

1 Relational Model

Relational Model: Basic Structure

- *relation* is a table
- *tuple* is a row in the table

In relational databases we enforce the condition:

- for all relations r , the domains of all attributes are *atomic*;
domain is atomic if elements in it are indivisible units

Example: set of integers is atomic, but set of all sets of integers is not atomic

Example: Bank Schema

- ***branch*** = (*branch-name*, *assets* , *branch-city*)
- ***customer*** = (*Cust-ID*, *customer-name*, *street* , *city* , *zip*)
- ***deposit*** = (*branch-name*, *account-number* , *Cust-ID*, *balance*)
- ***loan*** = (*branch-name*, *loan-number* , *Cust-ID*, *amount*)

Bank Schema... contd.

- key for ***customer*** is *Cust-ID* (similar to SSN)
- key for ***deposit*** – *account-number* ?
 - what if joint ownership of account – which customer ID associated with account ?
- key for ***branch*** – is *branch-name*, assumes unique name for each branch
- key for ***loan*** – *loannumber*

Bank Schema Example Example: The deposit relation and the scheme. The scheme states the attributes (structure) of the relation.

- $Deposit\text{-}scheme = (branch\text{-}name, account\text{-}number, Cust\text{-}ID, balance)$
- to specify the domains we may need to say $branch\text{-}name :string$ etc.
- $deposit(Deposit\text{-}scheme)$ denotes that **deposit** is a relation on the $Deposit\text{-}scheme$

Relational Model: Basic Structure

If t refers to a tuple then $t[i]$ is the value of the i -th attribute of tuple t ; *i.e.*, if $t = (v_1, v_2, \dots, v_i, \dots, v_n)$ is the tuple then $t[i] = v_i$.

Example: if t is the first tuple in **deposit** relation then $t[branch\text{-}name]$ is the value of t on $branch\text{-}name$ attribute, and thus

$t[branch\text{-}name] = \text{“Downtown”}$

Relational Model: Database Scheme..contd..

- two relation schemes may have same attributes (i.e., domain set D_i)
- this allows us to relate tuples of different relations
 - $Cust\text{-}ID$ appears in deposit and customer schemes
 - can use this to find cities of depositors of Downtown branch
 - note that different names could have been used (cust-ID and CID)
 - look into **deposit** to find all depositors of Downtown
 - for each customer look into **customer** to find their city

Relational Model: Database Scheme How to pick relation schemes ?

Example: for banking example, can have one relation scheme with $Account\text{-}info\text{-}scheme = (branch\text{-}name, account\text{-}number, Cust\text{-}ID, customer\text{-}name, balance, street, city)$

- if customer has several accounts then must list address several times: waste of storage
- if customer has not provided address then must use *null* values

Remark: it is not always possible to remove *null* values

Database Design: will discuss methods to choose relation schemes

- what is a “good schema” ? Can we formally define what we mean by “good schema” ? Is there a mathematical model for defining this concept ?
- this will be covered in our discussion of normal forms and the concept of functional dependencies.

Entity Integrity, Referential Integrity, Foreign keys

- **entity integrity constraint:** states no primary key can be null
primary key has to be used to identify tuples
- **referential integrity constraint:** specified between two relations; used to maintain consistency among tuples of relations
 - referential int. const.: tuple in one relation that refers to another relation must refer to an existing tuple in that relation

Relational Model: Query Languages

- a *query language* is a language in which a user requests information from the database
- they can be *procedural* or *nonprocedural*
- *procedural:* user specifies what data and specifies instructions to compute desired result
- *nonprocedural:* user specifies what data without giving specific procedure for computing result

Formal Query languages: relational algebra (proc.), and tuple calculus, domain calculus (non-proc.)

Commercial query languages: SQL, QUEL, QBE....

Relational Algebra

- proposed by Codd in his seminal work on relational database model.
- theoretical basis on which commercial query languages were developed
- set of mathematical operators that can be composed, and are able to compose, modify, and combine tuples within different relations
 - set union: $A \cup B$
 - set intersection: $A \cap B$
 - cross product: $A \times B$
 - set difference: $A - B$
 - above are standard set operations, we then add some “new ones”:
 - Select operator σ_{θ} to select tuples that satisfy predicate θ
 - Project operator π_{list} to project attributes in *list*.
 - Rename operator $\rho_P(R)$ to rename relation R to new name P
- some additional operations: Join, intersection
- insertion, deletion and update operations to modify the relations/database
- concept of views

Relational Algebra: Expressions Relational algebra operations operate on relations and produce relations (i.e., *closed algebra*).

Formal syntax for relational algebra: an expression in relational algebra (RA) is defined recursively:

(Basis) basic expression (query) in rel. alg. consists of either one of:

- a relation in the database
- a constant relation

(Recursion step) let E_1 and E_2 be RA expressions, then the following are also RA expressions

- $E_1 \cup E_2$
- $E_1 - E_2$
- $E_1 \times E_2$
- $\sigma_P(E_1)$, where P is a predicate on attributes in E_1
- $\pi_S(E_1)$ where S is a list consisting of some of the attributes in E_1
- $\rho_x(E_1)$ where x is the new name for E_1
- can you write a Context free grammar to specify the syntax of this language ??

Writing Queries in Relational Algebra A query is applied to *relation instances*, and the result of the query is also a relation instance. The schemas of input relations are fixed (but query will run regardless of instance). The schema for the result is also fixed and determined by the the definition of query language constructs.

- Each operator in RA accomplishes some operation on the data

how to write the query? Procedure ?

- First, figure out which tables store the data we need
- next, figure out the conditions on/properties of the data
- finally, use the RA operators to “extract” the data you need

<i>branch-name</i>	<i>loan-number</i>	<i>Cust-ID</i>	<i>amount</i>
Downtown	15	1111	1000
Downtown	41	9999	2000

Table 1: Result of Query

Relational Algebra: The Select Operation

The *select* operation selects tuples from a relation R that satisfy a given predicate, and is denoted by σ . predicate appears as subscript: $\sigma_P(R)$

used to select tuples in a relation that satisfy the selection condition

Example: select all tuples of *loan* relation where branch is “Downtown”

$$\sigma_{branch-name="Downtown"}(loan)$$

Result of query given by:

Relational Algebra: Fundamental Operations

- allow comparisons =, \neq , >, <, \leq , \geq on predicate
- larger predicates formed using logical connectives:
and (\wedge) and *or* (\vee)
- selection predicate may include comparison between attributes

Example: to find tuples from loan relation with loans greater than 1200 made by Downtown branch:

$$\sigma_{(branch-name="Downtown" \wedge amount > 1200)}(loan)$$

Example: find tuples from *customer* where customer name is the same as the street name

$$\sigma_{customer-name=street}(customer)$$

Relational Algebra: Project Operations

Sometimes we do not want all the columns of the result tuple; only want to “project” some of them.

Project operation π : selects specified attributes of tuples; attributes to be projected are specified in the subscript. $\pi_{attribute-list}(R)$ projects attributes in *attribute – list* from relation R

Example: generate list of customer IDs and the names of the branches where they have loans.

$$\pi_{branch-name,cust-ID}(loan)$$

: Composing operators to form more complex queries. Input to relational operator is a relation X ; this relation X can be output of another query. Relational algebra allows composition of RA operators– for example, $\pi(\sigma R)$.

Example: Find customers (their ID numbers) who have loans larger than \$1200 at Downtown branch; the argument to π can be a relational algebra expression.

$$\pi_{cust-ID}(\sigma_{branch-name="Downtown" \wedge amount > 1200}(loan))$$

Relational Algebra: Product/Join Operation The \times operation allows combining information from different relations – it is the set product operation.

- $R_1 \times R_2$, is the collection of tuples which results from the cross product of the two sets R_1 and R_2 .

<i>Cust-ID</i>	banker-name
1111	Johnson
4444	Jones
9999	Johnson

Table 2: *Client* Relation

- if R_1 has k attributes (A_1, A_2, \dots, A_k) and R_2 has n attributes (B_1, B_2, \dots, B_n) then $R_1 \times R_2$ has attributes $(A_1, A_2, \dots, A_k, B_1, B_2, \dots, B_n)$
- resulting schema is concatenation of the two schemas
- refer to attribute A_i of relation R_j as $R_j.A_i$;
relation name dropped if no ambiguity in attribute name

Example note: tuple of $R_1 \times R_2$ is constructed by associating all possible pairs of tuples - one each from R_1 and R_2

- if $|R_1| = n_1$ and $|R_2| = n_2$ then $|R_1 \times R_2| = n_1 \times n_2$

Example: *client* relation relates banker names to their client IDs

schema: *client - scheme* = (*Cust-ID*, *banker - name*) with table:

the scheme for $r = \textit{client} \times \textit{customer}$ is:

(client.*Cust-ID*, client.banker-name, customer.*Cust-ID*, customer.*customer-name*, customer.street, customer.zip)

Note that the *Cust-ID* attribute is repeated twice – once for each relation.

Example: Find all clients of banker Johnson and the city in which they live. If we write the

query as:

$$\sigma_{\textit{banker-name}=\textit{"Johnson"}}(\textit{client} \times \textit{customer})$$

- relation will contain tuples where *customer.Cust-ID* and *client.Cust-ID* are different; thus, customers who are not clients of Johnson will be included.
- how to resolve this ? We need to place a conditional on the tuples – i.e., a select predicate.

Answer: select tuples from above query where the two attributes agree, *client.Cust-ID* = *customer.Cust-ID*

Note: notation to identify each column of custid – relation.attribute. The *client.Cust-ID* refers to *Cust-ID* attribute from *client* relation, while *customer.Cust-ID* refers to *Cust-ID* from *customer* relation.

$\sigma_{\text{banker-name='Johnson'}}$

$(\sigma_{\text{client.Cust-ID=customer.Cust-ID}}(\text{client} \times \text{customer}))$

Conditional Join The previous example showed the concept of a conditional join – i.e., a select predicate applied immediately to the output of the cross product. Since this is a common sequence of operations, it is usually called a *conditional join* with the notation \bowtie_{θ} and interpreted as:

$$R \bowtie_{\theta} S = \sigma_{\theta}(R \times S)$$

The condition can be any predicate defined on $R \times S$; in the above example this condition was *client.Cust-ID* = *customer.Cust-ID* and the query can be rewritten as:

$\sigma_{\text{banker-name='Johnson'}}$

$(\text{client} \bowtie_{\text{client.Cust-ID=customer.Cust-ID}} \text{customer})$

Relational Algebra: Rename Operation

- What if a query needed to access the same table twice.

Example: Find names and IDs of customers who live on same street and same city as customer with cust-ID 6666.

Street and city of customer with ID 6666 is obtained from the expression \mathbf{X} :

$$\mathbf{X} = \pi_{street,city}(\sigma_{Cust-ID="6666"}(customer))$$

- given above result, with expression denoted \mathbf{X} , search again in customer to find tuples with same street and city
- now consider following query: $\sigma_P(customer \times \mathbf{X})$

P must specify equality of *street* and *city*;

cannot use *customer.street* to specify which *street* value we refer to since both *street* values are taken from customer.

- The rename operation allows renaming of a relation to prevent ambiguity. The operation $\rho_x(r)$ returns relation r under the name x .
- use rename operator to create copy of customer called *customer2*
- use this copy, customer2, to obtain street and city of custid 6666.
- refer to the street and city attributes as *customer2.street* and *customer2.city*.

$$\begin{aligned} &\pi_{customer.Cust-ID, customer.customer-name} \\ &(\sigma_{customer2.street=customer.street \wedge customer2.city=customer.city} \\ &(customer \times (\pi_{street,city}(\sigma_{Cust-ID=6666} \\ &(\rho_{customer2}(customer)))))) \end{aligned}$$

General Rename Operator In general we may want to rename the attributes and/or the entire relation. The general rename operator has the syntax $\rho_{S(B_1, B_2, \dots, B_n)}(\mathbf{R})$ where the

relation R has attributes (A_1, A_2, \dots, A_n) . The resulting relation has exactly the same tuples as R , but the name of the relation is S and the attributes of the result relation S are named B_1, B_2, \dots, B_n in order from left to right. If we only want to change the name of the relation to S and leave the attributes as they are in R then we can just say $\rho_S(R)$.

Relational Algebra: Set Operations

- the set operations work the same way as they would with sets of elements
- The **union** of two relations R and S , denoted $R \cup S$, is a relation that includes all tuples that are either in R or S or in both. Duplicate tuples are eliminated.
- The **difference** of two relations R and S , denoted $R - S$, is a relation that includes all tuples that are in R but not in S .

conditions for Set operations for two relations R and S : the “types” must match

- relations R and S must be of the same arity; same number of attributes
- domains of the i -th attribute of R and i -th attribute of S must be the same

Set Operations: Examples Example : Find (1) all customers of the Downtown branch and (2) find customers with account but no loan.

first must find customers who have loans or deposits.

loans: $\pi_{Cust-ID}(\sigma_{branch-name="Downtown"}(loan))$

deposit: $\pi_{Cust-ID}(\sigma_{branch-name="Downtown"}(deposit))$ answer for (1) is:

$\pi_{Cust-ID}(\sigma_{branch-name="Downtown"}(loan))$

$\cup \pi_{Cust-ID}(\sigma_{branch-name="Downtown"}(deposit))$

table containing one column with entries: 1111,8888,9999,5555

answer for (2) is

$$\pi_{Cust-ID}(\sigma_{branch-name="Downtown"}(loan))$$

$$- \pi_{Cust-ID}(\sigma_{branch-name="Downtown"}(deposit))$$

Set Operations: Examples

Example: Find the largest account balance.

- use concept of “not largest” balance

Example: Largest Account balance

- compute temporary relation which has accounts which are not the largest
- these are accounts with balance less than some other account

$$\pi_{deposit.balance}(\sigma_{deposit.balance < d.balance}(deposit \times \rho_d(deposit)))$$

let relation from above query be X ; from $deposit$ relation remove all tuples from X :

$$\pi_{balance}(deposit) - X$$

Relational Algebra: Additional Operations

- fundamental operations of relational algebra are sufficient to express *any* query
- however writability of language suffers (common queries may be lengthy to express)
- therefore: introduce new operations,
these can be expressed as composition of fundamental operations

Additional Operations

- Set Intersection \cap
- Joins \bowtie
 - conditional join \bowtie_P

– natural join

- Division \div
- Assignment \leftarrow

The Set Intersection Operation Find elements (from tuples) that satisfy two relations; elements that appear in both sets.

- $A \cap B = A - (A - B)$
- **Example:** Find all customers with *both* a loan and an account at the Downtown branch;

$\Pi_{Cust-ID}(\sigma_{branch-name="Downtown"}(loan))$

\cap

$\Pi_{customer-name}(\sigma_{branch-name="Downtown"}(deposit))$

Join Operation

- cartesian product results in a large set of tuples, some of which may not be relevant. A cartesian product with no accompanying selection condition is very rarely used.
- introduce join operation \bowtie_c
- combination of product and select σ_c
- $R \bowtie_c S = \sigma_c(R \times S)$
- Based on the conditions, we have two special cases of conditional joins : (1) **Equijoin** is when the condition is equality and (2) **natural join** is a “special” type of equijoin where equality is forced on all attributes with the same name.

Natural Join Operation

Definition 1 Natural join is a binary relation that is a special case of equijoin.

It forms a cartesian product of its arguments, performs selection forcing equality on all those attributes, with same names, common to both and removes duplicate columns.

Natural Join: Example

- Query: Find all customers (name and ID) who have a loan at the bank and the cities in which they live.

$$\Pi_{loan.Cust-ID, customer.name, city}(\sigma_{borrow.Cust-ID=customer.Cust-ID}(loan \times customer))$$

using natural join operation this is written as:

$$\Pi_{loan.Cust-ID, customer.name, city}(loan \bowtie customer)$$

since *Cust-ID* is a common attribute, natural join considers only pairs that have the same value on customer name **Example:** Find assets and name of all branches which have depositors

(customers with an account) living in New York.

$$\Pi_{branch-name, assets}(\sigma_{customer.city='NewYork'}(customer \bowtie deposit \bowtie branch))$$

note: natural join is associative

Natural Join: Formal Definition let $r.A$ denote value on attribute A and let $R \cap S$ denote attributes common to R and S

and let $R \cap S = \{A_1, \dots, A_n\}$

$R \cup S$ is union of attributes of R and S and denotes attribute set for natural join

$$r \bowtie s = \Pi_{R \cup S}$$

$$(\sigma_{r.A_1=s.A_1 \wedge \dots \wedge r.A_n=s.A_n} r \times s)$$

- Sometimes, Equijoin is defined to also project out the common attributes specified in the equality condition.

Example: find all customers who have both an account and a loan at Downtown branch. (this was written using set intersection before).

$$\Pi_{Cust-ID}(\sigma_{branch-name='Downtown'}(loan \bowtie deposit))$$

- which query is better ?
- note the importance of query optimization (later!)

Division Operation

- the **division** operation \div suited for queries that have a phrase “for all”
- semantics: $r_1 \div r_2$ is set of x values (unary tuples) such that for every y value in (tuple in) r_2 there is a tuple (x, y) in r_1 .

Example: Division

Example: Find all customers who have an account at all branches in Brooklyn.

all branches in Brooklyn found by: $r_2 =$

$\Pi_{branch-name}(\sigma_{branch-city="Brooklyn"}(branch))$

all *Cust-ID*, *branch-name* pairs at which customer has account:

$r_1 = \Pi_{Cust-ID,branch-name}(deposit)$

to find customers in r_1 with every branch name in r_2 , use divide operation: $r_1 \div r_2$

Division: Formal Definition Formally, $r(R)$ and $s(S)$ be relations with $S \subseteq R$

- $r \div s$ is relation on schema $R - S$, and tuple is in $r \div s$ if and only if:
 1. $t \in \Pi_{R-S}(r)$
 2. for every tuple $t_s \in s$ there is a tuple $t_r \in r$ satisfying (a) $t_r[S] = t_s[S]$ and (b) $t_r[R - S] = t$.

Assignment Operator

- use assignment to temporary variable: $x \leftarrow y$. this does not result in a relation being displayed to user. result of expression on to right is assigned to relation variable on left.
- can write RA expressions in sequential manner

Modifying Database: Operators three operations: **delete**, **insert**, and **update**

- **delete:** expressed by $r \leftarrow r - E$ to delete tuple E
example: delete all of customer with *Cust-ID* 3333 (Monk) accounts:
 $deposit \leftarrow deposit - \sigma_{Cust-ID=3333}(deposit)$
- **insert:** expressed by $r \leftarrow r \cup E$ to insert tuple
example: $deposit \leftarrow deposit \cup \{Downtown, 732, 2111, 5000\}$
- **update:** using operator $\delta_{A \leftarrow E}(r)$ to update tuple E in relation r
example: $\delta_{balance \leftarrow balance * 1.05}(deposit)$

- above updates all tuples
- can specify condition on tuples selected – but can update only one relation in one statement
- more on updates in SQL

Relational Model: Views

- conceptual model shows all relations in the database
- not desirable for all users to see entire database
- security consideration require certain data be hidden from some users
- to match user's needs, need to create personalized collection of relations

Example: clerk who needs to see loan number but has no need to see loan amount must only see the relation:

$\Pi_{branch-name, loan-number, Cust-ID}(loan)$

Example: for advertising division of branch, to list all customers of the bank (with account or loan).

Relational Model: Views

Definition 2 *Any relation that is not part of the conceptual model, or database schema, but is made visible to a user via a query expression is called a “virtual relation” or a **view**.*

note: since the view relation is not stored, a view may have to be recomputed for each requesting query – i.e., *macro*.

to define view:

create view *v* as \langle query-expression \rangle

shall return to this during SQL discussion

Relational Model: The Relational Calculus Relational algebra provides a sequence of procedures that generates answer to query

Relational calculus is a nonprocedural language: describes information required without specifying procedure for extracting the information

Relational calculus (RC) is equivalent in expressiveness to Relational algebra (RA), and is based on a subset of first-order logic – it is declarative without an implicit order of evaluation. While most relational DBMS systems use relational algebra internally, query languages such as SQL use concepts from relational calculus.

Relational calculus comes in two variants of relational calculus – (1) tuple relational calculus (TRC) and (2) Domain relational calculus (DRC). The calculus has variables, constants, comparison operators, logical connectives and quantifiers.

- In TRC, the variables range over (i.e., get bound to) tuples
- in DRC, variables range over domain elements (i.e., field/attribute values)
- both are simple subsets of first order logic

Expressions in the calculus are called formulas. An answer tuple is essentially an assignment of constants to variables that make the formula evaluate to true.

Relational Tuple Calculus.

A *tuple variable* is a variable whose values can be tuples of a relational schema.

A query/formula in the tuple relational calculus is expressed as:

- $\{t \mid P(t)\}$
- t is a tuple variable
- $P(t)$ is a property of tuple t ; it is a predicate formula that describes properties of the tuple variable (thereby defining the possible values of t).
- result is the set of all tuples t for which predicate P is true

Relational Calculus Formula

A formula in TRC can be defined recursively as:

- (B) *Atomic Formula* is of the form $t \in R$ or $X \text{ op } Y$ or $X \text{ op constant}$ where R is a relation and op is from $>, <, +, \leq, \geq, \neq, \dots$
- Formula is
 - an atomic formula or
 - $\neg p, p \wedge q, p \vee q$ where p and q are formulas or
 - $\exists X(p(X))$ where variable X is *free* in $p(X)$ or
 - $\forall X(p(X))$ where variable X is *free* in $p(X)$
- the use of *quantifiers* $\exists X, \forall X$ is said to *bind* X
 - a variable that is not bound is *free*

More on complex queries in RC

Starting with atomic predicates, we can build up new predicates by the following rules:

- Logical connectives: If p and q are predicates, then so are $p \wedge q, p \vee q, \neg p$, and $p \Rightarrow q$
 - $(x > 2) \wedge (x < 4)$
 - $(x > 2) \wedge \neg(x > 0)$
- Existential quantification: If p is a predicate then so is $\exists x(p)$
 - $\exists x.(x > 2) \wedge (x < 4)$ also written as $\exists x((x > 2) \wedge (x < 4))$
- Universal quantification: If p is a predicate then so is $\forall x.p$ (also written as $\forall x(p)$)
 - $\forall x(x > 2)$
 - $\forall x(\exists y(y > x))$

Recall some logical equivalences that you saw earlier (in CS123):

- $p \Rightarrow q \equiv \neg p \vee q$
 - whenever p is true, q must also be true.
- $\forall x.p(x) \equiv \neg \exists x.\neg p(x)$
 - p is true for all x (and therefore there cannot exist any x where p is false).
 - note that the right hand side condition can be easier to check

Free and Bound Variables in RC Formulas

The use of quantifiers, \forall and \exists , is said to *bind* the variables – it limits their domain.

- A variable is *bound* in a predicate p when p is of the form $\forall x$ or $\exists x$.
- A variable occurs *free* in p if it occurs in a position where it is not bound by an enclosing \forall or \exists .
 - x is free in $(x > 2)$
 - x is bound in $\exists x(x > y)$
- Important restriction: the variables that appear to the left of $|$ must be the *only* free variables in the formula $p(\dots)$
- Implication: the values that the free variables can legally take on are the results of the query.
- When a variable is bound, one can replace it with some other variable without altering the meaning of the expression, providing there are no name conflicts. Otherwise, the variable is defined outside our “scope”
 - $\exists x(x > 2)$ is equivalent to $\exists y(y > 2)$

Relational Calculus: Safety of Expressions

- a tuple relational calculus may generate an infinite query

$$\{t \mid \neg(t \in \text{loan})\}$$

there are infinitely many tuples outside *loan*

for each tuple relational formula P , we define a *domain* $\text{dom}(P)$ which is the set of all values referenced by P .

- A query is *safe* if no matter how we instantiate the relations, it always produces a finite answer.
- domain independent: answer is the same regardless of the domain in which it is evaluated

Unfortunately, both this definition of safety and domain independence are *semantic* conditions and are *undecidable* therefore, there are *syntactic conditions* that are used to guarantee “safe” formulas.

- One solution is for each tuple relational formula P , define the domain $\text{dom}(P)$ which is the set of all values references by P
- The formulas that are expressible in real query languages based on relational calculus are all safe. Many DB languages include additional features, such as recursion, that must be restricted in certain ways to guarantee termination and consistent answers.

Examples

- Find tuples (i.e., with attributes *branch-name*, *loannumber*, *Cust-ID*, and *amount*) for loans of over \$1200:
- what is the “type” of the elements in the result, i.e., where do they come from ?
- what is the property of the elements ?

$$\{t \mid t \in \text{loan} \wedge t[\text{amount}] > 1200\}$$

- Suppose we want to find only customer ID attribute.
we need tuples on *Cust-ID* for which there is a tuple in *borrow* with same customer name and with *amount* > 1200.
- this is an example where the type has to be inferred
- For this case use the notation:
 $\exists t \in r(Q(t))$
there exists tuple *t* in relation/set *r* which satisfies predicate *Q* n
- Query in calculus is:
 $\{t \mid \exists s \in \text{loan}(t[\text{Cust-ID}] = s[\text{Cust-ID}] \wedge s[\text{amount}] > 1200)\}$
- tuple *t* is defined only on *Cust-ID* attribute since that is only attribute for which condition is specified for *t*, thus relation on *Cust-ID* .

How about cross products in TRC ?

Example: Find customer names of customers who have loans greater than 1200. To find names, we have to query the customer relation.

- for custID with loans we had the query
 $\{t \mid \exists s \in \text{loan}(s[\text{custID}] = t[\text{custID}] \wedge s[\text{amount}] > 1200)\}$
- How about name ? It exists in Customer relation.
- For tuple *c* ∈ *Customer* what property does *c* have ?
 - its custID must be same as the customer with loan greater than 1200, *i.e.*, $c[\text{custid}] = s[\text{custid}]$
- therefore we get the query:

- $\{t | \exists s \in loan, \exists c \in customer (s[*custID*] = c[*custID*] \wedge s[*amount*] > 1200) \wedge t[*name*] = c[*name*]\}$

Domain Relational Calculus uses domain variables that take on values from an attribute's domain rather than values for tuple

A query in DRC has the form:

$$\{\langle x_1, x_2, \dots, x_n \rangle | p\}$$

- predicate p is a boolean expression over x_1, x_2, \dots, x_n
- answer includes all tuples that make the formula true
- variables come from domain of the attributes in the relation schema (in contrast to tuple calculus where variables are tuples).

DRC formulas:

- *Atomic Formula*:
 - $\langle x_1, x_2, \dots, x_n \rangle \in R$ or $X \text{ op } Y$ or $X \text{ op } constant$
 - op is one of $<, >, \leq, \geq, =, \neq$
- *Formula*: (similar definition to tuple calculus)
 - an atomic formula or
 - $\neg p, p \wedge q, p \vee q$ where p and q are formulas or
 - $\exists X(p(X))$ where variable X is *free* in $p(X)$ or
 - $\forall X(p(X))$ where variable X is *free* in $p(X)$

Example: find branch name, loan number, customer ID and amount for loans over 1200

$$\{\langle b, l, c, a \rangle \mid \langle b, l, c, a \rangle \in loan \wedge a > 1200\}$$