Universidade Federal da Fronteira Sul - UFFS
Computer Science - Chapecó
Database I
Prof.: Denio Duarte

Fast Guide to Relational Algebra

In the examples of this guide, we use the following database schema:
`client(`*clientid*`,cname,caddress)` `employee(`*empid*`,empname,hdate,sal)`
`product(`*prdid*`,descr,price)`
`invoice(`*inumber*`,clientid(client),empid(employee),billdate,billaddress)`
`invoiceitem(`*inumber*`(invoice),`*prdid*`(product),qty,sellprice)`

Algebra operations

Projection (π) - is a primitive unary operator of the form $\pi_{a_1,\ldots,a_n}(R)$ where $a_1, \ldots, a_n$ are attributes from the relation R. This operator creates a new temporary relation with only the attributes $a_1, \ldots, a_n$ of R.
For example, $\pi_{cname,address}(client)$ creates a new relation (the name of this relations is unknown) that is a copy from the relation *client* with the attributes *cname* and *address*.
It is possible to create an attribute using a valid expression: $\pi_{empname,sal*1.1}(employee)$. This query returns all employee's name and salary raised to 10%.
Selection (σ) - is a primitive unary operator of the form $\sigma_{\varphi}(R)$ where φ is propositional formula that consists of atoms connected by logical operators ($\wedge$, $\vee$ and $\neg$). This operator returns all tuples of R that satify φ.
For example, $\sigma_{price>20}(product)$ creates a new relation (the name of this relations is unknown) that is a copy from the relation *product* without the tuples that do not satisfy the formula (condition) $price > 20$.
Cross product ($\times$) - is a primitive binary operator (set operator) of the form $R \times S$ that multiplies R by S, *i.e.*, each tuple from R is combined with all tuples from S. The resulting relation R' will have all attributes from R and S (*i.e.*, $attr(R) \cup attr(S)$), and the cardinality of R' is $card(R) \times card(S)$, that is, the number of tuples of R' is the product of the tuples of R and S.
For example, $product \times invoiceitem$ creates a relation with the following attributes: *prdid*, *descr*, *price*, *inumber*, *prdid*, *qty*, *sellprice* and its cardinality is $card(product) \times card(invoiceitem)$.

Join operations: is a binary operator derived from the operators $\times$ and σ.

Equi Join ($\bowtie_{\varphi}$) - The operations $R \bowtie_{\varphi} S$ is implemented by $\sigma_{\varphi}(R \times S)$ where φ is the join condition of the form $r_{a_1} = s_{a_1} \wedge \ldots \wedge r_{a_n} = s_{a_n}$ where r_{a_i} are attributes from R and s_{a_i} are attributes from S. The join condition can be a single condition.
For example, $product \bowtie_{product.prdid=invoiceitem.prdid} invoiceitem$ returns tuples from *product* and *invoiceitem* where a tuple t from *product* has a product code ($t[prdid]$) appearing in a tuple s from *invoiceitem*, that is, $product \bowtie_{product.prdid=invoiceitem.prdid} invoiceitem = \{t[attr(product), attr(invoiceitem)] \mid t \in product \wedge s \in invoiceitem \wedge t[prdid] = s[prdid]\}$

Natural Join ($\bowtie$) - $R \bowtie S$ is similar to equi join but the join condition is ommitted since the join condition is built using attributes sharing the same name in both relations, R and S. Thus, it is mandatory that both relation have a same subset of attributes otherwise the operation becomes a cross product.
For example, the equi join $product \bowtie_{product.prdid=invoiceitem.prdid} invoiceitem$ from the previous example can be transformed into $product \bowtie invoiceitem$ since both relations, *product* and

invoiceitem, share the same name attributes in the join condition. Be carefull, if the relations in the natural join have attributes with the same name but with different semantics, the operation will not work properly.

Theta Join ($\bowtie_\theta$) - $R \bowtie_\theta S$ is similar to equal join but theta condition can have $=$, $\neq$, $<$, $>$, $\leq$ and $\geq$. Notice that equi join is a specialization of theta join since the operation $R \bowtie_{a=b} S$ can be called a theta join. On the other hand, $R \bowtie_{a>b} S$ is a theta join but it is not a equi join.

Outer Joins - join operation always returns tuples that match in both relations through the join condition. However, sometimes it is necessary that tuples that do not match in both relations return when a join is performed. Given the join operation $R \bowtie S$, the user can expect three cases: (i) tuples from R that do not match in S must appear in the answer, (ii) tuples from S that do not match in R must appear in the answer and (iii) tuples from R and S that do not match in both relations must appear in the answer.

Left Join ($\bowtie$) - this operations is used for the first case above.
For example, $product \bowtie_{product.prdid=invoiceitem.prdid} invoiceitem$ returns all tuples from *products* and tuples from *invoiceitem* that the join condition matches. The resulting relation will have attributes from *product* and *invoiceitem*, in the case where tuples from *product* do not match to tuples in *invoiceitem*, all attributes from *invoiceitem* will have $NULL$ value.

Right Join ($\bowtie$) - this operations is used for the second case above. In our running example database, we do not have a situation where right join is useful. A silly example can be: $invoiceitem \bowtie_{product.prdid=invoiceitem.prdid} product$ where we just exchange the relations positions. The result is the same of the previous example (left join).

Full Join ($\bowtie$) - this operations is used for the third case above. That is, tuples from both relations that do not match the join condition. In our running example database, we do not have a situation where full joins are useful. However, we can figure out a situation like that: returns all clients and employees, if there exists invoices associated to them, returns them, too. $client \bowtie_{client.prdid=invoicetem.prdid} invoiceitem \bowtie_{invoiceitem.empid=employee.empid} employee$.

Union ($\cup$) - is a primitive binary set operator of the form $R \cup S$ that builds a new relation R' with the distinct tuples from R and S (R' will not have duplicate tuples). The cardinality of R' is the sum of tuples from R and S. The union operation is possible only and if only R and S have the same set of attributes, that is, $attr(R) = attr(S)$. Formally, $R \cup S = \{t \mid t \in R \vee t \in S\}$.
Difference ($\setminus$ or $-$) - is a primitive binary set operator of the form $R \setminus S$ (or $R - S$) that builds a new relation R' with tuples from R that do not appear in S (R' will not have duplicate tuples). The difference operation is possible only and if only R and S have the same set of attributes, that is, $attr(R) = attr(S)$. Formally, $R \setminus S = \{t \mid t \in R \wedge t \notin S\}$.
Intersection ($\cap$) - is a binary set operator of the form $R \cap S$ that builds a new relation R' with tuples that appear in R and in S (R' will not have duplicate tuples). The intersection operation is possible only and if only R and S have the same set of attributes, that is, $attr(R) = attr(S)$. Formally, $R \cap S = \{t \mid t \in R \wedge t \in S\}$. Intersection is not a primitive set operator since $R \cap S = R \setminus (R \setminus S)$.
Division ($\div$) - is a binary set operator of the form $R \div S$ that builds a new relation R' with tuples of R that match to all tuples from S. The division operation is possible only and if only S have a subset of attributes of R. The matching is done using all atributes of S since $attr(S) \subset attr(R)$. This operation is used for finding tuples like: *returns, if exists, the actors that play a role in all movies directed by Alfred Hitchcok*, In our running example, the query could be: *returns, if exists, a client that appear in all invoices of january, 2011. It means that in january, 2011, only one client bought in our magazine.*
This query is $\pi_{prdid,name}(client) \div \pi_{prid}(\sigma_{billdate>=01/01/2011 \wedge billdate<01/02/2011}(invoice))$. Remark that we have to project both relation to satisfy division constraint: $attr(S) \subset attr(R)$.
The resulting relation schema will have the attributes from R that do not appear in S, *i.e.*,

$att(R) - attr(S)$.

Aggregation (group function - $\mathcal{G}$) - group functions are extended relational algebra operations that take a collection of values (from tuples) and return a single value as a result. As others algebra operators, group operators result a new relation.

count ($\mathcal{G}_{count(<attr>)(R)}$) - counts the number of tuples of R. Generaly, $<attr>$ is replaced by $*$ as parameter. Suppose that $card(client) = 1200$, that is, *client* has 1.200 tuples, the operation $\mathcal{G}_{count(*)}(client)$ the result is 1200.

sum ($\mathcal{G}_{sum(<attr>)(R)}$) - sum the values of $< attr >$ of the tuples of R and returns the result. $<attr>$ must be a numerical type.
For example, $\mathcal{G}_{sum(sellprice)}(client)$ sums all values of *sellprice* and returns the result (a single row).

avg ($\mathcal{G}_{avg(<attr>)(R)}$) - sum the values of $<attr>$ of the tuples of R and returns the average of the sum. $<attr>$ must be a numerical type.
For example, $\mathcal{G}_{avg(sellprice)}(invoiceitem)$ performs the average of all values of *sellprice* and returns the result (a single row).

max/min ($\mathcal{G}_{max(<attr>)(R)}$ $\mathcal{G}_{min(<attr>)(R)}$) - returns the highest/lowest value of $<attr>$ in the tuples of R.
For example, $\mathcal{G}_{max(price)}(product)$ returns the most expensive product stored in the database.

Group functions can be used together with the others algebra operators. For example, if we want to know how many invoices are created in 20/08/2010 we could perform:

$$\mathcal{G}_{count(*)}(\sigma_{billdate=20/08/2010}(invoice))$$

the same reasoning can be used to sum the salary of all employees hired in 10/10/2011:

$$\mathcal{G}_{sum(sal)}(\sigma_{hdate=10/10/2011}(employee))$$

Group functions can be also grouped by attributes. For example, if you want to count invoices by client, the command could be $clientid\mathcal{G}_{count(*)}(invoice)$. In this example, *count* function first groups all distinct *clientid* and counts the number of tuples for each *clientid*. If there exists 30 distinct values of *clientid* in *invoice*, the previous query would return 30 tuples with two columns: one representing the value of *clientid* and another representing the amount of tuples with the same value of *clientid*.

Renaming operator (ρ) - the rename operator is used to rename a relation and its attributes. Syntax: $\rho_{newname}(R)$ (R is named, after renaming operation, *newname*). To rename attributes, we use the following syntax: $\rho_{newname(att_1,...,att_n)}(R)$.
For example, $client \bowtie_{clientid=cid} \rho_{iv(id,cid,bdate,baddress}(\pi_{inumber,clientif,billdate,billaddress}(invoice))$. The projection over invoice selects four attributes and the rename operator renames the relation to *iv* and the four attributes to *id*, *cid*, *bdate* and *baddress*, respectively. Thus, the rename operator builds a new relation $iv(id, cid, bdate, baddress)$.