The relational data model is based on considering normalized tables as mathematical relationships. Powerful query languages can be defined over such mathematical relationships based on a form of mathematical logic called predicate calculus. The initial relational database query language (called Alpha) had pure predicate calculus syntax (logical expression). It was claimed that the predicate calculus syntax was difficult to understand for non-mathematicians, and therefore the query language SQL was developed having a more natural-language syntax (English-like). The SELECT statement in SQL is the basic query primitive, in fact, the SELECT statement is syntactically sugared predicate calculus based on Alpha. Over time, SQL was extended in many ways and now includes also constructs that are more relational algebra-oriented.

The mathematical logic on which most query languages for relational databases is based is called tuple calculus. The tuple calculus is characterized by variables in queries being bound to tuples, corresponding rows in tables. In SQL, this means that the FROM clause binds variables to rows of the tables identified by the FROM clause. It is proved that all queries that can be expressed in relational algebra can also be expressed in tuple calculus, and vice versa. One says that a query language is relationally complete if it can express all queries that can be expressed in tuple calculus or relational algebra.

SQL is a relationally complete query language. Actually, the SQL language contains much more than what is required for a query language to be relationally complete. This means that one can express queries in SQL which cannot be expressed in tuple calculus. An example of what one can express in SQL that cannot be expressed in basic tuple calculus or relational algebra is aggregate functions, such as SUM and COUNT. Another important extension is that the result of a query in the tuple calculus is a set of tuples, which in SQL is generalized to the so-called bags of tuples where the same row (tuple) can occur more than once in the result of an SQL query. We therefore need a more general formal language than the relational algebra or tuple calculus to represent full SQL.

In the rest of this compendium, we discuss how a subset of SQLs SELECT statement can be expressed as tuple calculus. In order to make it simple, we explain is through a small example.

Suppose we have the tables (relations):

- \textit{STAFF (SSN, NAME, SALARY, DEPT, PHONE)}
- \textit{DEPARTMENT (DNO, DNAME)}

\footnote{The original name of the SQL was SEQUEL, Structured English Query Language. The name SEQUEL proved copyright protected and was changed to SQL.}
The first example query is:

*Find the name and telephone numbers for all staff who have salaries in excess of 100000.*

The query can be expressed in SQL as:

```
SELECT S.NAME, S.PHONE
FROM STAFF S
WHERE S.SALARY > 100000;
```

In tuple calculus the query is expressed as:

```
{s.name, s.phone | staff(s) s.salary> 100000}
```

In general, a query in tuple calculus has the form:

```
{proj (t1, t2, ..) | r (t1, t2, ...) c (t1, t2, ...)}
```

The expression

```
r(t1, t2, ...)
```

is called a *range predicate* and specifies how the variables used in the query are bound to tuples (=rows) in the specified relations (=tables). In the example, we bind the variable `s` to tuples in the relation `staff`, so the range predicate is thus `staff(s)`. The *FROM* clause in SQL defines range predicates.

The term

```
c(t1, t2, ...)
```

is called the *search condition* and selects the tuples `t1, t2, ...` to be searched for by specifying constraints on returned tuple attributes. In the example, the search condition is `s.salary>100000`, i.e. the attribute `salary` in tuples bound to variable `s` must be greater than 100000. The *WHERE* clause in SQL defines search conditions.

The term

```
proj(t1, t2, ..)
```

is called the *projection* and specifies the attributes of the selected tuples to be returned by the query. In the example, the projection is `s.name,s.telephone`, i.e. the query returns the attributes `name` and `telephone` of selected tuples `s`. In general, the projection specifies a list of attributes for some tuple variables. The *SELECT* clause in SQL defines projections.

Queries over multiple relations is specified by using a binding predicate being a conjunction of simple (atomic) binding predicates from the relations used in the query. In SQL, this means that multiple tables are specified in the *FROM* clause. Such conjunctive binding predicates represents the Cartesian product of tuples in the specified relations.

Suppose we want to query:

*Find the name, phone number and title of persons who have higher salaries than 100,000*

The query is expressed in SQL as:

```
SELECT S.NAME, S.PHONE, D.DNAME
FROM STAFF S, DEPARTMENT D
WHERE S.SALARY > 100000 AND S.DEPT = D.DNO;
```
In tuple calculus this is expressed as:

\{
  s.name, s.telephone, d.dname |
  staff(s) \land department(d) \land
  s.salary > 100000 \land s.dept = d.dno
\}

In this case, the range predicate is \textit{staff(s)} \land \textit{department(d)}. It binds the variables \textit{s} and \textit{d} to form the Cartesian product of tuples (rows) in relations (tables) \textit{staff} and \textit{department}. The variables are then used in the search to join the relations \textit{staff} and \textit{department} and to restrict the tuples \textit{s} in the relation \textit{employee} to those whose attribute \textit{s.salary} is greater than 100000.

The example SQL statements all have explicit variable names specified for all tables in the \textit{FROM} clauses. It is recommended to always use in SQL the format \textit{T.A} to explicitly specify the value of attribute \textit{A} of tuple \textit{T}. In general, one can often omit the tuple variable names in SQL if it is unambiguous. For example, it is also permissible to express the previous query without tuple variables as:

\begin{verbatim}
SELECT NAME, PHONE, AVDNAMN
FROM STAFF, DEPARTMENT
WHERE SALARY > 100000 AND DEPT = DNO;
\end{verbatim}

We say we have a \textit{complete SQL query} if we specified all tuple variables in the \textit{FROM} clause and in attribute references. A complete SQL query can be directly translated to the corresponding tuple calculus query. A non-complete SQL query can always be transformed into a complete one by introducing explicit tuple variable names.

Besides allowing easy translation into tuple calculus, using complete SQL queries has the advantage of making it possible to add new columns in the tables without changing the SQL queries. If we e.g. add column \textit{PHONE} also in the table \textit{DEPARTMENT} the above non-complete SQL becomes ambiguous since there is a column named \textit{PHONE} in both tables. The corresponding complete SQL query remains correct. We note that complete queries are less sensitive to database schema changes, they have better data independence.

In general, a predicate in tuple calculus is made up of \textit{atoms}, which are simple predicates (conditions) that may be in the following formats:

1. A \textit{binding atom} with format \textit{r(t)} binds the variable \textit{t} to each of the tuples in relation \textit{r}, e.g. \textit{staff(s)}.

2. An atom with format \textit{t.A op k}, where \textit{op} is one of the comparison operators \textless, \textless=, =, \neq, \geq, \geq and \textit{k} is a constant, defines a comparison of an attribute with a constant, e.g. \textit{s.salary > 100000}.

3. An atom with format \textit{t.A op t.B} defines a comparison between two attribute values, e.g. \textit{s.dept = d.dno}.

A \textit{formula} in tuple calculus is either an atom or several formulas connected with the logic operators \texttt{\&} (and), \texttt{\lor} (or), and \texttt{\neg} (not). A formula can thus be one of the following:

4. An atom as above.

5. If \textit{F₁} and \textit{F₂} are formulas then \textit{F₁ \& F₂}, \textit{F₁ \lor F₂} and \texttt{\neg F₁} are formulas too, e.g. \textit{staff(s)} \& \textit{s.salary > 100000}. 

\textbf{Note:} This text is an excerpt from a larger document discussing tuple calculus and SQL queries. The full context is recommended for a comprehensive understanding.
A formula can also be connected with the quantifiers $\forall$ and $\exists$:

6. If $F$ is a formula then $(\exists t \ F)$ is a formula. $\exists$ denotes there exists and the formula $F$ is said to be existentially quantified.

7. If $F$ is a formula then $(\forall t \ F)$ is also a formula. $\forall$ denotes for all and the formula $F$ is said to be universally quantified.

As an example, suppose we want to express the query:

*Find those staff members for which there exist a department where they work.*

The query can be expressed in tuple calculus as:

$$\{s.\text{name} \mid\text{staff}(s) \land (\exists d \text{ department}(s) \land d.\text{dno} = s.\text{dept})\}$$

In SQL the query can be expressed as:

```sql
SELECT S.NAME
FROM STAFF S
WHERE EXISTS(SELECT *
            FROM DEPARTMENT D
            WHERE D.DNO = S.DEP);```

Universal quantification cannot be directly express in SQL as existential quantification. However, for relational completeness it is required that universal quantification can be expressed.

So how is universal quantification expressed in SQL? For example, suppose we want to query:

*Find those departments where all staff members earn more than 50000.*

The simplest way to handle universal quantification is to instead transform the query into the corresponding negative query:

*Find those departments where no staff member earns less than 50000.*

This query can be formulated using in tuple calculs as:

$$\{d.\text{dname} \mid (\text{department}(d) \land \neg (\exists s \text{ staff}(s) \land s.\text{dept} = d.\text{dno} \land d.\text{salary} < 50000))\}$$

Notice that the result from this query includes the departments having no staff at all. If you don’t want to include the departments without staff you will have to extend the query.
In SQL the reformulated query can be expressed as:

```
SELECT D. DNAME
FROM DEPARTMENT D
WHERE NOT EXISTS (SELECT *
    FROM STAFF S
    WHERE S.DEPT = D.DNO AND
    D.SALARY < 50000);
```