CMPT 354

Relational Algebra

Relational Algebra

- Introduction
- Relational Algebra Operations
  - Projection and Selection
  - Set Operations
  - Joins
  - Division
- Tuple Relational Calculus

Relational Query Languages

- Query languages allow the manipulation and retrieval of data from a database
- The relational model supports simple, but powerful query languages that
  - Have a strong formal foundation, and
  - Allow for optimization
- A query language is not a general purpose programming language
  - They are not Turing complete, and
  - Are not intended for complex calculations

Formal Query Languages

- Two mathematical query languages form the basis for SQL (and other query languages)
  - Relational Algebra – a *procedural* query language that is relatively close to SQL
  - Relational Calculus – a *non-procedural* language
- Understanding formal query languages is important for understanding
  - The origins of SQL
  - Query processing and optimization

Procedural vs. Non-procedural

- A procedural query consists of a series of operations
  - Which describe a step-by-step process for calculating the answer
  - e.g. giving precise directions on how to get somewhere
    - "…, turn left at the blue sign, drive for 1.5 miles, turn right at the cat, then left again at the red tower, …"
- A non-procedural (or declarative) query describes *what* output is desired, and not *how* to compute it
  - e.g.. giving the address someone is to go to
    - "go to the TD bank at 15th. and Lonsdale"

Results of a Query

- A query is applied to a relation instance, and the result of a query is a relation instance
  - The schema of the result relation is determined by the input relation and the query
  - Because the result of a query is a relation instance, it can be used as input to another query

Referring to Fields

- Fields in an instance can be referred to either by position or by name
  - Positional notation is easier for formal definitions
  - Using field names make queries easier to read
- The schema of the result is inherited from the input relations

Relational Algebra

Selection

- The selection operator, σ (sigma), specifies the *rows* to be retained from the input relation
- A selection has the form: $\sigma_{condition}(relation)$, where *condition* is a Boolean expression
  - Terms in the condition are comparisons between two fields (or a field and a constant)
  - Using one of the comparison operators: $<, \leq, =, \neq, \geq, >$
  - Terms may be connected by $\wedge$ (and), or $\vee$ (or),
  - Terms may be negated using $\neg$ (not)

Selection Example

$\sigma_{birth < 1981}(Customer)$

Customer

sin	firstName	lastName	birth
111	Buffy	Summers	1981
222	Xander	Harris	1981
333	Cordelia	Chase	1980
444	Rupert	Giles	1955
555	Dawn	Summers	1984

sin	firstName	lastName	birth
333	Cordelia	Chase	1980
444	Rupert	Giles	1955

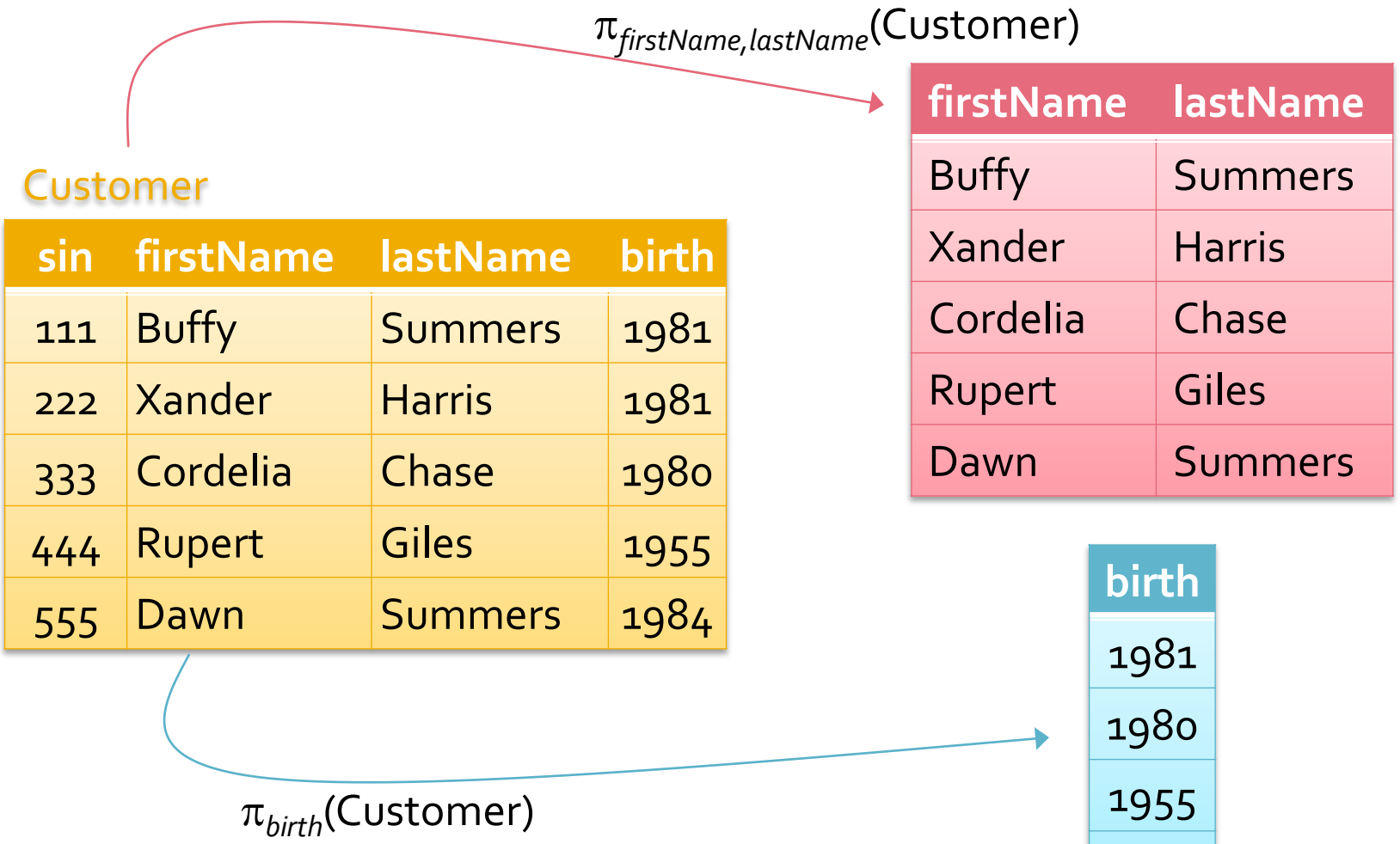
$\sigma_{lastName = "Summers"}(Customer)$

sin	firstName	lastName	birth
111	Buffy	Summers	1981
555	Dawn	Summers	1984

Projection

- The projection operator, π (pi), specifies the columns to be retained from the input relation
- A selection has the form: $\pi_{columns}(relation)$
  - Where *columns* is a comma separated list of column names
  - The list contains the names of the columns to be retained in the result relation

Projection Example



Selection and Projection Notes

- Selection and projection eliminate duplicates
  - Since relations are sets
  - In practice (SQL), duplicate elimination is expensive, therefore it is only done when explicitly requested
- Both operations require one input relation
- The schema of the result of a selection is *the same as* the schema of the input relation
- The schema of the result of a projection contains just those attributes in the projection list

Selection and Projection

$\pi_{sin, firstName}(\sigma_{birth < 1982 \wedge lastName = "Summers"}(Customer))$

Customer

sin	firstName	lastName	birth
111	Buffy	Summers	1981
222	Xander	Harris	1981
333	Cordelia	Chase	1980
444	Rupert	Giles	1955
555	Dawn	Summers	1984

intermediate relation

sin	firstName	lastName	birth
111	Buffy	Summers	1981

sin	firstName
111	Buffy

Set Operations

- Relational algebra includes the standard set operations:
  - Union, $\cup$
  - Intersection, $\cap$
  - Set Difference, $-$
  - Cartesian product (Cross product), $\times$

These are all binary operators

- All relational algebra operations can be implemented using five basic operations
  - *Selection*, *projection*, *union*, *set difference* and *Cartesian product*

Set Operations Review

$A = \{1, 3, 6\}$	$B = \{1, 2, 5, 6\}$	
Union ($\cup$)	$A \cup B \equiv B \cup A$	$A \cup B = \{1, 2, 3, 5, 6\}$
Intersection($\cap$)	$A \cap B \equiv B \cap A$	$A \cap B = \{1, 6\}$
Set Difference($-$)	$A - B \neq B - A$	$A - B = \{3\}$ $B - A = \{2, 5\}$
Cartesian Product ($\times$)	$A \times B \equiv B \times A^*$	$A \times B = \{(1,1),(1,2),(1,5),(1,6),(3,1),(3,2),(3,5),(3,6),(6,1),(6,2),(6,5),(6,6)\}$

* not strictly true, as pairs are ordered

Union Compatible Relations

- Union, set difference, and intersection can only be performed on *union compatible* operands
- Two relations, *R* and *S*, are union compatible if
  - They have the same number of fields
  - Corresponding fields, from left to right, have the same domains
- For convenience, we will assume that the result relation will inherit the field names of *R*
  - The schema of the result relation will be the same as the schema of *R*

Union Compatible Relations

Intersection of the Employee and Customer relations

Customer

sin	firstName	lastName	birth
111	Buffy	Summers	1981
222	Xander	Harris	1981
333	Cordelia	Chase	1980
444	Rupert	Giles	1955
555	Dawn	Summers	1984

Employee

sin	firstName	lastName	salary
208	Clark	Kent	80000.55
111	Buffy	Summers	22000.78
412	Carol	Danvers	64000.00

The two relations are not union compatible as birth is a DATE and salary is a REAL

We can carry out preliminary operations to make the relations union compatible

$\pi_{sin,\ firstName,\ lastName}(\text{Customer}) \cap \pi_{sin,\ firstName,\ lastName}(\text{Employee})$

Union

R

sin	firstName	lastName
111	Buffy	Summers
222	Xander	Harris
333	Cordelia	Chase
444	Rupert	Giles
555	Dawn	Summers

S

sin	firstName	lastName
208	Clark	Kent
111	Buffy	Summers
412	Carol	Danvers

$R \cup S$

sin	firstName	lastName
111	Buffy	Summers
222	Xander	Harris
333	Cordelia	Chase
444	Rupert	Giles
555	Dawn	Summers
208	Clark	Kent
412	Carol	Danvers

Returns all records in either relation (or both)

Intersection

R

sin	firstName	lastName
111	Buffy	Summers
222	Xander	Harris
333	Cordelia	Chase
444	Rupert	Giles
555	Dawn	Summers

S

sin	firstName	lastName
208	Clark	Kent
111	Buffy	Summers
412	Carol	Danvers

$R \cap S$

sin	firstName	lastName
111	Buffy	Summers

Only returns records that are in both relations

Set Difference

R

sin	firstName	lastName
111	Buffy	Summers
222	Xander	Harris
333	Cordelia	Chase
444	Rupert	Giles
555	Dawn	Summers

S

sin	firstName	lastName
208	Clark	Kent
111	Buffy	Summers
412	Carol	Danvers

R – S

sin	firstName	lastName
222	Xander	Harris
333	Cordelia	Chase
444	Rupert	Giles
555	Dawn	Summers

R – S returns all records in R that are not in S

S – R

sin	firstName	lastName
208	Clark	Kent
412	Carol	Danvers

Cartesian Product

- The *schema* of the result of $R \times S$ has one attribute for each attribute of the input relations
  - All of the fields of R, followed by all of the fields of S
  - Names are inherited if possible (i.e. if not duplicated)
    - If two field names are the same a *naming conflict* occurs and the affected columns are referred to by position
- The result *relation* contains one record for each pair of records $r \in R$, $s \in S$
  - Each record in R is paired with each record in S
    - If R contains m records, and S contains n records, the result relation will contain $m * n$ records

Cartesian Product Example

$\sigma_{lastName = "Summers"}(Customer)$

sin	firstName	lastName	birth
111	Buffy	Summers	1981
555	Dawn	Summers	1984

Account

acc	type	balance	sin
01	CHQ	2101.76	111
02	SAV	11300.03	333
03	CHQ	20621.00	444

$\sigma_{lastName = "Summers"}(Customer) \times Account$

1	firstName	lastName	birth	acc	type	balance	8
111	Buffy	Summers	1981	01	CHQ	2101.76	111
111	Buffy	Summers	1981	02	SAV	11300.03	333
111	Buffy	Summers	1981	03	CHQ	20621.00	444
555	Dawn	Summers	1984	01	CHQ	2101.76	111
555	Dawn	Summers	1984	02	SAV	11300.03	333
555	Dawn	Summers	1984	03	CHQ	20621.00	444

Renaming

- It is sometimes useful to assign names to the results of a relational algebra query
- The rename operator, ρ (rho) allows a relational algebra expression to be renamed
  - $\rho_x(E)$ names the result of the expression, or
  - $\rho_{x(A1,A2,\ldots,An)}(E)$ names the result of the expression, and its attributes

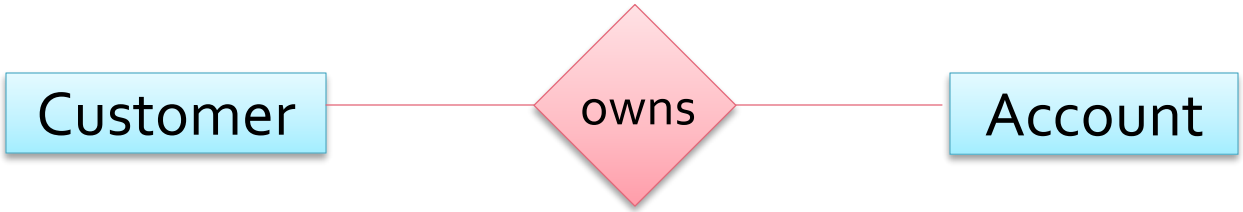
Largest Balance

- To find the largest balance first find accounts which are less than some other account
  - By performing a comparison on the Cartesian product of the account table with itself
    - The *Account* relation is referred to twice so with no renaming we have an ambiguous expression:
    - $\sigma_{account.balance < account.balance}(\text{Account} \times \text{Account})$
  - So rename one version of the Account relation
    - $\sigma_{account.balance < d.balance}(\text{Account} \times \rho_d(\text{Account}))$
- Then use set difference to find the largest balance

$$\pi_{accNumber,\ balance}(\text{Account}) - \pi_{account.accNumber,\ account.balance}(\sigma_{account.balance < d.balance}(\text{Account} \times \rho_d(\text{Account})))$$

Natural Joins

- It is often useful to simplify some queries that require a Cartesian product
- There is often a natural way to join two relations
  - e.g., finding data about customers and their accounts
    - *Owns* has foreign keys that reference *Account* and *Customer*
  - Compute the Cartesian product of *Customer* and *Owns*
    - Select the tuples where the primary key of *Customer* equals the foreign key attribute in *Owns*
    - Then repeat, using *accNumber*, with the result and *Account*



Natural Join Operation

- A natural join (denoted by ⋈) combines a Cartesian product and a selection
  - The selection consists of equality on all attributes that appear in both relations
  - Duplicate fields are dropped from the result relation
- The natural join of two tables with *no fields in common* is the Cartesian product
  - Not the empty set

i.e. with matching names and domains

Natural Join Example

Customer

sin	firstName	lastName	birth
111	Buffy	Summers	1981
222	Xander	Harris	1981
333	Cordelia	Chase	1980
444	Rupert	Giles	1955
555	Dawn	Summers	1984

Employee

sin	firstName	lastName	salary
208	Clark	Kent	80000.55
111	Buffy	Summers	22000.78
396	Dawn	Allen	41000.21
412	Carol	Danvers	64000.00

Customer ⋈ Employee

sin	firstName	lastName	birth	salary
111	Buffy	Summers	1981	22000.78

Theta Joins

- A theta join is an extension of the natural join operation
  - That combines any selection with a Cartesian product
  - Denoted as $R \bowtie_\theta S$ where $R \bowtie_\theta S \equiv \sigma_\theta(R \times S)$
- Each record in one relation is paired with each record of the other relation (a Cartesian product)
- Rows are only included in the result if they meet the join condition

Theta Join Example

Customer

sin	firstName	lastName	birth
111	Buffy	Summers	1981
222	Xander	Harris	1981
333	Cordelia	Chase	1980
444	Rupert	Giles	1955
555	Dawn	Summers	1984

Employee

sin	firstName	lastName	salary
208	Clark	Kent	80000.55
111	Buffy	Summers	22000.78
412	Carol	Danvers	64000.00

Customer $\bowtie_{\text{Customer.sin} < \text{Employee.sin}}$ Employee

1	2	3	birth	5	6	7	salary
111	Buffy	Summers	1981	208	Clark	Kent	80000
111	Buffy	Summers	1981	412	Carol	Danvers	64000
222	Xander	Harris	1981	412	Carol	Danvers	64000
333	Cordelia	Chase	1980	412	Carol	Danvers	64000

Division

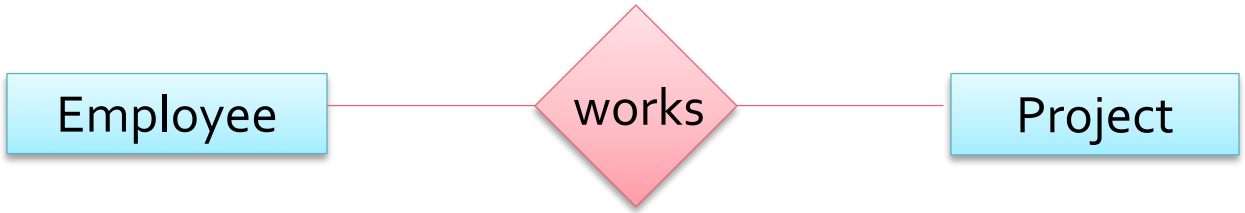
- Division is useful for queries which return records associated with *all* of the records in some subset
  - Find people who like all types of music
    - Country *and* Western??
  - Find tourists who have visited all of the provinces in Canada
- This operator is not always implemented in DBMSs
- But it can be expressed in terms of the basic set operators
  - Implementing a division query in SQL is a *fun* exercise

Division Example

- If *R* has two fields, *x* and *y*, and *S* has one field, *y*, then
  - $R \div S$ is equivalent to: $\pi_x(R) - \pi_x((\pi_x(R) \times S) - R)$
- R = {(a,1), (a,2), (a,3), (b,2), (b,3), (c,1), (c,3), (d,2)}
- S = {1, 2, 3}
- $\pi_x(R) \times S$ =
  - {(a,1), (a,2), (a,3), (b,1), (b,2), (b,3), (c,1), (c,2), (c,3), (d,1), (d,2), (d,3)}
- $(\pi_x(R) \times S) - R$ =
  - {(b,1), (c,2), (d,1), (d,3)}
- $\pi_x((\pi_x(R) \times S) - R)$ =
  - {b, c, d}
- $\pi_x(R) - \pi_x((\pi_x(R) \times S) - R)$ = {a, b, c, d} – {c, b, d} =
  - {a}

Division Example ...

- Find the *sin*s of employees who have worked on *all* of a company's projects
  - Assume that *Works* is a many-to-many relationship between *Employee* and *Project*
  - *Works* = {*sin*, *projectName*}
- The schemas of the input relations have to be carefully chosen
  - $Works \div \pi_{projectName}(Project)$



Other Operators

- There are additional relational algebra operators
  - Usually used in the context of query optimization
- Duplicate elimination – δ
  - Used to turn a bag into a set
- Aggregation operators
  - e.g. sum, average
- Grouping – γ
  - Used to partition tuples into groups
    - Typically used with aggregation

Summary

- Relational algebra is a procedural language that is used as the internal representation of SQL
- There are five basic operators: selection, projection, cross-product, union and set difference
  - Additional operators are defined in terms of the basic operators: intersection, join, and division
- There are often several equivalent ways to express a relational algebra query
  - These equivalencies are exploited by query optimizers to re-order the operations in a relational algebra expression

Tuple Relational Calculus

Non – Procedural Languages

- Both relational algebra and SQL are procedural languages
  - Where the query specifies how the result relation is computed
- Non-procedural languages just specify what the result should contain
  - Tuple Relational Calculus
  - Domain Relational Calculus

Tuple Relational Calculus

- A tuple relational calculus query is expressed as
  - $\{t \mid P(t)\}$
    - The set of tuples t, such that P is true for t
- In such a query t is a tuple variable and $P(t)$ is a formula that describes t
- For example: find all customers with incomes greater than 40000
  - $\{t \mid t \in Customer \wedge t.income > 40000\}$
    - The set of tuples t, such that t is in Customer and t's income is greater than 40000
- The query languages is a subset of first order logic

Formulae

- Tuple relational calculus queries are of the form
  - $\{t \mid P(t)\}$
- Where P is a *formula*, and formulae are built up using these rules
  - An atomic formula is a formula
  - If P_1 is a formula so are $\neg P_1$ and (P_1)
  - If P_1 and P_2 are formulae so are $P_1 \wedge P_2$, $P_1 \vee P_2$ and $P_1 \Rightarrow P_2$
  - If $P_1(s)$ is a formula with a free tuple variable s, and r is a relation, then $\exists s \in r(P_1(s))$ and $\forall s \in r(P_1(s))$ are formulae

Atomic Formulae

- If *Relation* is a relation name
- And *R* and *S* are tuple variables
  - And *a* is an attribute of *R* and *b* an attribute of *S*
- And if *op* is an operator in the set {<, ≤, =, ≠, ≥, >}
- Then an *atomic formula* is one of:
  - $R \in Relation$
  - *R.a op S.b*
  - *R.a op constant* or *constant op R.a*
- Tuple variables are either *free* or *bound*
  - A bound variable is quantified by $\exists$ *or* $\forall$

There Exists …

- The symbol ∃ means *there exists*
- Suppose we want to specify that only certain attributes are to be returned by a query
  - Return the customer's first and last names for customers with incomes greater than 40000
    - $\{t \mid \exists s \in Customer\ (s.income > 40000 \wedge t.firstName = s.firstName \wedge t.lastName = s.lastName)\}$
    - The set *t* such that there exists an *s* in Customer for which *t* has the same first and last names and the value of *s* for its income attribute is greater than 40000

Meanings

- The quantifiers $\exists$ and $\forall$ have these meanings
  - $\exists$ is the existential quantifier
    - Which means … there is some …
  - $\forall$ is the universal quantifier
    - Which means … for all …
- Like relational algebra, different tuple relational calculus queries may be equivalent
  - $P_1 \wedge P_2$ is equivalent to $\neg(\neg(P_1) \vee \neg(P_2))$
  - $\forall t \in r(P_1(t))$ is equivalent to $\neg \exists t \in r (\neg(P_1(t))$
    - For all t in r where condition – is equivalent to: there is no t in r where condition is not true
  - $P_1 \Rightarrow P_2$ is equivalent to $\neg(P_1) \vee P_2$

Multi Relation Example

- Schemata
  - Project = {pNumber, pName, description, npv, irr, advisorSIN}
  - Company = { companyName, city, …}
  - Undertakes = {pNumber, companyName, monitorSIN}

Return the project numbers, names and company names of projects that are monitored by employee 222 and that are undertaken by Vancouver companies (only include Vancouver company names in the result).

$\{t \mid \exists p \in Project\ \exists u \in Undertakes\ \exists c \in Company\ (p.pNumber = u.pNumber \wedge u.monitorSIN = 222 \wedge c.companyName = u.companyName \wedge c.city = \text{“Vancouver”} \wedge t.pNumber = p.pNumber \wedge t.pName = p.pName \wedge t.cName = c.cName)\}$

Query Builders and QBE

- Query-By-Example (QBE) is a query language that works in a non-procedural way
  - QBE was also the name of an early DBMS that used the language
  - In QBE queries are written as (and look like) tables
- Some modern DBMS have query builders that are similar to QBE
  - Graphical interfaces that allow the user to select what a query result should contain