The Relational Data Model: Fundamental Operations

1 Overview

- Now that we've looked at the structure of a relational database, we will look at its fundamental *operations* — that is, what you can do with that database.
- We can think of relational operations on multiple levels:
  - Pure mathematical abstractions: this would be the relational algebra, and if you go non-procedural, you can sort of think of the relational calculus in this manner — ultimately these form the basis for reasoning about a relational database.
  - Programmatic interaction: when actually interacting with a real live database, we need a programming language — and many programming languages are geared toward database operation. These are typically categorized as *query* languages; more formally there are *data manipulation languages* (DMLs) and *data definition languages* (DDLs).
    - A short list of query languages: SQL, QBE, Datalog, "pure" languages that attempt to translate the relational algebra or calculus directly
- For this handout, we focus on the foundation of it all: the basic operations of the relational algebra.

2 Fundamental Operations

- We have seen that a relational database is a set of relations r, each of which follows a particular relation schema R.
- The fundamental operations define what we can do with these relations. Note that all of these operations take relations as operands and produce relations as results — this is the mathematical property of *closure*.
- Some of these operations are *unary*: they take a single relation as an operand. Other operations are *binary*: they require two relation operands.

2.1 Select

- The *select* operation is a unary operation on a relation r that is denoted by the lowercase Greek letter sigma (σ) and includes a predicate P. It is written as:

  $$\sigma_P(r)$$

  The select operation results in a new relation consisting only of the tuples t in r for which P is true.

- P is a boolean expression that refers to attributes in r and compares them either to constant values or other attributes via $=, \neq, <, \leq, >, \geq$.

- Individual comparisons can be combined using logical *and* ($\wedge$), *or* ($\vee$), and *not* ($\neg$). These operators have the boolean meanings that you probably know and love by now.

- You can think of the select operation as a form of *horizontal partitioning* of the relation.

2.2 Project

- The *project* operation is a unary operation on a relation r that is denoted by uppercase Greek letter pi (Π) and includes an attribute list A. It is written as:

  $$\Pi_A(r)$$

  For all tuples t in r, the project operation results in a new relation consisting of $t[A]$.

- A is written as a comma-separated list of attributes.

- Remember that relations are sets, so tuples t_1 and t_2 that are not equal in r but have the same values for A (that is, $t_1[A] = t_2[A]$) become a single tuple in $\Pi_A(r)$.

- Remember again that the set of all relations is *closed* under these operations; thus, you can *compose* them as needed. For example, a query on a MMORPG database such as "Retrieve the names of all players who belong to server *Dethecus*" can be written as a select followed by a project (assume the relation is named $players$):

  $$\Pi_{player\_name}(\sigma_{server=\text{“}Dethecus''}(players))$$

2.3 Union

- The *union* operation is a binary operation on relations r and s that is denoted by the same symbol as in set theory: $\cup$. It is written as:

  $$r \cup s$$

  The result is a new relation such that $\forall\ t_r \in r$ and $t_s \in s$, $t_r \in r \cup s$ and $t_s \in r \cup s$.

- We diverge from set theory in that the union of relations only makes sense if they have *compatible* schemas. Thus, union only applies if all of the following conditions are true:
  - r and s have the same *arity* — meaning that they have the same number of attributes n. Notationally, if A_r is the set of r's attributes and A_s is the set of s's attributes, then $|A_r| = |A_s|$.
  - $\forall\ 1 \leq i \leq n$, the domain of attribute i in r must be the same as the domain of attribute i in s.
- Note how the requirements are essentially *number of attributes* and *corresponding domains*. Attribute names aren't involved.

2.4 Set-Difference

- The *set-difference* operation is a binary operation on relations r and s that is denoted by $-$. It is written as:

  $$r - s$$

  The result is a new relation consisting of tuples t such that $t \in r$ but $t \notin s$.
- As with union, set-difference only makes sense for compatible relations: same arity, corresponding domains.

2.5 Cartesian-Product

- The *Cartesian-product* operation is a binary operation on relations r_1 and r_2 that is denoted by $\times$. It is written as:

  $$r_1 \times r_2$$

  If r_1 has schema R_1 and r_2 has schema R_2 (that is, $r_1(R_1)$ and $r_2(R_2)$, $r = r_1 \times r_2$ is a relation whose schema R is the concatenation of R_1 and R_2. The resulting relation r contains all tuples t such that $\exists\ t_1 \in r_1$ and $t_2 \in r_2$ for which $t[R_1] = t_1[R_1]$ and $t[R_2] = t_2[R_2]$.
- When specifying the schema for some $r_1 \times r_2$ in which r_1 and r_2 have attributes of the same name, we use *dot notation* to prepend the attribute name with the relation name:

  $$R_1 = (tom, dick, harry)$$
  $$R_2 = (harry, hausen)$$
  $$R = (tom, dick, r_1.harry, r_2.harry, hausen)$$
- We can take the Cartesian product of a relation with itself. However, we need to distinguish the attributes that came from the first "instance" of the relation from those that came from the second "instance." For this, we *rename* one of the operands, then use the same dot notation (see Section 2.6).

- It's easy to figure out how many tuples are in $r_1 \times r_2$: if r_1 has n_1 tuples and r_2 has n_2 tuples, then $r_1 \times r_2$ has $n_1 n_2$ tuples.
- Note how the Cartesian product combines *all* tuples in r_1 with *all* tuples in r_2 — not very useful. In practice, we perform a select on the Cartesian product, choosing a condition that expresses a meaningful connection between some attribute(s) of r_1 and r_2:

  $$\sigma_{westplayer.player=eastplayer.player}(westplayer \times eastplayer)$$

  This Cartesian product followed by a select has a name of its own — it is called a *join*. More on joins later.

2.6 Rename

- Note how, while we start out with a set of named relations in a relational database, by default, the relations that are *results* of relational operators don't have names. For long sequences of operations, or for binary operations that take the same relations as operands, the *rename* operation presents a solution that eliminates the ambiguity or confusion that may emerge in those situations.
- The rename operation is denoted by the lowercase Greek letter rho (ρ), specifies a new name x, and accepts any relational expression E as its operand:

  $$\rho_x(E)$$

  E is *any* relational expression, so E covers everything from a single named relation to a complex sequence of operations on multiple relations.
- It is possible to rename not only a relation but also its attributes. In that case, we can provide an attribute name list $(a_1, a_2, \ldots, a_n)$ — where n is the arity of E — along with the new relation name x:

  $$\rho_{x(a_1,a_2,\ldots,a_n)}(E)$$

- A little footnote — remember that in the pure mathematical theory of relations, attributes don't have names; they go by number, in the order that they are listed in a relation schema. We accommodate positions by prepending a \$ sign before the position. Thus, $\sigma_{\$2=\$3}(r \times r)$ refers to all tuples in the Cartesian product of r with itself such that the values of the second and third attributes of $r \times r$ are equal.
  - Positional notation eliminates the need to rename, but it's hard for people to read, requiring top-of-your-head knowledge of how attributes are ordered in a schema. So generally, we don't use positional notation.

3 Formal Definition of Relational Algebra

- We now have all of the definitions that we need to formally define relational algebra — specifically, what comprises a valid expression in this algebra.
- A basic expression in the relational algebra is either:
  - A relation in a database
  - A constant (literal) relation: written as a comma-separated sequence of tuples enclosed in braces { }. The tuples themselves are comma-separated sequences of values, enclosed in parentheses ( ).
- A general expression in the relational algebra is a composition of smaller subexpressions. If E_1 and E_2 are expressions in the relational algebra, then so are these:

$$E_1 \cup E_2$$

$$E_1 - E_2$$

$$E_1 \times E_2$$

$$\sigma_P(E_1) \text{ where } P \text{ is a predicate on the attributes of } E_1$$

$$\Pi_S(E_1) \text{ where } S \text{ is a list of some of the attributes of } E_1$$

$$\rho_x(E_1) \text{ where } x \text{ is the new name for the result of } E_1$$

- And that's it — everything that you can do with relations can be decomposed ultimately into the above expressions.
- While we're being mathematical, think about the other properties involved in an algebra:
  - You've already seen how these operations have *closure* within the set of all relations: for a unary operation o and relation r, $o(r)$ is a relation, and for a binary operation $\cdot$ and relations r_1 and r_2, $r_1 \cdot r_2$ is a relation.
  - Which operations are *associative* — that is, for a given operation $\cdot$, $(r_1 \cdot r_2) \cdot r_3 = r_1 \cdot (r_2 \cdot r_3)$?
  - Are there *identity elements* for these operations? That is, for a given operation $\cdot$, is there a relation r_i such that $r_i \cdot r = r$ and $r \cdot r_i = r$ $\forall$ relations r?
  - Are there *inverse elements* in these operations? That is, for a given operation $\cdot$ and $\forall$ relations r, is there a relation r^{-1} such that $r \cdot r^{-1} = r_i$ and $r^{-1} \cdot r = r_i$?

  If a binary relational operation has these properties, then you have a *group* with respect to that operation and the set of all relations!
- If you take two relational operations, say + and $\cdot$, and the following conditions hold for relations:

  - $+$ and $\cdot$ are closed,
  - $+$ and $\cdot$ are associative,
  - $+$ and $\cdot$ are *commutative* — that is, $r_1 + r_2 = r_2 + r_1$ and $r_1 \cdot r_2 = r_2 \cdot r_1$,
  - $\cdot$ is *distributive* over $+$ — defined as $r_1 \cdot (r_2 + r_3) = (r_1 \cdot r_2) + (r_1 \cdot r_3)$ and $(r_1 + r_2) \cdot r_3 = (r_1 \cdot r_3) + (r_2 \cdot r_3)$,
  - $+$ has an identity element and inverse elements as defined above

  ... you have a *commutative ring*!

- If the $\cdot$ operation in a commutative ring also has an identity element, then you have a *commutative ring with identity*.

- If the $\cdot$ operation in a ring has inverse elements for every relation except for the relation identified as the identity element for $+$, then you have a *field*.

- *Why do you care?* Once you have identified a group, ring, or field, then that opens a whole slew of known truths about those constructs (theorems, lemmas) — and in many cases, these truths lead to powerful yet practical algorithms, optimizations, and implementations when you finally start to develop a relational database management system.