A Logic and Semantics for SQL Queries

David Posner, ParAccel, November 2012

Introduction. The problem addressed in this paper occurred to the author about twenty years ago while attending a seminar concerning the logical query transformations used in a certain commercial SQL compiler. To one not knowing much about SQL but having studied logic the process seemed straightforward. As illustrated in the diagram below, we start with a query $q_1$ and map it to a logical formula $F_1$ in an appropriate theory. We then apply the theory to find a logically equivalent formula $F_2$ and reinterpret $F_2$ as a query $q_2$ which is then equivalent to $q_1$, i.e., has the same result set.

One troubling feature in the presentation was the fact that the allowed transformations included De Morgan’s laws and elimination of double negation. This is troubling because together they would seem to imply excluded middle. We can argue as follows. $\neg A$ and $A$ are certainly contradictory and so we have $\neg (\neg A)$ and so applying De Morgan we have $(\neg A \lor \neg A)$ and finally, eliminating $\neg \neg A$, $(A \lor \neg A)$. This is troubling of course because excluded middle can fail in SQL. Fortunately, consistency is saved in the end because $A$ and $\neg A$ are not actually contradictory according to the evaluation rules of SQL. On thought, this makes sense. SQL is a computation system not a logical theory. We can’t expect a computation system to go off and do theorem proving. But that calls into question the process illustrated in the diagram and raises the question of how we go about determining what transformations are valid. The only viable option for dealing with the “NOT” problem, besides giving up the idea that $A$ and $\neg A$ are contradictory, is to accept the fact that the SQL “NOT” operator will not translate directly to logical $\neg$. But then how should we interpret “NOT” along with the rest of the SQL query constructors. One of the claims of SQL is that it is based in logic but if the “logical” operators can’t actually be interpreted in logic then the claim has no basis. This led to author to ask whether there is a logic to which SQL queries can be faithfully mapped which began an off and on again quest to find such a logic. The ground we adopted for this quest is that we have to accept SQL as is. Ignoring behavior that we might view as “warts” should not be an option.

In addition to the “NOT” problem there are at least two other issues with SQL that seem to pose serious obstacles to our program. One of these has to do with the “IS NULL” operator. “IS NULL” is a kind of introspective operator which appears to require the theory to reason about its own state of knowledge. In effect it requires the theory decide whether it has enough information to deduce the existence of a value for an expression. The second issue has to do with “table” (subquery) expressions which occur for example when using quantifiers. We can illustrate the problem with an example. Consider a bombs table used to track our progress in searching for bombs.

<table>
<thead>
<tr>
<th>Building</th>
<th>Location</th>
<th>Has_Bomb</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘Building I’</td>
<td>‘Kitchen’</td>
<td>False</td>
</tr>
<tr>
<td>‘Building II’</td>
<td>‘Bathroom’</td>
<td>True</td>
</tr>
<tr>
<td>‘Building I’</td>
<td>‘Bathroom’</td>
<td>NULL</td>
</tr>
<tr>
<td>‘Building I’</td>
<td>‘Conference Room I’</td>
<td>False</td>
</tr>
</tbody>
</table>

It would be natural to want to know what buildings are “safe” and to formulate the question as the query:

```sql
SELECT DISTINCT Building FROM bombs b1 WHERE NOT EXISTS (SELECT Location FROM bombs b2 WHERE b1.Building = b2.Building AND b2.Has_Bomb);
```
Unfortunately the result of this query would assure us that ‘Building I’ is safe in spite of the fact that the situation in the bathroom is unknown. One might have expected the EXISTS clause in that case to return NULL so that NOT EXISTS would also have returned NULL so that under the rules governing SELECT expressions, ‘Building I’ would have been excluded. The problem is that the same rules apply to the SELECT in the subquery so that information about NULL values, e.g. that the bathroom was not searched, cannot propagate out to the quantifier. From the point of view of logic the effect of SELECT in an SQL subquery is to map the expression from a non-classical to a classical logic. Thus our program seems to require formulating a logic in which the semantics can change from non-classical to classical within the clauses of an expression.

Fortunately there is one issue often thought to cause problems for interpreting SQL semantics that does not seem to pose a serious problem. This is the problem of the so-called “duplicate” rows. One might imagine that this would undermine the “set-theoretic foundation” of relational databases. Handling this issue seems relatively easy. If the system can distinguish two rows then there is obviously data attached to the rows which distinguishes them and we can model this in terms of “private” columns, e.g. “RowID”, if we like. One slight issue concerning duplicates is that certain operators like UNION and some of the aggregation functions take positions with respect to “duplicates” and potential duplicates which can occur when rows have NULL fields. We need to ensure that these various positions are consistent with the logic, e.g., the logic must be able to leave open the possibility that two potentially identical objects are or are not identical.

Given the special problem of SQL semantics it is perhaps not surprising that a solution to the problem leads us to a somewhat exotic area of mathematics. The area to which we will resort is called categorical logic and in particular the theory of topos. The idea of categorical logic is to formulate the elements of logic, i.e., formal systems and models, in terms of the stuff of category theory, i.e., objects, maps, and functors. The theory of topos can be viewed as a non-classical generalization of set theory which allows the possibility of objects with “partial existence” and hence “partial truth values.” This seems to be just the kind of thing we need to capture SQL semantics. This led the author to an article called “The Logic of Topoi” by Michael Fourman [LoT]. There is a particular construction in the theory of topos which is striking to one familiar with relational databases. The construction is called a pullback.

In a topos (singular of topos), given any objects $A,B$ and maps $f:A\to C$, $g:B\to C$ with domains $A$ and $B$ respectively, and common codomain $C$, there is a unique (up to isomorphism) object $E$ together with maps $\Pi_A, \Pi_B$ mapping $E$ into $A$ and $B$ respectively such that:

1. The diagram in (b) below commutes (the composite maps $f\circ \Pi_A = g\circ \Pi_B$);

2. For any object $D$ together with maps $\Pi_A', \Pi_B'$ mapping $D$ to $A$, $B$ respectively with $f\circ \Pi_A' = g\circ \Pi_B'$ there is a unique map $e:D\to E$ such that the diagram in (c) commutes (all composites with common domain and codomain are equal.)

$E$ is called the pullback of $f$ and $g$ and the associated maps $\Pi_A$ and $\Pi_B$ are called its projections. (Note the use of the dashed line in diagram (c) to denote a map whose existence is implied.)

Here are brief explanations of the terminology used above. A category is a kind of directed graph. The nodes are called objects and the edges are called maps or sometimes arrows or morphisms. The source of a map is called its
domain and the target is called its codomain. If $f$ is a map with domain $A$ and codomain $B$ we may write \( f:A \rightarrow B \) and say $f$ is a map \textit{from} $A$ \textit{to} $B$. Maps are closed under composition, i.e., given objects $A$, $B$, and $C$ there is an operator \( \circ \) which given $f:A \rightarrow B$ and $g:B \rightarrow C$ returns a map $g \circ f:A \rightarrow C$. Every object $A$ has an identity map $id_A:A \rightarrow A$ such that for any map $f$ with codomain $A$, $f \circ id_A = f$ and for every map $g$ with codomain $A$, $id_A \circ g = g$. The canonical example of a category, sometimes called \textit{SET}, has sets as objects, and pairs \( \langle f,C \rangle \) where $f$ is a function and $C$ is a set containing the range of $f$ as maps. Category theory characterizes systems and structures in terms of the existence and behavior of maps. A map $f:A \rightarrow B$ is an \textit{isomorphism} if there exists a map $g:B \rightarrow A$ such that $g \circ f = id_A$ and $f \circ g = id_B$.

Typically category theory only characterizes structures up to isomorphism classes and thus focuses on behavior rather than representational details.

It becomes clear why the pullback construction seems relevant to our quest when we consider its meaning in the \textit{SET} category. In \textit{SET} we can have $f:a \rightarrow b$ be a subset of the Cartesian product $A \times B$ and take $\Pi_a$ and $\Pi_b$ to be the respective component functions. The requirement that the diagram in (b) commute means that for \( \langle a,b \rangle \in E \) we have $f(a) = g(b)$. The requirement associated with diagram (c) implies that $E$ must consist of all such pairs. Thus in \textit{SET}, the pullback is exactly the \textit{equijoin} of $A$ and $B$ with respect to $f$ and $g$.

Here are some additional categorical generalizations of set theoretic concepts required to define a topos. For more details please see \textit{[LoT]} and references cited there or any standard references on category theory.

\textbf{1-1 function (injection)}: An \textit{injection} is a map which “discriminates” between distinct maps into its domain. Formally, a map $i:A \rightarrow B$ is an injection if for any $C$ and any $f,g:C \rightarrow A$ if $(i \circ f) = (i \circ g)$ then $f = g$.

\textbf{Subset (subobject)}: A is a subobject of $B$ if there is an injection from $A$ to $B$.

\textbf{Surjection (Onto function)}: An \textit{epimorphism} is a map which “discriminates” between distinct maps out of its codomain. Formally, a map $s:B \rightarrow A$ is an epimorphism if for any $C$ and any $f,g:A \rightarrow C$ if $(f \circ i) = (g \circ i)$ then $f = g$.

Notice that the definition of epimorphism can be obtained from the definition of injection by reversing the arrows and the order of composition (and v.v.). In this case we say the definitions are \textit{dual} to one another. Every categorical definition has a dual. (For example the dual of pullback is called, not surprisingly, a \textit{pushout}.)

There is a kind of “opposite sides” law for pullbacks. In the pullback diagram above, if $f$ or $g$ is an injection then the “opposite” projection is also an injection. For example if $f$ is injective then $\Pi_b$ is injective. In a topos, the kind of category that is of interest to us here, the analogous law is also true for epimorphisms, e.g., if $f$ is epimorphic then so is $\Pi_b$. When interpreting a pullback as an SQL equijoin the result on injections is useful when reducing a semijoin to the projection of a join and the result on epimorphisms may be used, when the hypothesis holds, to reduce an outer join to an inner join.

\textbf{Cartesian Product (Direct Product)}: The direct product of objects $A$ and $B$, written $A \times B$, is an object together with projection maps, $\Pi_A:A \times B \rightarrow A$, $\Pi_B:A \times B \rightarrow B$ which exactly captures all the ways of pairing information about $A$ with information about $B$. More precisely, for any object $C$ and maps $f:C \rightarrow A$, $g:C \rightarrow B$ there is a unique map $\langle f,g \rangle:C \rightarrow A \times B$ such that $f = \Pi_A \circ \langle f,g \rangle$ and $g = \Pi_B \circ \langle f,g \rangle$.

The dual to the direct product is the \textit{direct sum} denoted $A \oplus B$. In \textit{SET} the direct sum can be represented as the tagged union \( \{ \langle i,x \rangle \mid i = 0 \text{ and } x \in A \text{ or } i = 1 \text{ and } x \in B \} \). Thus direct sums correspond to partitions. The maps in the direct sum corresponding to the projections $\Pi_A$ and $\Pi_B$ are the \textit{injections} $i_A:A \rightarrow A \oplus B$ and $i_B:B \rightarrow A \oplus B$ which are in fact injective. If $f:A \rightarrow C$ and $g:B \rightarrow C$ then there is a unique map $f \oplus g:A \oplus B \rightarrow C$ with $f = f \circ (i_A \circ \langle f,g \rangle)$ and $g = (i_B \circ \langle f,g \rangle) \circ g$. Direct sum corresponds to the OUTER UNION in SQL. Note that if $f:A \rightarrow C$, then a partition of $C$, $C = \bigsqcup_i C_i$, induces via pullbacks, a partition of $A = \bigsqcup_i A_i$ and we have $f = \bigsqcup_i f_i$ where $f_i$ is the \textit{restriction} of $f$ to $A_i$ obtained by composing $f$.
with the injection from \( A \) to \( A \). Now suppose we also have \( g:B\rightarrow C \). Then we can express the pullback of \( A \) and \( B \) on \( f \) and \( g \) as \( \exists P \) where \( P \) is the pullback of \( A \) and \( B \) on \( f,g \). Thus we direct sum provides a representation for distributed operations in general and joins in particular with the injections representing the maps that merge the components.

**Singleton Set**

A terminal object is an object 1 with the property that for every object \( A \) there is one and only one map from \( A \) to 1. The dual concept is an initial object, 0 such that for every object \( A \) there is one and only one map from 0 to \( A \). In SET 1 can be represented by any single element set and 0 is represented (uniquely) by the empty set. Note that \( 1\times B = B \) for every \( B \) (taking \( \Pi_b \) to be \( i_b \) and \( \Pi_1 \) to be the unique map from \( B \) to 1). We will use “!” to name maps with codomain 1.

**Boolean Values**

A subobject classifier is specified by an object \( \Omega \) together with an injection \( 1\rightarrow \Omega \) such that for any object \( B \) and any subobject of \( B \), \( A \), with injection \( m:A\rightarrow B \), there is a unique map \( c_A \) making the following diagram a pullback.

\[
\begin{array}{ccc}
A & \rightarrow & B \\
\downarrow & & \downarrow \\
! & \downarrow c_A & \Omega \\
\end{array}
\]

\( \Omega \) represents the possible truth values in the logic of the topos. We call \( c_A \) the classifier of \( A \). In SET the subobject classifier has two values, usually taken to be 0 and 1 (for false and true), and the classifier of \( A \) is the characteristic function which takes the value 1 for elements of \( A \) and 0 otherwise. As we shall see, for an SQL database “with NULL values” the truth value object will have many more than two values. Maps into \( \Omega \) correspond to predicates in SQL and the pullback above corresponds to a predicate join. Note that since maps into 1 are unique, any map with domain 1 is an injection and so the projection into \( B \) of a pullback of 1 and \( B \) with respect to true and any map \( p:B\rightarrow \Omega \) will be an injection and so any pullback of \( B \) “along true” will be a subobject of \( B \). We call a map from \( B \) into \( \Omega \) a predicate on \( B \). We thus have a 1-1 correspondence between subobjects of \( B \) and predicates on \( B \).

**Power sets**

An exponential object is an internal representation of the maps between two objects. In SET an exponential object is the set of all functions between two sets. In a category in which the terminal object and direct products exist we define the exponential for maps from \( A \) to \( B \) as an object \( B^A \) together with a map \( ev:A\times B^A \rightarrow B \) called an evaluation map, such that for any object \( X \) and map \( f:X\times A\rightarrow B \) there exists a unique map \( \lambda A: f:X\rightarrow B^A \) such that \( ev_{i_X, \lambda A.f} = f \). So in particular, \( \Omega^B \) is an internal representation of the predicates on \( B \) and so generalizes the notion of power set. In SQL, power sets are needed for group by and aggregation operators and to support table domains. For example consider a map \( \text{dept} : \text{EMP} \rightarrow \text{DEPT} \). Then taking the pullback of \( \text{dept} \) and \( \text{dept}^{\text{DEPT}} \) we get a subobject of \( \text{EMP}\times\text{DEPT} \) which corresponds to a predicate \( \text{EMP.dept=DEPT.dept}; \text{EMP}\times\text{DEPT} \rightarrow \Omega \) and so we have \( \lambda \text{EMP}(\text{EMP.dept=DEPT.dept});\text{DEPT} \rightarrow \Omega^\text{EMP} \) which maps \( \text{DEPT} \) to the power set of \( \text{EMP} \). This corresponds to the SQL GROUP BY operator, as in “SELECT ... FROM EMP,DEPT WHERE EMP.dept=DEPT.dept GROUP BY DEPT.dept”. As we shall see, we will also need power set in our logic in order to represent SQL NULL semantics correctly.

A topos is a category that supports all the objects and constructions mentioned above. (For a proper definition see [LoT: Chapter 5].) A topos is thus a generalization of SET which supports the operators of relational algebra and so a candidate for a theory in which to interpret SQL queries. Our problem now is to show that the semantics of SQL can be interpreted in the logic of topoi. To do this we will construct a model of SQL semantics and show that that model is in fact a topos. The material for carrying this out is laid out almost perfectly in the paper Sheaves and Logic by Michael Fourman and Dana Scott [S&L]. Our development will follow that paper almost exactly.

**A Semantic Model for SQL**

Nothing in the following discussion should be construed as having philosophical content. Our goal is to create a model of SQL which captures the behavior of SQL. We are not concerned here in whether that is a good behavior or a bad behavior and we are not interested in indoctrinating users with this
model. Its function is to define a logic for reasoning about SQL which faithfully captures the behavior of SQL queries.

The sensible place to start looking for an SQL semantics of course is in set theory. Before the introduction of NULL this is basically what we had and ordinary predicate calculus was the appropriate logic. We can view the effect of adding NULLs to SQL as introducing “holes” into the sets. In order to formalize this notion it helps to view an SQL database as defined by a formal language. We are going to be vague about the details of the language for the time being. Our immediate goal is to explain concepts. We assume we have designated constants and operator symbols for all data values and operators. For each occurrence of a NULL value we add a new constant and allow these constants to participate in terms and specify that when these constants are evaluated they will be treated like NULL. For example, if c is a constant inserted for a NULL value the expression “2+c” has no value.

The crucial semantic question is how will we interpret these new constants? The usual interpretation is that the new constants represent “missing information”, i.e., they have values but those values aren’t known. Clearly, this interpretation does not match the evaluation rules of SQL. If a term x has a value then surely “x=x” must evaluate to true whether we know the value of x or not. This is certainly our assumption when solving equations involving unknowns. Similarly, under the assumption that all the terms in a predicate expression A have values, even if some are unknown, then certainly A is either true or false, even if we don’t know which. Thus (A OR NOT A) must hold regardless of what we know. After all, isn’t the point of logic is to tell us what formulas are true regardless of specific interpretation? The missing information interpretation still leaves us with classical logic and so not SQL.

An alternative to the “missing information” semantics is the “doesn’t have a value yet” semantics. The idea in this model is that the system will evolve over time eventually assigning values to previously unassigned constants, but leaving everything else the same. Thus terms don’t have values until a value is decided upon. Until that time, terms involving unassigned constants are meaningless. The problem with this interpretation is that while (A V NOT A) may not be true now it will eventually be true. So we wind up either assuming the law of the excluded middle or somehow introducing time dependence into our semantics. To fix this we must allow the possibility that some constants and other closed terms will never be assigned values. Under this model “x=x” and “(A V NOT A)” may never become true because the relevant terms never have a value. (Following Fourman [LoT] we will sometimes say that a term with no value doesn’t exist.)

Under this “potentially never decided” semantics we view our database as an evolving system. We can picture the possible evolutionary paths of the database in this model as a tree of possible extensions of the system. Each node is derived from its parent by possibly but not necessarily assigning values to constants not yet assigned values in the state of the parent. This of course is a very bushy tree as there may be separate branches for every value we might assign to a given constant. As we move up any particular path on this tree the terms and predicates of the system evolve. A useful image for this process is that of trying to infer information about a complex scene while viewing it through an out of focus lens. Many of the objects in our view will be blurry. Moving up a path is analogous to focusing the lens. As we do so some blurs may converge to a clearly defined point, some blurs may become distinct, some may disappear altogether, some apparently distinct blurs will merge into a single blur, and some blurs will just stay blurry. Our problem is to infer only that about which we can be confident. For example, though we will want to be able to express the (misleading) SQL form of the bomb in building query discussed earlier we want the natural form of the query (without the SELECT conversion) to behave correctly.

We can control the semantics thus obtained by constraining the tree of potential states associated with nodes on the tree. The usual “closed world” assumption for example is imposed by not allowing the creation of new potential elements in tables. In order to assign a precise meaning to the terms and formulas of this evolving system we want to associate a mathematical model with every potential state. In this structure we will assign a meaning to each term of the system (even those which have no meaning!) This tree of potential models will thus capture all the evolutionary possibilities of the system. The logical constraint that inferences along a path must be consistent will then correspond to a continuity constraint on the associated path of structures. The interpretations of expressions within these structures will evolve as the structures evolve. In general three truth values is not enough to maintain continuity. We can’t simply say a term is undefined. We need a representation of how
undefined the term is and how “true” some formula is and these representations must evolve continuously. For this to be the case the interpretation of a term or formulas must account for all of its evolutionary possibilities. Our goal now is to make these intuitions precise.

Let σ and τ be nodes on the potential extensions tree. We say τ is an extension of σ and write σ ≤ τ if τ = σ or τ is above σ on the tree. Our trees grow up so that further up the tree corresponds to possibly more defined. Let B_σ denote { τ | σ ≤ τ}. We will sometimes call B_σ the restriction to σ or the basic neighborhood of σ. A set of potential extensions O is open if B_σ is contained in O whenever σ is in O, i.e. if σ is in the collection then all extensions of σ are in the collection. We are going to take these open sets of potential extensions as our truth values. Clearly the empty set, which we take as false, is open as is the set of all possible extensions, which we take as true. Open sets are closed under unions and intersections and so the open sets form a topology.

We take the truth value of a statement S, denoted ||S||, to be the set of all σ such that S is true in the state associated with σ for all τ in B_σ. Requiring truth values to be open is how we enforce the continuity constraint on the evolution of structures mentioned above. Note in particular that we do not include a given extension in the truth set of a statement if there is a possibility that the statement will cease to be true in some further extension. With one class of exceptions the SQL evaluation rules already ensure that once an expression becomes defined on any path in the possible extensions tree it will stay defined and the value will not change. The exceptions are expressions which involve some form of “IS NULL” test.

We use Ω to represent the collection of open sets, aka “truth values,” in the extension tree topology. We write u ≤ v if u is contained in v. Of course ≤ defines a partial ordering on Ω and we use 0 or false for the empty set and 1 or true for the set of all extensions. Clearly 0 and 1 are the bottom and top of the partial ordering. We next interpret the logical operators over Ω. We define meet and join operators ∧ and ∨ on Ω as intersection and union respectively. Clearly Ω is a lattice under ∧ and ∨, i.e. ∧ is the greatest lower bound operator and ∨ is the least upper bound operator, and ∧ and ∨ distribute over one another, e.g. u ∧ (v ∨ w) = (u ∧ v) ∨ (u ∧ w). Note that the set complement of an open set is not in general open so we define ~u to be { σ | B_σ is contained in the set complement of u}. Equivalently, ~u is the largest open set contained in the set complement of u. Note that in general while (u ∧ ~u) = 0, we do not have u ∨ ~u = 1 so in contrast to classical logic, Ω under these definitions, is not a Boolean algebra. However we can define an inference structure as follows. For u and v in Ω define u→v as U {w | (u ∧ w) ≤ (v ∧ w)}. Think of this as the set of possible extensions σ such that in the restriction to B_σ, u implies v, i.e any extension in u is in v. Note that → is transitive, u→0 = ~u, and if u ≤ v then u→v = 1, so in particular for all u, u→1 = 1. Further we have the analog of the inference rule modus ponens, u ∧ (u→v) = v. A lattice with these properties is called a Heyting algebra, after the logician Arend Heyting who was a strong proponent of intuitionistic logic.

Heyting algebras play the same role for intuitionistic logic that Boolean algebras play for classical logic. Using the fact that Ω is closed under arbitrary intersections as well as unions we use ∀ and ∃ for the intersection and union operators on Ω, thus providing topological interpretations of the logical quantifiers. A lattice closed under meets and joins of arbitrary sets of elements is said to be complete. Thus Ω is an example of a complete Heyting Algebra.

For any truth value u and any potential extension τ let u_τ = u∧B_τ. Also take Ω_τ = { u_τ | u ∈ Ω}. We call these the restriction of u and Ω to τ.

For any potential extension σ, we use |σ| to denote the state associated with σ. States associated with potential extensions are called potential states. If σ_1 and σ_2 are potential states we say σ_2 extends σ_1 and write σ_1 ≤ σ_2 if σ_1 = σ_2 or σ_2 only differs from σ_1 in that some constants undefined in σ_1 are defined in σ_2. By constraint, σ ≤ τ will imply |σ| ≤ |τ|. Note however that the reverse is not true. Our tree is “intentional”. Except for stated constraints we can specify nodes and states any way we like. (It’s our tree!). In general there may be lots of incompatible nodes (with respect to the tree ordering) with the same state. We say σ is final if |τ| = |σ| for all τ extending σ. We assume that for every node σ there is a final node τ with σ ≤ τ and |σ| = |τ|. This assumption will ensure that we have enough neighborhoods to handle NULL semantics correctly. For any term α and potential extension τ, we write τ(α)↓ if α takes a value ("NULL is not considered a value) in |τ| and in that case we use τ(α) to denote the value.
Our goal is to associate structures with each potential extension with which we will define the associated interpretations of the database. Our plan is then to define a translation of SQL queries into the language, L, of these structures, and show that equivalence with respect to all interpretations of the translations associated with two queries implies result set equivalence of the queries. We will then appeal to the results of [S&L] to show that the logic of topoi is sound with respect to these interpretations.

We will develop L and its models in stages. We start with terms constructed using operator symbols and constants representing the defined values of data types and occurrences of “NULL.” We are not including terms for tuples or sets at this point. Our first step is to interpret equality in Ω.

Let |L| denote the set of closed terms of L and let α and β be such terms. We define the truth value of α=β, denoted ||α=β||, to be {τ|for all τ extending σ we have τ(α) ⊥, τ(β) ⊥, and τ(α) = σ(α) = τ(β) = σ(β) }. It is clear that ||α=β|| is open. ||α=α|| is called the extent of α and can be understood as the truth value of “α exists.” The extent of α consists of all elements in which α takes a stable hence meaningful value. The “Ω-analogues” of symmetry and transitivity hold, i.e., ||α=β|| = ||β= α|| and for any α, β, and γ, || α=β || ∧ || β=γ || ≤ || α=γ ||. We extend our interpretation of equality to a notion of equivalence, ≡, which also takes “undefinedness” into account by defining ||α≡β|| = {τ|for all τ extending σ we have either τ(α) ⊥ and τ(β) ⊥ or τ(α) ⊥ and τ(β) ⊥ and τ(α) = τ(β)}.

It is clear that ||α≡β|| is open and that the Ω-versions of symmetry and transitivity hold. Further we have ||α≡ α || = 1. A set A equipped with an equality function ||. . . = . . . ||: A × A → Ω, where Ω can be any complete Heyting algebra, which is symmetric and transitive in the sense defined here is called an Ω-set. As discussed in [S&L] an Ω-set is a suitable domain for a model of intuitionistic first order logic.

Our next step is for each term α and each potential extension τ to assign an interpretation of α at τ, Iα(τ), in such a way that for all α, β, and τ, ||αβ|| = 1, iff Iα(τ) = Iβ(τ). For any term α we define a function S(α): |L| → Ω, by S(α)(β) = ||α=β|| and for any extension τ, define Sτ(α): |L| → Ω, by Sτ(α)(β) = ||α=β||. We view S(α) as an Ω predicate that measures the extent to which a given term β is equal to α and so serves as the restriction of S to extensions of τ. Using the notation of sets we interpret these predicates as the global and local interpretations of || β = α ||. For this reason we call S(α) the singleton of α. Suppose ||αβ|| = 1. For any Y we have ||αβ || ∧ || Y || ≤ ||αβ Y ||, and so, since ||αβ|| = 1 and (1,∧u) = u, for any u, we have ||βY|| ≤ ||αβ Y ||. Applying the same argument with α and β reversed we get ||α=Y|| ≤ ||βY||, and so ||α=Y|| = ||βY||. It follows that ||α=Y|| = ||βY||, for all Y and so S(α) = S(β). On the other hand suppose S(α) = S(β). So in particular we have S(α)(α) = S(β)(α) = S(α)(β) = S(β)(β). Let σ be any extension of τ. We want to show that σ is in ||α=β||. By our “final node” assumption we can assume that σ is final. If one of α or β is defined at σ, say α, then σ is in S(α)(α) and so must be in S(α)(β) and so is in ||α=β|| and so is in ||α=β||. Otherwise both are undefined at σ and so again σ is in ||α=β||. Thus the condition stated at the beginning of the paragraph is satisfied for Iα(σ) = S(α).

Continuing to follow [S&L] (into a somewhat difficult technical argument), we abstract the notion of singleton as follows. A function I: |L| → Ω is a singleton if for any terms β and γ, I(β)∧||β=γ|| ≤ ||γ|| and (ii) I(β)∧||Y|| ≤ ||β=γ||. Note that if I is a singleton then for any extension τ, so is the restriction of I to τ, Iτ defined by Iτ(β) = I(β)∧βτ. For part (i) we have I(β)∧||β=γ|| ≤ B, ∧ ||Y|| ≤ ||β=γ|| and similarly for the second part of the definition. The collection of singletons forms an Ω-set under the equality map, ||| = | | = Y | = Y | ∧ | Y | with Y ranging over |L|.

The fact that S(α) satisfies this definition for any term α follows from the symmetry and transitivity laws for || . = . ||. Further we have |||S(α)=S(β)|| = |||α=β|| for if σ is in ||α=β|| then σ is in ||S(α)(α)∧S(β)(β)|| and if σ is in ||S(α)(γ)∧S(β)(γ)|| for some Y then by transitivity σ is in |||α=β||. An Ω-set is complete if every singleton is equal to S(α) for some α. For the purposes of serving as a model of intuitionistic logic the importance of completeness is that it allows definition of terms by definite description, by which we mean terms of the form “x of the y satisfying ϕ(x)” which we interpret as “the unique x satisfying ϕ(x).” In SQL definite descriptions correspond to scalar selects (though with the table expression issue discussed in the introduction.) In fact the Ω-set A = { | : | L | is a singleton over |L| } (it can be shown that if A is a singleton over Ω then A = S(I) where I is defined by I(α) = ϕ(S(α)) [S&L: 4.18]. It is shown in [S&L: 5.18] that extending the domain to A does not change the interpretation of terms and sentences from L(A) and so without loss we will take A to be the set of singletons over |L| and for any aeA and any extension τ use aτ to denote the restriction of a to τ. Thus for any term α we have |α| = S(α) and |α|τ = Sτ(α).
At this point we can add relation symbols to our language and extend our model to be a model of first order intuitionistic logic [S&L: Chapter 5]. We interpret an n-ary relation symbol in L as an n-ary relation on A where we define an n-ary relation on A to be a function R:A^n→Ω such that for all a, b, i = 0,...,n-1, from A, ||a_i=b_i|| ∧...∧ ||a_n=b_n|| ∧ R(a_0,...,a_n)≤ R(b_0,...,b_n). This requirement can be understood as the generalization to Ω-sets of the classical notion of extensionality. For any extension τ we define the restriction of R to τ, R by R(a_0,...,a_n) = R(a_0,...,a_n) ∩ ABτ. As shown in [S&L: 5.4] the extensionality requirement is equivalent to the "evolutionary consistency" requirement that for any a_0,...,a_n-1= a in A, and any extension τ, R(a) = R(a) ∧ B_τ. R is said to be strict if for all a, R(a_0,...,a_n) ≤ ||a_0=a_0|| ∧...∧ ||a_n=a_n||, i.e. for R to hold for a given tuple of elements, all elements of the tuple must exist. Strictness guarantees that when passing from A to Â the interpretation of R extends uniquely [S&L: 5.3]. Similar definitions define an n-ary operation and strictness for n-ary operations.

Extending our model to include appropriate n-ary relations allows us to interpret first order sentences over L as truth values and terms constructed using definite descriptions [S&L: 5.13,5.19] interpreting logical operators on Ω as described earlier. We will call such an interpretation an Ω-model. A formula ϕ(x_0,...,x_n) is said to be valid in an Ω-model if for every set of closed terms α_0,...,α_n, the interpretation of ϕ(α_0,...,α_n), ||ϕ(α_0,...,α_n)|| = 1. From [S&L: 5.14,5.20] we have that intuitionistic first order logic is sound with respect to these interpretations, i.e., if ϕ is provable in intuitionistic first order logic (IFOL) then ϕ is valid in every Ω-model. (Note that to avoid notational complexity we have not equipped L with multiple "sorts" which could correspond to the multiple column types of SQL. This could of course be done, in which case the definition of an Ω-model would be extended to include multiple Ω-sets, one for each sort assignment of sorts to variables and constants and so on. Alternatively we could just treat all database values as coming from a single "sort", say byte vectors.)

Even if we were willing to give up operators generating and operating on aggregations however, intuitionistic first order logic would not be adequate to describe SQL semantics. The problem is related to strictness. For example, suppose T is a table with columns "x", "y", and "z" and just one element <x=2,y=NULL,z=4> to be a function R:AXB→C on scalar domains naturally lifts to an operator h^~ on the corresponding singleton domains by mapping a product of singletons to a singleton of products and taking the image of that under h as the value of h^~. Of course the value will be the empty singleton if one of the arguments is the empty singleton.

In order to deal with NULL elements in tuples we need a mechanism for representing potentially non-existent data in terms of data which is guaranteed to exist. This problem is similar to the one that caused us to introduce singletons over A×B then I is a singleton over A×B if I(a,b) is a singleton on A and I(b) is a singleton on B. We have ||<a,b>=<c,d>|| = ||a=c|| ∧ ||b=d||. The reader can easily check that this defines an Ω-set. We take <a,b>=<a,b,> and S(<a,b,c><c,d>) = S(a)c ∧ S(b)d. Further if I is a singleton over A×B then I(a,b) is a singleton on A and I(b) is a singleton on B.

So our next step is to add tuples and sets to our models and extend our logic to higher types. The relevant constructions are from [S&L: 4.8(iv) for products and 4.14(vi) for power set]. We start with products. Let A and B be complete Ω-sets. We define an Ω-set A×B = {<a,b> | a∈A ∧ b∈B ∧ ||a|| = ||b||}. Note that we are requiring the components to have the same extent. We have ||<a,b>=<c,d>|| = ||a=c|| ∧ ||b=d||. The reader can easily check that this defines an Ω-set. We take <a,b>=<a,b,> and S(<a,b,c><c,d>) = S(a)c ∧ S(b)d. Further if I is a singleton over A×B then I is a singleton on A and I(B) is a singleton on B.
over \( B \) and so \( I = S(<a,b>) \) for \( I_a \) where \( I_a = S(a) \) and \( I_b = S(b) \). Thus \( A \times B \) is complete. The coordinate maps are easily seen to be extensional and strict.

For \( A \) a complete \( \omega \)-set, the power set of \( A \), \( P(A) \), is defined as the set of predicates \( p:A \rightarrow \omega \) satisfying (i) extensionality: for all \( a,b \in A \), \( p(a) \land |a-b| \leq p(b) \) and (ii) strictness: for all \( a \in A \), \( p(a) \leq |a=a| \). For \( p,q \) in \( P(A) \) we define \( ||p=q|| = \forall a(p(a) \rightarrow q(a)) \land (q(a) \rightarrow p(a)) \). For \( p \in P(A) \), \( \tau \) an extension, we define \( p:\tau:A \rightarrow \omega \) by \( p(\tau)(a) = p(a) \wedge B_\tau \). If \( I \) is a singleton over \( P(A) \) then \( I = S(p) \) where \( p \) is defined by \( p(a) = \exists q(||q||)(a) \). Under these definitions it can be shown that \( P(A) \) is a complete \( \omega \)-set.

We now extend \( L \) and \( A \) to include sorts and corresponding terms for tuples and power sets for all finite levels of the power set hierarchy giving us a language and a model for intuitionistic higher order logic (IHOL). This is the model in which we wish to interpret SQL queries. The constraints on the restrictions of our interpretations to extensions ensure that the models associated with extensions will evolve consistently along any potential path. In this higher order logic we will map SQL relations to constant terms whose sort is the power set of tuples of power sets of scalar sorts. Formulas in this logic are interpreted as predicates, i.e., maps into \( \omega \) on these tuples of singletons. In particular, sentences (formulas without free variables) are interpreted as truth values.

Rather than interpreting SQL directly in the model we will map queries into a theory of IHOL. The details of the particular choice of theory don’t really matter but for definiteness we can take it to be the theory of [LoT: Chapter 3]. The mapping into the theory is not especially elegant and probably not all that useful. Its main purpose is to justify the interpretation of SQL in the theory of topoi. The advantage of the categorical interpretation is in its abstraction. In mapping to the theory we are forced to deal with unpleasant representational details that are magically abstracted away in the categorical version. In the course of showing that IHOL defines a topos (Chapters 3 and 4 of [LoT]) Fourman demonstrates that IHOL is more than powerful enough to represent relational database constructions. We will confine ourselves here to showing how to deal with the “peculiarities” of SQL which of course mostly center on the interpretation of NULL values.

So far we have been purposely vague about what we mean by SQL as SQL varies over time and vendor. To avoid argument we will exploit tautological reasoning by taking SQL to be the language defined by exactly those operators for which we define a translation. To keep things simple we will not include tables of tables etc. though, unless the standards committees have done something strange, those domains should cause no problem. We also won’t include any kind of mutable data types (aka “objects”) as those really are strange and violate our semantic assumptions. For any SQL expression \( S \) (term or predicate) we use \( S^1 \) to denote the translation. For SELECT statements (aka queries) the goal of the translation is to construct a formula satisfying the equivalence condition discussed above, i.e., a tuple is in the result set iff the corresponding closed term provably satisfies the translation.

In order to distinguish “duplicates” we will assume that the underlying tuple types for terms representing relations include a special component usually called “rowid”. For each table in the database we add as an axiom \( \forall t(R(t) \leftrightarrow (t=t_0 \land \forall v(t=t_1 \lor \cdots \lor t_v=t_{v_0})) \) where \( R \) is the constant representing the table and the \( t_i \) are constant terms for tuples representing the rows of the table (whose components are constants for singleton sets.) As before, we introduce distinct constants for occurrences of NULL values in these tuples. In mapping SQL to the logic we need to distinguish between the component of a row and the value of that component. In representing a row \( r \) as tuple in the logic we will map “\( r.\text{col} \)” to \( I_a.(R_\text{col}(r)(a)) \). Note that \( R_\text{col}(r) \) exists but \( r.\text{col} \) may not. We will also add axioms asserting that row ids of distinct rows are distinct, i.e., using \( R_\text{rowid} \) as the name of the row id component projection for the tuple sort of \( R \), \( \forall t_1,t_2((R(t_1) \land R(t_2) \land R_\text{rowid}(t_1)=R_\text{rowid}(t_2)) \rightarrow t_1=t_2) \). When translating SELECT statements we include the row id column when we want to preserve “duplicates”. When we want to eliminate “duplicates” as in SELECT DISTINCT or SET operators (UNION, INTERSECT, etc.) we treat the SELECT as a GROUP BY over the distinct columns and map rowid to the least rowid in the group thus preserving the ability to distinguish rows in follow on projections. We will also imagine that we have added axioms asserting that all the data operations of SQL are “strict”, i.e., take NULL if any of their arguments are NULL and enough axioms to compute the value of any expression of the form \( S(c_0,\ldots,c_n) \) where \( S \) is a built-in operator or predicate. (For predicates we’ll need to include negative as well as positive assertions.)
Let $\tau$ be a final extension of the state of the database. The goal of the translation is to translate an SQL query $q$ to a formula $\varphi(x)$ in one free variable $x$ so that

$$(**)$$ For any value $v$, $v$ is in the result set of $q$ if and only if $\varphi(`v')$ is true at $\tau$ where `$v'$ is a closed term denoting $v$.

Assuming this is the case then for any queries $q_1$ and $q_2$ with translations $\varphi_1$ and $\varphi_2$ if $\varphi_1$ and $\varphi_2$ are logically equivalent they will evaluate equivalently at $\tau$ and so $q_1$ and $q_2$ are guaranteed to have equivalent result sets in the database. In the following we will use $S^*$ to denote the translation of an SQL expression $S$ to our logic. Note that in the final extension, all NULL column entries evaluate to the empty set and thus are equal. Thankfully this exactly matches the SQL standard for the handling of NULL columns in SELECT DISTINCT and related operators. In terms of the theory evaluation at $\tau$ can be simulated by adding as axioms “$\neg(N_i = N_j)$” for all NULL constants.

Our main problem is how to translate explicit and “implicit” NULL tests. By explicit occurrences of NULL tests we mean predicates of the form “$v$ IS NULL” or “$v$ IS NOT NULL” or terms involving operators like “NVL” and “COALESCE”. These are easy because we can translate “$v$ IS NULL” to “$v = \text{NULL}$” and “$v$ IS NOT NULL” to “$v = \text{NULL}$”. These will evaluate correctly in a final extension of the current state. Implicit NULL tests occur in the SQL NOT operator and table expressions, (subqueries and derived tables.) In these cases the NULL test is (implicitly) applied to a predicate expression rather than a data term. Fortunately, under the SQL evaluation rules, we can reduce NULL tests on predicate expressions to NULL tests on terms. Let $F(x_0, \ldots, x_n)$ be a predicate expression in SQL with free variables $x_0, \ldots, x_n$. We construct a predicate $F_{\text{NULL}}(x_0, \ldots, x_n)$ in the same free variables such that (i) $F_{\text{NULL}}$ is never NULL and (ii) For any closed terms $t_0, \ldots, t_n$, $F(t_0, \ldots, t_n)$ evaluates to NULL iff $F_{\text{NULL}}(t_0, \ldots, t_n)$ evaluates to TRUE. We construct $F_{\text{NULL}}$ recursively. If $F$ is a quantifier expression we take $F_{\text{NULL}}$ to be FALSE because quantifier expressions in SQL can never evaluate to NULL. We also take $F_{\text{NULL}}$ to be FALSE if $F$ is an explicit NULL test on a term. If $F$ is of the form NOT $G$, we take $F_{\text{NULL}}$ to be $G_{\text{NULL}}$. If $F = G$ OR $H$ we take $F_{\text{NULL}} = (G_{\text{NULL}} \text{ AND } H_{\text{NULL}})$ OR $(G_{\text{NULL}} \text{ AND } (\neg H_{\text{NULL}}))$ OR $(H_{\text{NULL}} \text{ AND } (\neg G_{\text{NULL}}) \text{ AND } (\neg H_{\text{NULL})))$ and similarly for $F = G$ AND $H$. Finally if $F = p(a_0, \ldots, a_n)$ where $p$ is a “built in” predicate and $a_0, \ldots, a_n$ are SQL terms we take $F_{\text{NULL}} = ((a_0 \text{ OR } a_0) \text{ AND } \ldots) \text{ OR } (a_0 \text{ OR } a_0))$ where $a \text{ OR } b$ is defined in SQL as $((a \text{ IS NULL}) \text{ OR } (b \text{ IS NULL}) \text{ OR } (a = b))$ and $a \text{ OR } b$ is $\neg(a = b)$.

We use the $F_{\text{NULL}}$ mapping to handle the translation of built in predicates, the SQL NOT operator and table expressions as follows. In order to represent the undecided expressions we add a 0-ary predicate “$?()$” to our logical theory with no corresponding axioms. $?()$ is thus an unspecified truth value that will act as a stand-in for “NULL”. A formula involving $?()$ that is logically valid for our theory, i.e. true in all models, will therefore remain true under any interpretation of $?()$ and so is independent of $?()$ which is exactly the behavior we want for “NULL”. (We could avoid introducing $?()$ by eliminating uses of “NULL” in the translation process, e.g. at the final SELECT boundary but this seems simpler.) We translate a simple built-in predicate expression $F = p(s_0, \ldots, s_n)$ to $(\neg(F_{\text{NULL}} \text{ OR } ?()) \text{ OR } (F_{\text{NULL}} \text{ AND } ?()))$. We will translate “NOT $F$” to $(\neg(F_{\text{NULL}} \text{ OR } ?()) \text{ OR } (F_{\text{NULL}} \text{ AND } ?()))$. We translate a table expression of the form, “SELECT ... WHERE $F(...)”, to $(\neg(F(...) \text{ OR } ?()) \text{ OR } (F(...) \text{ AND } ?()))$ thus explicitly eliminating partial values.

Numerical aggregation operators can be represented as terms for single valued predicates on $P(A)\times N$ where $N$ is the sort of numerical values and $P(A)$ is the power sort over the sort being aggregated. We can assume we have terms for all such possible operators and sufficient axioms to compute them over finite sets including how to handle NULL values. An expression of the form $\text{SELECT } \text{agg}(...)$ ... WHERE $F$ maps to $\text{I}_y \text{AGG}(... | F(...) \text{ AND } \neg(F_{\text{NULL}}(...)))y$ where $\text{AGG}$ is the relational term representing agg. GROUP BY operators are interpreted as maps to a power sort of a power sort. For example, an expression like “SELECT $h$, $\text{agg}(t)$ WHERE $F(t)$ GROUP BY $h$” can be translated to $\langle ch, n, p : \exists g, P(T) \{ g = \langle t : F(t) \text{ AND } h(t) = h_0 \rangle \text{ AND } \exists t, g(t) \text{ AGG}_n(g, n) \} \rangle$ where $\text{AGG}_n$ is the particular relational representation of the aggregator which aggregates subsets of $T$ by $h$. Here $h$ and $\text{AGG}_n$ operate on singletons and sets of singletons respectively. (The trickiest part of the translation is figuring out when you want scalars and when you want singletons.)

The last peculiarity we deal with is OUTER JOIN. The idea of outer join is to take an ordinary join and extend it so that the projection onto one of the components of the join is surjective. There are various ways of doing this but
the most useful, we believe, from the standpoint of reasoning about queries, is to express the outer join as an inner join with an extension of the opposite component.

In the diagram above we obtain the left outer join \(A,B\) by extending \(B\) and \(g\) to \(B^{*}\) and \(g^{*}\) so that the projection into \(A\) of the inner join (aka pullback) of \(A,B^{*}\) with respect to \(f,g^{*}\) is onto \(A\). The maps from \(B\) to \(B^{*}\) and \(IJ(A,B)\) to \(LOJ(A,B)\) are injections. Categorically we take \(B^{*}\) to be “minimal” in that any other extension having the associated maps will factor uniquely through \(B^{*}\). Roughly we do this by adjoining, via a direct sum, \(D=f[A] + g[B]\) where \(f[A]\) and \(g[B]\) are the images of \(A\) and \(B\) under \(f\) and \(g\). The NULL value traditionally taken as the second component in the outer join is just a marker, i.e. it doesn’t represent any kind of potentially evolving element, which we can just take to be the empty set. As indicated on the diagram, any map on \(B\), e.g. \(h:B \to E\), extends to \(B^{*}\) and so to \(LOJ\) which is implemented in SQL by mapping extra rows to NULL. (Note that we aren’t suggesting computing outer joins this way. The point is that the result is isomorphic to the standard computation and so we can reason about outer join using standard properties of inner joins, like associativity and commutativity. Expressing this construction in the logic is analogous to the definitions in [LoT: Chapter 3] and the axioms for aggregates described above.

The verification of (***) is handled by showing that in the theory augmented with the axioms declaring null constants non-existent, call the theory \(T\), we have for every closed expression of the form \(S^{i}\),

(###)  
(i) if \(S\) is a term then either \(T_{r}\) proves \(\langle\langle S^{i}=S^{j}\rangle\rangle\) in which case \(S\) evaluates to NULL or \(T_{r}\) proves \(\langle\langle S^{i}=d\rangle\rangle\) where \(d\) is the constant representing the value of \(S\);

(ii) if \(S\) is a predicate expression then \(T_{r}\) proves \(S_{\text{NULL}}\) if \(S\) evaluates to NULL and \(T_{r}\) proves \(\langle\langle S_{\text{NULL}}\rangle\rangle\) otherwise.

(iii) if \(S\) is a predicate, then \(T_{r}\) proves \(S^{i}\) or \(T_{r}\) proves \(\langle\langle \sim S^{i}\rangle\rangle\) or \(T_{r}\) proves \(\langle\langle \sim S^{i}\leftrightarrow ?()\rangle\rangle\) exactly in accordance with whether \(S\) evaluates to true, false, or NULL.

The proof, by induction on the structural complexity of \(S\), is outlined in the appendix.

If we accept the fact that the translation of SQL into IHOL is faithful we can now forget about IHOL and instead use the interpretation in category theory, i.e., in the theory of topoi, described in [S&L] and [LoT]. For example the diagram for outer join above can be used to validate some standard observations about outer join. Suppose \(p:E \to \Omega\) is a filter on \(E\) and we extend it to \(p^{*}=p \oplus \text{false}\) then \(p^{*}\) can filter the LOJ and we see that the result is the same as filtering the inner join on \(p\). On the other hand if we filter the LOJ by a filter on \(A\) then the filter can be pushed inside the LOJ and more generally any inner join against the outer join by a map only referencing \(A\) then the join can be pushed inside the outer join.

In the previous examples we have interpreted a query as the construction of a predicate. What we want to do now is interpret a query as a map on power sets. We can view this interpretation as the analogue of relational algebra. The intuitive idea is that of “lifting” a construction to the power set using exponentials. For example, for objects \(A,B\) and \(C\) we have a map: \(A \times B \times C^{B} \to \Omega^{A} \times \Omega^{B} \to \Omega^{A \times B}\) defined by \(\langle\langle a,b,c \rangle\rangle \mapsto \text{eval}_{A} \times \text{eval}_{B} \times c\) where \(\text{eval}_{A}\) and \(\text{eval}_{B}\) are the maps which evaluate the projections to \(A \times B\) and \(B \times C\) respectively and \(\text{eval}_{C}\) is the classifier of \(\langle \text{id}_{A}, \text{id}_{C}, b \rangle : C \to A \times B \times C\). Using this we obtain a map: \(C^{B} \times C^{B} \to \Omega^{A \times B}\). We can view this as the internal representation of the join operator. Similarly we can construct a map corresponding to the product operator which maps \(\Omega^{A} \times \Omega^{B} \to \Omega^{A \times B}\) using the classifier of \(\langle\langle \text{id}_{A}, \text{id}_{B}, b \rangle\rangle\). “Projections” are trickier but we also have maps: \(\Omega^{A} \times B^{B} \to \Omega^{B}\) which construct images of subobjects and similarly for all the other constructors of relational databases. (The internal version of group by for
example is a map taking \( \Omega^A \times B^A \rightarrow P(B \times \Omega^A) \), using \( P \) to denote the power sort object.) To illustrate these ideas we will look at an example from [ASE15]. Most of [ASE15] is concerned with search strategies and we have nothing to add here on that topic. What we do want to look at the categorical representation of the query plan for the example constructed in the paper.

```sql
SELECT l2, sum(13) AS s_l3, o3, o4
FROM C, O, L
WHERE c1-o2 AND o1-l2
    AND c2='automotive' AND o3 < date 'yesterday' AND 14 > date 'yesterday'
GROUP BY l2, o3, o4 ORDER BY s_l3 desc, o3;
```

The plan from [ASE15] makes use of indexes iC21 on i(2,1), iO2134 on O(2,1,3,4) and iL243 on L(2,4,3). \( O_2, L_4 \) etc. name the codomains of the column maps. For typographical reasons we use \( P(A) \) below for power set of \( A \) (instead of \( \Omega^A \)). While the diagram has the shape of a tree it can be viewed as linear with implicit products of the objects at each level and tupling of the maps. The annotations on objects, e.g. \( \{o2,o1,o3,o4\} \), represent the structure of the objects, in this case the order structure, and the maps, e.g. “merge_join”, represent structure preserving “morphisms” as is usual in applications of category theory.

**References.**


Appendix: Additional Remarks and Examples

Here are more details on the proof of \#\#\. We assume (\#\#) holds for all subexpressions of \( S^i \). If \( S^i \) is a constant or an expression of the form \( f(s_{0}, ..., s_{n}) \) where \( f \) is an operation then the axioms of \( T \) and the induction hypothesis assure that (i) holds for \( S^i \). If \( S^i \) results from a predicate of the form \( p(s_{0}, ..., s_{n}) \) where \( p \) is a built-in predicate symbol then \( S^i \) has the form \( (\neg F_{\text{null}} \land \tau()) \lor (S_{0} \land S_{1}) \). By the induction hypothesis applied to the \( s_{0}^i \) and the definition of \( F_{\text{null}} \) for built-in predicate expressions this will be provably equivalent to \( \tau() \) if any of the \( s_{i} \) evaluate to \( \text{null} \) and \( p(s_{0}^i, ..., s_{n}^i) \) otherwise. In the latter case the inductive hypothesis ensures that the \( s_{i} \) are provably equivalent to data value constants and the axioms for \( p \) will then ensure that \( T \) correctly proves either \( p(s_{0}^i, ..., s_{n}^i) \) or \( \neg p(s_{0}^i, ..., s_{n}^i) \). If \( S \) is of the form “\( P \lor Q \)” or “\( P \land Q \)” then \( S^i = (P^\lor Q^\lor) \), or \( (P^\land Q^\land) \). By induction hypothesis \( T \) correctly proves the equivalence of \( P^i \) and \( Q^i \) to one of true, false, or \( \tau() \) and so does the same for \( S^i \). Further, applying the induction hypothesis for (ii) to \( F_{\text{null}} \) and \( Q_{\text{null}} \) and the definition of \( S_{\text{null}} \) it is clear that \( S_{\text{null}} \) will satisfy (ii). Suppose now that \( S \) is of the form “\( \neg P \)”\). Then \( S^i \) has the form \( (\neg P^i \land \neg P^\lor) \lor (P^i \land \neg \tau()) \). Applying the induction hypothesis (ii) to \( F_{\text{null}} \) and the observation that \( S \) evaluates to \( \text{null} \) just in case \( P \) evaluates to \( \text{null} \), we see that either \( T \) proves \( P^\lor \) in which case \( S \) evaluates to \( \text{null} \) and \( S^i \) is provably equivalent to \( \tau() \) or \( T \) proves \( \neg P^\lor \) in which case \( T \) proves either \( P^\land \) and \( P \) evaluates to true or \( T \) proves \( \neg P^\land \) and \( P \) evaluates to false and hence (iii) will be satisfied for \( S \). (iii) is satisfied in this case as \( S_{\text{null}} = P_{\text{null}} \). Expressions of the form “\( S \) IS NOT \( \text{null} \)” or “\( S \) IS \( \text{null} \)” translate to \( S_{\text{null}} \) and \( \neg S_{\text{null}} \) respectively. By induction hypothesis (ii) is satisfied for \( S_{\text{null}} \) and so (iii) is satisfied for the respective cases. Further, \( F_{\text{null}} \) is the false predicate for both these cases.

Now suppose \( S \) has the form “\( \text{SELECT} \, s_{1}^i(\tilde{a}_{1}), ..., s_{n}^i(\tilde{a}_{n}) \, \text{FROM} \, T_{1}, ..., T_{m} \, \text{WHERE} \, F(b_{1}, ..., b_{n}) \)” where the \( \tilde{a}_{i} \) are the lifted forms of scalar functions on tuples of column entries, i.e.\. they represent maps from tuples of singletons to singletons of tuples, and \( F \) is a predicate on column values from the \( T_{i} \) (We are not grouping or aggregating in this case.)\. The translation can be divided into three parts. The first represents \( \langle t_{1}, ..., t_{n} : F(b_{1}, ..., b_{n}) \rangle \); the second part projects these to tuples of column entries \( \langle \tilde{a}_{1}, ..., \tilde{a}_{n} \rangle \) and the third part projects these via \( \langle s_{1}, ..., s_{n} \rangle \\rangle \). The last two parts are not problematic. In the first part the translation replaces \( F(b_{1}, ..., b_{n}) \) with \( F(^{\langle \ldots \rangle}) \lor \neg F_{\text{null}}(^{\langle \ldots \rangle}) \) thus assuring that the values are fully decided, i.e.\. true or false. Since the \( t_{i} \) come from \( T_{i} \) and the axioms of \( T \) fully specify the tuples of the \( T_{i} \) we can enumerate the tuples satisfying \( F(^{\langle \ldots \rangle}) \lor \neg F_{\text{null}}(^{\langle \ldots \rangle}) \) into an explicit set and \( T \) can prove that this is exactly the set of tuples satisfying \( F(^{\langle \ldots \rangle}) \lor \neg F_{\text{null}}(^{\langle \ldots \rangle}) \). Further by the induction hypothesis these correspond exactly to the tuples in the result set of this part of the query. Of course the \( \text{NULL} \) test for selects is always false. Finally we consider the translations of \( \text{DISTINCT}, \text{GROUP BY}, \) aggregates and set operations (\( \text{UNION}, \text{INTERSECTION}, \text{etc.} \)). As noted earlier the SQL rules on grouping which identify \( \text{NULL} \) values hold in \( T \).

It always seems surprising when work which is apparently motivated by pure science, such as the categorial representation of logic, so directly applies to an engineering problem. The particular case described in this paper makes sense at an intuitive level we believe. Databases are pragmatic structures designed to model real world systems whose descriptions are almost always partial. Intuitionistic logic is a kind of “pragmatic logic” designed to reason about mathematical systems for which knowledge is necessarily partial. (The connections between intuitionistic logic and category theory which trace back to connections between logic and topology probably give more cause for surprise.)

We should point out that the program carried out in this paper doesn’t exactly match the program discussed in the introduction. In the introduction we described a process which involves translating a query into a formula of logic; reasoning in the logic to obtain a logically equivalent formula; and finally translating the formula back into a query. This would probably require a characterization of which formulas can be translated back to queries and perhaps some constraint on the logic which would guarantee that derived formulas are translatable. (On the other hand if our goal is to create a plan and we are reasoning about plans then we probably don’t care whether the result can be translated back to SQL.) We should also point out that while we proved soundness for the theory with respect to query evaluation equivalence, we did not prove completeness.

The fact that a SQL database can be interpreted as a topos and the fact that a topos is equivalent to a theory as Fourman shows in [LoT], leads to the following idea. In this equivalence of an IHOL theory and a topos, truth values are represented as equivalence classes of sentences. (The structure of these equivalence classes under
provability is called the Lindenbaum-Tarski algebra.) This provides a natural way of explicitly representing an $\Omega$-predicate. We could represent the result of a query as a table of data rows together with a column for the partial truth value and truth values would be represented as sentences. If the truth value of a row becomes false we would eliminate it. Otherwise we would just carry it along. For example, if we had a row $<5,\text{NULL}>$ in a table $(x,y)$ then this row would be retained in the result “set” of a query for $x=5$, $y=2$ with truth value “$y=2$”. If the system being modeled evolved in a manner consistent with the semantics underlying SQL then query updates would only require reevaluating rows with partial truth values.

Note that the equivalence of topoi and theories of IHOL discussed in [LoT] implies that any consistent extension of IHOL defines a topos and so has a model which is a topos. Thus we can extend the semantics of SQL in any consistent way without contradicting results about SQL derived from the theory of topoi. One way to think of this is to generalize our conception of the potential extension tree by taking the “state” associated with a potential extension to be a consistent theory extending the theory of its parent and ensuring that all such consistent extensions are possible. For example one might want to vary the way SQL treats NULL values in aggregations or GROUP BY or the SET operations perhaps by declaring them all distinct then this would correspond to a different final extension and we would simply modify ### to apply to that extension.

Though the high degree of abstraction of category theory is often intimidating to us mortals it does provide a framework capable of unifying apparently disparate systems. A possible application might be the current dichotomy provoked by the “NoSQL” approach to data management. When data manipulation is represented purely in terms of maps and their behavior without regard to underlying representation then all that matters are the behaviors of the maps and their costs. After all, “map-reduce” is semantically a grouped aggregate and the underlying representation of “tables” in relational databases is often not tabular. Within a common planning framework one could consider options involving various combinations of representations and strategy including “relational” and “non-relational” components.

Here is another example of a query represented in categorical form. A standard optimization for GROUP BY is to push the grouping below a join. A map $f:A \to C$ induces a map $f^1: \Omega^C \to \Omega^A$, the preimage map. Composing this with the singleton map $\{\cdot\}:C \to \Omega^C$ we obtain the GROUP BY $f$ map which maps $C$ into $\Omega^A$. If $f=g \circ h$ then $f^1=h^1 \circ g^1$. Consider applying this to the case of a join. So in general we can first group $A$ by $g$ but that will only be of use to us if we can collapse the groups to a single row for the purposes of applying $\Pi_A^{-1}$. For this we need to know that equivalence with respect to $g$ implies equivalence with respect to $c$, i.e. each $g$-class determines a unique $c$ class. This follows if $c$ factors through $g$, i.e., there exists $h$ such that $c = h \circ g$. 

![Diagram of categorical optimization for GROUP BY](https://via.placeholder.com/150)