\ [ASB_{m_k}?]\\ \mathsf{FROM}\ Rel_1\ [AS\ V_1?]\ldots\ Rel_n\ [AS\ V_n?]\\ \mathsf{WHERE}\ \phi\end{aligned}$$

- If Q_1 and Q_2 are $\mathsf{SQL}_{\mathsf{Subquery}}$ queries with matching attributes in the SELECT clause, then Q_1 UNION Q_2 is an $\mathsf{SQL}_{\mathsf{Subquery}}$ query.

The notion of an alias being free in an $\mathsf{SQL}_{\mathsf{Subquery}}$ query or WHERE clause is the obvious one: roughly speaking, the free aliases or those that do not occur in any FROM clause.

Semantics of SQL with subqueries. The semantics of an SQL query is now more complex, since it must take into account free variables. The semantics will be analogous to that for relational calculus, where we have to keep track of a variable mapping. In giving the semantics below, we will assume that the keyword DISTINCT is always present in the SELECT clause – in this way we can give a purely relational account of the semantics, without discussing bag semantics.

The main complication is that the semantics must be given relative to a database instance and an *alias mapping* σ of free aliases to tuples.

We will have two different semantic functions, whose definitions are mutually recursive.

- We write $[Q]_{D,\sigma}$ for the table obtained by evaluating Q on D with alias mapping σ
- We write $D, \sigma \models \rho$, where ρ is a WHERE clause, when ρ holds relative to D and σ.

Example 5 *Consider a variant of the correlated subquery from Example 4:*

$Q =$ SELECT DISTINCT E.Studid FROM Enrollment E WHERE E.Studid $\neq$ S.Studid

To say what the output of Q is, we need a database and a mapping of the free alias S to some tuple. For example, suppose D is the database where the Enrollment *table has one row each for* Studid $= 31$, Studid $= 45$, *and* Studid $= 57$. *Assume σ maps the alias S to a* Student *tuple with* Studid $= 57$. *Then the result $[Q]_{D,\sigma}$ would be a table having rows with* Studid $= 31$ *and* Studid $= 45$. *If instead we had a mapping σ taking the alias S to a* Student *tuple with* Studid $= 45$, *then $[Q]_{D,\sigma}$ would be a table with two rows, one with* Studid $= 31$ *and one with* Studid $= 57$.

Consider the $\mathsf{SQL}_{\mathsf{Subquery}}$ *WHERE clause*

$X.\mathsf{Studid}$ *IN* SELECT DISTINCT $E.\mathsf{Studid}$ FROM Enrollment E WHERE $E.\mathsf{Studid} = S.\mathsf{Studid}$

This has two free aliases, X *and* S*. To say whether the clause holds, we need a database and a mapping* σ *assigning both* X *and* S *to tuples. If* D *is as above,* S *maps to a* Student *tuple with* $\mathsf{Studid} = 57$ *and* X *maps to a tuple with* $\mathsf{Studid} = 48$*, then the clause does not hold. We write* $D, \sigma \not\models \rho$*.*

We now give the official definition of the two semantic relationships $[Q]_{D,\sigma}$ and $D, \sigma \models \rho$, which are by mutual induction on the number of symbols in Q or ρ.

The inductive step for a SELECT clause on an instance D relative to an alias mapping σ is roughly "as before": we iterate over all sequences of assignments of tuples t_i to aliases V_i in the FROM clause – for each assignment, the mapping σ is extended to a new alias mapping $\sigma' = \sigma, V_1 \mapsto t_1 \ldots V_n \mapsto t_n$: that is, extending σ by assigning V_i to t_i. The WHERE clause is then evaluated for each extended alias mapping σ' – by a recursive call to the algorithm for $D, \sigma \models \rho$. Finally, we take every sequence that satisfies the WHERE clause and restrict it to the columns in the SELECT clause, eliminating duplicates and renaming if needed.

We now give the inductive definition for checking whether $D, \sigma' \models \rho$, where σ' is the extended alias mapping and ρ is the WHERE clause. We use the usual inductive rules for boolean operations. We also use the usual semantics for equalities and inequalities, substituting each variable V with the value $\sigma'(V)$. The interesting case is when the WHERE clause is one of the new subquery constructs. In this case, the inductive rules are:

- $D, \sigma' \models X.a$ IN Q iff $\sigma'(X).a$ is a value in the table $[Q]_{D,\sigma'}$, and similarly for C IN Q.
- $D, \sigma' \models X.a$ NOT IN Q iff $\sigma'(X).a$ is not a value in $[Q]_{D,\sigma'}$ and similarly for C NOT IN Q.
- $D, \sigma' \models$ EXISTS Q iff $[Q]_{D,\sigma'} \neq \emptyset$
- $D, \sigma' \models$ NOT EXISTS Q iff $[Q]_{D,\sigma'} = \emptyset$

One can check that these rules capture the informal meaning of the semantics of subqueries.

The meaning of query equivalence. We now want to state that $\mathsf{SQL}_{\mathsf{Subquery}}$ queries are "the same as" relational calculus queries. To explain what this means, we have to handle the annoying distinction between positional tables and named tables. In the translation, we have also to deal with the distinction in SQL queries between "free aliases" and aliases that are bound using the FROM clause. The following notion on the relational calculus side is helpful:

A *parameterized relational calculus query* has the same syntax as a relational calculus query, namely:

$$\{\langle x_1 \ldots x_m \rangle | \phi\}$$

but ϕ can contain free variables $x_1 \ldots x_n$ which may be a larger set than the returned variables $x_1 \ldots x_m$. In addition a subset of the free variables of ϕ $x_{i_1} \ldots x_{i_p}$ are distinguished as being *parameter variables*.

For example,

$$Q' = \{\langle \mathsf{Studid} \rangle | \ \exists \mathsf{Courseid}\ \mathsf{Enrollment}(\mathsf{Studid}, \mathsf{Courseid}) \wedge \mathsf{Studid} \neq \mathsf{Studid}'\}$$

is a parameterized query, where Studid' is a parameter variable.

Intuitively, the SQL query

$$Q = \mathsf{SELECT\ DISTINCT}\ E.\mathsf{Studid}\ \mathsf{FROM}\ \mathsf{Enrollment}\ E\ \mathsf{WHERE}\ E.\mathsf{Studid} \neq S.\mathsf{Studid}$$

is equivalent to the query Q' in relational calculus, with the attribute $S.\mathsf{Studid}$ of the free alias corresponding to the parameter Studid'. This "equivalence" is relative to some renamings taking positions to attribute names.

Suppose Q is an $\mathsf{SQL}_{\mathsf{Subquery}}$ query returning a table with attributes $A_1 \ldots A_n$ and having free aliases $V_1 \ldots V_k$. We will translate Q to a parameterized relational calculus query $Q' = \{\langle x_1 \ldots x_n \rangle | \phi'\}$. We will now say what it means for Q and Q' to be the same, taking into account the distinction between positional and named notation. This requires us to translate both the input and output of the query back and forth between the two notations.

An input mapping for an $\mathsf{SQL}_{\mathsf{Subquery}}$ query Q and a parameterized DRC formula Q' as above is a function m that: i) associates each position in the input schema of Q' with an attribute in some table of the input schema of Q and 2) associates each parameter variable of Q' with an expression $V_i.a_i$ where V_i is a free alias of Q, such that every expression of the form $V_i.a_i$ where V_i is free in Q is in the range of m. That is, m tells us how to get from an input database and parameter values for Q' to an input and values for attributes of free alias that are inputs to Q.

Given such an input mapping m, if we have a sequence of values $\vec{c} = c_{i_1} \ldots c_{i_p}$, where $P = x_{i_1} \ldots x_{i_p}$ are the parameter variables of Q' we let $m(\vec{c})$ be the corresponding assignment of values to the terms $V_i.a_i$. We can thus consider $m(\vec{c})$ as an alias mapping for Q, assigning the other attribute values of the V_i arbitrarily. If we have a positional database instance I, we let $m(I)$ be the corresponding instance in named notation obtained by applying m to each tuple in I.

An output renaming m' associates every attribute returned by Q with a number, in such a way that the image of the returned attributes cover a set of the form $[1, j]$ for some j. That is, m' converts an output of Q back to an output of Q'. Given a collection of tuples O for the output schema of Q, we let $m'(O)$ be the corresponding relation in positional notation.

We say that Q *is equivalent to* Q' *modulo* m, m' iff:

For every relational database I in positional notation, for every sequence $\vec{c}$ of values for the parameter variables of Q',

$$m'([Q_{m(I),m(\vec{c})}]) = \{\langle t_1 \ldots t_n \rangle) | I, \vec{t} \models \phi' \wedge \bigwedge_{j \leq p} t_{i_j} = c_{i_j}\}$$

That is, the evaluation of the SQL query relative to the alias mapping gives the same result as the relational calculus query when the parameters are substituted accordingly.

We have a similar notion for a WHERE clause ϕ being equivalent to a relational calculus formula ϕ' relative to an input renaming m. An input renaming in this case maps any free variable x_i of ϕ' to an expression $V_i.a_i$ in ϕ. Equivalence in this context means:

For every relational database I in positional notation, for every assignment v of values to the free variables of ϕ',

$$m(I), m(v) \models \phi \leftrightarrow I, v \models \phi'$$

The equivalence results. We are now ready to make a formal claim that SQL with subqueries can be translated *into* relational calculus.

Theorem 2 *[Translation of SQL into Relational Calculus] For every* $\mathsf{SQL}_{\mathsf{Subquery}}$ *query Q there is a relational calculus query Q' that is equivalent to it relative to some input renaming m_1 and output renaming m_2. Similarly every* $\mathsf{SQL}_{\mathsf{Subquery}}$ *WHERE clause ϕ there is a parameterized relational calculus formula ϕ' equivalent to it relative to some input renaming.*

Proof We give a translation function SQLToRC that maps:

- Every $\mathsf{SQL}_{\mathsf{Subquery}}$ query to a parameterized relational calculus query, input renaming m_1, and output renaming m_2:
- Every $\mathsf{SQL}_{\mathsf{Subquery}}$ WHERE clause to a parameterized relational calculus formula which is equivalent up to an input renaming.

The step for SELECTs uses the same translation idea as in the case of basic SELECT queries (e.g. going through relational algebra) but now using an inductive call to translate the WHERE clause. The variables that are associated with alias terms that are not bound in the FROM clause will become parameter variables in the resulting parameterized DRC formula.

We give only the step for the new subquery constructs.

- EXISTS Q, NOT EXISTS Q

  Let $Q', m_1, m_2 = \mathsf{SQLToRC}(Q)$ be a parameterized relational calculus query, input renaming, output renaming such that Q' is equivalent to Q relative to m_1 and m_2. Let $Q' = \{< x_1 \ldots x_m > | \phi\}$. Then we translate EXISTS Q to

  $$\exists x_1 \ldots \exists x_m \ \phi$$

  That is, the above is a relational calculus formula equivalent to the WHERE clause EXISTS Q, relative to the input renaming m_1' that restricts m_1 to the parameter variables of ϕ, which will be the only free variables in $\exists x_1 \ldots \exists x_m \ \phi$.

  Analogously we translate NOT EXISTS Q to:

$\neg\exists x_1 \ldots \exists x_m\ \phi$

That is, the above is a safe relational calculus formula equivalent to the WHERE clause NOT EXISTS Q relative to m'_1.

- IN, NOT IN

  Inductively let $Q', m_1, m_2 = \mathsf{SQLToRC}(Q)$ be a relational calculus query, input renaming, and output renaming such that Q' is equivalent to Q relative to m_1, m_2. Let $Q' = \{< x > |\phi\}$. Then we translate $V.a$ IN Q simply to: ϕ.

  That is, ϕ is a relational calculus formula equivalent to the WHERE clause $V.a$ IN Q relative to the input mapping extending m_1 by mapping x to $V.a$.

  Analogously, we translate V NOT IN Q simply to $\neg\phi$.

□

Example 6 *Consider again the query:*

SELECT DISTINCT S.Studid FROM Student S WHERE *NOT EXISTS*
(SELECT E.Studid FROM Enrollment E WHERE E.Studid = S.Studid)

Earlier we have stated that we require all of our SELECT clauses to have the keyword DISTINCT *in them, in order to compare to relational query languages. But here for brevity we drop* DISTINCT *from the subquery, where it has no impact on the semantics of the top-level query.*

We translate the subquery

SELECT E.Studid FROM Enrollment E WHERE E.Studid = S.Studid

to the parameterized query:

$\{\langle \mathsf{Studid}\rangle |\ \exists \mathsf{Courseid}\ \mathsf{Enrollment}(\mathsf{Courseid}, \mathsf{Studid}) \wedge \mathsf{Studid} = \mathsf{Studid}'\}$ *along with corresponding input and output renamings.*

Applying the inductive rule above, we map the WHERE clause:

NOT EXISTS (SELECT DISTINCT E.Studid FROM Enrollment E WHERE E.Studid = S.Studid)

to the formula

$\phi(\mathsf{Studid}') = \neg\exists \mathsf{Studid}\ \exists \mathsf{Courseid}\ \mathsf{Enrollment}(\mathsf{Courseid}, \mathsf{Studid}) \wedge \mathsf{Studid} = \mathsf{Studid}'$

note that Studid' *is still free in this formula.*

The last step involves performing the translation of basic SELECTs into RC^+ *on the top-level SELECT, then plugging in* $\phi(\mathsf{Studid}')$ *as an additional conjunct. Applied here, this would result in the query:*

$$\{\langle \mathsf{Studid}'\rangle |\exists \mathsf{LastName}\ \mathsf{Student}(\mathsf{Studid}', \mathsf{LastName}) \wedge$$
$$\neg\exists \mathsf{Studid}\ \exists \mathsf{Courseid}\ \mathsf{Enrollment}(\mathsf{Courseid}, \mathsf{Studid}) \wedge \mathsf{Studid} = \mathsf{Studid}'\}$$

Translating back to SQL from Relational Calculus. Above we have seen that SQL with subqueries can be translated to the Domain Relational Calculus. What about the converse? It is easy to see that SQLBasSel queries are safe, so they can not express every relational calculus formula. Can SQLBasSel express every safe relational calculus formula? Using our previous results on algebra versus calculus, we can see that this is the case.

Theorem 3 *[Translation from Safe Relational Calculus to SQL] For every safe relational calculus query Q there is a $\mathsf{SQL}_{\mathsf{Subquery}}$ query Q' that is equivalent to it relative to some input renaming m_1 and output renaming m_2.*

Proof We will prove this by showing that every relational algebra query can be translated to $\mathsf{SQL}_{\mathsf{Subquery}}$. The statement above follows because we know from our prior results that relational algebra can express every safe relational calculus query.

We can translate relational algebra to SQLBasSel by induction on the structure.

The base cases are already done in Theorem 1. The inductive cases for the RA^+ operators are straightforward. For example, for a projection $\pi_B(Q)$, we let Q' be the translation of Q, and then we remove from the SELECT clause any attributes that are not in B.

We focus on the inductive case of the difference operator, which is where we will use subqueries.

Consider $\rho_1 - \rho_2$, and let Q_1 and Q_2 be equivalent $\mathsf{SQL}_{\mathsf{Subquery}}$ queries, which exist by induction. We also simplify by assuming that both Q_1 and Q_2 consist of a single SELECT statement – in general they can be rewritten as unions of SELECTs.

Let $Q_1 =$ SELECT $V_1.A_1 \ldots V_j.A_j$ FROM F_1 WHERE C_1.

Let $Q_2 =$ SELECT $W_1.B_1 \ldots W_j.B_j$ FROM F_2 WHERE C_2.

We assume that the aliases used in Q_1 and Q_2 are disjoint, which can be arranged by renaming.

Let Q_2' be formed from Q_2 by replacing the WHERE clause C_2 of Q_2 by

C_2 AND $(W_1.B_1 = V_1.A_1$ AND $\ldots$ $W_j.B_j = V_j.A_j)$

Note that Q_2' now is a correlated SQL subquery, since it has aliases that are not bound. If there is no WHERE clause C_2 then we just add a WHERE clause $(W_1.B_1 = V_1.A_1$ AND $\ldots$ $W_j.B_j = V_j.A_j)$.

Let Q_1' be formed from Q_1 by replacing C_1 by:

C_1 AND NOT EXISTS Q_2'

Again, if C_1 is empty, we omit it from above along with AND.

Then Q_1' is the desired query.

□

Example 7 *Consider the relational algebra query* $\pi_{\text{Studid}}\text{Student} - \pi_{\text{Studid}}\text{Enrollment}$.

Applying the base case of the algorithm, we translate Student *to:*

$$\text{SELECT DISTINCT } S.\text{Studid} \ldots \text{ FROM Student } S$$

Above, ... *includes other attributes of* Student.

Applying the inductive rule for projection, we translate: $Q_1 = \pi_{\text{Studid}}\text{Student}$ *to:*

$$\text{SELECT DISTINCT } S.\text{Studid FROM Student } S$$

Similarly we arrive at a translation of $\pi_{\text{Studid}}\text{Enrollment}$ *as*

$$Q_2 = \text{SELECT DISTINCT } E.\text{Studid FROM Enrollment } E$$

We now perform the first step above for difference, obtaining

$$Q_2' = \text{SELECT DISTINCT } E.\text{Studid FROM Enrollment } E \text{ WHERE } E.\text{Studid} = S.\text{Studid}$$

We then perform the final step, producing the query:

$Q_1' =$ SELECT DISTINCT S.Studid FROM Student S WHERE $NOT\ EXISTS$ (SELECT DISTINCT E.Studid FROM Enrollment E WHERE E.Studid $= S$.Studid)

One can verify that this is indeed a correct translation of the original relational algebra query, relative to the appropriate mappings.

Combining Theorems 2 and 3, along with our prior results, we get:

Theorem 4 *The following are equivalent in expressiveness up to renaming:*

- *Relational algebra*
- *Safe Relational Calculus*
- *Active-Domain Relational Calculus*
- $\text{SQL}_{\text{Subquery}}$

4 The Bottom Line

We have shown that SQL with subqueries is equivalent (modulo renamings, assumptions about DISTINCT, etc.) to Relational Algebra, which is equivalent to Safe Relational Calculus and to Active Domain Relational Calculus. What are the advantages of each formalism?

SQL syntax has by far the most intuitive syntax for users – thus the most important translations are the ones from SQL to other languages. The main problem with SQL is it is not conducive to inductive algorithms – you will notice

this in the translations from SQL, which are very ad-hoc. SQL syntax is fragile, This is the reason why it is important to translate out of SQL into a more compositional syntax.

Relational algebra is the simplest language to manipulate computationally, but is unintuitive for users.

The strength of Relational Calculus is that it is the easiest language to analyze. For example, if we want to know whether a transformation rule holds, it is easiest to say at the DRC level. If we want to know whether a query can be expressed in SQL (as defined in this note) it is easiest to see that by considering whether it can be expressed in DRC. In fact, one there are a number of general theorems in the literature about what can and cannot be expressed in first-order logic that can normally be applied.