SQL Database queries and their equivalence to predicate calculus

Russell Impagliazzo, with assistance from Cameron Helm

November 3, 2013

1 Warning

This lecture goes somewhat beyond Russell’s expertise, so he might make mistakes. He’s happy to answer questions, but he might have to repeat the questions to more of an expert. If you are intrigued by this topic, the course to follow up with is CSE 132, and the faculty to talk to are Alin Deutsch, Yannis Papakonstaninou and Victor Vianu.

2 Introduction

It’s become a huge cliche to say the world is entering the domain of big data. Data has ALWAYS been important! Databases are about how to we store and organise information so we can access it. According to wikipedia, the first commercial databases were introduced in the 1960’s, as businesses like IBM started looking for ways to use computers to automate their record keeping, and allow more sophisticated access to the information they were already maintaining. The relational database approach that I will sketch below was introduced by Cobb in 1970. The SQL language was developed in implementing Cobb’s approach around 1978-79. Relational databases are good for handling structured data, data that is relatively homogenous in what parts mean what. More recent work has been in extending the model to handle unstructured data.

At the most simple level, any string of 0’s and 1’s is data, but that data has little meaning by itself. In order to relate the data inside our computer to facts outside the computer, we need to keep more information about what the data is intended to mean. On the other hand, actually defining “meaning” is a deep philosophical question that is beyond the paygrade of database system designers. As a compromise, we will let the format of the data annotate its intended meaning, without needing to define exactly what the data means. Thus, as long as the people accessing the data use the same vocabulary the database designers do, the answers will be correct without having to completely understand the questions.
The relational database model gives meaning to the information via structure, but makes the structure flexible enough to allow a wide variety of types of information to be stored. It’s a compromise between ease of use, generality, and meaningfulness. This is done by giving data items and their components suggestive names and making sure the data is put in a homogenous format so that things with same name are treated the same by computations over the data. Relationships between the fields are the main way we store data.

3 An illustrative example: storing information

Say we are designing a database for human resources for our company. Here are a few of the many things our database needs to keep track of:

1. Who works for our company?
2. How much do they make?
3. Who supervises who?
4. Which employees are signed up for which benefits?

This of course oversimplifies the issue. There are many more possible pieces of data to store and many sources of complexity. (Real companies can have 1000’s of employees, subcontractors that do not work directly for the company, subsidiary companies, employees acquired from corporate takeovers and still on benefit plans not available to others in the company, ...). For our purposes, I’ll try to err on the side of over-simplification.

Not even clear how to organise the idea of who works for our company, concerns of duplicate names come up. We add a unique identifier as well (key). Here’s how we might handle some of these questions in a relational database: We can create a table, Employees, storing two keys, name and a unique id like social security numbers, to start. If we feel like it, we can add more keys, like position and salary, to the same table, or store them on different tables.

For example, the table employees might have entries (called “rows” or “tuples”) such as:

<table>
<thead>
<tr>
<th>Name</th>
<th>SSN</th>
<th>Position</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>John Smith</td>
<td>100</td>
<td>Sr. Vice President</td>
<td>$192039</td>
</tr>
<tr>
<td>John Smith</td>
<td>101</td>
<td>Jr. Peon</td>
<td>$19203</td>
</tr>
<tr>
<td>Jane Doe</td>
<td>201</td>
<td>Middle manager</td>
<td>$79203</td>
</tr>
</tbody>
</table>

Each column would have a suggestive field name, and a type. For example, although we use numbers for both social security number and salary, the types (string of digits for social security number, integer for salary) would specify what kinds of comparisons could be made. For example, 0110 and 110 are the same as integers, but not as strings.

If everyone in the company had a single manager they reported to, we could just add a Manager field to the above table. But that’s not always the case. Some employees might not report to anyone, others might report to multiple
supervisors. So instead, we could add a table expressing the “Reports To” relationship between employees:

<table>
<thead>
<tr>
<th>Employee</th>
<th>Manager</th>
<th>ESSN</th>
<th>MSSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>John Smith</td>
<td>Jane Doe</td>
<td>101</td>
<td>201</td>
</tr>
<tr>
<td>John Smith</td>
<td>John Smith</td>
<td>101</td>
<td>100</td>
</tr>
<tr>
<td>Jane Doe</td>
<td>John Smith</td>
<td>201</td>
<td>100</td>
</tr>
</tbody>
</table>

In this table, we can have an employee with no managers (no entry) or an employee with many managers (many entries).

Items: relations, tables of "tuples". Tuple would represent a row in the table. No matter the table, the are the same type of object. By presenting the data in this way, we successfully preserve a lot of meaning. The way we access the data shouldn’t depend on order in the table (how it is stored on a device etc).

Relationship database model is only one way to store information though, we have made progress in more advanced models but the "common core" remains the same.

4 Accessing information

How would we go about using the information stored in our tables? For example, say we the list of positions divided into Management positions such as Sr. Vice President and Middle manager, and non-management positions such as Jr. Peon. We want to find any employees not in managerial positions that are actually supervising employees.

Here’s how we would write such a search in the Structured Query Language (SQL). The basic syntax of a query in SQL has three parts: Select (what type of object do you want to find?), From: (what relations should we be searching through?), and the Where? (Under what conditions do you want to put the data in your answer).

Select From Where

The information we want is in neither table we created earlier, the first table will tell us who are managers and the second will tell who is supervising another employee. But in SQL, we can make queries that join this information together. To design the SQL query, it is helpful to look at the logical structure of the information we want.

Let x be a person who is a supervisor but not in a managerial position. To say that x is a supervisor, is to say there exists (∃) some employee y such that (y,x) is in Report. We also want that x.Position is not managerial. Note that when we search through Reports, we get the whole entry in the table, not just an employee number directly. Let’s call that whole entry that contains x and y “z”. So here’s how we write the query in SQL:

Selectx.SSN FromEmployee x, Reports z Where(z.MSSN = x.SSN) AND(x.PositionnotinManagerial)
To me, since I have a math background, the syntax of SQL is reminiscent of
the standard notation for describing sets. We often write:
\[ \{ f(x) \mid x \in \text{Domain} \text{ so that } P(x) \} \]
for example \( \{ p^2 \mid p \in \mathbb{Z}, p \text{ is prime} \} \) as the
set of square of prime numbers. Here, the \( f \) part is what we are “Selecting”, the
Domain is where we are selecting it “From”, and the property \( P \) corresponds to
the Where condition. If this is a helpful way to understand SQL queries, great.
If it is confusing, just ignore this point.

4.1 Nested queries

Queries produce answers in the same table format used to hold data. So to
make more complex queries, it is often useful to introduce tables we create as
intermediate steps. For example, we can also write our previous query in two
steps: create a list of managers from Reports; look through this list in Employees
to see which ones don’t have managerial positions:

\[
\begin{align*}
\text{CreateSupervisors} & : \text{Select} \ z.MSSN \ \text{From} \ Reportsz \\
\text{CreateNon-managerial} & : \text{Select} \ x.SSN \ \text{FROM} \ Employeesx \ \text{Where} (x.\text{position}\notin \text{ManagerialPosition})
\end{align*}
\]

And then intersect them by:

\[
\text{Select} \ x \ \text{From} \ Supervisorsx, \ Non-managerialy \ \text{Where} (x = y)
\]

4.2 Negations and universal quantifiers

But we are still missing some expressive power/abilities.

Define a Middle Manager to be a manager where everyone who reports to
that employee is a manager as well!

The logical structure here involves a universal \( \forall \) quantifier, where our previ-
ous queries were only existential. By using the “Where \( x \) not in Table” construc-
tion on a previously computed Table, we can negate the property creating the
table. Since the negation of an existential statement is a universal statement,
we can use this to write queries for universal quantifiers.

So like before, we can create a list of Supervisors

\[
\begin{align*}
\text{CreateSupervisors} & : \text{Select} \ z.MSSN \ \text{From} \ Reportsz \\
\text{CreateNon-supervisors} & : \text{Select} \ x.SSN \ \text{From} \ Employeesx \ \text{Where} (x.\text{SSN}\notin \text{Supervisors})
\end{align*}
\]

We can then negate this to get the list of non-supervisors:

\[
\begin{align*}
\text{CreateNon-supervisors} & : \text{Select} \ x.SSN \ \text{From} \ Employeesx \ \text{Where} (x.\text{SSN}\notin \text{Supervisors})
\end{align*}
\]

Then we can see who manages someone on the non-supervisor list:

\[
\begin{align*}
\text{CreateLowlevelmanager} & : \text{Select} \ z.MSSN \ \text{From} \ Reportsz \ \text{Where} (z.\text{ESSN}\in \text{Non-supervisors})
\end{align*}
\]

4
And then prune out this list from the Supervisors to get rid of those that manage non-supervisors:

\[ \text{Select } x \ \text{From } \text{Supervisors} \ \text{Where} x \notin \text{Lowlevelmanager} \]

5 General connection

Claim: A new relation \( R(x_1, \ldots, x_k) \) is creatable by a sequence of basic SQL queries IFF it is expressible in FO logic, by finite set of quantifiers and boolean expressions.

The proof is by induction on the number of tables created in one direction, and on the number of symbols in the formula in the other. To go from queries to FO logic, we express all intermediate tables as formulas, and then each new construction without “not in” gives one more existential quantifier. Each “not in” changes all the existential quantifiers to universal quantifiers and vice versa, using the logical rule \( \neg \forall x P(x) \iff \exists x \neg P(x) \). In the other direction, we do the same process in reverse, creating queries that simulate all of the subformulas of a FO statement, and then adding one more existential quantifier via a new select, or a universal quantifier by taking the complement table, doing a select, and taking the complement again.

To be fair, I should point out that, while this connection was the essence of Cobb’s original plan for relational databases, SQL has evolved by adding new constructions such as “average” or “sum” that go beyond first order predicate calculus. So this connection is no longer exactly true. However, it is still useful to be able to dissect the logical structure of queries when presenting them in SQL.

6 Query optimization

Predicate logic becomes vital in query optimization, in going beyond what you want to compute to also consider how to compute it. When we have a list of intermediate tables, the sizes of those tables could dwarf the size of the final output. If the computer can compute the answer you want without actually constructing those tables, all the better. Humans are not very good at finding efficient ways of expressing queries. Query optimizers are programs that look for logically equivalent formulations with smaller intermediate tables. To design a query optimizer, computer scientists need to be first able to generate many logically equivalent ways of writing the query, and second, approximate the size of the intermediate tables for each one. The first requires some fairly deep understanding of logic, enough to automate logical reasoning. The second often is data-dependent (How many middle managers will a large company have, compared to upper managers?). So query optimization is not a completely solved problem; many competing approaches exist. However, it is also a vital part of any database system, because different plans to answer queries can have
widely different time requirements. One plan might take seconds, an equivalent one hours or even days.

This is because intermediate queries could create some HUGE tables depending on the type of table created (pairs of items would create $r^2$ rows compared to the initial $r$)! While there is no universal panacea for this problem, a good query optimizer can often save us, by automatically replacing such a bad plan with a logically equivalent one that will minimize the intermediate steps.

7 Extra credit project

Describe a schema (format) for an SQL database for scheduling at a job fair. Upon entering the fair, students fill out a form giving their names, student ids, minimum expected salary, a list of qualifications (e.g., “Skilled in Java programming”), and categories of job they are interested in, e.g., “web development”. Companies have a set of positions they want to fill, each with a salary range, category (as above), and set of requirements (types match types of student qualifications). They also have a number of representatives, each with a name and employee id. The university provides interview rooms and time slots for interviews. Your database should keep track of schedules for both applicants and corporate representatives. You should be able to answer queries such as: What times does a given student or representative already have interviews? If a student and representative request an interview, give a list of times and places where the interview could occur (without conflicting with either of their schedules or other scheduled interviews). List the positions within a given company that a student is qualified for that meet their salary expectations. List the students that have the qualifications and might be interested in a given position. Show how to write these as queries for your schema.