Relational Query Languages

Artyom Desyatnikov

November 3, 2016

Contents

1 History

In early 1960-x, with the advance of the first computers arose the need to maintain the large dynamic collection of data. Conveniently, this demands were met with the increasing availability of direct-access data storage devices. From then on the database history begins. The first databases aimed to provide the user with a means of programmatically store and traverse data organized in some structure. Such was, for example, Information Management System developed in 1966 by IBM. It operated an XML-like tree of nodes, each containing multiple homogeneous entries. The API allowed to "manually" navigate from a node to either its parent or child and browse their contents on the way. That was why such databases were called navigational. It was left for the user to organize their model in a manner to be efficiently traversable. While this alone was said to be difficult in some cases, the evidence says it turned even more complicated to maintain should the requirements change to form new unexpected connections in the model.

The concept of relational database was introduced in 1970 by Edgar Codd in attempt to describe an implementation-agnostic database model that would make for a query language both declarative (in contrast to IMS' procedural approach) and adequately powerful. In fact, there were 2 universal query languages introduced. In the paper "A Relational Model of Data for Large Shared Data Banks" (1970) Codd presented **relational algebra**, a simple formal language to define steps of a query execution. Later in 1970 Codd published another paper featuring **relational calculus**, a higher-level query language that essentially was the first order logic applied to a database entities. Last but not the least, Codd proved that whatever can be expressed in relational calculus, could as well be expressed in relational algebra and presented an algorithm of such translation. Thus, relational calculus was intended as a user input language while the translation algorithm was to convert the query into an algebraic form suitable for execution.

Codd pointedly criticized the existing DBMSs for the fact their queries and responses lacked abstraction over the storage implementation details. What he, in turns, suggested was to interpose an implementation-unaware query language, namely relational algebra, between an application requirements and data storage specificities. Given the flexibility, and therefore production value, Codd's approach proposed it is a small wonder that his works were quickly picked up by the INGRES project and soon by IBM themselves. Thus began the dominance of relational databases, and it wasn't until mid

2000-s that it began to fade.

2 Relational Model

But the central notion of Codd's works was not the mere query language itself, but the so-called relational database model. It is this model that Codd suggested as a replacement for the network and hierarchal databases of 1960-x.

In order to approach the notion, first, let us revise the mathematical concept of relation. The k-ary relation over $(D_1, D_2, ..., D_k)$ is defined as a subset of $D_1 \times D_2 \times ... \times D_k$. This means that of all possible combinations of values from the corresponding domains we select only those satisfying some condition, and call the set of them a relation.

The elements of the Cartesian product, taking the form of $(d_1, ..., d_k)$, are called **tuples**. The elements of tuples are called **attributes**. Then, a k-ary relation can as well be said to be a set of tuples from $D_1 \times D_2 \times ... \times D_k$.

The relational database model is simply a set of named relations with a schema associated to each. The schema is a list of attribute schemata, each specifying a name and a domain. The relation with the schema $[(name : a_1, domain : D_1), ...(name : a_k, domain : D_k)]$ is naturally required to be defined on $D_1 \times D_2 \times ... \times D_k$.

For example, schema:

Author (AID integer, name string, age integer),
Paper (PID string, title string, year integer),
Write (AID integer, PID string)

and instance:

$\{< 142, “Knuth”, 73 >, < 123, “Ullman”, 67 >, ...\}$,
$\{< “181140pods”, “Querycontainment”, 1998 >, ...\}$,
$\{< 123, “181140pods” >, < 142, “193214algo” >, ...\}$.

The notion of a relation comes very similar to what we are used to think of as a table. A tricky difference one should be aware of, though, is that being technically a set, a relation cannot contain a pair of duplicate tuples.

3 Relational algebra

Now let us turn our attention towards relational algebra, a language designed to query data from relational model. The language essentially consists of several operations on relations that yield in turns other relations. Thus, a query in relational algebra is a possibly nested application of these predefined operations to either relations of a database or constant tuple sets (like $< 1, ``a" >, < 2, ``b" >$) as the atomic values. Naturally, a query result is also bound to be a relation.

3.1 Basic operations

The set of operations is redundant in a way, there being a subset of them sufficient to express the rest. The list of them is as follows:

- **Selection** σ

  Selects the tuples from the relation matching some criteria. The criteria is restricted to either equality or inequality between two attributes or between an attribute and a constant. For example, $\sigma_{name=``Knuth"}(Author)$ will yield a relation containing a single tuple $< 142, ``Knuth", 73 >$.

- **Projection** π

  Selects only the specified subset of attributes tuples from the relation matching some criteria. The criteria is restricted to either equality or inequality between two attributes or between an attribute and a constant. $\pi_{AID}(Write)$, for instance, will return IDs of all authors who have written a paper. Note, that given the nature of set, there will be only a single occurrence of each AID, even though the author have written more than one paper.

- **Rename** ρ

  Renames the specified set of attributes. For example, $\rho_{AID \to AuthorID}$. It is commonly put to use in order to make attribute name unique for the next operation.

- **Cartesian product** $\times$

  Yields all possible combination of tuples from two relations merged together. Note that it is required the attribute names be unique within a relation. To avoid ambiguity performing Cartesian product on two relations sharing a common attribute name is prohibited. As an illustration, to have all possible authors older than Knuth, one might first select the rows associated with Knuth:

$$K = \sigma_{name=\text{“}Knuth\text{”}}(Author)$$

and then inspect the Cartesian product of the Knuth's age and all other authors

$$O = \rho_{age \to knuthsAge}(\pi_{age}(K)) \times Author$$

searching for age value more than the Knuth's:

$$\pi_{AID,name,age}(\sigma_{age>knuthsAge}(O)).$$

- **Intersection and Union**

  These two are straightforward analogs of their set-theoretic versions. Again, their application is restricted to pairs of relations having the same attribute names, which in turns can be achieved by renaming them and dropping off the extra ones with projection.

The listed operations along with all their possible compositions is what constitutes the relational algebra. More formally, an algebraic structure is defined as a carrier set and a list of operations on it. Thus, the relational algebra of database D is an algebraic structure consisting of the six aforementioned operations acting and a closure of all relations from D under these operations serving as a carrier set.

3.2 Additional operations

There are several additional handy operations in relational algebra which are though perfectly expressible through the six basic ones. They are given such credit for the fact at the database engine level they can be implemented more efficiently than their translations and hence are to be preferred during optimizations.

Take **natural join**, for example. It yields the set of all combinations of tuples of the two given relations with their common attribute equal.

Employee

Name	EmpId	DeptName
Harry	3415	Finance
Sally	2241	Sales
George	3401	Finance
Harriet	2202	Sales

Dept

DeptName	Manager
Finance	George
Sales	Harriet
Production	Charles

Employee $\bowtie$ Dept

Name	EmpId	DeptName	Manager
Harry	3415	Finance	George
Sally	2241	Sales	Harriet
George	3401	Finance	George
Harriet	2202	Sales	Harriet

The natural join has its straightforward implementation as

$$\sigma_{A1=A2}(\rho_{A\to A1}R_1 \times \rho_{A\to A2}R_2)$$

However it would leave us having to traverse all the possible tuple combinations, which can and in most cases must be avoided in a real DBMS. Instead of blindly searching for the matching tuples one can index R_1 by the attribute A to achieve $o(|R_1|\ max_{r_1\in R_1}|\sigma_{A=r_1.A}R_2)|$ join complexity, essentially $o(|R_1|)$ in most cases, instead of $o(|R_1|\ |R_2|)$.

Probably the most challenging of the lot is the **division** operator. The result of $R_1 \div R_2$ with the attribute sets A_1 and A_2 respectively is defined to have attributes $A_1 \setminus A_2$. It contains the values of the attributes unique to R_1 which have all the combinations with entries from R_2 present in R_1. It can be thought of as a rought equivalent of a universal quantifier. For the sake of illustration, here is how the division can be used to determine which students have completed all the given tasks:

Completed

Student	Task
Fred	Database1
Fred	Database2
Fred	Compiler1
Eugene	Database1
Eugene	Compiler1
Sarah	Database1
Sarah	Database2

DBProject

Task
Database1
Database2

Completed ÷ DBProject

Student
Fred
Sarah

Again, the division can be implemented in terms of the basic relational algebraic operations. In this particular case it would be:

$$\pi_{Student}(Completed) - \pi_{Student}(Completed - (DBProject \times \pi_{Student}(Completed)))$$

.

4 Relational calculus

Having discussed the basics of relational algebra, we should note that creation of a low-level query execution planning language as it is was not the final goal

Codd had in mind. In his second paper he presented the readers with yet another language for querying data from relational databases - the relational calculus. The two languages had an almost completely unrelated syntax while bearing similar semantics.

By semantics of a query we mean a function from all possible databases to relations resulting from the query execution on a given database. The semantics of the whole query language therefore is the union of all its words' semantics. If two languages' semantics are equal sets such languages are said to be equivalent in expressive power and if semantics of one is the superset of another's, it is said to be more expressive. The proof of expressive equivalence of languages is usually given in form of a bijection between them, or just an injection when expressive superiority is sought.

The aim of Codd's second work was to provide a human-readable language reducible to relational algebra. Since he also did not cherish the idea of losing any of algebra's expressive power by doing so, it was actually to be expressively equivalent to the relational algebra. And indeed relational calculus, was proven to be such.

Now in fact there are two languages that the term "relational calculus" might stand for. Codd gave his proof regarding the so-called **tuple calculus**, while the second version known as **domain calculus** was introduced 7 years later by two French computer scientists Michel Lacroix and Alain Pirotte. The two languages undoubtedly have equivalent expressive power but the formal proof of the statement is beyond our consideration.

The **tuple calculus** consists of queries in the form of

$$\{t_1.attr_1, ...t_k.attr_{i_k} | \phi(t_1, ..., t_k)\},$$

where $t_1, ..., t_n$ represent tuples and ϕ is a second-order logic formula with relations of a database as unary predicates and attribute equalities and inequalities $t_i.attr_1 \; \rho \; t_j.attr_2$ as binary predicates.

Imagine the database from the division example being subjected to:

$$\begin{aligned}\{compEntry.Student \mid Completed(compEntry) \wedge \\ \forall proj DBProj(proj) \rightarrow (\exists comp \wedge \\ Completed(comp) \wedge \\ comp.Task = proj.Task \wedge \\ comp.Student = compEntry.Student)\}.\end{aligned}$$

Not exactly as clear as Codd probably expected, it is still obviously equivalent to $Completed \div DBProject$.

The same semantics can be expressed in **domain calculus** as well, its query also being a first-order formula, but with attributes as variables and k-ary relations instead of unary ones.

$$\{student | (\exists studentTaskCompleted(student, studentTask)) \wedge \\ (\forall taskDBProj(task) \rightarrow Completed(student, task))\}.$$

Note that some queries however are obviously inapplicable to the real-world databases, such as $\{p.PID | \neg Paper(p)\}$. These queries basically require the database to give all PIDs in the world, and the only 'world' the database is aware of comes in form of the domains. An actually correct response to the query stated would be a column, containing the union of all attribute domains of the database except for $Paper$'s PID. Cases like this would be extremely difficult to implement and actually useless. If nothing else, we would not want our database to output all $256^{2^{30}}$ strings it can represent. Thus, to possess any interest to us, the queries are to yield results independent of attribute domains. At least, so long as the domains hold all the values from relation - otherwise the database would be invalid.

To determine whether a given query is domain-dependent is an undecidable problem, i. e., there is no algorithm that can tell whether a given relational calculus query is domain dependent [2]. This is however unnecessary in pursuit our goal. Instead some simple-to-check restriction could be imposed onto the tuple calculus queries in order to guarantee their domain independence. Since one is interested not to lose any of its expressive power in the process, one would have to ensure, that for every domain-independent query there is an equivalent one conforming such restriction. Thus is the range-restricted queries, where each quantifier is followed by some relation restricting its variable: $\forall x \in P$ (shorthand for $\forall x(P(x) \rightarrow ...)$) or $\exists x \in P$ (shorthand for $\exists x(P(x) \wedge ...)$). This way the only source of tuples to draw from would be strictly defined and domain independent. One way to convert a query known to be domain-independent into a range-restricted form is to specify the so-called active domain for each of its variables. The active domain of a query is defined as a set of tuples from all possible relations together with constants referenced in the query. If done so, the quantifier expressions will take the form $\forall(\exists)x \in \alpha(x)$, where $\alpha(x)$ is a massive disjunction of $R(x)$ for all relations R in the database as well as one constant relation holding all possible constants from the query for attributes of x referenced in the query. Then the disjunctions can be trivially separated so that the query fits the definition of range-restricted one.

5 Codd's Theorem

The main point of Codd's second paper was to prove range-restricted query language to be equivalent to relational algebra. Besides, Codd's proof was constructive, that is to say it suggested an algorithm converting a range-restricted query into that of relational algebra. The backwards conversion does not pose much of a problem and will be left as an exercise to the reader.

As for the forward conversion, the formal algorithm starts with first rewriting a query in either prenex normal form, i. e. with all its quantifiers placed in front. The relation in question is initially set to simply be the Cartesian product of all relations of the query. After that it is filtered according to the predicate-free part. Then the process starts that gradually shapes the relation into the desired relational algebraic expression, by gradually applying one of the predefined formulas to the quantifiers, right to left.

The list of the said formulas is usually regarded as tedious and therefore omitted. A thorough demonstration of the conversion process, however, is given in [3].

6 Limitations

As powerful as the two languages might seem, there are certain limitations to their expressiveness. Consider such metric as the Erdős number, the numeric characteristic of a person that in a way represents their "collaborative distance" to Paul Erdős, one of the most productive authors of mathematical papers. The Erdős number of 0 is assigned to Paul Erdős himself. Those, who have at least once collaborated with him at writing a paper are said to have the Erdős number of 1. Their collaborators receive the number 2 unless they have already got 1 or 0. And so on.

Let us write a query for the Author-Paper-Write database from the previous examples, that selects IDs of authors with the Erdős number less or equal than 1. First, we should select the papers written by Erdős:

$$A_0 = \pi_{AID}(\sigma_{name=\text{“Erdős”}} Author)$$

$$P_1 = \pi_{PID}(A_0 \bowtie Write).$$

The desired set of author IDs will be:

$$A_1 = \pi_{AID}(P_1 \bowtie Write).$$

Mechanically incrementing the indices we will be able to define a query for an arbitrary large Erdős number. Note, that while doing so the expression will become increasingly nested but nevertheless perfectly executable. However, in case one tries to construct a query selecting the authors with the infinite Erdős number, i.e. whose “collaboration path” will never lead to Paul Erdős, one is bound to fail [1].

There is a whole classes of problems unsolvable by means of relational algebra and calculus, some of them dealing with recursive structures or graph algorithms likewise to the Erdős number example, others related to grouping and aggregation of tuples and some more.

7 A note on SQL

The SQL was intended as an end-user language featuring the relational model aspects, making for easy relational calculus and algebra-like queries. Due to the lack of Codd's influence SQL turned out not to follow the concept of either of the two (even three) query languages discussed. Instead they have chosen to go for the mixture of all the approaches adding some extra features on top of that, which lead to the language becoming extremely redundant. Still, it was more than enough to implement all relational operations in it which made SQL at least as expressively powerful as relational algebra.

Furthermore, such popular needs as that for aggregation or recursive queries have been tended to in SQL, thus making the language superior to the relational algebra in terms of expressive power.

References

[1] Reinhard Pichler, *Datenbanktheorie*, Sommersemester 2016 http://www.dbai.tuwien.ac.at/staff/pichler/dbt/

[2] Phokion G. Kolaitis University of California, Santa Cruz & IBM Research-Almaden, *Relational Databases, Logic, and Complexity*, 2009 https://users.soe.ucsc.edu/∼kolaitis/talks/gii09-final.pdf

[3] C.J. Date, *An Introduction to Database Systems*