CSE 4020/5260
Database Systems

Instructor: Fitzroy Nembhard, Ph.D.

Week 4 & 5

The Relational Model

FLORIDA TECH
FLORIDA'S STEM UNIVERSITY®

Distribution

- All slides included in this class are for the exclusive use of students and instructors associated with Database Systems (CSE 4020/5260) at the Florida Institute of Technology
- Redistribution of the slides is not permitted without the written consent of the author.

The Relational Model

- Structure of Relational Databases

- Relational Algebra

Reading:

=> Chapter 2

=> Chapter 6, sections 1 & 2 (3 is optional).

Structure of a Relational Database

- *A relational database consists of a collection of* ***tables,*** *each of which is assigned a* ***unique name and*** *stores information about a set of entities*
- *A table has columns (attributes), each presents a specific type of information about the table*
  - $R = (A_1, A_2, \ldots, A_n)$ *is a relation schema*

    Example: instructor (ID, name, dept_name, salary)
- *Each row of a table records information about one entity, called also a tuple*
- ***A relationship between n values is represented mathematically by an n-tuple of values***

Basic Structure

- Formally, given sets D_1, D_2, …. D_n a *relation* r is a subset of $D_1 \times D_2 \times \ldots \times D_n$

- Thus, a relation is a *set* of *tuples* (a_1, a_2, …, a_n) where each $a_i \in D_i$

- Example:

  cust-name = {Jones, Smith, Curry, Lindsay}
  cust-street = {Main, North, Park}
  cust-city = {Harrison, Rye, Pittsfield}

  r = {(Jones, Main, Harrison),
  (Smith, North, Rye),
  (Curry, North, Rye),
  (Lindsay, Park, Pittsfield)}

Relations are Unordered

- Since a relation is a *set*, the order of tuples is irrelevant and may be thought of as arbitrary.

- In a real DBMS, tuple order is typically very important and not arbitrary.

- Historically, this was/is a point of contention for the theorists.

Table vs. Relation

- In a DBMS, a relation is represented or stored as a table.

- The Relation:

  { (10101,Srinivasan,Comp. Sci, 65000),
  (12121,Wu,Finance,90000),
  (15151,Mozart,Music,40000),
  :
  (98345,Kim,Elec. Eng., 80000) }

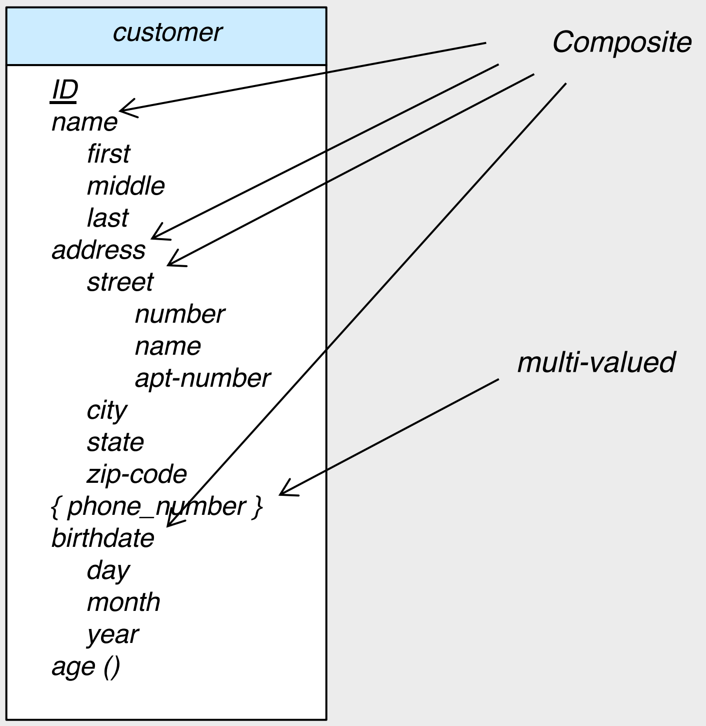
- The Table:

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Attribute Types

- Each attribute of a relation has a name.
- The set of allowed values for each attribute is called the *domain* of the attribute.
- Attribute values are required to be *atomic*, that is, indivisible.
- This will differ from ER modeling, which will have:
  - Multi-valued attributes
  - Composite attributes

Recall ER- Modeling



The Evil Value "Null"

- The special value *null* is an implicit member of every domain.
- Thus, tuples can have a *null* value for some of their attributes.
- A null value can be interpreted in several ways:
  - value is unknown
  - value does not exist
  - value is known and exists, but just hasn't been entered yet
- The null value causes complications in the definition of many operations.
- We shall consider their effect later.

Relation Schema

- Let $A_1, A_2, \ldots, A_n$ be attributes. Then $R = (A_1, A_2, \ldots, A_n)$ is a *relation schema*.

  Customer-schema = (*customer-name, customer-street, customer-city*)

- Sometimes referred to as a *relational schema* or *relational scheme*.

Database

- A database consists of multiple relations: (example)

  account - account information
  depositor - depositor information, i.e., who deposits into which accounts
  customer - customer information

- Storing all information as a single relation is possible:

  bank(*account-number, balance, customer-name*, ..)

- This results in:
  - Repetition of information (e.g. two customers own an account)
  - The need for null values (e.g. represent a customer without an account).

Relational Schemes

- Banking enterprise: (keys underlined)

  customer (<u>*customer-name*</u>, *customer-street*, *customer-city*)

  branch (<u>*branch-name*</u>, *branch-city*, *assets*)

  account (<u>*account-number*</u>, *branch-name*, *balance*)

  loan (<u>*loan-number*</u>, *branch-name*, *amount*)

  depositor (<u>*customer-name*</u>, <u>*account-number*</u>)

  borrower (<u>*customer-name*</u>, <u>*loan-number*</u>)

Relational Schemes

- University enterprise:

classroom (<u>building</u>, <u>room-number</u>, capacity)

department (<u>dept-name</u>, building, budget)

course (<u>course-id</u>, title, dept-name, credits)

instructor (<u>ID</u>, name, depart-name, salary)

section (<u>course-id</u>, <u>sec-id</u>, <u>semester</u>, <u>year</u>, building, room-number, time-slot-id)

teaches (<u>ID</u>, <u>course-id</u>, <u>sec-id</u>, <u>semester</u>, <u>year</u>)

student (<u>ID</u>, name, dept-name, tot-cred)

takes (<u>ID</u>, <u>course-id</u>, <u>sec-id</u>, <u>semester</u>, <u>year</u>, grade)

advisor (<u>s-ID</u>, <u>i-ID</u>)

time-slot (<u>time-slot-id</u>, <u>day</u>, <u>start-time</u>, end-time)

prereq (<u>course-id</u>, <u>prereq-id</u>)

Relational Schemes

- Employee enterprise:

  employee(<u>person-name</u>, street, city)

  works(<u>person-name</u>, company-name, salary)

  company(<u>company-name</u>, city)

  manages(<u>person-name</u>, manager-name)

Query Languages

- Language in which user requests information from the database.

- Recall there are two categories of DML languages
  - procedural
  - non-procedural
- Query languages can be categorized as
  - Imperative *(example: Python, C and Java)*
  - Functional *(example: FQL, Geoquery, Kleisli, XMorph)*
  - Declarative *(example: Ruby, R and Haskell)*

- "Pure" languages:
  - Relational Algebra (functional query language, according to the current version of the book)
  - Tuple Relational Calculus (declarative)
  - Domain Relational Calculus (declarative)

- Pure languages form the underlying basis of "real" query languages.

Imperative Language Code Example

- Find the name of the customer with customer_id 192-83-7465:

```
try{
        Statement st = connection.createStatement();
        ResultSet rs = st.executeQuery("select customer.customer_name" +
                                       "from customer" +
                                       "where customer.customer_id = '192-83-7465'");
        while(rs.next){
            String s = rs.getString(1);
        }
    } catch (SQLException e){}
```

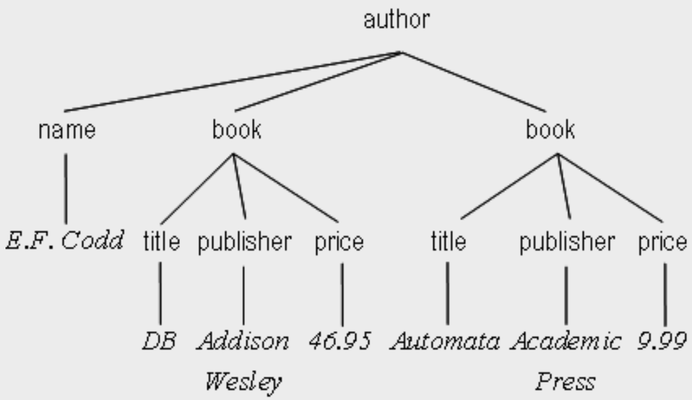
- Resultset declares what data is needed, which are included in the line of the SQL query:

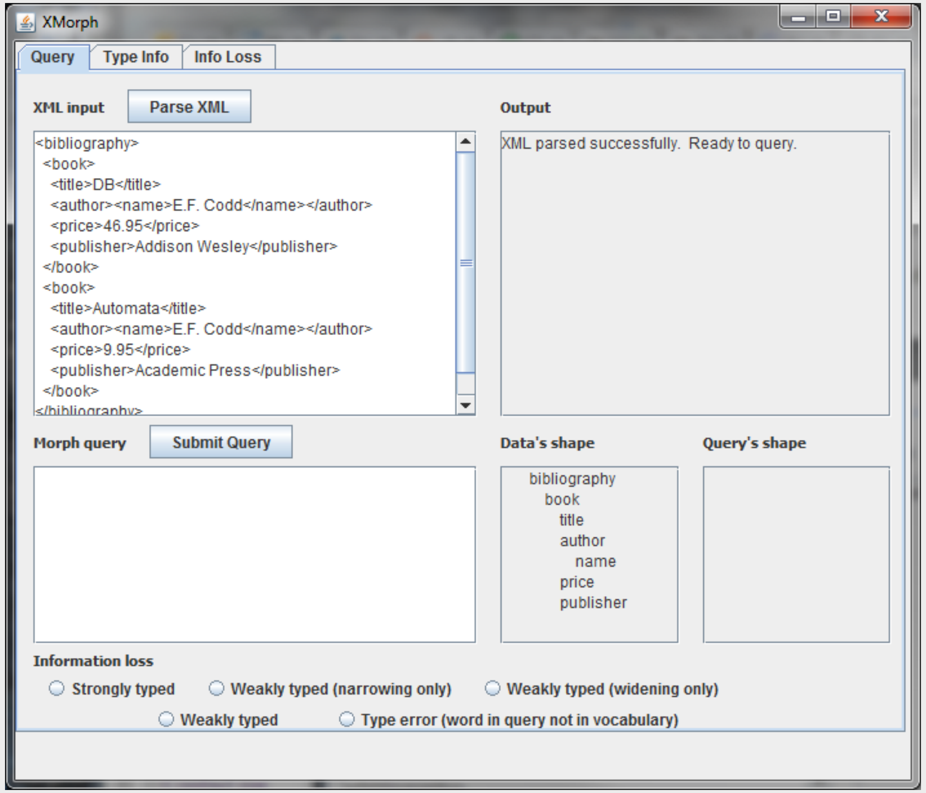
  select customer.customer-name **from** customer **where** customer.customer-id = '192-83-7465'

- The while loop states the way to retrieve the data.

Functional Language Code Example

- XMorph is an XML data transformation language and implementation. It is a functional query language (i.e., similar to a query algebra rather than a tuple calculus)





- Query 1: List authors by title

```
MORPH author [ name title ]
```

- Query 2: List titles only for the author named 'E. F. Codd’:

```
MORPH author [
    name, WHERE value == 'E.F. Codd’
    title
    ]
```

Source: https://cs.usu.edu/people/CurtisDyreson/XMorph

Declarative Language Code Example

- Haskell Database Connectivity (HDBC) is a library that provides a common abstraction or interface to different database engines like sqlite, mysql and postgres. The communication to each database engine is handled by the database driver.

```
//Connect to an SQLLite Database
import Database.HDBC
import Database.HDBC.Sqlite3

> conn <- connectSqlite3 "zotero.sqlite"
conn :: Connection
>

//A quick query

> :t quickQuery
quickQuery
  :: IConnection conn =>
     conn -> String -> [SqlValue] -> IO [[SqlValue]]
>

//Run the Query

> quickQuery conn "SELECT tagID, name FROM tags WHERE tagID > 80 LIMIT 15" []
```

Source: https://caiorss.github.io/Functional-Programming/haskell/DatabaseHDBC.html

Relational Algebra

- Functional query language (according to the book)

- Six basic operators:
  - select (σ)
  - project (Π)
  - union ($\cup$)
  - set difference ($-$)
  - cartesian product ($\times$)
  - rename (ρ)

Relational Algebra

- Each operator takes one or more relations as input and results in a new relation.

- Each operation defines:
  - Requirements or constraints on its parameters.
  - Attributes in the resulting relation, including their types and names.
  - Which tuples will be included in the result.

Select Operation – Example

- Relation *r*

A	B	C	D
α	α	1	7
α	β	5	7
β	β	12	3
β	β	23	10

- $\sigma_{A=B \wedge D > 5}(r)$

A	B	C	D
α	α	1	7
β	β	23	10

Tuple Relational Calculus – A Quick Look

- A declarative query language based on mathematical logic, where each query is of the form:

$$\{\, t \mid P(t) \,\}$$

  - Read as "the set of all tuples *t* such that predicate *P* is true for *t*"
  - P is a formula similar to that of predicate calculus
  - $\exists\, t \in r\ (Q(t)) \equiv$ "there exists" a tuple t in relation r such that Q(t) is true
  - $\forall t \in r\ (Q(t)) \equiv$ Q(t) is true "for all" tuples t in relation r

$$\{\, t \mid \exists\, t \in customer\ (t[customer\text{-}name] = \text{“Smith”})\}$$

$$\{\, t \mid \forall u \in account\ (u[balance] > 1000 \wedge u[branch\text{-}name] = \text{“Perryridge”})\,\}$$

- While we do not cover Tuple Relational Calculus in detail in this class, its syntax is used to define many of the operations done in relational algebra.

Tuple Relational Calculus – A Quick Example

- Find the *loan-number, branch-name,* and *amount* for loans of over $1200

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

Schema
branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

Domain Relational Calculus – A Quick Look

- A declarative query language based on mathematical logic, where each query is of the form:

  $\{ < x_1, x_2, x_3, ..., x_n > / \; P\,(x_1, x_2, x_3, ..., x_n\,)\,\}$

  where, $< x_1, x_2, x_3, \ldots, x_n >$ represents resulting domains variables and

  $P\,(x_1, x_2, x_3, \ldots, x_n\,)$ represents the condition or formula equivalent to the Predicate calculus.

- While we do not cover Domain Relational Calculus in detail in this class, some real query languages are built upon its constructs.

Domain Relational Calculus – A Quick Example

- Find the *loan-number, branch-name,* and *amount* for loans of over $1200

$$\{< l, b, a > \mid < l, b, a > \in loan \wedge a > 1200\}$$

Schema
branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

Difference Between Tuple Relational Calculus and Domain Relational Calculus

Tuple Relational Calculus (TRC)	Domain Relational Calculus (DRC)
In TRC, the variables represent the tuples from a specified relation.	In DRC, the variables represent the values drawn from a specified domain.
A tuple is a single element of relation. In a database, a tuple is equivalent to a row.	A domain is equivalent to column data type and any constraints on the value of the data.
Filtering variable uses a tuple of a relation.	Filtering is done based on the domain of the attributes.
Notation : $\{T \mid P\ (T)\}$ or $\{T \mid \text{Condition}\ (T)\}$	Notation : $\{\ a_1, a2, a_3, \ldots, a_n \mid P\ (a_1, a_2, a_3, \ldots, a_n)\}$
Example : $\{T \mid \text{EMPLOYEE}\ (T)\ \text{AND}\ T.\text{DEPT\_ID} = 10\}$	Example : $\{\ \mid\ <\text{EMPLOYEE}>\ \text{DEPT\_ID} = 10\ \}$

Select Operation

- Notation:

$$\sigma_p(r)$$

where *p* is a ***selection predicate*** and *r* is a relation (or more generally, a relational algebra expression).

- Defined as:

$$\sigma_p(\mathbf{r}) = \{t \mid t \in r \textbf{ and } p(t)\}$$

where *p* is a formula in propositional logic consisting of ***terms*** connected by: $\wedge$ (**and**), $\vee$ (**or**), $\neg$ (**not**), and where each term can involve the comparison operators: $=, \neq, >, \geq, <, \leq$

** Note that, in the book's notation, the predicate p cannot contain a subquery.*

Select Operation, Cont.

- Example:

$$\sigma_{\text{branch-name="Perryridge"}}(account)$$

$$\sigma_{\text{customer-name="Smith"} \wedge \text{customer-street = "main"}}(customer)$$

- Logically, one can think of selection as performing a table scan, but technically this may or may not be the case, i.e., an index may be used; that's why relational algebra is most frequently referred to as non-procedural.

Schema
branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

Project Operation – Example

- Relation *r*:

A	B	C
α	10	1
α	20	1
β	30	1
β	40	2

- $\Pi_{A,C}\ (r)$

A	C
α	1
α	1
β	1
β	2

=

A	C
α	1
β	1
β	2

Project Operation

- Notation:

$$\Pi_{A1,\ A2,\ \ldots,\ Ak}\ (r)$$

  where A_1, A_2 are attribute names and r is a relation.

- The result is defined as the relation of k columns obtained by erasing or excluding the columns that are not listed.
- Duplicate rows are *removed* from the result, since relations are *sets*.

- Example:

$$\Pi_{\mathit{account\text{-}number},\ \mathit{balance}}\ (\mathit{account})$$

Note, however, that account is not actually modified.

Schema
branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

Project Operation

- The projection operation can also be used to reorder attributes.

$$\Pi_{\textit{branch-name, balance, account-number}}\ (\textit{account})$$

As before, however, note that account is not actually modified; the order of the attributes is modified only in the result of the expression.

Schema
branch (*branch-name*, *branch-city*, *assets*)
customer (*customer-name*, *customer-street*, *customer-city*)
account (*account-number*, *branch-name*, *balance*)
loan (*loan-number*, *branch-name*, *amount*)
depositor (*customer-name*, *account-number*)
borrower (*customer-name*, *loan-number*)

Union Operation – Example

- Relations *r, s:*

r

A	B
α	1
α	2
β	1

s

A	B
α	2
β	3

$r \cup s$

A	B
α	1
α	2
β	1
β	3

Union Operation

- Notation: $r \cup s$
- Defined as:

$$r \cup s = \{t \mid t \in r \text{ or } t \in s\}$$

- Union can only be taken between *compatible* relations.
  - r and s must have the *same arity (same number of attributes)*
  - attribute domains of r and s must be compatible (e.g., 2nd attribute of r deals with "the same type of values" as does the 2nd attribute of s)
- Example: find all customers with either an account or a loan

$$\Pi_{customer\text{-}name}\ (depositor) \ \cup \Pi_{customer\text{-}name}\ (borrower)$$

Schema
branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

Union Operation – Example

- Relations *Paternity, Maternity:*

Father	*Child*
Adam	*Cain*
Adam	*Abel*
Abraham	*Isaac*
Abraham	*Ishmael*

Paternity

Mother	*Child*
Eve	*Cain*
Eve	*Seth*
Sarah	*Isaac*
Hagar	*Ishmael*

Maternity

Paternity $\cup$ Maternity?

Set Difference Operation

- Relations *r, s:*

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

$r - s$

A	B
α	1
β	1

Set Difference Operation, Cont.

- Notation $r - s$

- Defined as:

$$r - s = \{t \mid t \in r \textbf{ and } t \notin s\}$$

- Set difference can only be taken between *compatible* relations.
  - *r* and *s* must have the *same arity*
  - attribute domains of *r* and *s* must be compatible

- Note that there is no requirement that the attribute names be the same.
  - So, what about attribute names in the result?
  - Similarly for union.

Cartesian-Product Operation

- Relations *r, s*:

r

A	B
α	1
β	2

s

C	D	E
α	10	a
β	10	a
β	20	b
γ	10	b

r x *s*:

A	B	C	D	E
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b

Cartesian-Product Operation, Cont.

- Notation $r \times s$

- Defined as:

$$r \times s = \{tq \mid t \in r \textbf{ and } q \in s\}$$

- In some cases, the attributes of *r* and *s* are disjoint, i.e., that $R \cap S = \varnothing$.

- If the attributes of *r* and *s* are not disjoint:
  - Each attribute's name has its originating relations name as a prefix.
  - If *r* and *s* are the same relation, then the rename operation can be used.

Rename Operation

- The rename operator allows the results of an expression to be renamed.

- The operator appears in two forms:

  $\rho_x(E)$ - returns the expression E under the name X

  $\rho_{x\,(A1,\, A2,\, \ldots,\, An)}(E)$ - returns the expression E under name X, with attributes renamed to $A1, A2, \ldots, An$

- Typically used to resolve a name class or ambiguity.

Rename Operation – Example

- Relations *Paternity, Maternity:*

Father	*Child*
Adam	*Cain*
Adam	*Abel*
Abraham	*Isaac*
Abraham	*Ishmael*

Paternity

Mother	*Child*
Eve	*Cain*
Eve	*Seth*
Sarah	*Isaac*
Hagar	*Ishmael*

Maternity

Paternity ∪ Maternity?

ρ***Father->Parent(Paternity)*** $\cup$ ρ***Mother->Parent(Maternity)***

Parent	*Child*
Adam	*Cain*
Adam	*Abel*
Abraham	*Isaac*
Abraham	*Ishmael*
Eve	*Cain*
Eve	*Seth*
Sarah	*Isaac*
Hagar	*Ismael*

Composition of Operations

- Expressions can be built using multiple operations

A	B
α	1
β	2

r

C	D	E
α	10	a
β	10	a
β	20	b
γ	10	b

s

$r \times s$

A	B	C	D	E
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b

$\sigma_{A=C}(r \times s)$

A	B	C	D	E
α	1	α	10	a
β	2	β	10	a
β	2	β	20	b

Formal (recursive) Definition of a Relational Algebraic Expression

- A basic expression in relational algebra consists of one of the following:
  - A relation in the database
  - A constant relation

- Let E_1 and E_2 be relational-algebra expressions. Then the following are also relational-algebra expressions:
  - $E_1 \cup E_2$
  - E_1 - E_2
  - $E_1 \times E_2$
  - σ_p (E_1), P is a predicate on attributes in E_1
  - $\prod_s(E_1)$, S is a list consisting of attributes in E_1
  - ρ_x (E_1), x is the new name for the result of E_1

Banking Example

- Recall the relational schemes from the banking enterprise:

  branch (branch-name, branch-city, assets)
  customer (customer-name, customer-street, customer-city)
  account (account-number, branch-name, balance)
  loan (loan-number, branch-name, amount)
  depositor (customer-name, account-number)
  borrower (customer-name, loan-number)

Example Queries

- Find all loans of over $1200 (a bit ambiguous).

$$\sigma_{amount > 1200}(loan)$$

- Find the loan number for each loan with an amount greater than $1200.

$$\Pi_{loan\text{-}number}(\sigma_{amount > 1200}(loan))$$

Schema
branch (*<u>branch-name</u>*, *branch-city*, *assets*)
customer (*<u>customer-name</u>*, *customer-street*, *customer-city*)
account (*<u>account-number</u>*, *branch-name*, *balance*)
loan (*<u>loan-number</u>*, *branch-name*, *amount*)
depositor (*<u>customer-name</u>*, *<u>account-number</u>*)
borrower (*<u>customer-name</u>*, *<u>loan-number</u>*)

Example Queries

- Find the names of all customers who have a loan, an account, or both.

$$\Pi_{customer\text{-}name}\,(borrower) \cup \Pi_{customer\text{-}name}\,(depositor)$$

- Find the names of all customers who have a loan and an account.

$$\Pi_{customer\text{-}name}\,(borrower) \cap \Pi_{customer\text{-}name}\,(depositor)$$

Schema
branch (<u>branch-name</u>, branch-city, assets)
customer (<u>customer-name</u>, customer-street, customer-city)
account (<u>account-number</u>, branch-name, balance)
loan (<u>loan-number</u>, branch-name, amount)
depositor (<u>customer-name</u>, <u>account-number</u>)
borrower (<u>customer-name</u>, <u>loan-number</u>)

Example Queries

- Find the names of all customers who have a loan at the Perryridge branch.

$$\Pi_{customer\text{-}name}\ (\sigma_{branch\text{-}name=\text{“Perryridge”}}\ (\sigma_{borrower.loan\text{-}number\ =\ loan.loan\text{-}number}(borrower\ x\ loan)))$$

Schema
branch (*<u>branch-name</u>*, *branch-city*, *assets*)
customer (*<u>customer-name</u>*, *customer-street*, *customer-city*)
account (*<u>account-number</u>*, *branch-name*, *balance*)
loan (*<u>loan-number</u>*, *branch-name*, *amount*)
depositor (*<u>customer-name</u>*, *<u>account-number</u>*)
borrower (*<u>customer-name</u>*, *<u>loan-number</u>*)

Example Queries

- Alternative - Find the names of all customers who have a loan at the Perryridge branch.

$$\Pi_{customer\text{-}name}(\sigma_{loan.loan\text{-}number\ =\ borrower.loan\text{-}number}(borrower \times \sigma_{branch\text{-}name\ =\ \text{“Perryridge”}}(loan)))$$

Schema
branch (<u>branch-name</u>, branch-city, assets)
customer (<u>customer-name</u>, customer-street, customer-city)
account (<u>account-number</u>, branch-name, balance)
loan (<u>loan-number</u>, branch-name, amount)
depositor (<u>customer-name</u>, <u>account-number</u>)
borrower (<u>customer-name</u>, <u>loan-number</u>)

Example Queries

- Find the names of all customers who have a loan at the Perryridge branch but no account at any branch of the bank.

$$\Pi_{customer\text{-}name}\,(\sigma_{branch\text{-}name\,=\,\text{“Perryridge”}}\,(\sigma_{borrower.loan\text{-}number\,=\,loan.loan\text{-}number}\,(\text{borrower x loan})))$$
$$-\ \Pi_{customer\text{-}name}(\text{depositor})$$

- A general query writing strategy – start with something simpler, and then enhance.

Schema
branch (<u>branch-name</u>, branch-city, assets)
customer (<u>customer-name</u>, customer-street, customer-city)
account (<u>account-number</u>, branch-name, balance)
loan (<u>loan-number</u>, branch-name, amount)
depositor (<u>customer-name</u>, <u>account-number</u>)
borrower (<u>customer-name</u>, <u>loan-number</u>)

Example Queries

- Find the largest account balance:
  - Requires comparing each account balance to every other account balance.
  - Accomplished by performing a Cartesian product between account and itself.
  - Unfortunately, this results in ambiguity of attribute names.
  - Resolved by renaming one instance of the *account* relation as *d.*

$$\Pi_{balance}(account) - \Pi_{account.balance}(\sigma_{account.balance < d.balance}(account \times \rho_{d}(account)))$$

Schema
branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

Additional Operations

- The following operations do not add any “power,” or rather, capability to relational algebra queries, but simplify common queries.
  - Set intersection
  - Natural join
  - Theta join
  - Outer join
  - Division
  - Assignment

- All of the above can be defined in terms of the six basic operators.

Set-Intersection Operation

- Notation: $r \cap s$
- Defined as:

$$r \cap s = \{ t \mid t \in r \textbf{ and } t \in s \}$$

- Assume:
  - r, s have the *same arity*
  - attributes of r and s are compatible
- In terms of the 6 basic operators:

$$r \cap s = r - (r - s)$$

Set-Intersection Operation, Cont.

- Relation r, s:

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

- $r \cap s$

A	B
α	2

Natural-Join Operation

- Notation: $r \bowtie s$

- Let *r* and *s* be relations on schemas *R* and *S* respectively.
- $r \bowtie s$ is a relation that:
  - Has all attributes in $R \cup S$
  - For each pair of tuples t_r and t_s from *r* and *s*, respectively, if t_r and t_s have the same value on ***all*** attributes in $R \cap S$, add a "joined" tuple *t* to the result.

- Joining two tuples t_r and t_s creates a third tuple *t* such that:
  - *t* has the same value as t_r on attributes in *R*
  - *t* has the same value as t_s on attributes in *S*

Natural-Join Example

- Relational schemes for relations *r* and *s*, respectively:

  $R = (A, B, C, D)$

  $S = (E, B, D)$ -- Note the common attributes, which is typical.

- Resulting schema for $r \bowtie s$:

  (A, B, C, D, E)

- In terms of the 6 basic operators $r \bowtie s$ is defined as:

  $$\Pi_{r.A,\ r.B,\ r.C,\ r.D,\ s.E}\ (\sigma_{r.B\ =\ s.B\ \wedge\ r.D\ =\ s.D}\ (r\ \ x\ \ s))$$

- More generally, computing the natural join equates to a Cartesian product, followed by a selection, followed by a projection.

Natural Join Example

- Relations *r, s*:

A	B	C	D
α	1	α	a
β	2	γ	a
γ	4	β	b
α	1	γ	a
δ	2	β	b

r

B	D	E
1	a	α
3	a	β
1	a	γ
2	b	δ
3	b	$\in$

s

- Contents of $r \bowtie s$:

A	B	C	D	E
α	1	α	a	α
α	1	α	a	γ
α	1	γ	a	α
α	1	γ	a	γ
δ	2	β	b	δ

Natural Join – Another Example

- Find the names of all customers who have a loan at the Perryridge branch.

  Original Expression:

  $$\Pi_{customer\text{-}name}\ (\sigma_{branch\text{-}name=\text{“Perryridge”}}\ (\sigma_{borrower.loan\text{-}number\ =\ loan.loan\text{-}number}(borrower\ x\ loan)))$$

  Using the Natural Join Operator:

  $$\Pi_{customer\text{-}name}(\sigma_{branch\text{-}name\ =\ \text{“Perryridge”}}(borrower \bowtie loan))$$

- Specifying the join explicitly makes it look nicer, plus it helps the query optimizer.

Schema
branch (<u>branch-name</u>, branch-city, assets)
customer (<u>customer-name</u>, customer-street, customer-city)
account (<u>account-number</u>, branch-name, balance)
loan (<u>loan-number</u>, branch-name, amount)
depositor (<u>customer-name</u>, <u>account-number</u>)
borrower (<u>customer-name</u>, <u>loan-number</u>)

Natural Join – Another Example

- Find the instructor IDs for those who teach in the Crawford building.

$$\Pi_{ID}(\sigma_{building = \text{“Crawford”}}(teaches \bowtie section))$$

- In this case, the natural join is on four attributes
  – *course_id, section_id, semester,* and *year*.

section

course_id	sec_id	semester	year	building	room_number	time_slot_id
BIO-101	1	Summer	2017	Painter	514	B
BIO-301	1	Summer	2018	Painter	514	A
CS-101	1	Fall	2017	Packard	101	H
CS-101	1	Spring	2018	Packard	101	F
CS-190	1	Spring	2017	Taylor	3128	E
CS-190	2	Spring	2017	Taylor	3128	A
CS-315	1	Spring	2018	Watson	120	D
CS-319	1	Spring	2018	Watson	100	B
CS-319	2	Spring	2018	Taylor	3128	C
CS-347	1	Fall	2017	Taylor	3128	A
EE-181	1	Spring	2017	Taylor	3128	C
FIN-201	1	Spring	2018	Packard	101	B
HIS-351	1	Spring	2018	Painter	514	C
MU-199	1	Spring	2018	Packard	101	D
PHY-101	1	Fall	2017	Watson	100	A

teaches

ID	course_id	sec_id	semester	year
10101	CS-101	1	Fall	2017
10101	CS-315	1	Spring	2018
10101	CS-347	1	Fall	2017
12121	FIN-201	1	Spring	2018
15151	MU-199	1	Spring	2018
22222	PHY-101	1	Fall	2017
32343	HIS-351	1	Spring	2018
45565	CS-101	1	Spring	2018
45565	CS-319	1	Spring	2018
76766	BIO-101	1	Summer	2017
76766	BIO-301	1	Summer	2018
83821	CS-190	1	Spring	2017
83821	CS-190	2	Spring	2017
83821	CS-319	2	Spring	2018
98345	EE-181	1	Spring	2017

Theta-Join Operation

- Notation: $r \bowtie_\theta s$

- Let *r* and *s* be relations on schemas *R* and *S*, respectively, and let θ be a predicate.

- Then, $r \bowtie_\theta s$ is a relation that:
  - Has all attributes in $R \cup S$ *including duplicate attributes*.
  - For each pair of tuples t_r and t_s from *r* and *s*, respectively,
    if θ evaluates to true for t_r and t_s, then add a "joined" tuple *t* to the result.

- In terms of the 6 basic operators, $r \bowtie_\theta s$ is defined as:

  $\sigma_\theta (r \times s)$

Theta-Join Example #1

- Example:

  $R = (A, B, C, D)$

  $S = (E, B, D)$

- Resulting schema:

  $(r.A, r.B, r.C, r.D, s.E, s.B, s.D)$

Theta Join – Example #2

- Consider the following relational schemes:

  Score = (<u>ID#</u>, <u>Exam#</u>, Grade)

  Exam = (<u>Exam#</u>, Average)

- Consider the following query:

  “Find the ID#s for those students who scored less than average on some exam.”

$$\Pi_{Score.ID\#}\,(Score \bowtie_{Score.Exam\# = Exam.Exam\# \,\wedge\, Score.Grade < Exam.Average} Exam)$$

- Note the above could also be done with a natural join, followed by a selection.

Theta Join – Example #3

- Consider the following relational schemes: (Orlando temperatures)

  Temp-Avgs = (<u>Year</u>, Avg-Temp)

  Daily-Temps-2010 = (<u>Date</u>, High-Temp)

- Consider the following query:

"Find the days during 2010 where the high temperature for the day was higher than the average for some prior year."

- Looks ugly, perhaps, but phrasing the query this way does have benefits for query optimization.

$$\Pi_{Date}\ (Daily\text{-}Temps\text{-}2010 \bowtie_{Daily\text{-}Temps\text{-}2010.High\text{-}Temp\ >\ Temp\text{-}Avgs.Avg\text{-}Temp\ \wedge\ Temp\text{-}Avgs.Year\ <\ 2010}\ Temp\text{-}Avgs)$$

Outer Join

- An extension of the join operation that avoids loss of information.
- Computes the join and then adds tuples from one relation that do not match tuples in the other relation.
- Typically introduces *null* values.

Outer Join – Example

- Relation *loan:*

loan-number	*branch-name*	*amount*
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

- Relation *borrower:*

customer-name	*loan-number*
Jones	L-170
Smith	L-230
Hayes	L-155

Outer Join – Example

- **Inner Join**

  loan ⋈ *Borrower*

loan-number	*branch-name*	*amount*	*customer-name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith

- **Left Outer Join**

  loan ⟕ *Borrower*

loan-number	*branch-name*	*amount*	*customer-name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	*null*

Loan

loan-number	*branch-name*	*amount*
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

Borrower

customer-name	*loan-number*
Jones	L-170
Smith	L-230
Hayes	L-155

Outer Join – Example

- **Right Outer Join**

  loan ⟖ *borrower*

loan-number	*branch-name*	*amount*	*customer-name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-155	*null*	*null*	Hayes

- **Full Outer Join**

  loan ⟗ *borrower*

loan-number	*branch-name*	*amount*	*customer-name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	*null*
L-155	*null*	*null*	Hayes

Loan

loan-number	*branch-name*	*amount*
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

Borrower

customer-name	*loan-number*
Jones	L-170
Smith	L-230
Hayes	L-155

Example Left-Outer Join

- Consider the following relational schemes:

  Student = (<u>SS#</u>, Address, Date-of-Birth)

  Grade-Point-Average = (<u>SS#</u>, GPA)

- Consider the following query:

  "Create a list of all student SS#s and their GPAs. Be sure to include <u>all</u> students, including first semester freshman, who do not have a GPA."

- Solution:

  $\Pi_{SS\#, GPA}(Student ⟕ Grade\text{-}Point\text{-}Average)$

Outer Join

- In terms of the 6 basic operators (plus natural join ☺), let *r(R)* and *s(S)* be relations:

$$r ⟕ s = (r - \prod_R (r \bowtie s)) \times \{(null, null, \ldots, null)\} \cup (r \bowtie s)$$

where $\{(null, null, \ldots, null)\}$ is on the schema $S - R$

Division Operation

- Notation: $r \div s$

- Suited to queries that require "universal quantification," e.g., include the phrase *"for all."*

Division Operation

- Let r and s be relations on schemas R and S respectively where $S \subseteq R$.

  Assume without loss of generality that the attributes of R and S are:

  $R = (A_1, \ldots, A_m, B_1, \ldots, B_n)$

  $S = (B_1, \ldots, B_n)$

  The A_i attributes will be referred to as *prefix* attributes, and the B_i attributes will be referred to as *suffix* attributes.

  The result of $r \div s$ is a relation on schema

  $R - S = (A_1, \ldots, A_m)$

  where:

  $r \div s = \{ t \mid t \in \prod_{R-S}(r) \wedge \forall u \in s \, (tu \in r) \}$

Division – Example #1

Relations *r*, *s*:

r

A	B
α	1
α	2
α	3
β	1
γ	1
δ	1
δ	3
δ	4
$\in$	6
$\in$	1
β	2

s

B
1
2

r_{attr} - s_{attr}

A
α
α
α
β
γ
δ
δ
δ
$\in$
$\in$
β

s

B
1
2

r

A	B
α	1
α	2
α	3
β	1
γ	1
δ	1
δ	3
δ	4
$\in$	6
$\in$	1
β	2

$r \div s$:

A
α
β

γ, δ and $\in$ are not returned because when paired with s, the following are true:

$\{(\gamma, 1), (\gamma, 2)\} \not\subseteq r$
$\{(\delta, 1),(\delta, 2)\} \not\subseteq r$
$\{(\in, 1), (\in, 2)\} \not\subseteq r$
To be in r, the following must hold: $\forall\, u \in s\ (tu \in r)$

Division – Example #2

Relations *r*, *s*:

r

A	B	C	D	E
α	a	α	a	1
α	a	γ	a	1
α	a	γ	b	1
β	a	γ	a	1
β	a	γ	b	3
γ	a	γ	a	1
γ	a	γ	b	1
γ	a	β	b	1

s

D	E
a	1
b	1

$r \div s$:

A	B	C
α	a	γ
γ	a	γ

Division – Example #3

Relations *r*, *s*:

r

A	B	C	D	E
α	a	α	a	1
α	a	γ	a	1
α	a	γ	b	1
β	a	γ	a	1
β	a	γ	b	3
γ	a	γ	a	1
γ	a	γ	b	1
γ	a	β	b	1

s

B	D
a	a
a	b

$r \div s$:

A	C	E
α	γ	1
γ	γ	1

Division Operation (Cont.)

- In terms of the 6 basic operators, let *r(R)* and *s(S)* be relations, and let $S \subseteq R$:

$$r \div s = \Pi_{R-S}(r) - \Pi_{R-S}((\Pi_{R-S}(r) \times s) - \Pi_{R-S,S}(r))$$

  To see why:

  - $\Pi_{R-S,S}(r)$ simply reorders attributes of r
  - $\Pi_{R-S}(\Pi_{R-S}(r) \times s) - \Pi_{R-S,S}(r))$ gives those tuples t in $\Pi_{R-S}(r)$ such that for some tuple $u \in s$, $tu \notin r$.

- Property:
  - Let $q = r \div s$
  - Then q is the largest relation satisfying $q \times s \subseteq r$

Example Queries

- Consider the following query:

  *"Find the names of **all customers** who have an account at both the 'Downtown' and the 'Uptown' branches."*

- Query 1:

  $$\Pi_{CN}(\sigma_{BN=\text{"Downtown"}}(depositor \bowtie account)) \cap \Pi_{CN}(\sigma_{BN=\text{"Uptown"}}(depositor \bowtie account))$$

- Query 2:

  $$\Pi_{customer\text{-}name,\ branch\text{-}name}(depositor \bowtie account) \div \rho_{temp(branch\text{-}name)}(\{(\text{"Downtown"}), (\text{"Uptown"})\})$$

Schema
branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

Example Queries

- Consider the following (more general) query:

  "Find all customers who have an account at all branches located in the city of Brooklyn."

Schema
branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

- How could Query 1 be modified for this scenario?

- How about Query 2?

$$\Pi_{customer\text{-}name,\ branch\text{-}name}(depositor \bowtie account) \div \Pi_{branch\text{-}name}(\sigma_{branch\text{-}city = \text{"Brooklyn"}}(branch))$$

- By the way, what would (should) be the result of the query if there are no Brooklyn branches?

FLORIDA TECH

Assignment Operation

- The assignment operator ($\leftarrow$) provides an easy way to express complex queries.

- Example (for $r \div s$):

$$temp1 \leftarrow \Pi_{R-S}\ (r)$$
$$temp2 \leftarrow \Pi_{R-S}\ ((temp1 \times s) - \Pi_{R-S,S}\ (r))$$
$$result \leftarrow temp1 - temp2$$

*Do the exercises on the employee/works/company/manages DB!

*And also the exercises on the university DB!

$$r \div s = \Pi_{R-S}\ (r) - \Pi_{R-S}\ (\ (\Pi_{R-S}\ (r) \times s) - \Pi_{R-S,S}(r))$$

Extended Relational Algebra Operations

- Generalized Projection

- Aggregate Operator

Generalized Projection

- Extends projection by allowing arithmetic functions in the projection list.

$$\Pi_{F1,\ F2,\ \ldots,\ Fn}(E)$$

- E is any relational-algebra expression
- Each of F_1, F_2, …, F_n are arithmetic expressions involving constants and attributes in the schema of E.

Recall project from the 6 basic operators

$\Pi_{A1,\ A2,\ \ldots,\ Ak}\ (r)$

where A_1, A_2 are attribute names and r is a relation.

Generalized Projection

- Consider the following relational scheme:

  credit-info=(customer-name, limit, credit-balance)

- Give a relational algebraic expression for the following query:

  "Determine how much credit is left on each person's line of credit; Also determine the percentage of their credit line that they have already used."

$$\Pi_{\text{customer-name, limit – credit-balance, (credit-balance/limit)*100}}\ (\text{credit-info})$$

- We can also rename attributes

$$\Pi_{\text{customer-name, (limit – credit-balance) as credit-available}}(\text{credit-info})$$

Aggregate Functions

- An *aggregation function* takes a collection of values and returns a single value:

avg	-	average value
min	-	minimum value
max	-	maximum value
sum	-	sum of values
count	-	number of values

- Other aggregate functions are provided by most DBMS vendors.

- Not all aggregate operators are numeric, e.g., some apply to strings.

Hey, Mr. DB Here!
Check out
https://dev.mysql.com/doc/refman/8.0/en/string-functions.html for
example string functions.

The Aggregate Operator

- Aggregation functions are used in the *aggregate operator*:

$$_{G1,\ G2,\ \ldots,\ Gn}\ \mathcal{G}\ _{F1(\ A1),\ F2(\ A2),\ldots,\ Fn(\ An)}\ (E)$$

  - E is any relational-algebra expression.
  - G_1, G_2 …, G_n is a list of attributes on which to group (can be empty).
  - Each F_i is an aggregate function.
  - Each A_i is an attribute name.

Aggregate Function – Example

- Relation *r*:

A	*B*	*C*
α	α	7
α	β	7
β	β	3
β	β	10

$\mathcal{G}_{\,sum(c)}\,(r)$

sum-C
27

- Could also add *min*, *max*, and other aggregates to the above expression.

$\mathcal{G}_{\,sum(c),\ min(c),\ max(c)}\,(r)$

sum-C	*min-C*	*max-C*
27	3	10

Grouping – Example

- Grouping is somewhat like sorting, although not identical.

- Relation *account* grouped by *branch-name*:

account-number	*branch-name*	*balance*
A-102	Perryridge	400
A-374	Perryridge	900
A-224	Brighton	175
A-161	Brighton	850
A-435	Brighton	400
A-201	Brighton	625
A-217	Redwood	750
A-215	Redwood	750
A-222	Redwood	700

Aggregate Operation – Example

- Grouping and aggregate functions frequently occur together.

- A list of branch names and the sum of all their account balances:

$$_{branch\text{-}name}\, g\, _{sum(balance)}\ (account)$$

branch-name	*balance*
Perryridge	1300
Brighton	2050
Redwood	2200

Aggregate Operation: Grouping on Multiple Attributes

- Consider the following relational scheme:

  History = (<u>Student-Name</u>, <u>Department</u>, <u>Course-Number</u>, Grade)

- Sample data:

Student-Name	Department	Course-Number	Grade
Smith	*CSE*	*1001*	*90*
Jones	*MTH*	*2030*	*82*
Smith	*MTH*	*1002*	*73*
Brown	*PSY*	*4210*	*86*
Jones	*CSE*	*2010*	*65*
	:		

Aggregate Operation: Grouping on Multiple Attributes

- Consider the following query:

  "Construct a list of student names and, for each name, list the average course grade for each department in which the student has taken classes."

Smith	*CSE*	*87*
Smith	*MTH*	*93*
Jones	*CHM*	*88*
Jones	*CSE*	*75*
Brown	*PSY*	*97*
	:	

- Recalling the schema:

  History = (<u>Student-Name</u>, <u>Department</u>, <u>Course-Number</u>, Grade)

- Answer:

  $_{\textit{student-name, department}}\ \mathcal{g}\ _{\textit{avg(grade)}}\ (\textit{History})$

Aggregate Operation: Grouping on Multiple Attributes

- Adding *count(Course-Number)* would tell how many courses the student had in each department. Similarly, *min* and *max* could be added.

$$_{student\text{-}name,\ department}\ \mathcal{G}\ _{avg(grade),\ count(Course\text{-}Number),\ min(Grade),\ max(Grade)}(History)$$

Aggregate Operation: Grouping on Multiple Attributes

- Would the following two expressions give the same result?

$$_{student\text{-}name,\ department}\ \mathcal{G}\ _{avg(grade),\ count(Course\text{-}Number),\ min(Grade),\ max(Grade)}(History)$$

$$_{department,\ student\text{-}name}\ \mathcal{G}\ _{avg(grade),\ count(Course\text{-}Number),\ min(Grade),\ max(Grade)}(History)$$

Aggregate Operation: Naming Attributes

- Note that the aggregated attributes do not have names?

$$\mathcal{G}_{\ sum(c),\ min(c),\ max(c)}\ (r)$$

sum-C	*min-C*	*max-C*
27	3	10

Aggregate Operation: Naming Attributes

- Note that the aggregated attributes do not have names?

$$\mathcal{G}_{\,\mathbf{sum(c),\ min(c),\ max(c)}}\,(r)$$

?	?	?
27	3	10

- Aggregated attributes can be renamed in the aggregate operator:

$$_{branch\text{-}name}\,\mathcal{G}_{\,\mathbf{sum}(balance)\ \mathbf{as}\ sum\text{-}balance}\,(account)$$

FLORIDA TECH

Aggregate Functions and Null Values

- Null values are controversial.
- Various proposals exist in the research literature on whether null values should be allowed and, if so, how they should affect operations.

  (Example: http://www.dbazine.com/ofinterest/oi-articles/pascal27/)
- Null values can frequently be eliminated through normalization and decomposition.

Aggregate Functions and Null Values

- How nulls are treated by relational operators:
  - For duplicate elimination and grouping, null is treated like any other value, i.e., two nulls are assumed to be the same.
  - Aggregate functions (except for *count*) simply ignore null values.
- The above rules are consistent with SQL.
- Note how the second rule can be misleading:
  - Is *avg(grade)* actually a class average?

Null Values and Expression Evaluation

- Null values also affect how selection predicates are evaluated:
  - The result of any arithmetic expression involving *null* is *null.*
  - Comparisons with *null* returns the special truth value *unknown.*
  - Value of a predicate is treated as *false* if it evaluates to *unknown.*

$$\sigma_{balance*100 > 500}\ (account)$$

- For more complex predicates, the following three-valued logic is used:
  - OR: (*unknown* **or** *true*) = *true*
    (*unknown* **or** *false*) = *unknown*
    (*unknown* **or** *unknown*) = *unknown*
  - AND: (*true* **and** *unknown*) = *unknown*
    (*false* **and** *unknown*) = *false*
    (*unknown* **and** *unknown*) = *unknown*
  - NOT: (**not** *unknown*) = *unknown*

$$\sigma_{(balance*100 > 500) \text{ and } (branch\text{-}name = \text{“Perryridge”})}(account)$$

Schema
branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

Null Values and Expression Evaluation, Cont.

- Why doesn’t a comparison with *null* simply result in *false*?

- If *false* was used instead of *unknown*, then:

$$not\ (A < 5)$$

would not be equivalent to:

$$A >= 5$$

Why would this be a problem?

- How does a comparison with *null* resulting in *unknown* help?

Modification of the Database

- The database contents can be modified with operations:
  - Deletion
  - Insertion
  - Updating

- These operations can all be expressed using the assignment operator.
  - Some can be expressed other ways too.

Deletion

- A deletion is expressed in relational algebra by:

$$r \leftarrow r - E$$

  where r is a relation and E is a relational algebra query.

- The deletion of a single tuple is expressed by letting E be a constant relation containing one tuple.

- Only whole tuples can be deleted, not specific attribute values.

Deletion Examples

- Forget referential integrity for the moment…

"Delete all account records with a branch name equal to Perryridge."

$$account \leftarrow account - \sigma_{\textit{branch-name} = \text{"Perryridge"}}(account)$$

"Delete all loan records with amount in the range of 0 to 50."

$$loan \leftarrow loan - \sigma_{amount \geq 0 \text{ and } amount \leq 50}(loan)$$

Schema
branch (*branch-name*, *branch-city*, *assets*)
customer (*customer-name*, *customer-street*, *customer-city*)
account (*account-number*, *branch-name*, *balance*)
loan (*loan-number*, *branch-name*, *amount*)
depositor (*customer-name*, *account-number*)
borrower (*customer-name*, *loan-number*)

Deletion Examples

- Now suppose we want to maintain proper referential integrity…

  "Delete all accounts at branches located in Needham"

- **Version #1:**

$$r_1 \leftarrow \sigma_{\textit{branch-city} = \text{"Needham"}} (\textit{account} \bowtie \textit{branch})$$

$$r_2 \leftarrow \Pi_{\textit{account-number, branch-name, balance}} (r_1)$$

$$r_3 \leftarrow \Pi_{\textit{customer-name, account-number}} (\text{depositor} \bowtie r_2)$$

$$\textit{account} \leftarrow \text{account} - r_2$$

$$\textit{depositor} \leftarrow \text{depositor} - r_3$$

Schema
branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

Alternative Versions

- **Version #2:**

$r_1 \leftarrow \Pi_{branch\text{-}name}\ (\sigma_{branch\text{-}city\ =\ \text{“Needham”}}\ (branch))$

$r_2 \leftarrow \Pi_{account\text{-}number}\ (\Pi_{account\text{-}number,\ branch\text{-}name}(account) \bowtie r_1)$

$account \leftarrow account - (account \bowtie r_2)$

$depositor \leftarrow depositor - (depositor \bowtie r_2)$

- **Version #3:**

$r_1 \leftarrow (\sigma_{branch\text{-}city\ <>\ \text{“Needham”}}\ (depositor \bowtie account \bowtie branch))$

$account \leftarrow \Pi_{account\text{-}number,\ branch\text{-}name,\ balance}(r_1)$

$depositor \leftarrow \Pi_{customer\text{-}name,\ account\text{-}number}(r_1)$

Schema
branch (<u>branch-name</u>, branch-city, assets)
customer (<u>customer-name</u>, customer-street, customer-city)
account (<u>account-number</u>, branch-name, balance)
loan (<u>loan-number</u>, branch-name, amount)
depositor (<u>customer-name</u>, <u>account-number</u>)
borrower (<u>customer-name</u>, <u>loan-number</u>)

Alternative Versions

- **Version #4:**

$$r_1 \leftarrow account \bowtie \sigma_{branch\text{-}city <> \text{"Needham"}} (branch)$$

$$account \leftarrow \Pi_{account\text{-}number,\ branch\text{-}name,\ balance}(r_1)$$

$$depositor \leftarrow \Pi_{customer\text{-}name,\ account\text{-}number}(depositor \bowtie r_1)$$

- Which version is preferable?

- Note that the last two do not fit the authors' pattern for deletion, i.e., as a set-difference.

Schema
branch (<u>branch-name</u>, branch-city, assets)
customer (<u>customer-name</u>, customer-street, customer-city)
account (<u>account-number</u>, branch-name, balance)
loan (<u>loan-number</u>, branch-name, amount)
depositor (<u>customer-name</u>, <u>account-number</u>)
borrower (<u>customer-name</u>, <u>loan-number</u>)

Insertion

- In relational algebra, an insertion is expressed by:

$$r \leftarrow r \cup E$$

where r is a relation and E is a relational algebra expression.

- The insertion of a single tuple is expressed by letting E be a constant relation containing one tuple.

Insertion Examples

- Insert information in the database specifying that Smith has \$1200 in account A-973 at the Perryridge branch.

$$account \leftarrow account \cup \{(\text{A-973}, \text{“Perryridge”}, 1200)\}$$
$$\text{depositor} \leftarrow depositor \cup \{(\text{“Smith”}, \text{A-973})\}$$

- Provide, as a gift, a \$200 savings account for all loan customers at the Perryridge branch. Let the loan number serve as the account number for the new savings account.

$$r_1 \leftarrow (\sigma_{branch\text{-}name\, =\, \text{“Perryridge”}}\,(borrower \bowtie loan))$$
$$account \leftarrow account \cup \prod\nolimits_{loan\text{-}number,\ branch\text{-}name, 200}\ (r_1)$$
$$\text{depositor} \leftarrow depositor \cup \prod\nolimits_{customer\text{-}name,\ loan\text{-}number}(r_1)$$

Schema
branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

Updating

- Generalized projection is used to change one or more values in a tuple.

$$r \leftarrow \Pi_{F1,\ F2,\ \ldots,\ Fl,}\ (r)$$

- Each F_i is either:
  - The ith attribute of r, if the ith attribute is not updated, or,
  - An expression, involving only constants and attributes of r, which gives a new value for an attribute, when that attribute is to be updated.

Update Examples

- Make interest payments by increasing all balances by 5 percent.

$$account \leftarrow \Pi_{AN,\ BN,\ BAL\ *\ 1.05}\ (account)$$

where *AN*, *BN* and *BAL* stand for *account-number*, *branch-name* and *balance*, respectively.

- Pay 6 percent interest to all accounts with balances over $10,000 and pay 5 percent interest to all others.

$$account \leftarrow \Pi_{AN,\ BN,\ BAL\ *\ 1.06}\ (\sigma_{BAL\ >\ 10000}\ (account)) \cup \Pi_{AN,\ BN,\ BAL\ *\ 1.05}\ (\sigma_{BAL\ \leq\ 10000}\ (account))$$

Schema
branch (<u>branch-name</u>, branch-city, assets)
customer (<u>customer-name</u>, customer-street, customer-city)
account (<u>account-number</u>, branch-name, balance)
loan (<u>loan-number</u>, branch-name, amount)
depositor (<u>customer-name</u>, <u>account-number</u>)
borrower (<u>customer-name</u>, <u>loan-number</u>)

Views

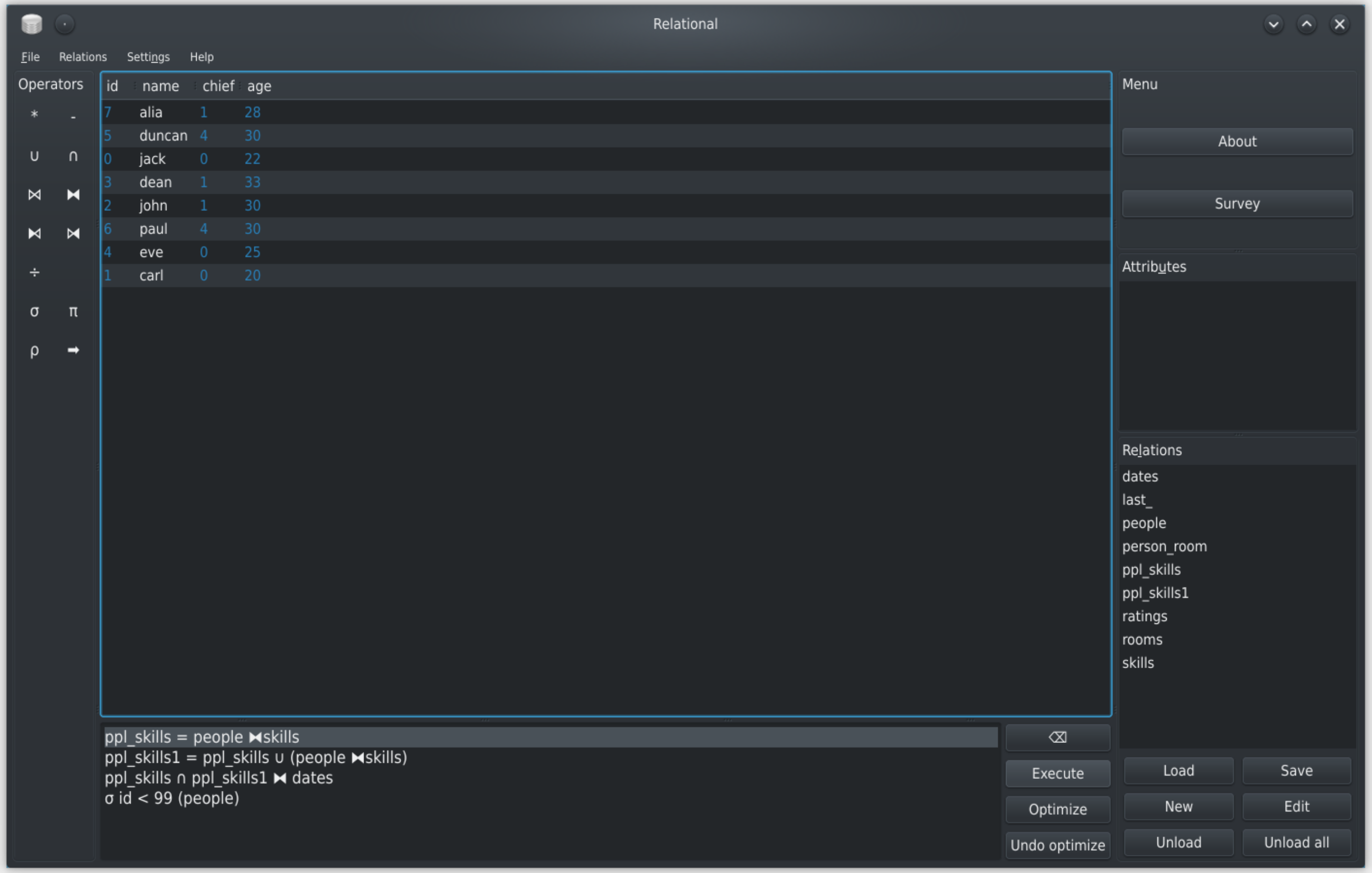
- Views are very important, but we will not consider them until chapter 3.

TOOLS FOR TESTING RELATIONAL ALGEBRA EXPRESSIONS

107

The Relational Tool

- Locate it here: https://ltworf.github.io/relational/



The Relational Tool – Syntax (Cont'd)

Example Syntax

Symbol	Name	Example	Notes
*	product	A * B	
-	difference	A - B	
∪	union	A ∪ B	
∩	intersection	A ∩ B	
÷	division	A ÷ B	
⋈	join	A ⋈ B	
⟕	left outer join	A ⟕ B	All outer joins use a python None value when they have no value to place.
⟖	right outer join	A ⟖ B	
⟗	full outer join	A ⟗ B	

FLORIDA TECH

The Relational Tool - Syntax

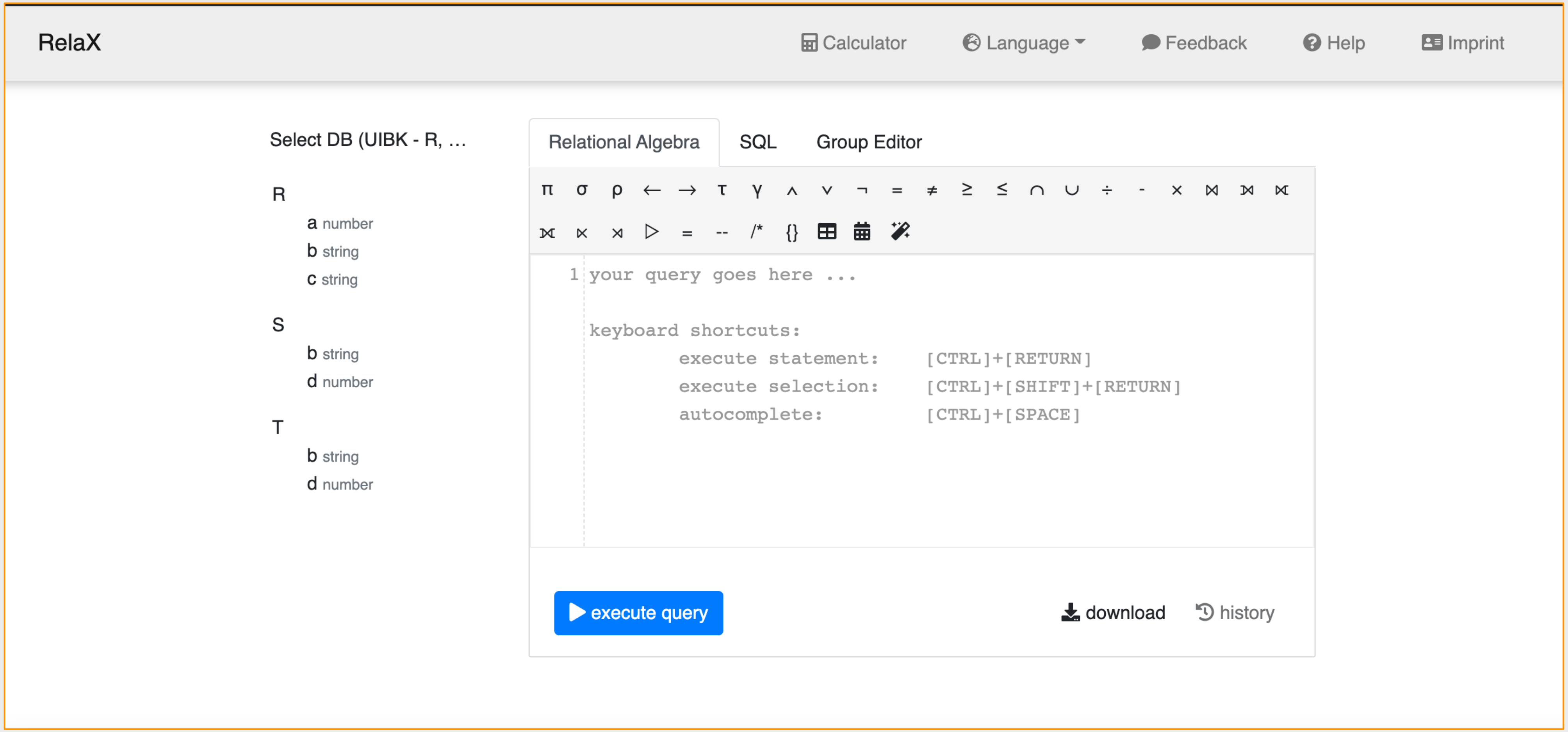
Example Syntax

Symbol	Name	Example	Note
σ	selection	σ id==index or rank>3 (A)	Expression must be written in python. The variables have the names of the fields in the relation. If the expression contains parenthesis, it **must** be surrounded by another pair of parenthesis.
π	projection	π name,age (A)	
ρ	rename	ρ old_name➡️new_name,age➡️old (A)	

Read more about the syntax here: https://ltworf.github.io/relational/allowed_expressions.html

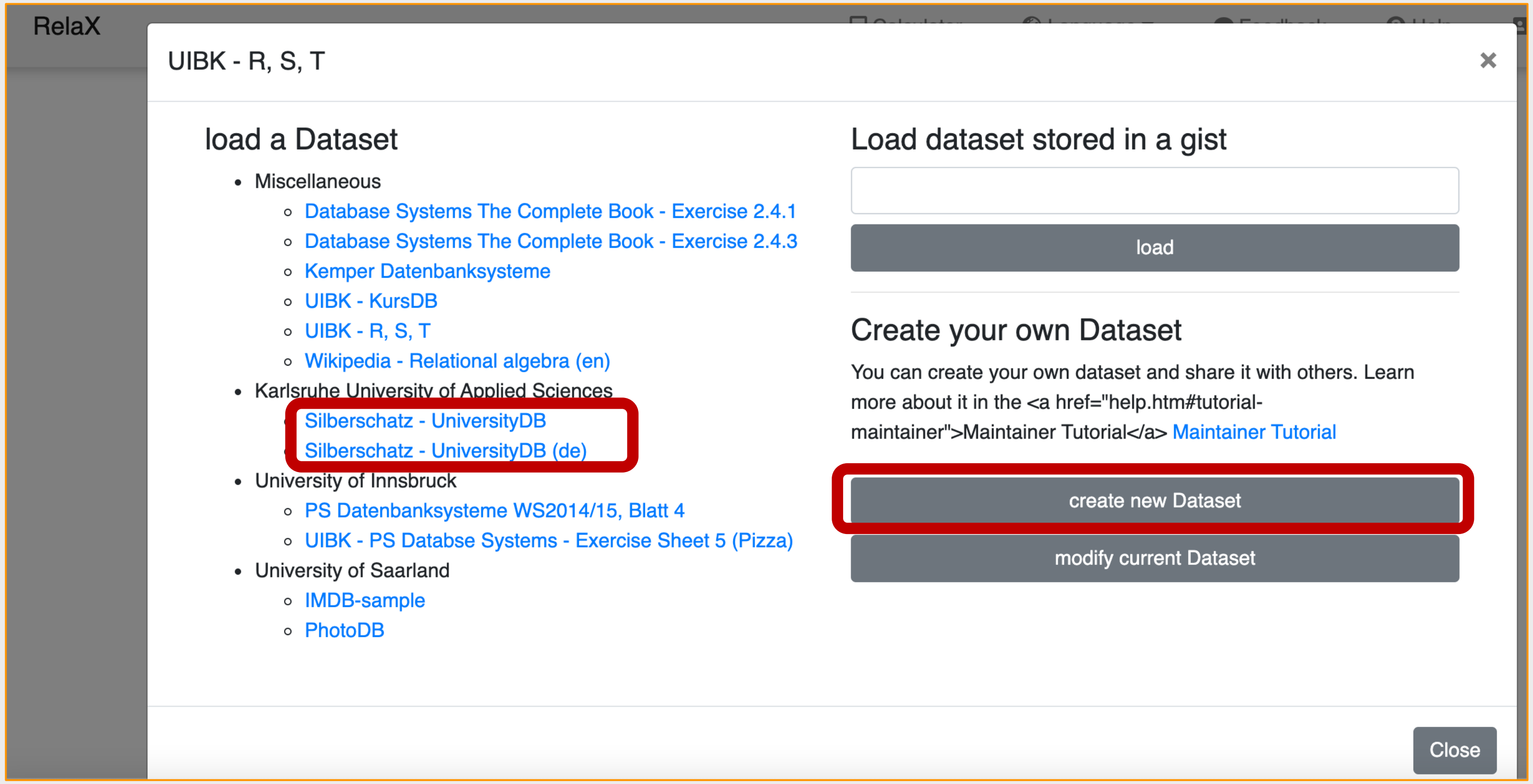
The RelaX Tool

- Locate it here: https://dbis-uibk.github.io/relax/landing



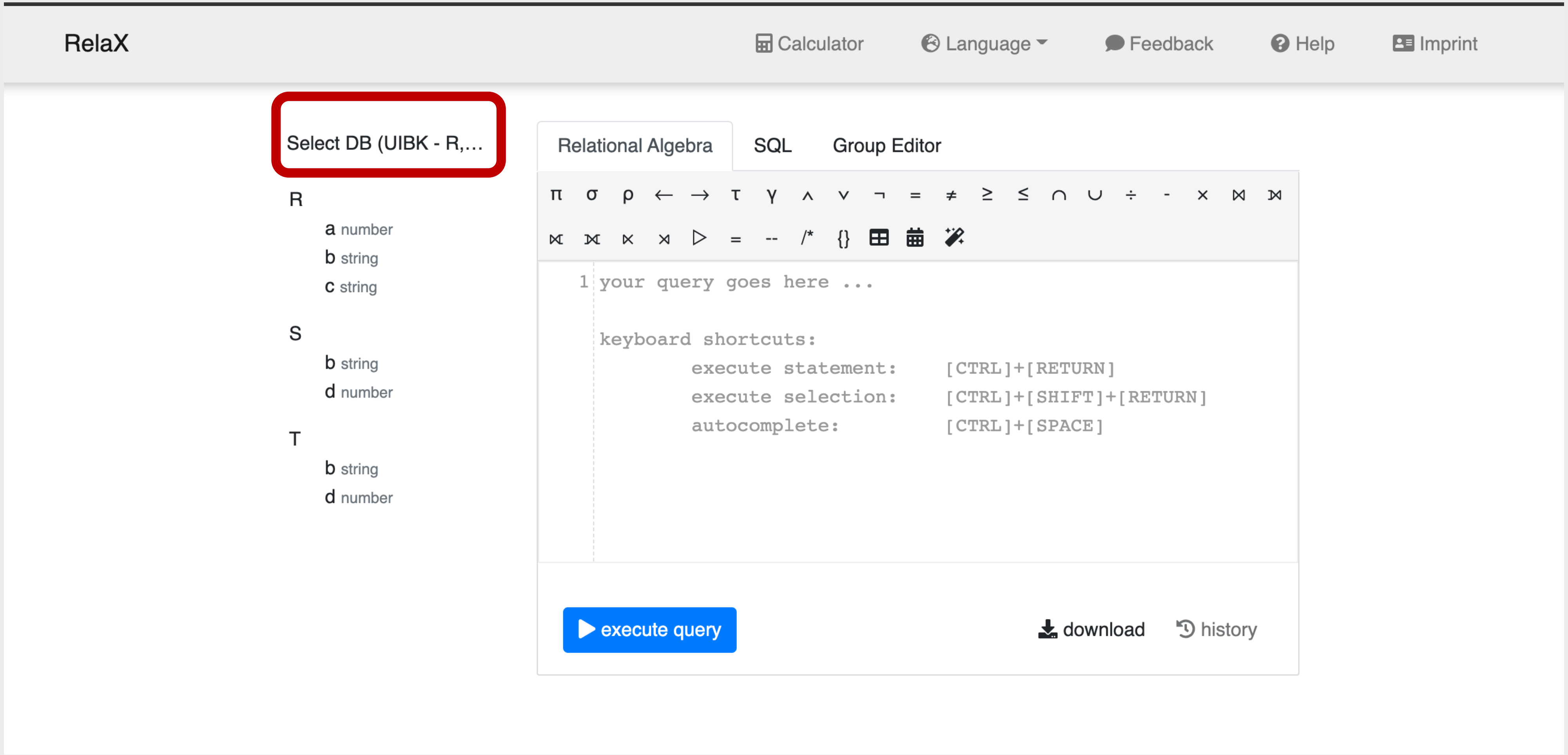
The RelaX Tool – Choosing a Database (schema)

- Notice the UniversityDB from our textbook in the list.
- You may aso add your own relation



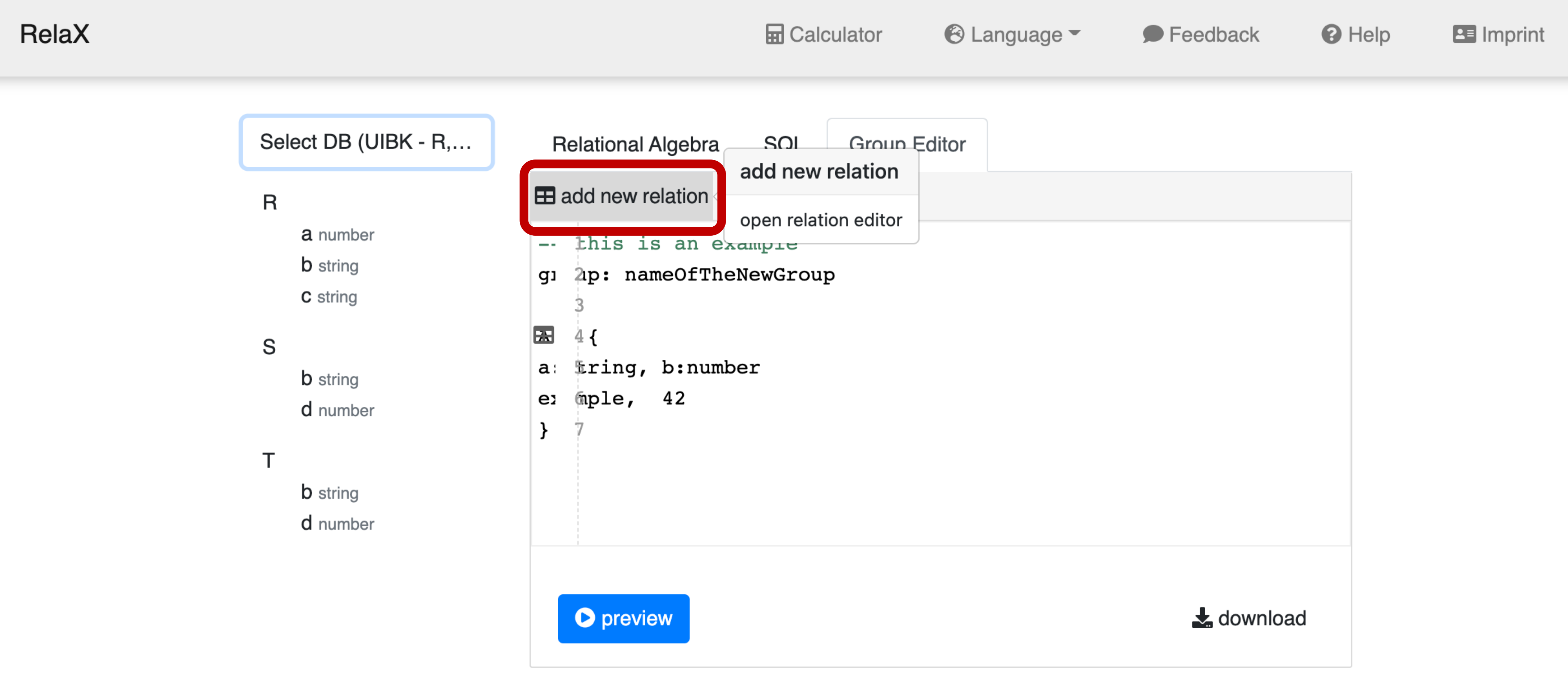
The RelaX Tool – Loading a dataset (relation) - 1

- Click Select DB to select a database or dataset



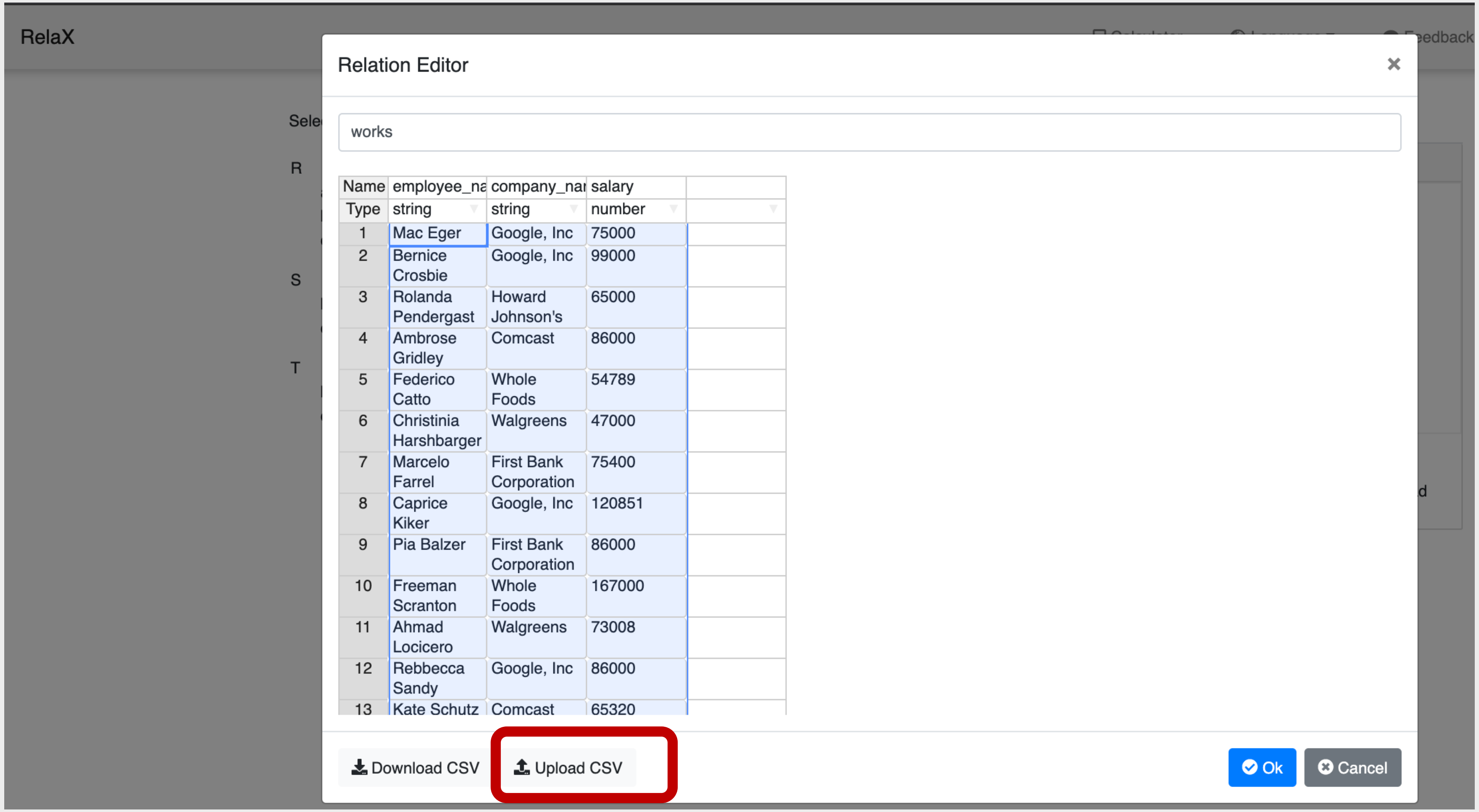
The RelaX Tool – Loading a dataset (relation) - 2

- Click "Add new Relation" to add your own dataset



The RelaX Tool – Loading a dataset (relation) - 3

- Paste your data in the cells provided or upload a CSV



The RelaX Tool – Loading a dataset (relation) - 4

- You may also load a relation using the format below

```
-- this is the works relation
group: works
works = {
employee_name:string, company_name:string, salary:number
"Mac Eger", "Google, Inc", 75000
"Bernice Crosbie", "Google, Inc", 99000
"Rolanda Pendergast", "Howard Johnson's", 65000
"}
```

The RelaX Tool – Loading a dataset (relation) - 5

- After loading data, click “Use Group in Editor” to execute a query

RelaX   Calculator   Language   Feedback

preview   download

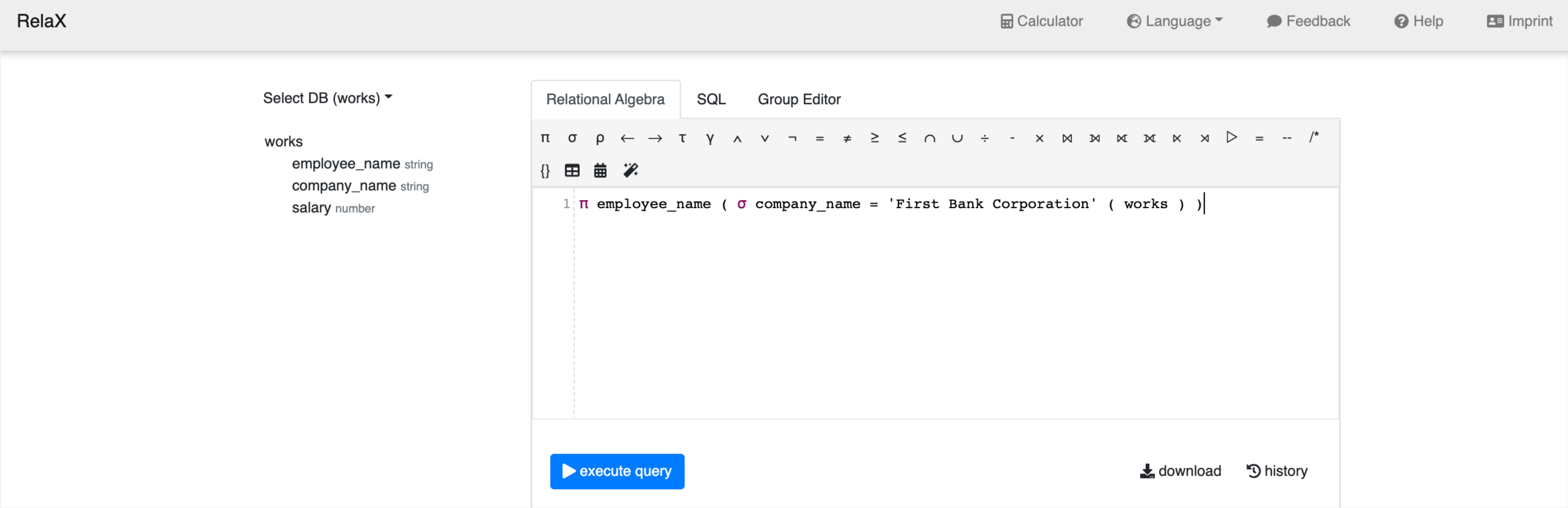
works use Group in editor

- works

works.employee_name	works.company_name	works.salary
'Mac Eger'	'Google, Inc'	75000
'Bernice Crosbie'	'Google, Inc'	99000
'Rolanda Pendergast'	'Howard Johnson's'	65000
'Ambrose Gridley'	'Comcast'	86000
'Federico Catto'	'Whole Foods'	54789
'Christinia Harshbarger'	'Walgreens'	47000
'Marcelo Farrel'	'First Bank Corporation'	75400
'Caprice Kiker'	'Google, Inc'	120851
'Pia Balzer'	'First Bank Corporation'	86000
'Freeman Scranton'	'Whole Foods'	167000
'Ahmad Locicero'	'Walgreens'	73008
'Rebbecca Sandy'	'Google, Inc'	86000
'Kate Schutz'	'Comcast'	65320
'Ilana Rodriquez'	'Whole Foods'	62000
'Wendell Nodine'	'Tmobile'	56989
'Ayesha Markowitz'	'Walgreens'	78231
'Vincent Sakai'	'First Bank Corporation'	200008
'Leeanne Caffey'	'Small Bank Corporation'	65000
'Jamel Pasha'	'Small Bank Corporation'	72000
'Kamala Harrow'	'Small Bank Corporation'	123000

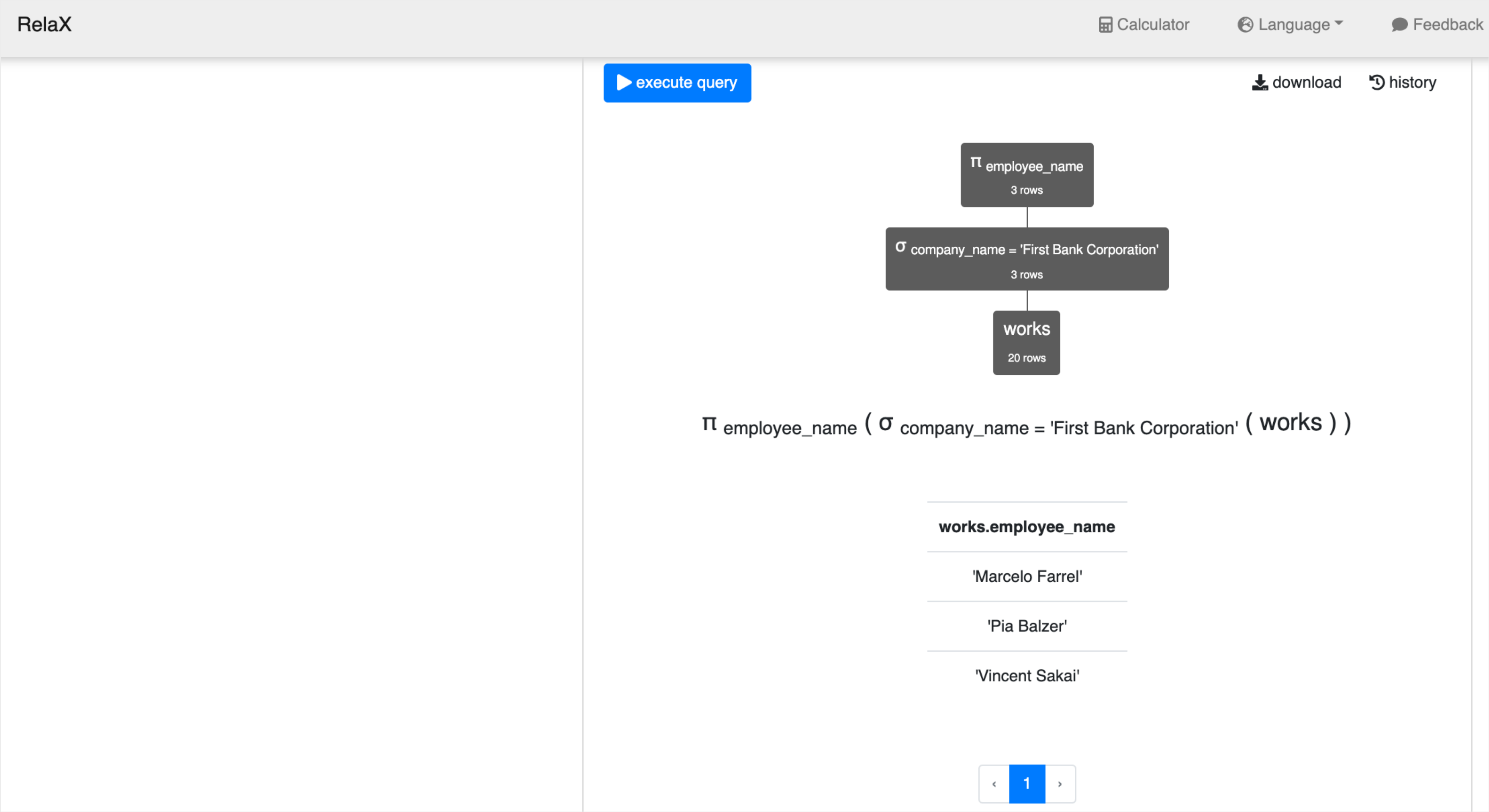
The RelaX Tool – Loading a dataset (relation) - 6

- Entering a Query in RelaX



The RelaX Tool – Loading a dataset (relation) - 7

- Output from RelaX



Relational Algebra Practice Problems

Consider the relational database for various employees (EmployeeDB):

- employee (employee-name, street, city)
- works (employee-name, company-name, salary)
- company (company-name, city)
- manages (employee-name, manager-name)

a) Find the names of all employees who work for First Bank Corporation.

b) Find the names and cities of residence of all employees who work for First Bank Corporation.

c) Find the names, street address, and cities of residence of all employees who work for First Bank Corporation and earn more than $10,000 per annum.

d) Find the names of all employees in this database who live in the same city as the company for which they work.

e) Find the names of all employees who live in the same city and on the same street as do their managers.

f) Find the names of all employees in this database who do not work for First Bank Corporation (assume that all employees work for exactly one company.)

g) Find the names of all employees who earn more than every employee of Small Bank Corporation.

h) Assume the companies may be located in several cities. Find all companies located in every city in which Small Bank Corporation is located.