In this lecture we will be starting to work on the problem how to migrate data from one table to the other, and how to do it in a sound way. We have already seen something about database normalization, but what we will be seeing today is slightly different, more basic if you want.

As a homework assignment you have seen the database for all of the Lufthansa Cargo flights. Let's concentrate on the projection of the large table that only contained flight numbers `FNR`, air plane types `ACTYPE` and the corresponding full name `ACTYPEFULLNAME`.

1 Worlds

Recall from many lectures ago, I said, that the database could be seen as a list of facts, and some rules that define the valid operations. We also named the facts. For example, we would use the following judgments to describe the database: `FNR`, `ACTYPE`, `ACTYPEFULLNAME` and `FLIGHT` for the relation that we are actually interested in. We can then look at the following list of facts,

```
LH101 : FNR, LH102 : FNR,
319 : ACTYPE, Airbus_319 : ACTTYPEFULLNAME,
t1 : FLIGHT LH101 319 Airbus_319,
t2 : FLIGHT LH102 319 Airbus_319
```

More abstractly. Let us refer to an arbitrary list of facts of this kind as Δ (pronounced delta), and we call it a world, because the facts listed in Δ simply describe what we know about a world and what we do not.

2 Operations

The worlds are not really static, they evolve over time. New facts are being added, old ones removed. Because we work with digital artifacts, it should be easy for us to define what operations we would like to support on a database. For example, consider the situation that we would like to schedule another flight, for example, `LH103` to our world Δ, but `LH103` does not fly an Airbus 319, but a Boing 737. Depending if the Boing 737 has already been introduced, we need to add a declaration of `ACTYPE` and a new declaration of `ACTYPEFULLNAME`.

We give operations names, such as `insert` f a b or `delete` f. Note the lower case syntax.

We use inference rules to describe the operational behavior of these operations. An inference rule consists of a line with multiple premisses above the line, and multiple conclusions below the line. We say that an inference rule fires, if all of the facts in the premiss list can be found in the database. When an inference rules fires, it removes all of the premisses and replaces them by the conclusions of the firing rule. We say Δ is valid, if all operations, such as `insert` and `delete` have been resolved.

The first inference rule describes how we add a new flight number using equipment that is already used on another flight. Let's say we need to add a flight f using an aircraft of type a. Furthermore, let's assume that in the database, there is already a flight f' that uses an aircraft of type a for which we know the name. Then the result is that the database needs to contain to items one for flight f and one for flight f'.

$$\frac{\texttt{FLIGHT}\ f'\ a\ b \quad \texttt{insert}\ f\ a\ b}{\texttt{FLIGHT}\ f'\ a\ b \quad \texttt{FLIGHT}\ f\ a\ b}\ \texttt{insert}_1$$

Another rule says that it is perfectly legal to add a declaration `FLIGHT` $f\ a\ b$ as long the world does not contain other declarations `FLIGHT` $?\ a\ ?$ that mention an aircraft of type a.

$$\frac{\texttt{insert}\ f\ a\ b}{\texttt{FLIGHT}\ f\ a\ b}\ \texttt{insert}_2$$

Similarly, we can imagine a delete operation.

$$\frac{\texttt{FLIGHT}\ f\ a\ b \quad \texttt{delete}\ f}{}\ \texttt{delete}$$

Depending on how complicated we would like to make things, these operations can get more and more complicated. For example, the `insert` operation may actually check for duplicates. Flight numbers should be unique. A `delete` only deletes one flight of a particular flight number, but if the database allows duplicate flights with the same number, we would actually have to iterate to remove all occurrences.

3 Functional Dependencies

Let's revisit the idea of functional dependencies. Without any additional knowledge about Δ, there is nothing we can do. If we know, whoever that there is a functional dependency in play, such as `ACTYPE` $\Rightarrow$ `ACTYPEFULLNAME` then we could try to decompose the judgment `FLIGHT` into two other judgments, let's call then `FLIGHT'` and `AIRCRAFT`, and we can then rewrite the Δ into another list of facts, let's call that Γ, staying in the Greek tradition.

A functional dependency is a good example of something that we can explain with our operations. Instead of saying that the current state of the database fulfills a functional dependency, it is a better thing to *prove* that with the given operations, starting from the empty world, all database states that are reachable with the operations will satisfy the functional dependencies. We use the history traces of the database to argue for its logical properties.

Now we can formally prove that the all possible database states reachable by our rules imply functional dependencies.

Theorem 3.1 *Let* Δ *valid and be derived according to the rules above. Then if* `FLIGHT` $f\ a\ b \in \Delta$ *and* `FLIGHT` $f'\ a\ b' \in \Delta$ *then* $b = b'$.

The proof goes by induction on the length of the rewriting steps.

4 Translation

Next, we will be looking into splitting FLIGHT into FLIGHT' and AIRCRAFT, where $\texttt{FLIGHT}'\ f\ a$ is relation between f :FNR and a : ACTYPE and AIRCRAFT is a relation between a : ACTYPE and ACTYPEFULLNAME. We write Γ for the new world to contrast it against Δ from above.

$$\frac{\texttt{AIRCRAFT}\ a\ b \quad \texttt{insert}\ f\ a\ b}{\texttt{AIRCRAFT}\ a\ b \quad \texttt{FLIGHT}'\ f\ a}\ \texttt{insert}'_1$$

Here, assuming that $\texttt{AIRCRAFT}\ a\ b'$ and $b \neq b'$, then then the above rule simply doesn't fire, and the world will never be valid.

The side condition for the next rule is that the world does not contain other declarations $\texttt{AIRCRAFT}\ a\ ?$ that mention an aircraft of type a.

$$\frac{\texttt{insert}\ f\ a\ b}{\texttt{AIRCRAFT}\ a\ b, \texttt{insert}\ f\ a\ b}\ \texttt{insert}'_2$$

Similarly, we can imagine a delete operation.

$$\frac{\texttt{FLIGHT}'\ f\ a \quad \texttt{delete}\ f}{}\ \texttt{delete}_1$$

Finally, we can define a relation between between Δ and Γ, which just tells us how to map declarations in Δ into declarations in Γ:

$$\begin{array}{lcll} [\Delta, \texttt{FLIGHT}\ f\ a\ b] & = & [\Delta], \texttt{FLIGHT}'\ f\ a & \text{if } \texttt{AIRCRAFT}\ a\ b \in [\Delta] \\ [\Delta, \texttt{FLIGHT}\ f\ a\ b] & = & [\Delta], \texttt{FLIGHT}'\ f\ a, \texttt{AIRCRAFT}\ a\ b & \text{if } \texttt{AIRCRAFT}\ a\ b \notin [\Delta] \end{array}$$

With this mapping we can show that

Theorem 4.1 *For all derivation $\mathcal{D}$ of Δ from the first set of rules above, we can translate it into a derivation $\mathcal{D}'$ of $[\Delta]$ with the second rules above.*

Proof: The proof again is a simple structural induction over the derivation of $\mathcal{D}$. We distinguish three cases.

Case: Last applied rule is $\texttt{insert}_1$: $\Delta = \Delta_0, \texttt{FLIGHT}\ f'\ a\ b, \texttt{FLIGHT}\ f\ a\ b$ and in the previous world, we have $\Delta' = \Delta_0, \texttt{FLIGHT}\ f'\ a\ b$, where we inserted operation $\texttt{insert}\ f\ a\ b$. By induction hypothesis, we have that there is a derivation $\mathcal{D}'$ of $[\Delta'] = [\Delta_0, \texttt{FLIGHT}\ f'\ a\ b]$. No matter if there is another flight $\texttt{FLIGHT}\ f''\ a\ b \in \Delta_0$ or not, $\texttt{AIRCRAFT}\ a\ b \in [\Delta']$. Thus after reinserting the insert operation into the world $\Gamma' = [\Delta'], \texttt{insert}\ f\ a\ b$ we may apply of $\texttt{insert}'_1$ and end up in the world $\Gamma = [\Delta'], \texttt{FLIGHT f a}$.

All that remains to show is that that $[\Delta] = \Gamma$:

$$\begin{aligned}[\Delta] &= [\Delta_0, \texttt{FLIGHT}\ f'\ a\ b, \texttt{FLIGHT}\ f\ a\ b] \\ &= [\Delta'], \texttt{FLIGHT}\ f\ a \\ &= \Gamma\end{aligned}$$

Cases: Last applied rule is $\texttt{insert}_2$ or `delete`. Analogous to the previous case.

□

What we have shown here, is that our operations are somehow compatible. Furthermore, we have shown that we can convince ourselves by checking the few operations, case by case, that no matter what the old database schema allowed us to derive, the new one will allow us to do exactly the same. Therefore, we can sleep tight at night.

5 Migration

The idea to change the operations while preserving meaning, e.g. related worlds are being mapped to related worlds, is the central idea to help us do data migration. When we look carefully at the proof of the previous theorem, we can actually read out how the different operations are translated. For example, the rule $\texttt{insert}_1$ is simply replaced by rule $\texttt{insert}_2$ (case 1), and similarly, $\texttt{insert}_2$ is replaced by rule $\texttt{insert}'_2$ followed by $\texttt{insert}'_1$ and `delete` is replaced by $\texttt{delete}'$. This observation helps us to formulate migration SQL expressions.

Assuming that all $\texttt{FLIGHT}\ f\ a\ b$ declarations in a world are stored in a table that was created as follows,

```
create table FLIGHT
(FNR varchar (5),
 ACTYPE varchar(3),
 ACTYPEFULLNAME varchar(30)
);
```

we can do the transformation by splitting this relation into $\texttt{FLIGHT}'\ f\ a$ and $\texttt{AIRCRAFT}\ a\ b$:

```
create table FLIGHT'
( FNR varchar (5),
  ACTYPE varchar(3)
);
insert into FLIGHT'
select FNR, ACTYPE from FLIGHT;
```

and

```
create table AIRCRAFT
( ACTYPE varchar(3),
  ACTYPEFULLNAME varchar(30)
);
insert into AIRCRAFT
select DISTINCT ACTYPE, ACTYPEFULLNAME from FLIGHT;
```

We see that while Δ was stored in table `FLIGHT`, we can convince ourselves that $[\Delta]$ is now appropriately stored in tables `FLIGHT'` and `AIRCRAFT`. What we have done is to use the knowledge that we have gained about how to relate Δ and Γ and simply expressed this relationship as SQL expressions.

In principle we can now handle any kind of migration. I would like to point out a few examples though.

Example 5.1 (Factoring) Under factoring, we understand the operation that we have described in this section. If we notice that redundant information is contained in a database (i.e. the database is neither in BCNF or 3NF), we can split a table in two smaller (just as we have learned it when we discussed normalization theory earlier this semester.

Example 5.2 (Finite expansion) Under finite expansion, we refer to the fact that that there is a field that can only take finitely many values, for example a record that includes a date interval, such as 12.12.2010 – 15.12.2010. Finite expansion may be applied, if a field can only take finitely many valued, such as {12.12, 13.12, 14.12, 15.12}, which means that that we need to duplicate record finitely many times (here four times, for every date in the interval).

Example 5.3 (Finite contraction) This situation is the opposite of finite expansion. The world can be partioned into several (possibly infinitely many) groups, where each group differs only in finitely many attributes. If it is possible to identify the groups (for example in form of an interval), then replace each partition by a unique representative in the world.