Appendix E: Expressiveness of Relational Model Query Languages

Elmasri/Navathe:

- 9.3 “The Tuple Relational Calculus”
- 9.4 “The Domain Relational Calculus”

I have also used the following sources:

- Silberschatz/Korth/Sudarshan: Database System Concepts, 3rd Ed., 1999. Section 3.3, 3.4, 5.3.
- Kemper/Eickler: Datenbanksysteme (in German), Ch. 2, Oldenbourg, 1997.
- Lipeck: Skript zur Vorlesung Datenbanksysteme (in German), Univ. Hannover, 1996.

Overview

1. Expressiveness of Relational Algebra (short)
2. Domain and Tuple Calculus (short)

Limitations of Relational Algebra (1)

- Not all queries can be expressed in relational algebra.
- I.e. although you have the data in the database, it is not possible to write a query in relational algebra which computes the desired result (for all database states).
- This has to be expected, since relational algebra is not computationally complete.

  I.e. it is not a general computational device. In contrast, programming languages like C or Pascal are computationally complete, every known algorithm can be coded in them.

  If relational algebra were computationally complete, it would be subject to the halting problem, and we could never know whether a query terminates.

Limitations of Relational Algebra (2)

- So sometimes you have to write application programs in C, not only for the user interface, but also for doing more complex evaluations on the data than RA and SQL can do.
- These limitations already start with quite natural queries. A typical example is the class of transitive closure queries.
- Suppose you have stored the direct supervisor of every employee:

  Employee(EmpNo, Name, Mgr → Employee, Sal)
- Then you cannot compute the set of all indirect supervisors (i.e. supervisor, supervisor of supervisor, 3rd level supervisor, etc.) by means of a relational algebra query.

Limitations of Relational Algebra (3)

- Many problems are of this kind:
  - Given the “parent” relationship, you want to compute the grandparents, great-grandparents, etc. (ancestors).
  - Suppose your company assembles parts from components, which in turn are assembled from smaller components. Elementary parts are bought from other companies.
    Now you want to compute a “bill of materials”: What is the overall cost for a recursively composed part?
  - You have stored which procedure calls which other procedure. You want to compute on which procedures a given procedure depends directly or indirectly.

Limitations of Relational Algebra (4)

- So all these problems are of a kind where you must follow paths of unknown length in order to compute a result tuple.

  You can solve these problems in relational algebra or SQL if you know a maximum length for the path, e.g. that there are only four hierarchy levels. Then you can join up to four employee records together. But the query will be clumsy.

- Deductive databases and the SQL3 standard proposal contain recursion to allow such queries, Oracle SQL has a special extension for simple tree-structured data.

  But the expressivity of database query languages will always be less than that of arbitrary programming languages at least as long as query evaluation is guaranteed to terminate.

Relationally Complete Languages (1)

- Relational Algebra is not the only formal query language proposed for the relational model.
- Since database states in the relational model are interpretations of first order logic, it is natural to use first order logic as a query language ($\rightarrow$ Relational Calculus).
- Now the surprising result is that relational algebra and relational calculus can express exactly the same queries.
- I.e. every relational algebra query can be translated into an equivalent query of relational calculus, and vice versa.

  Equivalent means that for all database states, both queries return the same set of answers.

Relationally Complete Languages (2)

- So although not all sensible queries can be expressed in relational algebra, the set of queries which can be expressed in relational algebra is not completely arbitrary.

  Different formalisms lead to the same set of expressible queries.

- A query language is called relationally complete if it has (at least) the same expressive power as relational algebra.

  I.e. every relational algebra query can be translated into an equivalent query in this language.

- SQL is relationally complete. SQL without aggregations (and ordering) can express the same queries as relational algebra.

Overview

1. Expressiveness of Relational Algebra (short)
2. Domain and Tuple Calculus (short)

Domain Relational Calculus (1)

- Suppose we have the two tables:

  Composer(<u>CName</u>: string, Born: number, Died: number)

  Piece(<u>ID</u>: number, Title: string, CName $\rightarrow$ Composer)
- Then this query returns the life data of "Bach":

$$\{X, Y \mid \text{Composer}(\text{'Bach'}, X, Y)\}$$

- So here variables (like X) range over domain elements.

  In this example, X and Y are numbers.
- The relations are used as predicates (functions which return a boolean value). Attributes are identified by position.

Domain Relational Calculus (2)

- Terms of the Domain Relational Calculus:
  - Datatype constants, e.g. 'Bach' or 1685,
  - Variables, e.g. X or Name,
  - Composed terms, e.g. Died − Born.
- Atomic formulas of the Domain Relational Calculus:
  - Relation($\text{Term}_1, \ldots, \text{Term}_n$),

    e.g. Composer('Bach', X, Y),

    It is possible to fill an unneeded column with a "_". Since columns are identified by position in this formalism, all columns must be mentioned.
  - Term Comparison-Operator Term, e.g. $X < 1700$.

Domain Relational Calculus (3)

- Formulas are atomic formulas or composed from formulas φ_1 and φ_2 (and possibly a variable X) in the following ways:
  - $\neg\varphi_1$ ("not φ_1"),
  - $\varphi_1 \wedge \varphi_2$ ("φ_1 and φ_2"),
  - $\varphi_1 \vee \varphi_2$ ("φ_1 or φ_2")
  - $\varphi_1 \rightarrow \varphi_2$ ("if φ_1 then φ_2"),
  - $\exists X: \varphi_1$ ("there is an X with φ_1"),
  - $\forall X: \varphi_1$ ("for all X holds φ_1")
- If you prefer, you can use keywords, e.g. **not** instead of $\neg$, **and** instead of $\wedge$, and **or** instead of $\vee$.

Domain Relational Calculus (4)

- Parentheses must be used to make the structure clear. Otherwise the structure is determined by the precedence of the operators ($\neg$ highest, then $\wedge$, $\vee$, $\rightarrow$, and finally $\exists$ and $\forall$).
  E.g. $\varphi_1 \vee \varphi_2 \wedge \varphi_3$ is understood as $\varphi_1 \vee (\varphi_2 \wedge \varphi_3)$, not as $(\varphi_1 \vee \varphi_2) \wedge \varphi_3$.
- Queries have the form: $\{\text{Term}_1, \ldots, \text{Term}_n \mid \text{Formula}\}$.
- Variables must be bound to some finite domain (safeness).
  E.g. $\{\text{X} \mid \text{X} < 1700\}$ would not be a legal query.
  But $\{\text{X} \mid \text{X} < 1700 \wedge \text{Composer}(\_, \_, \text{X})\}$ is an allowed query.
  Since variables are anyway bound to a database column (or multiple columns / single constants in rare cases), there is no need to declare explicitly a datatype for the variables.

Domain Relational Calculus (5)

Examples:

- Print the names of all composers born before 1700:
  $$\{\text{X} \mid \text{Composer}(\text{X}, \text{Y}, \_) \wedge \text{Y} < 1700\}$$
- Who wrote "The Four Seasons"? Print name and birth year:
  $$\{\text{Name}, \text{Birth} \mid \text{Composer}(\text{Name}, \text{Birth}, \_) \wedge \text{Piece}(\_, \text{'The Four Seasons'}, \text{Name})\}$$
- Which composers have written more than one piece stored in the database?
  $$\{\text{Name} \mid \text{Piece}(\_, \text{X}, \text{Name}) \wedge \text{Piece}(\_, \text{Y}, \text{Name}) \wedge \text{X} \neq \text{Y}\}.$$

Domain Relational Calculus (6)

- Which composers have written a violin concerto?
  $$\{\text{Name} \mid \text{Piece}(\_, \text{Title}, \text{Name}) \wedge (\text{Title } \mathbf{like} \text{ '\%Violin\%Concert\%'} \vee \text{Title } \mathbf{like} \text{ '\%Concert\%Violin\%'})\}.$$
- Who is the oldest composer (earliest birth year)?
  $$\{\text{Name} \mid \text{Composer}(\text{Name}, \text{X}, \_) \wedge \forall\, \text{Y}\colon (\text{Composer}(\_, \text{Y}, \_) \rightarrow \text{X} \leq \text{Y})\}$$
  or $$\{\text{Name} \mid \text{Composer}(\text{Name}, \text{X}, \_) \wedge \neg \exists\, \text{Y}\colon (\text{Composer}(\_, \text{Y}, \_) \wedge \text{Y} < \text{X})\}$$

Some Logical Equivalences (1)

Commutativity of $\wedge$ and $\vee$:

- $\varphi_1 \wedge \varphi_2 \equiv \varphi_2 \wedge \varphi_1$
- $\varphi_1 \vee \varphi_2 \equiv \varphi_2 \vee \varphi_1$

Associativity of $\wedge$ and $\vee$:

- $(\varphi_1 \wedge \varphi_2) \wedge \varphi_3 \equiv \varphi_1 \wedge (\varphi_2 \wedge \varphi_3)$
- $(\varphi_1 \vee \varphi_2) \vee \varphi_3 \equiv \varphi_1 \vee (\varphi_2 \vee \varphi_3)$

Distribution Law:

- $\varphi_1 \wedge (\varphi_2 \vee \varphi_3) \equiv (\varphi_1 \wedge \varphi_2) \vee (\varphi_1 \wedge \varphi_3)$
- $\varphi_1 \vee (\varphi_2 \wedge \varphi_3) \equiv (\varphi_1 \vee \varphi_2) \wedge (\varphi_1 \vee \varphi_3)$

Some Logical Equivalences (2)

Double Negation:

- $\neg\neg\varphi \;\equiv\; \varphi$

De Morgan's Law:

- $\neg(\varphi_1 \wedge \varphi_2) \;\equiv\; (\neg\varphi_1) \vee (\neg\varphi_2)$
- $\neg(\varphi_1 \vee \varphi_2) \;\equiv\; (\neg\varphi_1) \wedge (\neg\varphi_2)$
- $\varphi_1 \wedge \varphi_2 \;\equiv\; \neg\big((\neg\varphi_1) \vee (\neg\varphi_2)\big)$
- $\varphi_1 \vee \varphi_2 \;\equiv\; \neg\big((\neg\varphi_1) \wedge (\neg\varphi_2)\big)$

Meaning of Implication:

- $\varphi_1 \rightarrow \varphi_2 \;\equiv\; (\neg\varphi_1) \vee \varphi_2$

Some Logical Equivalences (3)

Duality of Quantifiers:

- $\neg(\forall \mathsf{X}\colon \varphi) \;\equiv\; \exists \mathsf{X}\colon (\neg\varphi)$
- $\neg(\exists \mathsf{X}\colon \varphi) \;\equiv\; \forall \mathsf{X}\colon (\neg\varphi)$
- $\forall \mathsf{X}\colon \varphi_1 \;\equiv\; \neg\big(\exists \mathsf{X}\colon (\neg\varphi_1)\big)$
- $\exists \mathsf{X}\colon \varphi_1 \;\equiv\; \neg\big(\forall \mathsf{X}\colon (\neg\varphi_1)\big)$

Commutativity of Quantifiers:

- $\forall \mathsf{X}\colon (\forall \mathsf{Y}\colon \varphi) \;\equiv\; \forall \mathsf{Y}\colon (\forall \mathsf{X}\colon \varphi)$
- $\exists \mathsf{X}\colon (\exists \mathsf{Y}\colon \varphi) \;\equiv\; \exists \mathsf{Y}\colon (\exists \mathsf{X}\colon \varphi)$

Tuple Relational Calculus (1)

- In tuple relational calculus, variables are not ranging over domain elements, but over tuples from relations.
- Tuple relational calculus is similar to SQL.

  But SQL also has elements of relational algebra.
- E.g. the following query returns the life data of “Bach”:

  $$\big\{\mathsf{X.Born},\ \mathsf{X.Died} \;\big|\; \mathsf{Composer(X)} \wedge \mathsf{X.CName} = \text{'Bach'}\big\}$$
- The only syntax changes are:
  - Terms now have the form “Variable.Attribute”.

    Constants and composite terms are allowed as before.
  - Atomic formulas now have the form “Relation(Variable)”.

    Comparisons of two terms are allowed as before.

Tuple Relational Calculus (2)

- Print the names of all composers born before 1700:

  $$\big\{\mathsf{X.CName} \;\big|\; \mathsf{Composer(X)} \wedge \mathsf{X.Born} < 1700\big\}$$
- Who wrote “The Four Seasons”? Print name and birth year:

  $$\big\{\mathsf{C.CName},\ \mathsf{C.Born} \;\big|\; \mathsf{Composer(C)} \wedge \mathsf{Piece(P)} \wedge \mathsf{C.CName} = \mathsf{P.CName} \wedge \mathsf{P.Title} = \text{'The Four Seasons'}\big\}$$
- Which composers have written more than one piece stored in the database?

  $$\big\{\mathsf{X.CName} \;\big|\; \mathsf{Piece(X)} \wedge \mathsf{Piece(Y)} \wedge \mathsf{X.Title} \neq \mathsf{Y.Title}\big\}.$$

Tuple Relational Calculus (3)

- Which composers have written a violin concerto?

$$\{\text{P.CName} \mid \text{Piece(P)} \land (\text{P.Title } \textbf{like} \text{ '\%Violin\%Concert\%'} \lor \text{P.Title } \textbf{like} \text{ '\%Concert\%Violin\%'})\}.$$

- Who is the oldest composer (earliest birth year)?

$$\{\text{X.CName} \mid \text{Composer(X)} \land \forall\, \text{Y: } (\text{Composer(Y)} \rightarrow \text{X.Born} \leq \text{Y.Born})\}$$

$$\{\text{X.CName} \mid \text{Composer(X)} \land \neg\, \exists\, \text{Y: } (\text{Composer(Y)} \land \text{Y.Born} < \text{X.Born})\}$$