WIKIPEDIA

Relational algebra

In database theory, **relational algebra** is a theory that uses algebraic structures with a well-founded semantics for modeling data, and defining queries on it. The theory has been introduced by Edgar F. Codd.

The main application of relational algebra is to provide a theoretical foundation for relational databases, particularly query languages for such databases, chief among which is SQL. Relational databases store tabular data represented as relations. Queries over relational databases often likewise return tabular data represented as relations. The main premise of the relational algebra is to define operators that transform one or more input relations to an output relation. Given that these operators accept relations as input and produce relations as output, they can be combined and used to express potentially complex queries that transform potentially many input relations (whose data are stored in the database) into a single output relation (the query results). Unary operators accept as input a single relation; examples include operators to filter certain attributes (columns) or tuples (rows) from an input relation. Binary operators accept as input two relations; such operators combine the two input relations into a single output relation by, for example, taking all tuples found in either relation, removing tuples from the first relation found in the second relation, extending the tuples of the first relation with tuples in the second relation matching certain conditions, and so forth. Other more advanced operators can also be included, where the inclusion or exclusion of certain operators gives rise to a family of algebras.

Contents

Introduction

Relational algebra received little attention outside of pure mathematics until the publication of E.F. Codd's relational model of data in 1970. Codd proposed such an algebra as a basis for database query languages. (See section Implementations.)

Five primitive operators of Codd's algebra are the *selection*, the *projection*, the *Cartesian product* (also called the *cross product* or *cross join*), the *set union*, and the *set difference*.

Set operators

The relational algebra uses set union, set difference, and Cartesian product from set theory, but adds additional constraints to these operators.

For set union and set difference, the two relations involved must be *union-compatible*—that is, the two relations must have the same set of attributes. Because set intersection is defined in terms of set union and set difference, the two relations involved in set intersection must also be union-compatible.

For the Cartesian product to be defined, the two relations involved must have disjoint headers—that is, they must not have a common attribute name.

In addition, the Cartesian product is defined differently from the one in set theory in the sense that tuples are considered to be "shallow" for the purposes of the operation. That is, the Cartesian product of a set of *n*-tuples with a set of *m*-tuples yields a set of "flattened" $(n + m)$-tuples (whereas basic set theory would have prescribed a set of 2-tuples, each containing an *n*-tuple and an *m*-tuple). More formally, *R* × *S* is defined as follows:

$$R \times S := \{(r_1, r_2, \dots, r_n, s_1, s_2, \dots, s_m) | (r_1, r_2, \dots, r_n) \in R, (s_1, s_2, \dots, s_m) \in S\}$$

The cardinality of the Cartesian product is the product of the cardinalities of its factors, that is, |*R* × *S*| = |*R*| × |*S*|.

Projection (Π)

A **projection** is a unary operation written as $\Pi_{a_1,\ldots,a_n}(R)$ where $a_1, \ldots, a_n$ is a set of attribute names. The result of such projection is defined as the set that is obtained when all tuples in *R* are restricted to the set $\{a_1, \ldots, a_n\}$.

Note: when implemented in SQL standard the "default projection" returns a multiset instead of a set, and the Π projection to eliminate duplicate data is obtained by the addition of the `DISTINCT` keyword.

Selection (*σ*)

A **generalized selection** is a unary operation written as $\sigma_\varphi(R)$ where *φ* is a propositional formula that consists of atoms as allowed in the normal selection and the logical operators ∧ (and), ∨ (or) and ¬ (negation). This selection selects all those tuples in *R* for which *φ* holds.

To obtain a listing of all friends or business associates in an address book, the selection might be written as $\sigma_{\text{isFriend = true} \lor \text{isBusinessContact = true}}(\text{addressBook})$. The result would be a relation containing every attribute of every unique record where isFriend is true or where isBusinessContact is true.

Rename (*ρ*)

A **rename** is a unary operation written as $\rho_{a/b}(R)$ where the result is identical to *R* except that the *b* attribute in all tuples is renamed to an *a* attribute. This is simply used to rename the attribute of a relation or the relation itself.

To rename the 'isFriend' attribute to 'isBusinessContact' in a relation, $\rho_{\text{isBusinessContact / isFriend}}(\text{addressBook})$ might be used.

Joins and join-like operators

Natural join (⋈)

Natural join (⋈) is a binary operator that is written as (*R* ⋈ *S*) where *R* and *S* are relations.[1] The result of the natural join is the set of all combinations of tuples in *R* and *S* that are equal on their common attribute names. For an example consider the tables *Employee* and *Dept* and their natural join:

Employee

Name	EmpId	DeptName
Harry	3415	Finance
Sally	2241	Sales
George	3401	Finance
Harriet	2202	Sales
Mary	1257	Human Resources

Dept

DeptName	Manager
Finance	George
Sales	Harriet
Production	Charles

Employee ⋈ Dept

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

Note that neither the employee named Mary nor the Production department appear in the result.

This can also be used to define composition of relations. For example, the composition of *Employee* and *Dept* is their join as shown above, projected on all but the common attribute *DeptName*. In category theory, the join is precisely the fiber product.

The natural join is arguably one of the most important operators since it is the relational counterpart of logical AND operator. Note that if the same variable appears in each of two predicates that are connected by AND, then that variable stands for the same thing and both appearances must always be substituted by the same value (this is a consequence of the idempotence of the logical AND). In particular, natural join allows the combination of relations that are associated by a foreign key. For example, in the above example a foreign key probably holds from *Employee*.*DeptName* to *Dept*.*DeptName* and then the natural join of *Employee* and *Dept* combines all employees with their departments. This works because the foreign key holds between attributes with the same name. If this is not the case such as in the foreign key from *Dept*.*Manager* to *Employee*.*Name* then we have to rename these columns before we take the natural join. Such a join is sometimes also referred to as an **equijoin** (see *θ*-join).

More formally the semantics of the natural join are defined as follows:

$$R \bowtie S = \{r \cup s \mid r \in R \ \wedge \ s \in S \ \wedge \ Fun(r \cup s)\} \qquad (1)$$

where *Fun(t)* is a predicate that is true for a relation *t* (in the mathematical sense) iff *t* is a function. It is usually required that *R* and *S* must have at least one common attribute, but if this constraint is omitted, and *R* and *S* have no common attributes, then the natural join becomes exactly the Cartesian product.

The natural join can be simulated with Codd's primitives as follows. Assume that $c_1,...,c_m$ are the attribute names common to *R* and *S*, $r_1,...,r_n$ are the attribute names unique to *R* and $s_1,...,s_k$ are the attribute names unique to *S*. Furthermore, assume that the attribute names $x_1,...,x_m$ are neither in *R* nor in *S*. In a first step we can now rename the common attribute names in *S*:

$$T = \rho_{x_1/c_1,\ldots,x_m/c_m}(S) = \rho_{x_1/c_1}(\rho_{x_2/c_2}(\ldots\rho_{x_m/c_m}(S)\ldots)) \qquad (2)$$

Then we take the Cartesian product and select the tuples that are to be joined:

$$P = \sigma_{c_1=x_1,\ldots,c_m=x_m}(R \times T) = \sigma_{c_1=x_1}(\sigma_{c_2=x_2}(\ldots\sigma_{c_m=x_m}(R \times T)\ldots)) \qquad (3$$

Finally we take a projection to get rid of the renamed attributes:

$$U = \Pi_{r_1,\ldots,r_n,c_1,\ldots,c_m,s_1,\ldots,s_k}(P) \qquad (4)$$

θ-join and equijoin

Consider tables *Car* and *Boat* which list models of cars and boats and their respective prices. Suppose a customer wants to buy a car and a boat, but she does not want to spend more money for the boat than for the car. The *θ*-join ($\bowtie_\theta$) on the predicate *CarPrice* ≥ *BoatPrice* produces the flattened pairs of rows which satisfy the predicate. When using a condition where the attributes are equal, for example Price, then the condition may be specified as *Price*=*Price* or alternatively (*Price*) itself.

Car

CarModel	CarPrice
CarA	20,000
CarB	30,000
CarC	50,000

Boat

BoatModel	BoatPrice
Boat1	10,000
Boat2	40,000
Boat3	60,000

$\underset{CarPrice \geq BoatPrice}{Car \bowtie Boat}$

CarModel	CarPrice	BoatModel	BoatPrice
CarA	20,000	Boat1	10,000
CarB	30,000	Boat1	10,000
CarC	50,000	Boat1	10,000
CarC	50,000	Boat2	40,000

If we want to combine tuples from two relations where the combination condition is not simply the equality of shared attributes then it is convenient to have a more general form of join operator, which is the *θ*-join (or theta-join). The *θ*-join is a binary operator that is written as $\underset{a\,\theta\,b}{R \bowtie S}$ or $\underset{a\,\theta\,v}{R \bowtie S}$ where *a* and *b* are attribute names, *θ* is a binary relational operator in the set $\{<, \leq, =, \neq, >, \geq\}$, *υ* is a value constant, and *R* and *S* are relations. The result of this operation consists of all combinations of tuples in *R* and *S* that satisfy *θ*. The result of the *θ*-join is defined only if the headers of *S* and *R* are disjoint, that is, do not contain a common attribute.

The simulation of this operation in the fundamental operations is therefore as follows:

$$R \bowtie_\theta S = \sigma_\theta(R \times S)$$

In case the operator *θ* is the equality operator (=) then this join is also called an **equijoin**.

Note, however, that a computer language that supports the natural join and selection operators does not need *θ*-join as well, as this can be achieved by selection from the result of a natural join (which degenerates to Cartesian product when there are no shared attributes).

In SQL implementations, joining on a predicate is usually called an *inner join*, and the *on* keyword allows one to specify the predicate used to filter the rows. It is important to note: forming the flattened Cartesian product then filtering the rows is conceptually correct, but an implementation would use more sophisticated data structures to speed up the join query.

Semijoin (⋉)(⋊)

The left semijoin is a joining similar to the natural join and written as *R* ⋉ *S* where *R* and *S* are relations.[2] The result is the set of all tuples in *R* for which there is a tuple in *S* that is equal on their common attribute names. The difference from a natural join is that other columns of *S* do not appear. For example, consider the tables *Employee* and *Dept* and their semijoin:

Employee

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

Dept

DeptName	Manager
Sales	Sally
Production	Harriet

Employee ⋉ *Dept*

Name	EmpId	DeptName
Sally	2241	Sales
Harriet	2202	Production

More formally the semantics of the semijoin can be defined as follows:

$$R \ltimes S = \{ t : t \in R \wedge \exists s \in S(Fun\,(t \cup s)) \}$$

where $Fun(r)$ is as in the definition of natural join.

The semijoin can be simulated using the natural join as follows. If $a_1, ..., a_n$ are the attribute names of R, then

$$R \ltimes S = \pi_{a_1,..,a_n}(R \bowtie S).$$

Since we can simulate the natural join with the basic operators it follows that this also holds for the semijoin.

In Codd's 1970 paper, semijoin is called restriction.[3]

Antijoin (▷)

The antijoin, written as $R \triangleright S$ where R and S are relations, is similar to the semijoin, but the result of an antijoin is only those tuples in R for which there is *no* tuple in S that is equal on their common attribute names.[4]

For an example consider the tables *Employee* and *Dept* and their antijoin:

Employee

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

Dept

DeptName	Manager
Sales	Sally
Production	Harriet

Employee ▷ Dept

Name	EmpId	DeptName
Harry	3415	Finance
George	3401	Finance

The antijoin is formally defined as follows:

$$R \triangleright S = \{ t : t \in R \land \neg\exists s \in S(Fun\ (t \cup s)) \}$$

or

$$R \triangleright S = \{ t : t \in R, \text{there is no tuple } s \text{ of } S \text{ that satisfies } Fun\ (t \cup s) \}$$

where $Fun\ (t \cup s)$ is as in the definition of natural join.

The antijoin can also be defined as the complement of the semijoin, as follows:

$$R \triangleright S = R - R \ltimes S \quad \textbf{(5)}$$

Given this, the antijoin is sometimes called the anti-semijoin, and the antijoin operator is sometimes written as semijoin symbol with a bar above it, instead of ▷.

Division (÷)

The division is a binary operation that is written as $R \div S$. Division is not implemented directly in SQL. The result consists of the restrictions of tuples in R to the attribute names unique to R, i.e., in the header of R but not in the header of S, for which it holds that all their combinations with tuples in S are present in R. For an example see the tables *Completed*, *DBProject* and their division:

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

If *DBProject* contains all the tasks of the Database project, then the result of the division above contains exactly the students who have completed both of the tasks in the Database project. More formally the semantics of the division is defined as follows:

$$R \div S = \{ t[a_1,...,a_n] : t \in R \land \forall s \in S\ ((t[a_1,...,a_n] \cup s) \in R) \} \quad \textbf{(6)}$$

where $\{a_1,...,a_n\}$ is the set of attribute names unique to R and $t[a_1,...,a_n]$ is the restriction of t to this set. It is usually required that the attribute names in the header of S are a subset of those of R because otherwise the result of the operation will always be empty.

The simulation of the division with the basic operations is as follows. We assume that $a_1,...,a_n$ are the attribute names unique to R and $b_1,...,b_m$ are the attribute names of S. In the first step we project R on its unique attribute names and construct all combinations with tuples in S:

$$T := \pi_{a_1,...,a_n}(R) \times S$$

In the prior example, T would represent a table such that every Student (because Student is the unique key / attribute of the Completed table) is combined with every given Task. So Eugene, for instance, would have two rows, Eugene → Database1 and Eugene → Database2 in T.

> EG: First, let's pretend that "Completed" has a third attribute called "grade". It's unwanted baggage here, so we must project it off always. In fact in this step we can drop 'Task' from R as well; the multiply puts it back on.
> $T := \pi_{\text{Student}}(R) \times S$ // This gives us every possible desired combination, including those that don't actually exist in R, and excluding others (eg Fred | compiler1, which is not a desired combination)

T

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

In the next step we subtract *R* from *T*

relation:

$U := T - R$

In *U* we have the possible combinations that "could have" been in *R*, but weren't.

> EG: Again with projections — *T* and *R* need to have identical attribute names/headers.
> $U := T - \pi_{\text{Student,Task}}(R)$ // This gives us a "what's missing" list.

T

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

R a.k.a. *Completed*

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

U
aka
T − R
aka
what's missing

Student	Task
Eugene	Database2

So if we now take the projection on the attribute names unique to *R*

then we have the restrictions of the tuples in *R* for which not all combinations with tuples in *S* were present in *R*:

$V := \pi_{a_1,\ldots,a_n}(U)$

> EG: Project *U* down to just the attribute(s) in question (Student)
> $V := \pi_{\text{Student}}(U)$

V

Student
Eugene

So what remains to be done is take the projection of *R* on its unique attribute names and subtract those in *V*:

$W := \pi_{a_1,\ldots,a_n}(R) - V$

> EG: $W := \pi_{\text{Student}}(R) - V$.

$\pi_{\text{Student}}(R)$

Student
Fred
Eugene
Sarah

V

Student
Eugene

W
aka
$(\pi_{\text{Student}}(R) - V)$
aka
desired result

Student

Fred
Sarah

Common extensions

In practice the classical relational algebra described above is extended with various operations such as outer joins, aggregate functions and even transitive closure.[5]

Outer joins

Whereas the result of a join (or inner join) consists of tuples formed by combining matching tuples in the two operands, an outer join contains those tuples and additionally some tuples formed by extending an unmatched tuple in one of the operands by "fill" values for each of the attributes of the other operand. Outer joins are not considered part of the classical relational algebra discussed so far.[6]

The operators defined in this section assume the existence of a *null* value, *ω*, which we do not define, to be used for the fill values; in practice this corresponds to the NULL in SQL. In order to make subsequent selection operations on the resulting table meaningful, a semantic meaning needs to be assigned to nulls; in Codd's approach the propositional logic used by the selection is extended to a three-valued logic, although we elide those details in this article.

Three outer join operators are defined: left outer join, right outer join, and full outer join. (The word "outer" is sometimes omitted.)

Left outer join (⟕)

The left outer join is written as *R* ⟕ *S* where *R* and *S* are relations.[7] The result of the left outer join is the set of all combinations of tuples in *R* and *S* that are equal on their common attribute names, in addition (loosely speaking) to tuples in *R* that have no matching tuples in *S*.

For an example consider the tables *Employee* and *Dept* and their left outer join:

Employee

Name	EmpId	DeptName
Harry	3415	Finance
Sally	2241	Sales
George	3401	Finance
Harriet	2202	Sales
Tim	1123	Executive

Dept

DeptName	Manager
Sales	Harriet
Production	Charles

Employee ⟕ *Dept*

Name	EmpId	DeptName	Manager
Harry	3415	Finance	ω
Sally	2241	Sales	Harriet
George	3401	Finance	ω
Harriet	2202	Sales	Harriet
Tim	1123	Executive	ω

In the resulting relation, tuples in *S* which have no common values in common attribute names with tuples in *R* take a *null* value, *ω*.

Since there are no tuples in *Dept* with a *DeptName* of *Finance* or *Executive*, *ω*s occur in the resulting relation where tuples in *Employee* have a *DeptName* of *Finance* or *Executive*.

Let r_1, r_2, ..., r_n be the attributes of the relation *R* and let {(*ω*, ..., *ω*)} be the singleton relation on the attributes that are *unique* to the relation *S* (those that are not attributes of *R*). Then the left outer join can be described in terms of the natural join (and hence using basic operators) as follows:

$$(R \bowtie S) \cup ((R - \pi_{r_1, r_2, \dots, r_n}(R \bowtie S)) \times \{(\omega, \dots \omega)\})$$

Right outer join (⟖)

The right outer join behaves almost identically to the left outer join, but the roles of the tables are switched.

The right outer join of relations *R* and *S* is written as *R* ⟖ *S*.[8] The result of the right outer join is the set of all combinations of tuples in *R* and *S* that are equal on their common attribute names, in addition to tuples in *S* that have no matching tuples in *R*.

For example, consider the tables *Employee* and *Dept* and their right outer join:

Employee

Name	EmpId	DeptName
Harry	3415	Finance
Sally	2241	Sales
George	3401	Finance
Harriet	2202	Sales
Tim	1123	Executive

Dept

DeptName	Manager
Sales	Harriet
Production	Charles

Employee ⟖ *Dept*

Name	EmpId	DeptName	Manager
Sally	2241	Sales	Harriet
Harriet	2202	Sales	Harriet
ω	ω	Production	Charles

In the resulting relation, tuples in *R* which have no common values in common attribute names with tuples in *S* take a *null* value, *ω*.

Since there are no tuples in *Employee* with a *DeptName* of *Production*, *ω*s occur in the Name and EmpId attributes of the resulting relation where tuples in *Dept* had *DeptName* of *Production*.

Let s_1, s_2, ..., s_n be the attributes of the relation S and let $\{(\omega, ..., \omega)\}$ be the singleton relation on the attributes that are *unique* to the relation R (those that are not attributes of S). Then, as with the left outer join, the right outer join can be simulated using the natural join as follows:

$$(R \bowtie S) \cup (\{(\omega, \dots, \omega)\} \times (S - \pi_{s_1, s_2, \dots, s_n}(R \bowtie S)))$$

Full outer join (⟗)

The **outer join** or **full outer join** in effect combines the results of the left and right outer joins.

The full outer join is written as R ⟗ S where R and S are relations.[9] The result of the full outer join is the set of all combinations of tuples in R and S that are equal on their common attribute names, in addition to tuples in S that have no matching tuples in R and tuples in R that have no matching tuples in S in their common attribute names.

For an example consider the tables *Employee* and *Dept* and their full outer join:

Employee

Name	EmpId	DeptName
Harry	3415	Finance
Sally	2241	Sales
George	3401	Finance
Harriet	2202	Sales
Tim	1123	Executive

Dept

DeptName	Manager
Sales	Harriet
Production	Charles

Employee ⟗ *Dept*

Name	EmpId	DeptName	Manager
Harry	3415	Finance	ω
Sally	2241	Sales	Harriet
George	3401	Finance	ω
Harriet	2202	Sales	Harriet
Tim	1123	Executive	ω
ω	ω	Production	Charles

In the resulting relation, tuples in R which have no common values in common attribute names with tuples in S take a *null* value, ω. Tuples in S which have no common values in common attribute names with tuples in R also take a *null* value, ω.

The full outer join can be simulated using the left and right outer joins (and hence the natural join and set union) as follows:

$$R ⟗ S = (R ⟕ S) \cup (R ⟖ S)$$

Operations for domain computations

There is nothing in relational algebra introduced so far that would allow computations on the data domains (other than evaluation of propositional expressions involving equality). For example, it is not possible using only the algebra introduced so far to write an expression that would multiply the numbers from two columns, e.g. a unit price with a quantity to obtain a total price. Practical query languages have such facilities, e.g. the SQL SELECT allows arithmetic operations to define new columns in the result `SELECT unit_price * quantity AS total_price FROM t`, and a similar facility is provided more explicitly by Tutorial D's `EXTEND` keyword.[10] In database theory, this is called **extended projection**.[11]:213

Aggregation

Furthermore, computing various functions on a column, like the summing up of its elements, is also not possible using the relational algebra introduced so far. There are five aggregate functions that are included with most relational database systems. These operations are Sum, Count, Average, Maximum and Minimum. In relational algebra the aggregation operation over a schema (A_1, A_2, ... A_n) is written as follows:

$$G_1, G_2, \dots, G_m \; g_{f_1(A_1'), f_2(A_2'), \dots, f_k(A_k')} \; (r)$$

where each A_j', $1 \le j \le k$, is one of the original attributes A_i, $1 \le i \le n$.

The attributes preceding the g are grouping attributes, which function like a "group by" clause in SQL. Then there are an arbitrary number of aggregation functions applied to individual attributes. The operation is applied to an arbitrary relation r. The grouping attributes are optional, and if they are not supplied, the aggregation functions are applied across the entire relation to which the operation is applied.

Let's assume that we have a table named `Account` with three columns, namely `Account_Number, Branch_Name` and `Balance`. We wish to find the maximum balance of each branch. This is accomplished by $_{\texttt{Branch\_Name}}G_{\text{Max}(\texttt{Balance})}(\texttt{Account})$. To find the highest balance of all accounts regardless of branch, we could simply write $G_{\text{Max}(\texttt{Balance})}(\texttt{Account})$.

Transitive closure

Although relational algebra seems powerful enough for most practical purposes, there are some simple and natural operators on relations that cannot be expressed by relational algebra. One of them is the transitive closure of a binary relation. Given a domain D, let binary relation R be a subset of $D \times D$. The transitive closure R^+ of R is the smallest subset of $D \times D$ that contains R and satisfies the following condition:

$$\forall x \forall y \forall z \left((x,y) \in R^+ \wedge (y,z) \in R^+ \Rightarrow (x,z) \in R^+\right)$$

There is no relational algebra expression $E(R)$ taking R as a variable argument that produces R^+. This can be proved using the fact that, given a relational expression E for which it is claimed that $E(R) = R^+$, where R is a variable, we can always find an instance $\boldsymbol{r}$ of R (and a corresponding domain **d**) such that $E(\boldsymbol{r}) \neq \boldsymbol{r}^+$.[12]

SQL however officially supports such fixpoint queries since 1999, and it had vendor-specific extensions in this direction well before that.

Use of algebraic properties for query optimization

Queries can be represented as a tree, where

- the internal nodes are operators,
- leaves are relations,
- subtrees are subexpressions.

Our primary goal is to transform expression trees into equivalent expression trees, where the average size of the relations yielded by subexpressions in the tree is smaller than it was before the optimization. Our secondary goal is to try to form common subexpressions within a single query, or if there is more than one query being evaluated at the same time, in all of those queries. The rationale behind the second goal is that it is enough to compute common subexpressions once, and the results can be used in all queries that contain that subexpression.

Here we present a set of rules that can be used in such transformations.

Selection

Rules about selection operators play the most important role in query optimization. Selection is an operator that very effectively decreases the number of rows in its operand, so if we manage to move the selections in an expression tree towards the leaves, the internal relations (yielded by subexpressions) will likely shrink.

Basic selection properties

Selection is idempotent (multiple applications of the same selection have no additional effect beyond the first one), and commutative (the order selections are applied in has no effect on the eventual result).

1. $\sigma_A(R) = \sigma_A \sigma_A(R)$
2. $\sigma_A \sigma_B(R) = \sigma_B \sigma_A(R)$

Breaking up selections with complex conditions

A selection whose condition is a conjunction of simpler conditions is equivalent to a sequence of selections with those same individual conditions, and selection whose condition is a disjunction is equivalent to a union of selections. These identities can be used to merge selections so that fewer selections need to be evaluated, or to split them so that the component selections may be moved or optimized separately.

1. $\sigma_{A \wedge B}(R) = \sigma_A(\sigma_B(R)) = \sigma_B(\sigma_A(R))$
2. $\sigma_{A \vee B}(R) = \sigma_A(R) \cup \sigma_B(R)$

Selection and cross product

Cross product is the costliest operator to evaluate. If the input relations have *N* and *M* rows, the result will contain NM rows. Therefore, it is very important to do our best to decrease the size of both operands before applying the cross product operator.

This can be effectively done if the cross product is followed by a selection operator, e.g. $\sigma_A(R \times P)$. Considering the definition of join, this is the most likely case. If the cross product is not followed by a selection operator, we can try to push down a selection from higher levels of the expression tree using the other selection rules.

In the above case we break up condition *A* into conditions *B*, *C* and *D* using the split rules about complex selection conditions, so that $A = B \wedge C \wedge D$ and *B* contains attributes only from *R*, *C* contains attributes only from *P*, and *D* contains the part of *A* that contains attributes from both *R* and *P*. Note, that *B*, *C* or *D* are possibly empty. Then the following holds:

$$\sigma_A(R \times P) = \sigma_{B \wedge C \wedge D}(R \times P) = \sigma_D(\sigma_B(R) \times \sigma_C(P))$$

Selection and set operators

Selection is distributive over the set difference, intersection, and union operators. The following three rules are used to push selection below set operations in the expression tree. For the set difference and the intersection operators, it is possible to apply the selection operator to just one of the operands following the transformation. This can be beneficial where one of the operands is small, and the overhead of evaluating the selection operator outweighs the benefits of using a smaller relation as an operand.

1. $\sigma_A(R \setminus P) = \sigma_A(R) \setminus \sigma_A(P) = \sigma_A(R) \setminus P$
2. $\sigma_A(R \cup P) = \sigma_A(R) \cup \sigma_A(P)$
3. $\sigma_A(R \cap P) = \sigma_A(R) \cap \sigma_A(P) = \sigma_A(R) \cap P = R \cap \sigma_A(P)$

Selection and projection

Selection commutes with projection if and only if the fields referenced in the selection condition are a subset of the fields in the projection. Performing selection before projection may be useful if the operand is a cross product or join. In other cases, if the selection condition is relatively expensive to compute, moving selection outside the projection may reduce the number of tuples which must be tested (since projection may produce fewer tuples due to the elimination of duplicates resulting from omitted fields).

$$\pi_{a_1,\ldots,a_n}(\sigma_A(R)) = \sigma_A(\pi_{a_1,\ldots,a_n}(R)) \text{ where fields in } A \subseteq \{a_1,\ldots,a_n\}$$

Projection

Basic projection properties

Projection is idempotent, so that a series of (valid) projections is equivalent to the outermost projection.

$$\pi_{a_1,\ldots,a_n}(\pi_{b_1,\ldots,b_m}(R)) = \pi_{a_1,\ldots,a_n}(R) \text{ where } \{a_1,\ldots,a_n\} \subseteq \{b_1,\ldots,b_m\}$$

Projection and set operators

Projection is distributive over set union.

$$\pi_{a_1,\ldots,a_n}(R \cup P) = \pi_{a_1,\ldots,a_n}(R) \cup \pi_{a_1,\ldots,a_n}(P).$$

Projection does not distribute over intersection and set difference. Counterexamples are given by:

$$\pi_A(\{\langle A=a, B=b\rangle\} \cap \{\langle A=a, B=b'\rangle\}) = \emptyset$$

$$\pi_A(\{\langle A=a, B=b\rangle\}) \cap \pi_A(\{\langle A=a, B=b'\rangle\}) = \{\langle A=a\rangle\}$$

and

$$\pi_A(\{\langle A=a, B=b\rangle\} \setminus \{\langle A=a, B=b'\rangle\}) = \{\langle A=a\rangle\}$$

$$\pi_A(\{\langle A=a, B=b\rangle\}) \setminus \pi_A(\{\langle A=a, B=b'\rangle\}) = \emptyset\,,$$

where *b* is assumed to be distinct from *b'*.

Rename

Basic rename properties

Successive renames of a variable can be collapsed into a single rename. Rename operations which have no variables in common can be arbitrarily reordered with respect to one another, which can be exploited to make successive renames adjacent so that they can be collapsed.

1. $\rho_{a/b}(\rho_{b/c}(R)) = \rho_{a/c}(R)$
2. $\rho_{a/b}(\rho_{c/d}(R)) = \rho_{c/d}(\rho_{a/b}(R))$

Rename and set operators

Rename is distributive over set difference, union, and intersection.

1. $\rho_{a/b}(R \setminus P) = \rho_{a/b}(R) \setminus \rho_{a/b}(P)$
2. $\rho_{a/b}(R \cup P) = \rho_{a/b}(R) \cup \rho_{a/b}(P)$
3. $\rho_{a/b}(R \cap P) = \rho_{a/b}(R) \cap \rho_{a/b}(P)$

Product and union

Cartesian product is distributive over union.

1. $(A \times B) \cup (A \times C) = A \times (B \cup C)$

Implementations

The first query language to be based on Codd's algebra was Alpha, developed by Dr. Codd himself. Subsequently, ISBL was created, and this pioneering work has been acclaimed by many authorities[13] as having shown the way to make Codd's idea into a useful language. Business System 12 was a short-lived industry-strength relational DBMS that followed the ISBL example.

In 1998 Chris Date and Hugh Darwen proposed a language called **Tutorial D** intended for use in teaching relational database theory, and its query language also draws on ISBL's ideas. Rel is an implementation of **Tutorial D**.

Even the query language of SQL is loosely based on a relational algebra, though the operands in SQL (tables) are not exactly relations and several useful theorems about the relational algebra do not hold in the SQL counterpart (arguably to the detriment of optimisers and/or users). The SQL table model is a bag (multiset), rather than a set. For example, the expression $(R \cup S) \setminus T = (R \setminus T) \cup (S \setminus T)$ is a theorem for relational algebra on sets, but not for relational algebra on bags; for a treatment of relational algebra on bags see chapter 5 of the "Complete" textbook by Garcia-Molina, Ullman and Widom.[11]

See also

- Cartesian product
- D (data language specification)
- D4 (programming language) (an implementation of D)
- Database
- Logic of relatives
- Object-role modeling
- Projection (mathematics)
- Projection (relational algebra)
- Projection (set theory)
- Relation
- Relation (database)
- Relation algebra

- Relation composition
- Relation construction
- Relational calculus
- Relational database
- Relational model
- Theory of relations
- Triadic relation
- Tutorial D
- Tuple relational calculus
- SQL
- Datalog
- Codd's theorem

References

1. In Unicode, the bowtie symbol is ⋈ (U+22C8).
2. In Unicode, the ltimes symbol is ⋉ (U+22C9). The rtimes symbol is ⋊ (U+22CA)
3. Codd, E.F. (June 1970). "A Relational Model of Data for Large Shared Data Banks". *Communications of the ACM*. **13** (6): 377–387. doi:10.1145/362384.362685 (https://doi.org/10.1145%2F362384.362685).
4. In Unicode, the Antijoin symbol is ▷ (U+25B7).
5. M. Tamer Özsu; Patrick Valduriez (2011). *Principles of Distributed Database Systems* (https://books.google.com/books?id=TOBaLQMuNV4C&pg=PA46) (3rd ed.). Springer. p. 46. ISBN 978-1-4419-8833-1.
6. Patrick O'Neil; Elizabeth O'Neil (2001). *Database: Principles, Programming, and Performance, Second Edition* (https://books.google.com/books?id=UXh4qTpmO8QC&pg=PA120). Morgan Kaufmann. p. 120. ISBN 978-1-55860-438-4.
7. In Unicode, the Left outer join symbol is ⟕ (U+27D5).
8. In Unicode, the Right outer join symbol is ⟖ (U+27D6).
9. In Unicode, the Full Outer join symbol is ⟗ (U+27D7).
10. C. J. Date (2011). *SQL and Relational Theory: How to Write Accurate SQL Code* (https://books.google.com/books?id=WuZGD5tBfMwC&pg=PA133). O'Reilly Media, Inc. pp. 133–135. ISBN 978-1-4493-1974-8.
11. Hector Garcia-Molina; Jeffrey D. Ullman; Jennifer Widom (2009). *Database systems: the complete book* (2nd ed.). Pearson Prentice Hall. ISBN 978-0-13-187325-4.
12. Aho, Alfred V.; Jeffrey D. Ullman (1979). "Universality of data retrieval languages". *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*: 110–119. doi:10.1145/567752.567763 (https://doi.org/10.1145%2F567752.567763).
13. C. J. Date. "Edgar F. Codd - A.M. Turing Award Laureate" (https://amturing.acm.org/award_winners/codd_1000892.cfm). *amturing.acm.org*. Retrieved 2020-12-27.

Further reading

Practically any academic textbook on databases has a detailed treatment of the classic relational algebra.

- Imieliński, T.; Lipski, W. (1984). "The relational model of data and cylindric algebras". *Journal of Computer and System Sciences*. **28**: 80–102. doi:10.1016/0022-0000(84)90077-1 (https://doi.org/10.1016%2F0022-0000%2884%2990077-1). (For relationship with cylindric algebras).

External links

- RAT. Software Relational Algebra Translator to SQL (http://www.slinfo.una.ac.cr/rat/rat.html)
- Lecture Videos: Relational Algebra Processing (http://www.databaselecture.com/processing.html) - An introduction to how database systems process relational algebra
- Lecture Notes: Relational Algebra (http://www.databasteknik.se/webbkursen/relalg-lecture/index.html) – A quick tutorial to adapt SQL queries into relational algebra
- Relational – A graphic implementation of the relational algebra (https://ltworf.github.io/relational/index.html)
- Query Optimization (http://www-db.stanford.edu/~widom/cs346/ioannidis.pdf) This paper is an introduction into the use of the relational algebra in optimizing queries, and includes numerous citations for more in-depth study.
- Relational Algebra System for Oracle and Microsoft SQL Server (http://www.cse.fau.edu/~marty#RADownload)
- Pireal – An experimental educational tool for working with Relational Algebra (https://centaurialpha.github.io/pireal/index.html)
- DES – An educational tool for working with Relational Algebra and other formal languages (http://des.sourceforge.net)
- RelaX - Relational Algebra Calculator (https://dbis-uibk.github.io/relax/) (open-source software available as an online service without registration)
- RA: A Relational Algebra Interpreter (https://users.cs.duke.edu/~junyang/ra2/)
- Translating SQL to Relational Algebra (http://mlwiki.org/index.php/Translating_SQL_to_Relational_Algebra)