Simple Database Mathematics and the Query Converter

by Aarne Ranta
http://www.cse.chalmers.se/~aarne/
March 2015

Preface

This document is written to satisfy three needs:

- fill in the formal definitions that were missing in the material used at an introductory database course, http://www.cse.chalmers.se/edu/year/2015/course/TDA357/VT2015/
- explain the ideas implemented in an emerging piece of software, the Query Converter (qconv) http://www.grammaticalframework.org/~aarne/query-converter/
- create a concise summary of the main mathematical concepts of databases

The writing started in March 2015, addressing the goals in this very order. Hence it has increasingly many holes when we go down the list.

The material is inherited from many sources:

- the slides and other documents created by earlier teachers (including at least Jonas Almström Duregård, Niklas Broberg, Rogardt Heldal, and Graham Kemp)
- the course book, *Database Systems: the Complete Book* by Garcia-Molina, Ullman, and Widom http://catalogue.pearsoned.co.uk/educator/product/Database-Systems-The-Complete-Book-International-Version/9780131354289.page
- research papers and Wikipedia articles
- a textbook draft by Jyrki Nummenmaa (in Finnish)

But the presentation and some of the notations are mine, and so are the many errors likely to be found in the first versions of this document.

Summary of the Query Converter

The program is started in a Unix shell with the command

```
qconv
```

This opens a qconv shell, with the following commands available:

```
<SQL>          run sql command
a <SQL>        show algebra for sql query
i <File>       execute SQL commands
d <File>       read and show design (E-R, schemas, English)
f <File>       read relation, analyse dependencies and keys
n <File>       read relation, normalize to BCNF and 4NF
x <XPath>      run xpath query (on the current database)
x              print database in xml
h              help
q              quit
```

Introduction

Following the tradition of the Chalmers database course, very similar to the standard textbook by Garcia-Molina et al., the material is presented in the following order:

1. database design:
    - Entity-Relationship models
    - schemas
    - functional dependencies and other constraints
    - normal forms
2. database construction and queries:
    - relational algebra
    - SQL
    - query compilation
3. data representations:
    - XML

Modelling vs. design vs. coding

Computer programming is sometimes referred to as **coding**, which means writing code in some programming language. In the case of databases, this usually means writing SQL. However, seeing programming simply as coding ignores some important work that has to be done before coding: **modelling** and **design**. We could say

programming = modelling + design + coding

which conceptually proceeds in this order. In practice, of course, the actual workflow may look different. For instance,

- Modelling and design have already been done by others, and the programmer just has to code.
- Modelling, design, and coding are made in parallel, so that we get into coding early with only a small part of the modelling and design done.
- The programming language used for coding is also used as a tool for modelling and design.

All of these can be valid reasons to emphasize coding and talk less about modelling and design. However, awareness of the three components is useful, because it helps us understand when flaws in the final code are actually flaws in the modelling or the design. This is particularly useful in the case of SQL, which does not express all the distinctions that can be made at the modelling phase.

How can we then distinguish these three components of programming (or, using a more general term, **software development**)? We should look at what each of the phases operates on: what its input and output are. Briefly:

- **Modelling** is modelling of reality. It takes **reality as input** and yields a **model as output**. The model specifies the **structure** of a piece of reality, in a precise way.
- **Design** is design of software. It takes a **model as input** and yields a **software blueprint as output**. This blueprint, unlike the model itself, takes into account the specific features of the programming language to be used in coding.
- **Coding** is the implementation of the software blueprint. It takes the **design as input** and yields **program code as output**.

Notice that each of these phases can be made mathematically precise and formal. Hence they differ from the original starting point, the reality itself. Modelling is hence essentially **formalization** of the informal reality. Formalization is a necessary step to have any task done by a computer. But it can also be useful for humans, since it increases our understanding of reality by forcing us to be more precise than usual.

Languages and formalisms

In database programming, the following formats (plus many others) are used for the different phases:

- Modelling: natural language, Entity-Relationship diagrams (E-R diagrams)
- Design: database schemas
- Coding: SQL, XML

The following transitions are commonly done between the formats:

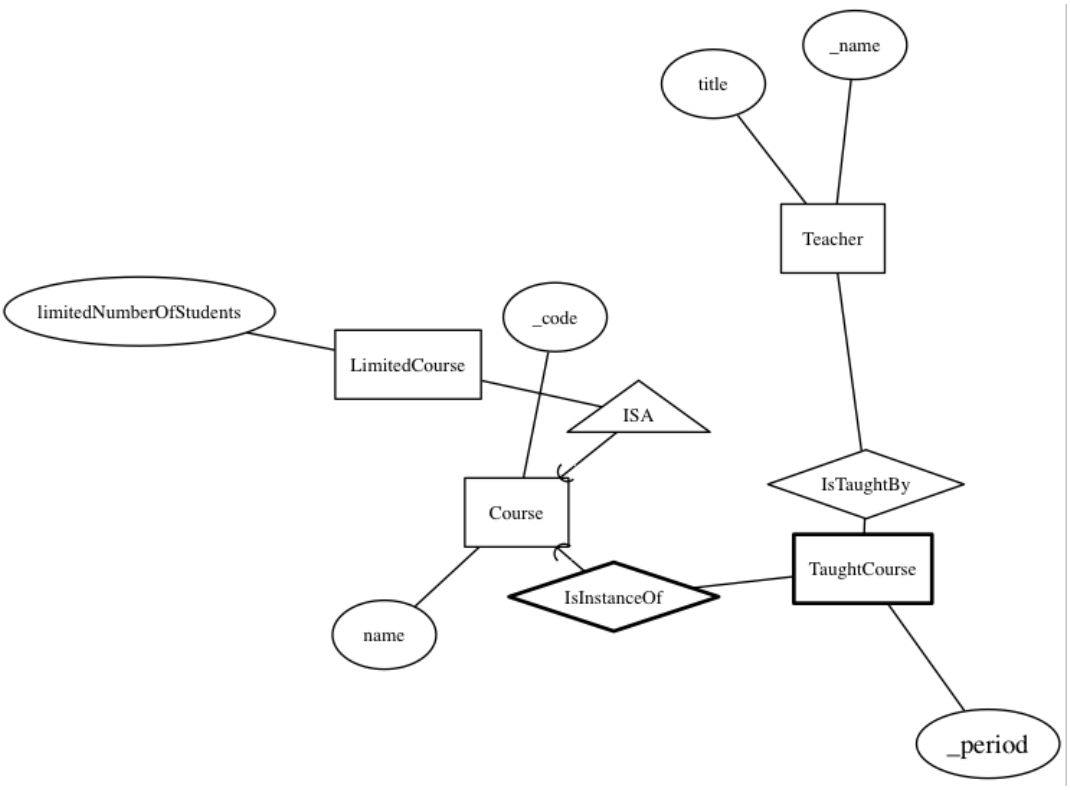
- reality -> natural language: needs knowledge and intelligence
- natural language -> E-R diagram: guided by heuristic rules
- E-R diagram -> database schema: mechanical
- (intermediate step:) database schemas can be improved by dependency analysis
- database schema -> SQL schema: needs some decisions e.g. types
- database schema -> XML: needs some decisions

We cannot say much about the modelling of reality in natural language. It is the skill that needs both knowledge of the domain and writing skills. Its outcome can also be that the piece of reality is not suitable for modelling with a database, but some other model can be better, e.g. a numerical model or a model with functions rather than static data. We take it for granted that, when starting to build a database, we are dealing with data that works with a database.

Natural language and E-R diagrams

Even though we cannot define how to transform reality into natural language, we can give some hints about how to use natural language in a way that makes the next phase, E-R models, easier. These hints are inspired by the creator of E-R models himself, Peter Chen, but they generalize his original presentation. The following table gives a conversion between a set of natural language sentences and the corresponding E-R model, which is shown graphically under the table:

Natural language	**E-R model notation in qconv**
A course has a code and a name.	ENTITY Course _code name
A teacher has a name and a title.	ENTITY Teacher _name title
A taught course of a course has a period.	WEAK ENTITY TaughtCourse Course IsInstanceOf _period
A taught course can be taught by a teacher.	RELATIONSHIP IsTaughtBy TaughtCourse -- Teacher
A limited course is a course that has a limited number of students.	ISA LimitedCourse Course limitedNumberOfStudent



The natural language descriptions on the left are mechanically generated from the E-R model component descriptions on the right. The graphical diagram is generated from the same descriptions. The descriptions thus have a formal notation, which was created *ad hoc* for the Query Converter (qconv) tool, and which supports the generation of both natural language and of graphical E-R diagrams.

To summarize: of the above components, only the “E-R model notation in qconv” was written by hand. The natural language and the diagram were produced by qconv.

The graphical diagram was produced by generating code for the Graphviz program. Graphviz is a very powerful general-purpose tool for drawing graphs, and by no means specialized on E-R diagrams. This explains why the layout is not what you might see in special-purpose tools. But it can certainly be improved by post-editing the Graphviz code generated by qconv.

The natural language generation is based on the choice of entity and attribute names. If they fall under suitable syntactic categories, the language comes out quite nice. As a rule of thumb:

- an **entity** is a **common noun** (CN), e.g. *course*
- an **attribute** is a CN that fits in the form “*the* CN *of X*”, e.g. *name* as in *the name of X*

- a **weak entity** is a CN modified by an adjectival phrase (AP), e.g. *taught course*, but it could also have the form "CN *of a* CN", e.g. *instance of a course*
- a **subentity** (ISA) is also a modified CN, e.g. *limited course*, but it could also have the form of a compound CN, e.g. *research course*
- a **relationship** is a transitive verb (TV), e.g. *teaches*, but more generally it can be any two-place predicate *P* that fits in the sentence "*X P Y*", e.g. *is taught by*.

The most difficult thing might be to decide between weak entities and subentities, because both can be expressed by modified nouns. Trying to test with the alternative forms may help: if "CN of a CN" works, then weak entity could be the right choice.

Now, could we *start* with a natural language description, parse it into an E-R description, and generate the diagram? This is certainly possible, but requires that the natural language follows a set of strict rules so that it can be parsed unambiguously. Such a set of rules is known as a **controlled natural language** (CNL). It is a long-term plan in the Query Converter to define a CNL for data modelling.

But even before we have the CNL, the above rules of thumb can be used as an intermediate step when creating E-R diagrams. The workflow could be as follows:

1. obtain a messy domain description (from a manager, a customer, or your teacher)
2. reformulate it by using more precise natural language (CNL)
3. from this, translate (almost mechanically) to an E-R description
4. let qconv generate the E-R diagram from the description

Of course, some people prefer working directly on graphical E-R tools where you see what you will get, and keep the relation to natural language completely informal. But actually creating an E-R description has another advantage: it can produce database schemas as well!

From E-R diagrams to database schemas

From the same notation as the natural language and the E-R diagram, qconv can also produce a database schema in a conventional notation. Here is the schema produced from the description above:

```
Course(_code,name)

Teacher(_name,title)

TaughtCourse(_period,_code)
  code -> Course.code
```

```
IsTaughtBy(_taughtCoursePeriod,_teacherName)
  taughtCoursePeriod -> TaughtCourse.period
  teacherName -> Teacher.name

LimitedCourse(_code,limitedNumberOfStudents)
  code -> Course.code
```

The algorithm that qconv follows is based on a document by Jonas Almström Duregård: http://www.cse.chalmers.se/edu/year/2015/course/TDA357/VT2015/TranslationV2.pdf

The E-R notation of qconv

If you want to try out the generation of E-R diagrams, natural language, and schemas, you of course need to know the notation that qconv uses for E-R descriptions. Here come the grammar:

```
Element ::=
  "ENTITY" Entity Attribute*
 | "WEAK" "ENTITY"  Entity StrongEntity Relationship Attribute*
 | "ISA" Entity SuperEntity Attribute*
 | "RELATIONSHIP" Relationship Entity  Arrow Entity Attribute*

Arrow ::= "--" | "->" | "-)"
Attribute ::= Ident | "_"Ident
Entity, StrongEntity, SuperEntity, Relationship ::= Ident
```

The notation is not (yet) quite complete for all E-R diagrams:

- It only supports two-place relationships. This may be OK because one can encode all relationships as two-place ones.
- It doesn't support weak relationships. They may actually be redundant, because a "weak relationship" just means the relationship between a week entity and the corresponding strong entity and is therefore determined by the context.
- The many/one arrows are only definable in one direction. This must be fixed.

How do you use the notation in qconv? You write it in a text file, such as edu.txt, and use the qconv command `d`

```
d edu.txt
```

In order to see the nice graphical diagram, you need to have the graphviz tool installed. Use your OS package manager (e.g. brew on Mac) to obtain it. The display is moreover now set to work on Mac OS and uses the “open” command. On Linux and Windows, you must manually open the generated file er-tmp.png by using that system’s viewer command. This must be solved later as a configuration option.

For the curious: the source code for model conversions is in the file

https://github.com/GrammaticalFramework/gf-contrib/blob/master/query-converter/Design.hs

The central datatype is `ERElement`, starting from line 87. All conversions go via this type, and may lose some information preserved in it.

Functional dependencies and independencies

A part of the domain modelling is to describe the dependencies among the data. For instance,

- that teacher depends on course and study period - a **functional dependency**
- that teachers and rooms are independent of each other - an **independency**, standardly called **multivalued dependency**

The standard material gives many explanations and examples, but concise mathematical definitions can also be useful. Thus:

Definition. A **tuple** has the form $\{A_1 = v_1,\ldots, A_n = v_n\}$, where $A_1,...,A_n$ are **attributes** and $v_1,...,v_n$ are their **values**.
Definition. The **signature** of the tuple, S, is the set of all its attributes $\{A_1,...,A_n\}$
If t is a tuple of signature S, the **projection** $t.A_i$ computes to the value v_i.
If X is a set of attributes $\{B_1,...,B_m\} \subseteq S$ and t is a tuple with signature S, we write t.X for the sequence of values $(t.B_1,...t.B_m)$.

Definition. A **relation** R of signature S is a set of tuples with signature S.

Example. Consider the following table.

country	capital	population	currency
Sweden	Stockholm	9	SEK
Finland	Helsinki	5	EUR
Estonia	Tallinn	2	EUR

- It has four attributes: country, capital, population, currency. The attributes are listed on the first line of the table, whereas the other lines each represent a tuple.
- The first tuple is
  {country = Sweden, capital = Stockholm, population = 9, currency = SEK}.
- If we call this tuple t, then t.country = Sweden.
- The signature is the set {country, capital, population, currency}.
- The whole relation is the set
  {
  {country = Sweden, capital = Stockholm, population = 9, currency = SEK},
  {country = Finland, capital = Helsinki, population = 5, currency = EUR},
  {country = Estonia, capital = Tallinn, population = 2, currency = EUR}
  }
- If t is the tuple with Sweden, as above, then

  t.{currency,population} = {currency = SEK, population = 9}

Definition. Assume X is a set of attributes and A an attribute, all belonging to a signature S. Then A is **functionally dependent on** X in a relation R, written X -> A, if

for all tuples t,u in R, if t.X = u.X then t.A = u.A

If $Y \subseteq S$, we write X -> Y to mean that X -> A for every A in Y.

Example. In the above table, even if extended with all countries of the world, we expect to get the following functional dependencies:

- country -> capital population currency
- capital -> country population currency

If we only have the above data, then functional dependencies hold between all pairs of attributes, except

- currency -> A

for any attribute A, because the currency EUR does not uniquely determine any of them.

Definition. Let X,Y,Z be disjoint subsets of a signature S such that $S = X \cup Y \cup Z$. Then Y is **functionally independent of** Z in R, written X ->> Y | Z, if

for all tuples t,u in R,
  if t.X = u.X
    then there is a tuple v in R such that
      [1] v.X = t.X
      [2] v.Y = t.Y
      [3] v.Z = u.Z

We write X ->> Y to mean X ->> Y | (S - X - Y).

Example. In the above table, we don't have enough data to prove any independencies. But thinking about the domain of countries, we could at least expect the following to hold:

- country ->> population
- country ->> currency

In other words, that the population and the currency of a country are independent of each other (and of the capital).

To see the power of these definitions, here is how easily you can prove a slightly surprising result saying that dependence implies independence:

Theorem. If X -> Y then X ->> Y.
Proof.
Assume that t,u are tuples in R such that t.X = u.X.
We select v = u.
This is a good choice, because
[1] u.X = t.X by assumption
[2] u.Y = t.Y by the functional dependency X -> Y
[3] u.Z = u.Z by reflexivity of identity.

Closures, keys, and superkeys

For the sake of analysis, a relation is charaterized by its signature S, its functional dependencies FD, and its multivalued dependencies MVD. We start with things where we don't need MVD.

Assume thus a signature (i.e. set of attributes) S and a set FD of functional dependencies.

Definition. An attribute A **follows from** a set of attributes Y, if there is an FD X -> A such that $X \subseteq Y$.

Definition. The **closure** of a set of attributes $X \subseteq S$ under a set FD of functional dependencies, denoted X+, is the set of those attributes that follow from X.

Algorithm. If $X \subseteq S$, then the closure X+, can be computed in the following way:

1. Start with X+ = X
2. Set New = {A | A $\in$ S, not A $\in$ X+, A follows from X+}
3. If New = Ø return X+, else set X+ = X+ U New and go to 1

Definition. The **closure** of a set FD of functional dependencies, denoted by FD+, is defined as follows:

$$FD+ = \{X \rightarrow A \mid X \in P(S), A \in X+, \text{not } A \in X\}$$

The last condition excludes **trivial functional dependencies**, where $A \in X$.

Definition. A set of attributes $X \subseteq S$ of attributes is a **superkey** if $S \subseteq X+$.

Example. In the above example, all sets that include either country or capital are superkeys:

- {country},{country,capital},{country,currency},{country,population,currency} etc

Definition. A set of attributes $X \subseteq S$ of attributes is a **key** if

- X is a superkey
- no proper subset of X is a superkey

Example. Only {country} and {capital} are keys in the above example.

Definition (**Boyce-Codd Normal Form).** A functional dependency X -> A **violates BCNF** if X is not a superkey and the dependency is not trivial. A relation **is in BCNF** if it has no BCNF violations.

Note. Any trivial dependency A -> A always holds even if A is not a superkey.

Definition. An attribute A is **prime** if A belongs to some key.

Example. The prime attributes in the above example are country and capital.

Definition (Third Normal Form). A functional dependency X -> A **violates 3NF** if X is not a superkey and A is not prime, and the dependency is not trivial. A relation **is in 3NF** if it has no 3NF violations.

Note. Any violation X -> A of 3NF is also a violation of BCNF, because it says that X is not a superkey. Hence, any relation that is in BCNF is also in 3NF.

Definition. An independence X ->> Y is **trivial** if $Y \subseteq X$ or $X \cup Y = S$.

Definition (Fourth Normal Form). An indepence X ->> Y **violates 4NF** if X is not a superkey and the independence is not trivial.

Note. If X -> A violates BNCF, then it also violates 4NF, because

1. it is an independence by the theorem above

2. it is not trivial because
    a. if {A} ⊆ X, then X -> A is a trivial FD and cannot violate BCNF
    b. if X U {A) = S then X is a superkey and X -> A cannot violate BCNF

Example. The above table of countries is in 4NF and hence also in BCNF and 3NF. This is because its only independences are also FDs, and all its FDs have superkeys on the left hand side. However, if we add one more column that states the equivalent of the currency in US dollars, we get a more interesting table:

country	capital	population	currency	inUSD
Sweden	Stockholm	9	SEK	0.12
Finland	Helsinki	5	EUR	1.06
Estonia	Tallinn	2	EUR	1.06

The value in USD is determined by the currency and does not depend on the other attributes. Hence we have the functional dependency

currency -> inUSD

But, since the currency does not determine the other attributes, {currency} is not a superkey. Hence the relation has a BCNF violation. In common sense terms, this corresponds to a problem in the table: the value of EUR in USD is repeated for each country that has EUR as currency, which is of course redundant.

Algorithm. Consider a relation R(S) with a set FD of functional dependencies. R can be brought to BCNF by the following steps:

1. a. If R has no BCNF violations, return R
   b. If R has a violating functional dependency X -> A, **decompose** R to two relations
      - R_1(X,A)
      - R_2(S-{A})
2. Apply step 1 to R_1 and R_2

Optimization. One can combine several violations with the same left-hand-side X to produce fewer tables. Then the violation X -> Y decomposes to R_1(X,Y) and R_2(S-Y). Or even further, to R_1(X+) and R_2(S - (X+) U X).

Example. The table above has the violation currency -> inUSD. The decomposition yields two relations, which don't have any violations left and are hence in BCNF:

R_1(currency,inUSD)

currency	inUSD
SEK	0.12
EUR	1.06

R_2(country,capital,population,currency)

country	capital	population	currency
Sweden	Stockholm	9	SEK
Finland	Helsinki	5	EUR
Estonia	Tallinn	2	EUR

Relation analysis in the Query Converter

The qconv command `f` reads a relation from a file and prints out relation info: the closure of functional dependencies, superkeys, keys, and normal form violations. The command `n` reads a relation from the same file format and prints out a decomposition in BCNF.

The format of these files is shown by
https://github.com/GrammaticalFramework/gf-contrib/blob/master/query-converter/fd-ex.txt

with the following contents:

```
A B C D
A -> B
B -> C
A -> C
B D -> A
```

Processing this file with the qconv command

```
f fd-ex.txt
```

produces the following output:

```
Attributes:
A B C D
```

Functional dependencies:
A -> B
B -> C
A -> C
B D -> A

Derived functional dependencies:
B D -> C
B C D -> A
A D -> B
A D -> C
A C -> B
A C D -> B
A B -> C
A B D -> C

Superkeys:
B D
B C D
A D
A C D
A B D
A B C D

Keys:
B D
A D

3NF violations:
B -> C
A -> C
A B -> C

BCNF violations:
B -> C
A -> B
A -> C
A C -> B
A B -> C

4NF violations:
B ->> C
A ->> B
A ->> C
A C ->> B
A B ->> C

The qconv command `n` reads a relation from a file and shows its decomposition to BCNF and 4NF. If there are no MVD's other than FD's, the 4NF is the same as the BCNF and is not shown separately. Here is an example with the same file as before:

```
n fd-ex.txt
```

produces the following output:

```
BCNF decomposition:

1. Attributes:
B C
Functional dependencies:
B -> C

2. Attributes:
A B
Functional dependencies:
A -> B

3. Attributes:
A D
Functional dependencies:
none
```

The Haskell code in

https://github.com/GrammaticalFramework/gf-contrib/blob/master/query-converter/Fundep.hs

is a direct rendering of the mathematical definitions. There is a lot of room for optimizations, but as long as the number of attributes is within the usual limits of textbook exercises, the naive algorithms work perfectly well.

TODO: Add 3NF decomposition; improve 4NF decomposition (can be very slow and even incomplete); add FD's and apply normal forms to the schemas produced by the E-R design command.

Relational algebra, SQL, and query compilation

Relational algebra interpreted in set theory

The mathematical model of relational databases is, not surprisingly, **relations**. Relations are sets of tuples, just like in the previous definitions of dependencies. In those definitions, the tuples were shown as **labelled records**, where each value is shown together with the attribute it corresponds to. However, in this chapter we will use **unlabelled tuples**, which are like records with the labels (=attributes) omitted. This data structure is more economic in practice, because we don't repeat storing the attributes for each tuple. It is also more in the tradition of mathematics, because unlabelled tuples correspond directly to **vectors**, and relations to subsets of **cartesian products**.

To access the value of a certain attribute in a labelled record, with a projection *t.a*, we search for the attribute *a* in *t*. To access a value in an unlabelled tuple, we use its **position** instead: we write *t[i]* where i is an integer that is shorter than or equal to the length of the tuple. The "signature" of an unlabelled relation is just the length of its tuples.

If we encode labelled records as unlabelled tuples, as is done in relational databases, we have to provide an **index**, which is a function that maps attributes to positions. When we convert a set of labelled records into a set of unlabelled tuples, we only need to store the attributes ones, namely in the index.

Let us proceed to the definitions. First out are the relations themselves, also known as **tables**.

Definition. A **relation** T is a subset of $V \times V \times \ldots \times V$. The **signature** of the relation is the whole set $V \times V \times \ldots \times V$.

Note. The signature could be coded simply as the number *n* of V's. However, more generally the signature is a product of distinct types, $V_1 \times V_2 \times \ldots \times V_n$, and this is what a typed language like SQL actually assumes. However, we can (mostly) ignore the distinctions between value types when discussing relational algebra.

Definition. A **tuple** of T is a sequence $t = \langle v_1, v_2, \ldots, v_n \rangle \in T$ where each $v_i \in V$.

Definition. An **attribute set** A for a relation T is a set of **identifiers** (defined e.g. as strings).

Definition. An **index** for A is a mapping $i \in A \rightarrow \{1,\ldots,n\}$

Definition. The projection from a tuple t with an attribute a can be written t.a, which means t[i(a)].

Now that we have defined the basic concepts of tables, we can proceed to defining the operators of **relational algebra**. Technically, these definitions are just one interpretation of relational algebra, which is an abstract structure. But this is, for the purposes of database, the natural interpretation.

Definition. If T is a relation with attribute set A and $B = \{b_1,...,b_m\} \subseteq A$, then the **projection** $\pi_B(T)$ is a relation with the attribute set B, defined

$$\pi_B(T) = \{<t.b_1,\ldots,t.b_m> \mid t \in T\}$$

Definition. If T is a relation of signature S and C : S -> Bool, then the **selection** $\sigma_C(T)$ is a relation with the same attribute set, defined

$$\sigma_C(T) = \{t \mid t \in T, C(t) = True\}$$

Definition. If T and U are relations with the same signature and attribute set, then the **union** $T \cup U$ is a relation, defined

$$T \cup U = \{t \mid t \in T \text{ or } t \in U\}$$

Definition. If T and U are relations with the same signature and attribute set, then the **intersection** $T \cap U$ is a relation, defined

$$T \cap U = \{t \mid t \in T \text{ and } t \in U\}$$

Definition. If T and U are relations with the same signature and attribute set, then the **difference** T - U is a relation, defined

$$T - U = \{t \mid t \in T \text{ and not } (t \in U)\}$$

Definition. If T and U are relations of signatures R and S, respectively, then the **cartesian product** T x U is a relation, defined

$$T \times U = \{t+u \mid t \in T, u \in U\}$$

where the + notation for **flattening** is defined

$$<v_1,...,v_m>+<w_1,...,w_n> = <v_1,...,v_m,w_1,...,w_n>$$

Note. The definitions of signature and attribute set for T × U are a bit tricky, because of the flattening of tuples. In the ordinary cartesian product in mathematics, an element would be

$$<<v_1,...,v_m>,<w_1,...,w_n>>$$

that is, a tuple of two tuples. This avoids the trouble of creating deep hierarchic tuples. But it creates the problems of also defining the signature and the attribute set. The signature is actually easy: just use the flattened product R x S of the two signatures. But the attribute set is trickier: since T and U can have partly the same attributes, such attributes become ambiguous.

The normal way to avoid this is by **renaming**: either rename those attributes of U that also appear in T (more “user-friendly”) or always routinely rename all attributes by **qualifying** them. For instance, all attributes a of T become 1a and attributes of U become 2a. Instead of 1 and 2, one might use the names ot T and U. But since is not necessarily the case that all relations have names, the latter alternative is not always available. The next definition introduces explicit renaming, which we will assume instead of such automatic tricks.

Definition. Let T be a relation with signature S and index map A. If B is also an index map for the signature S, then the **renaming** $\varrho_B(T)$ is a relation, defined

$$\varrho_B(T) = T$$

That is, exactly the same set of tuples. But now with the index map B instead of A.

Definition. If T and U are relations of signatures R and S, respectively, and C : R × S -> Bool, then the **theta join** $T \bowtie_C U$ is a relation, defined

$$T \bowtie_C U = \{t+u \mid t \in T, u \in U, C(t,u)=True\}$$

Note. The theta join is actually just a selection from the cartesian product, and could be equivalently defined

$$T \bowtie_C U = \sigma_C(T \times U)$$

The relational algebra notation is hence a bit misleading, as it suggests that theta join is a special case of the next operator, which turns out to be much more complex.

Definition. If T and U are relations with attribute sets A and B, respectively, then the **natural join** $T \bowtie U$ is a relation, defined

$$T \bowtie U = \{t+<t.c_1,..,t.c_k> \mid t \in T, u \in U, (\forall a \in A \cap B)(t.a=u.a)\}$$

where $\{c_1,...,c_k\} = B-A$.

Note. An equivalent definition can be given by using projection on top of selection from a cartesian product,

$$T \bowtie U = \pi_{\{a,...,am,c1,...,ck\}} (\sigma_{(\forall a \in A \cap B)(t.a=u.a)}(T \times U))$$

as well as projection on top of theta join

$$T \bowtie U = \pi_{\{a,...,am,c1,...,ck\}} (T \bowtie_{(\forall a \in A \cap B)(t.a=u.a)} U)$$

Thus we can conclude: natural join is a special case of theta join, which is a special case of the cartesian product.

Definition. Let T be a relation with attribute set A, $X = \{c_1,..,c_k\} \subseteq A$, and $G = \{g_1,...,g_m\}$ a set of **aggregation functions** g, i.e. functions that assign a value to a multiset of values collected from all projections with a given attribute a from a subset of T:

$$g_i \in P(T) \rightarrow V.$$

Then gamma_X,G(T) is a relation, defined

$$\gamma_{X,G}(T) =$$
$$\{<t.c1,...,t.ck, g_1(U), g_m(U)> \mid v \in \delta(\pi_X(T)), u \in \{t \mid t \in T, (\forall a \in X)(t.a = v.a)\}$$

where $\delta(T)$ is the set of distinct values from the multiset T, and just T if T is a set.

Todo. Outer joins, sorting, distinct. The status of these relational algebra is unclear, because outer joins use NULL values, sorting is meaningless on sets and multisets, and distinct is meaningless for sets.

Todo. Proper characterization of sets vs. multisets. The above definitions work for both.

Query compilation: from SQL to relational algebra

Executing SQL queries can be neatly done via relational algebra:

1. translate the query to an algebraic formula
2. optimize the algebraic formula
3. compute the set corresponding to the algebraic formula

The main advantages of this are:

- Relational algebra is a much simpler language than SQL, which reduces the work done at step (3).
- Relational algebra formulas obey general mathematical laws, which make (2) possible.

In a sense, relational algebra is the “machine language” of SQL, and query execution is similar to what for instance the execution of Java code does: first translate Java to JVM (Java Virtual Machine), then compute with JVM.

This is of course a slightly idealized picture, presented in textbooks and research papers. But the way queries are executed in the Query Converter (qconv) is exactly the steps (1) and (3); optimization (2) is still future work.

Before we can translate an SQL query to an algebraic formula, we must **parse** it. This means building an **abstract syntax tree** (AST). The AST is built by using a **grammar**, and qconv has its own SQL grammar, defined in

https://github.com/GrammaticalFramework/gf-contrib/blob/master/query-converter/MinSQL.bnf

The AST of SQL is then converted to the AST of a relational algebra formula, and there is a grammar for these formulas as well:

https://github.com/GrammaticalFramework/gf-contrib/blob/master/query-converter/RelAlgebra.bnf

Notice that the algebra grammar is written in LaTeX type-setting code, which means that it can be converted to nice layout with greek letters, mathematical symbols, and subscripts. The grammar can also parse relational algebra expressions written in LaTeX, but this facility is not used in qconv yet; we only use the printing of formulas from ASTs, and, more importantly, the interpretation of formulas to sets.

Notice also that the relational algebra grammar has 70 lines whereas SQL (even if not yet complete up to any standard) has 183 lines. This shows clearly that SQL is a more complex language.

The SQL-to-Algebra translation function is defined in

https://github.com/GrammaticalFramework/gf-contrib/blob/master/query-converter/Converter.hs

which mostly works by one-line conversion rules, except for the GROUP-related SQL constructs, which come out as complex combinations of γ, σ, and π. One-line conversion rules that only operate on the immediate subtrees of ASTs are called **compositional**, and it is obvious from the Converter module that the whole SELECT structure (starting from line 90) is not compositional, but this is mainly just because of the presence of aggregation. The other table expressions (start line 138) are compositional one-liners again.

In a DBMS system, the relational algebra translation is just a preparation for the query execution itself. However, qconv also provides the bonus of query translation. The command "a" applied to an SQL query, for instance

```
a SELECT * FROM Courses WHERE teacher LIKE "%e%" ;
```

renders the algebra expression in LaTeX, which converts it into a pdf file.

Note. A detail to notice is that while qconv is case-insensitive with the SQL keywords, it is case-sensitive with identifiers, such as table and attribute names. This behaviour may be changed to meet the usual standard.

Note. The query translator is incomplete, with many features of SQL missing. These come out as errors saying "not yet" and showing the internal AST.

Query execution: from relational algebra to tables

Due to the nice design of relational algebra, this step is very close to compositional:

https://github.com/GrammaticalFramework/gf-contrib/blob/master/query-converter/Algebra.hs

The conversion works in an **environment**, which maps names of tables into tables. This environment is changed every time qconv receives an SQL command other than a query: CREATE TABLE, INSERT, etc. The qconv program starts with an empty environment and builds it up during a session. The SQL commands can be imported from a file, for instance,

```
i course.sql
```

using the file

https://github.com/GrammaticalFramework/gf-contrib/blob/master/query-converter/course.sql

And of course, SQL commands can be just given on the command line, for instance,

```
SELECT * FROM Courses WHERE teacher LIKE "%e%" ;
INSERT INTO Teachers VALUES ('Abel','associate professor') ;
```

The final back-end of query execution are the set operations themselves. They are defined in the Haskell module

https://github.com/GrammaticalFramework/gf-contrib/blob/master/query-converter/Relation.hs

which uses the following datatype to represent tables:

```
data Table = Table {
  tname   :: Maybe Id,           -- name, if given
  tdata   :: [[Value]],          -- tuples, all of same length
  tindex  :: Map Id (Typ,Int),   -- from labels to types and indexes
  tlabels :: [Id]                -- labels in presentation order
  }
```

It should be noticed that we use neither sets nor multisets but **lists**. But this is OK as long as we don't exploit the order of elements in the rest of the code.

From relational database to XML

The last thing added to qconv is a rudimentary translation to XML,

https://github.com/GrammaticalFramework/gf-contrib/blob/master/query-converter/ToXML.hs

and a part of the XPath language,

https://github.com/GrammaticalFramework/gf-contrib/blob/master/query-converter/XPath.bnf

with an even more partial interpreter,

https://github.com/GrammaticalFramework/gf-contrib/blob/master/query-converter/XPath.hs

This only covers the part of XML that results as a direct translation from a relational database (that is, from tables as defined in the previous section).

The related qconv functionalities are two: first, you can show the current database as an XML dump, which consists of a DTD and a valid document. The command is simply

```
x
```

Second, you can make an XPath query on the current database, with a command like

```
x /QConvData/Teachers
```

However, the latter functionality is just under construction and does not yet contain attributes, tests, axes, etc.