Relational data model

Domain: a predefined set of atomic values: integers, strings,...

First Normal Form: domains consist of atomic values.

Attribute: a label.

Relation:

- relation schema
- relation instance

Textbook: chapters 3, 4 and 5

Schema and instance

Relation instance:

- a finite subset of the cartesian product of one or more domains
- can be viewed as a *table*, consisting of *columns* and *rows* (tuples)

Relation schema:

- relation name
- vector of all attributes (and the associated domains)
- integrity constraints

An instance of a schema is **legal** if it satisfies the integrity constraints of the schema.

Integrity constraints

The constraints guard against entering incorrect data into a database.

Type constraints:

- the columns of each instance of a schema have to correspond to the attributes of the schema (and be identically named)
- the values in each column have to come from the domain associated with the corresponding attribute or be null.

Key constraints.

Foreign key constraints.

Semantic constraints.

Key constraints

Key constraint: a set of attributes S (called a **key**) of the relation schema R.

An instance r **satisfies** a key constraint S if r does not contain a pair of tuples that agree on S but disagree on some other atribute of R.

Formally: for each tuple $t_1 \in r$, $t_2 \in r$

if $t_1[S] = t_2[S]$, then $t_1[A] = t_2[A]$ for every attribute A in R.

Minimality property: no subset of a key can be designated a key.

The key serves as a **unique identifier** of a tuple. It should be as small as possible (good database design).

There may be more than one key in a relation schema; one is selected as the *primary* key.

Entity integrity: the primary key cannot have a null value.

Superkey: a superset of a key.

Relational model is value-based

Property 1.

There cannot be two different "objects" (here: tuples) with identical attribute values.

Property 2.

The only way to reference an "object" (tuple) is by providing its key value.

These properties are *not* shared by the ER model, object-oriented models etc.

Foreign keys

Relation schemas R_1, R_2 (not necessarily distinct).

Foreign key constraint: a pair of sets of attributes (S_1, S_2) such that:

- $S_1 \subseteq R_1$, $S_2 \subseteq R_2$,
- S_2 is a key of R_2
- the number of attributes and their respective domains in S_1 and S_2 are the same.

A pair of instances (r_1, r_2) **satisfies** a foreign key constraint (S_1, S_2) if for every tuple $t_1 \in r_1$, $t_1[S_1] = t_2[S_2]$ for some tuple $t_2 \in r_2$ or $t_1[S_1]$ is null.

A primary key (or a part thereof) can be a foreign key at the same time (but then it can't be null).

Query languages for relational databases

Relational algebra:

- a set of algebraic operators,
- procedural (describe how to evaluate the query).

Relational calculus:

- domain calculus (first-order logic) or tuple calculus,
- declarative (query evaluation unspecified).

SQL:

- a mix of relational algebra and calculus.
- the standard query language of the existing DBMS.

Relational algebra

Query language for relational databases.

Consists of **operators** that can be nested to form **expressions**.

All the operators take relations as arguments and return relations as results.

Basic operators

Set-theoretic:

- union,
- set difference.

Relational:

- Cartesian product,
- selection,
- projection,
- renaming.

This is a **minimal** set of operators.

Derived operators are also defined.

Compatibility of arguments

Some binary operations require **compatibility** of arguments.

Two relations R_1 and R_2 are compatible if:

- $arity(R_1) = arity(R_2)$,
- the corresponding attribute domains in R_1 and R_2 are the same.

The compatibility of R_1 and R_2 can be determined solely on the basis of their schemas.

Union and difference

R_1, R_2 - compatible relations.

Union $(\cup)$ of R_1 and R_2:

- $arity(R_1 \cup R_2) = arity(R_1) = arity(R_2)$
- $t \in R_1 \cup R_2$ iff $t \in R_1$ or $t \in R_2$.

Difference $(-)$ of R_1 and R_2:

- $arity(R_1 - R_2) = arity(R_1) = arity(R_2)$
- $t \in R_1 - R_2$ iff $t \in R_1$ and $t \notin R_2$.

Cartesian product

$arity(R_1) = k_1$

$arity(R_2) = k_2$

Cartesian product($\times$):

- $arity(R_1 \times R_2) = arity(R_1) + arity(R_2)$
- $t \in R_1 \times R_2$ iff:
  - the first k_1 components of t form a tuple in R_1, and
  - the next k_2 components of t form a tuple in R_2.

Selection

A selection condition E is built from:

- comparisons between operands which can be constants or attribute names,
- Boolean operators: $\wedge$ (AND), $\vee$ (OR), $\neg$ (NOT).

Selection $\sigma_E(R)$:

- $arity(\sigma_E(R)) = arity(R)$,
- $t \in \sigma_E(R)$ iff $t \in R$ and t satisfies E.

Projection

$A_1, \ldots, A_k$: distinct attributes of R.

Projection $\pi_{A_1,\ldots,A_k}(R)$:

- $arity(\pi_{A_1,\ldots,A_k}(R)) = k$,
- $t \in \pi_{A_1,\ldots,A_k}(R)$ iff for some $s \in R$, $t[A_1] = s[A_1], \ldots, t[A_k] = s[A_k]$.

Renaming

$A_1, \ldots, A_n$: attributes of R

$B_1, \ldots, B_n$: new attributes.

Renaming $R(B_1, \ldots, B_n)$**:**

- $arity(R(B_1, \ldots, B_n)) = arity(R) = n$,
- $t \in R(B_1, \ldots, B_n)$iff for some $s \in R$, $t[B_1] = s[A_1], \ldots, t[B_n] = s[A_n]$.

Derived operators

1. Intersection.
2. Quotient.
3. θ-join.
4. Natural join.

Intersection

Intersection:

- $arity(R_1 \cap R_2) = arity(R_1) = arity(R_2)$
- $t \in R_1 \cap R_2$ iff $t \in R_1$ and $t \in R_2$.

Intersection is a derived operator:

$$R_1 \cap R_2 = R_1 - (R_1 - R_2).$$

Quotient

$A_1, \ldots, A_{n+k}$: all the attributes of R_1.

$A_{n+1}, \ldots, A_{n+k}$: all the attributes of R_2.

R_2 nonempty.

Quotient (division):

- $arity(R_1 \div R_2) = arity(R_1) - arity(R_2) = n$.
- $t \in R_1 \div R_2$ iff for all $s \in R_2$ there is a $w \in R_1$ such that
  - $t[A_1] = w[A_1], \ldots, t[A_n] = w[A_n]$, and
  - $s[A_{n+1}] = w[A_{n+1}], \ldots, s[A_{n+k}] = w[A_{n+k}]$.

Quotient is a derived operator:

$$\begin{aligned} R_1 \div R_2 \quad = \quad & \pi_{A_1,\ldots,A_n}(R_1) - \\ & \pi_{A_1,\ldots,A_n}(\pi_{A_1,\ldots,A_n}(R_1) \times R_2 - R_1) \end{aligned}$$

θ-join

θ: a comparison operator $(=, \neq, <, >, \geq, \leq)$.

$A_1, \ldots, A_n$: all the attributes of R_1.

$B_1, \ldots, B_k$: all the attributes of R_2.

θ-join:

- $arity(R_1 \underset{A_i \theta B_j}{\bowtie} R_2) = arity(R_1) + arity(R_2)$
- $R_1 \underset{A_i \theta B_j}{\bowtie} R_2 = \sigma_{A_i \theta B_j}(R_1 \times R_2)$

Natural join

$A_1, \ldots, A_n$: all the attributes of R_1.

$B_1, \ldots, B_k$: all the attributes of R_2.

m - the number of attributes common to R_1 and R_2.

Natural join:

- $arity(R_1 \bowtie R_2) = arity(R_1) + arity(R_2) - m$
- to obtain $R_1 \bowtie R_2$:
  1. select from $R_1 \times R_2$ the tuples that agree on all attributes common to R_1 and R_2;
  2. project duplicate columns out from the resulting tuples.

Relational calculus

Basic notions:

- constants;
- variables;
- database relation symbols;
- binary comparison symbols;
- boolean connectives;
- quantifiers.

Variables:

- atomic values (domain relational calculus), or
- tuples (tuple relational calculus).

Variables are either bound by a quantifier or free.

Syntax

Query conditions:

- atomic conditions:
  - $T \in R$ where R is a relation name and T is a tuple variable
  - $T.A\ \theta\ W.B$ where T and W are tuple variables, $\theta \in \{=, \neq, <, >, \geq, \leq\}$
  - $T.A\ \theta\ d$ where d is a constant.
- $C_1 \wedge C_2$, $C_1 \vee C_2$, $\neg C_1$, where C_1 and C_2 are query conditions
- $\forall T \in R(C)$ and $\exists T \in R(C)$ where T is a tuple variable, R a relation name and C a query condition.

Semantics

Query:

$$\{T \mid C\}$$

where C is a query condition and T is a tuple variable (or consists of different components of free tuple variables from C).

Query result:

given a database D, a query $\{T \mid C\}$ returns the set of all tuples which substituted for all the occurrences of T in C make it true in D.

Properties of quantifiers

$$\forall T_1 \in R(C) \equiv \neg\exists T_1 \in R(\neg C).$$

$$\forall T_1 \in R.\forall T_2 \in S.(C) \equiv \forall T_2 \in S.\forall T_1 \in R(C).$$

$$\exists T_1 \in R.\exists T_2 \in S.(C) \equiv \exists T_2 \in S.\exists T_1 \in R(C).$$

In general:

$$\forall T_1 \in R.\exists T_2 \in S.(C) \not\equiv \exists T_2 \in S.\forall T_1 \in R(C).$$

SQL

Supported by virtually all relational DBMS.

Standardized:

- SQL2 or SQL-92 (completed 1992);
- SQL3, SQL:1999, SQL:2003 (ongoing work).

Components:

- query language;
- data definition language;
- data manipulation language;
- integrity constraints and views;
- API's (ODBC, JDBC);
- host language preprocessors (Embedded SQL, SQLJ);
- ...

The query language is a hybrid of relational algebra and calculus.

Basic SQL queries

The basic form:

`SELECT` $A_1, \ldots, A_n$
`FROM` $R_1, \ldots, R_k$
`WHERE` C

This corresponds to the following relational algebra expression:

$$\pi_{A_1,\ldots,A_n}(\sigma_C(R_1 \times \cdots \times R_k))$$

Range variables

To refer to a relation more than once in the `FROM` clause, **range variables** are used.

Example.

```
SELECT R1.A, R2.B
FROM R R1,R R2
WHERE R1.B=R2.A
```

corresponds to:

$$\pi_{A,D}(R(A,B) \underset{B=C}{\bowtie} R(C,D)).$$

Manipulating the result

`SELECT *`: all the columns are selected.

`SELECT DISTINCT`: duplicates are eliminated from the result.

`ORDER BY` $A_1, \ldots, A_m$: the result is sorted according to $A_1, \ldots, A_m$.

E `AS` A can be used instead of an column A in the `SELECT` list to mean that the value of the column A in the result is determined using the (arithmetic or string) expression E.

Set operations

`UNION` set union.

`INTERSECT` set intersection.

`EXCEPT` set difference.

Notes:

- `INTERSECT` and `EXCEPT` can be expressed using other SQL constructs;
- some systems, e.g., SYBASE, do not support `INTERSECT` and `EXCEPT`.

Nested queries

A query Q can appear as a **subquery** in the `WHERE` clause which can now contain:

- A `IN` Q: for set membership ($A \in Q$);
- A `NOT IN` Q: for the negation of set membership ($A \notin Q$);
- $A\ \theta$ `ALL` Q: A is in the relationship θ to **all** the elements of Q ($\theta \in \{=, <, >, >=, <=, <>\}$);
- $A\ \theta$ `ANY` Q: A is in the relationship θ to **some** elements of Q;
- `EXISTS` Q: Q is nonempty;
- `NOT EXISTS` Q: Q is empty.

Notes:

- the subqueries can contain columns from enclosing queries;
- multiple occurrences of the same column name are disambiguated by choosing the closest enclosing `FROM` clause.

Aggregation

Instead of a column A, the `SELECT` list can contain the results of some aggregate function applied to all the values in the column A in the relation.

The supported functions are:

- `COUNT(`A`)`: the number of all values in the column A (with duplicates);
- `SUM(`A`)`: the sum of all values in the column A (with duplicates);
- `AVG(`A`)`: the average of all values in the column A (with duplicates);
- `MAX(`A`)`: the maximum value in the column A;
- `MIN(`A`)`: the minimum value in the column A.

`DISTINCT` A, instead of A, considers only distinct values.

Grouping

The clause

`GROUP BY` $A_1, \ldots, A_n$

assembles the tuples in the result of the query into groups with identical values in columns $A_1, \ldots, A_n$.

The clause

`HAVING` C

leaves only those groups that satisfy the condition C.

The `SELECT` list of a query with `GROUP BY` can contain only:

- the columns mentioned in `GROUP BY` (or expressions with those), or
- the result of an aggregate function, which is then viewed as applied group-by-group.

Building complex queries

A complex query can be broken up into smaller pieces using:

- nested queries in the `FROM` clause
- views.

A **view** is a computed relation whose contents are defined by an SQL query.

Create a view:

`CREATE VIEW` *View-name*`(`*Attr1*`, . . . ,`*Attrn*`)`
`AS` *Query*

Drop a view:

`DROP VIEW` *View-name*

Null values

Various interpretations: **unknown**, missing value, inapplicable, no information...

In SQL columns that not explicitly or implicitly designated as `NOT NULL` can assume **null values**.

Behavior of null values:

- comparisons return the **unknown** truth value if at least one of the arguments is null;
- `IS NULL` returns true;
- null values counted by `COUNT(*)`, discarded by other aggregate operators.

Three-valued logic

NOT	
T	F
F	T
?	?

AND	T	F	?
T	T	F	?
F	F	F	F
?	?	F	?

OR	T	F	?
T	T	T	T
F	T	F	?
?	T	?	?

Outer joins

To keep the tuples in the result if there are no matching tuples in the other argument of the join:

- `LEFT`: preserve only the tuples from the left argument
- `RIGHT`: preserve only the tuples from the right argument
- `FULL`: preserve the tuples from both arguments.

The result tuples are padded with nulls.

Syntax (in the `FROM` clause):

R_1 `OUTER JOIN` R_2 `ON` *Condition* `USING` *Columns*

Outer joins can be expressed using other SQL constructs.

Some DBMS, e.g., Oracle, use a different syntax for outerjoins.

Limitations of relational query languages

They cannot express queries involving transitive closure of binary relations:

- *"List all the ancestors of David."*
- *"Find all the buildings reachable from Bell Hall without going outside."*

Solution: *recursive* views.

Recursion in SQL3

A *recursive* view definition can refer to the view being defined.

Recursive views can be used in SQL3 queries.

If R is recursively defined, it should be preceded by `RECURSIVE`.

Syntax:

`WITH` R AS
definition of R
query to R

Example:

```
WITH RECURSIVE Anc(Upper,Lower) AS
     (SELECT * FROM Parent)
  UNION
     (SELECT P.Upper, A.Lower
      FROM Parent AS P, Anc AS A
      WHERE P.Lower=A.Upper)

SELECT Anc.Upper
FROM Anc
WHERE Anc.Lower='David';
```

Restrictions

R *depends* on Q: Q is used, directly or indirectly, in the definition of R.

Stratification restriction:

> No view can depend on itself through `EXCEPT` or aggregation.

Evaluating queries with recursive views

1. Initially, the contents of all views are empty.
2. Compute the new contents of the views, using database relations and the current contents of the views.
3. Repeat the previous step until no changes in view contents occur.

Why does this *terminate*?